Genetic Algorithms in Timetabling and Scheduling

Hsiao-Lan Fang

Ph.D.
Department of Artificial Intelligence
University of Edinburgh
1994

Abstract

This thesis investigates the use of genetic algorithms (GAs) for solving a range of timetabling and scheduling problems. Such problems are very hard in general, and GAs offer a useful and successful alternative to existing techniques.

A framework is presented for GAs to solve modular timetabling problems in educational institutions. The approach involves three components: declaring problem-specific constraints, constructing a problem-specific evaluation function and using a problem-independent GA to attempt to solve the problem. Successful results are demonstrated and a general analysis of the reliability and robustness of the approach is conducted. The basic approach can readily handle a wide variety of general timetabling problem constraints, and is therefore likely to be of great practical usefulness (indeed, an earlier version is already in use). The approach relies for its success on the use of specially designed mutation operators which greatly improve upon the performance of a GA with standard operators.

A framework for GAs in job shop and open shop scheduling is also presented. One of the key aspects of this approach is the use of specially designed representations for such scheduling problems. The representations implicitly encode a schedule by encoding instructions for a schedule builder. The general robustness of this approach is demonstrated with respect to experiments on a range of widely-used benchmark problems involving many different schedule quality criteria. When compared against a variety of common heuristic search approaches, the GA approach is clearly the most successful method overall. An extension to the representation, in which choices of heuristic for the schedule builder are also incorporated in the chromosome, is found to lead to new best results on the makespan for some well known benchmark open-shop scheduling problems. The general approach is also shown to be readily extendable to rescheduling and dynamic scheduling.

Acknowledgements

Mostly, I would like to thank my first supervisor Dr Peter Ross for his boundless and instructive help, criticism and patience. Without his supervision and expert knowledge in Artificial Intelligence and especially Genetic Algorithms, this thesis would not have been possible. I would also like to thank my second supervisor Dave Corne, who introduced me to Genetic Algorithms and gave me much invaluable help, comments and also careful proof-reading. Thanks also to my former second supervisor Dr Chris Mellish for the supervison of the early parts of my project.

Thanks to Drs Robert Fisher and Alan Smaill for providing the MSc examinations and lectures information. The former also provided much invaluable help throughout the timetabling part of my project and comments on parts of this dissertation. Thanks also to Howard Beck, Tim Duncan and Dr Pauline Berry (who has since left) in the Artificial Intelligence Applications Institute who taught me about the scheduling domain through seminars and discussions, and proof-read the early chapters of this dissertation.

I would also like to thank many of the colleagues and friends who have provided help, advice and companionship during my PhD study in the Department of Artificial Intelligence. In no particular order, thanks to: Chris Gathercole (for proof reading chapters of this dissertation), W W Vasconcelos, Rolando S. Carrera-S., Dr Kee Yin How, Albert Burger, Jessica Chen-Burger, Simon Liang, Chun Chi Wang, Ching Long Yeh and Cheng Hsiang Pan.

Finally, I thank my parents, my wife Hui-Chen and my daughter Shyuan-Yih for their encouragement and company, and also thank my employer, China Steel Corporation, Taiwan, R.O.C., especially the Artificial Intelligence/Expert Systems Projects and those who gave me the support during these last few years.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research unless otherwise indicated.

Hsiao-Lan Fang
Edinburgh
September 30, 1994

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Timetabling/scheduling problems are particularly challenging for Artificial Intelligence and Operations Research techniques. They are problems of time-based planning and combinatorial optimisation which tend to be solved with a cooperation of search and heuristics, which usually lead to satisfactory but sub-optimal solutions. Why are timetabling/scheduling problems so difficult?

1. They are computationally *NP-complete* problems [Garey & Johnson 79]. This means that there is no known deterministic polynomial time algorithm. Also, the space of possible solutions for most real problems is far too large for brute force or undirected search methods to be feasibly applied.

2. Advanced search techniques using various heuristics to prune the search space will not be guaranteed to find an optimal (or near optimal) solution. In other words, it is very difficult to design effective heuristics.

3. Timetabling/scheduling problems are often complicated by the details of a particular timetabling/scheduling task. A general algorithmic approach to a problem may turn out to be inapplicable, because of certain special constraints required in a particular instance of that problem.

4. Real world timetabling/scheduling problems often involve constraints that cannot be precisely represented or even precisely stated.

1.1 Genetic Algorithms (GAs)

Genetic Algorithms (GAs) are a group of methods which solve problems using algorithms inspired by the processes of neo-Darwinian evolutionary theory. In a GA, the performance of a set of candidate solutions to a problem (called 'chromosomes') are evaluated and ordered, then new candidate solutions are produced by selecting candidates as 'parents' and applying mutation or crossover operators which combine bits of two parents to produce one or more children. The new set of candidates is then evaluated, and this cycle continues until an adequate solution is found. In chapter 3, we will introduce GAs in more detail.

1.2 Timetabling Problems

A timetabling problem is a kind of problem in which events (exams, classes etc...) have to be arranged into a number of time slots, subject to various constraints. The need for powerful methods for solving a large timetabling problem is plain by considering the simple fact that with, say, e exams to be fitted into t time slots, there are t^e possible candidate timetables, which vary in optimality according to the constraints of the problem.

Conventional computer-based timetabling methods concern themselves more with simply finding the shortest timetable that satisfies all the constraints, usually using a *graph-colouring* algorithm and less with optimising over a collection of soft constraints. That is to find sets of exams that can be scheduled at the same time corresponds to finding a colouring such that adjacent nodes have different colours: each colour represents a time slot, and each edge a constraint that the two vertices which it connects must occupy different slots (have different colours). Knowledge-based and OR-based approaches to solving such problems are hard to develop, are often slow and can be inflexible because the architecture itself was based on specific assumptions about the nature of the problem. There are commercial software tools available, described in [Duncan 93], which can be used to build timetabling applications. It is more difficult to handle soft constraints such as preferences in these tools because they are usually based on straightforward *constraint satisfaction* techniques. Most often people still re-

Student	*Examination*
*s*1	e1,e2,e4,e6
*s*2	e1,e2,e4,e5
*s*3	e1,e2,e4,e5
*s*4	e1,e2,e5
*s*5	e4
*s*6	e4,e5
*s*7	e2,e4
*s*8	e2,e4,e6
*s*9	e3,e4,e6
*s*10	e3,e4,e6

Table 1.1: Example of student-exam data

day	*AM*		*PM*	
	0930-1100	1130-1300	1400-1530	1600-1730
day1	t1	t2	t3	t4
day2	t5	t6	t7	t8

Table 1.2: Example of empty time slots

sort to hand-crafted solutions starting from an earlier solution and making a sequence of modifications to cater for changed requirements. This typically leads to sub-optimal solutions that bring significant organisational or financial penalties.

This thesis begins by investigating the powerful techniques of Genetic Algorithms on this sort of problem. The GA approach can deal with hard and soft constraints in a uniform way; further, using the approach developed in this thesis, adding a constraint or changing the importance of a constraint simply corresponds to adding/altering a component of the fitness function, rather than extensively revising the algorithm itself. This will be explained in chapter 4.

A simple example of a timetabling problem is as follows:
Assume that there are ten students (s1..s10), six exams (e1..e6) and eight time slots (t1..t8) such that t1..t4 occur on day 1 and t5..t8 on day 2. Each day has four time slots with two in the morning and two in the afternoon. Each student takes different numbers of exams as in Table 1.1, where student 1 takes exams 1, 2, 4, 6, student 2 takes exams 1, 2, 4, 5, and so on. An empty timetable is shown in Table 1.2, with each slot labelled by its identifier.

day	*AM*		*PM*	
	0930-1100	1130-1300	1400-1530	1600-1730
day1	e1	e3		e4
day2	e2		e5	e6

Table 1.3: A possible exam timetable

The constraints in this example problem are that a student cannot take two different exams in the same time slot and taking three different exams on the same day should be strongly avoided. We must also avoid the same student taking two consecutive exams. Mapping this problem to graph colouring problem in Figure 1.1, each edge represents a not-at-the-same-time constraint. That is, e1e2, e1e4, e1e6, e2e4, e2e6 and e4e6 are the edges which arise from the fact that student s1 is taking exams e1, e2, e4 and e6, so no two can be at the same time.

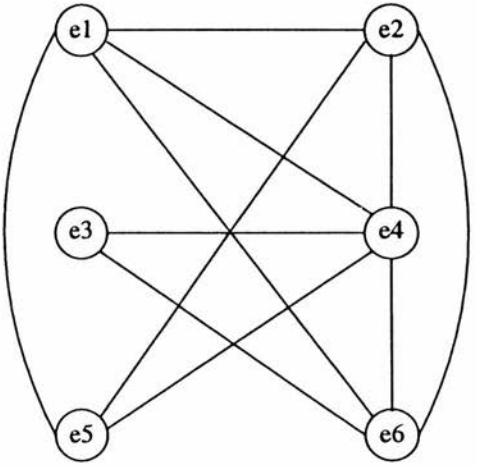

Figure 1.1: Graph colouring for a simple timetabling problem

One possible timetable is shown in Table 1.3, where exams e1, e2, e3, e4, e5 and e6 are put in time slots t1, t5, t2, t4, t7 and t8 respectively.

In this thesis, we will see how a GA solves such problems.

1.3 Scheduling Problems

A scheduling problem is one which involves: "the allocation of resources over time to perform a collection of tasks" [Baker 74]. In the general job-shop scheduling problem, there are j jobs and m machines; each job comprises a set of operations which must each be done on a different machine for different specified processing times, in a given job-dependent order. The job-shop scheduling problem (JSSP) is an extremely hard scheduling problem which frequently needs to be solved in various institutions and organisations. Efficient methods of solving it will have an important effect on profitability and product quality, but with the JSSP being among the worst members of the class of NP-complete problems, as pointed out in the appendix of [Garey & Johnson 79], there remains much room for improvement in current techniques for tackling it. Existing techniques include Artificial Intelligence approaches such as sophisticated *knowledge-based systems* described in [Fox & Smith 84] (combining the use of multiple abstraction levels and progressive constraint relaxation within a frame-based representation scheme), *constraint satisfaction problem, expert systems, neural network* and Operations Research approaches such as *mathematical programming, dynamic programming, branch & bound search, simulated annealing, tabu search*, and *bottleneck-based methods*. In general, the vast size of the search space coupled with the high occurrence of local minima in this kind of problem make it very hard for conventional knowledge-based and/or search-based methods to find near optima in reasonable time. The recent success of genetic algorithms (GAs) on optimisation problems have led some researchers to apply GAs to the JSSP.

The second aim of this thesis was to investigate the powerful techniques of GAs on this sort of problem. The GA approach can deal with several different objectives, for example maximum complete time or makespan (the complete time of a job is the time at which processing of the last operation of that job is completed and the makespan of a schedule is the largest of all the jobs' complete time), flow time (the total time a job spends in the shop), tardiness (the time a job is completed after the due-date) and earliness (the time a job is completed before the due-date) etc, in a uniform way. Further, adding or changing the importance of an objective simply corresponds to adding/altering a component of the fitness function, rather than extensively revising

the algorithm itself. This will be explained in chapter 6.

A simple example of job-shop scheduling problem taken from [French 82] is as follows: Algy, Bertie, Charlie, and Digby share a flat. Four newspapers are delivered to the house which are the Financial Times, the Guardian, the Daily Express, and the Sun. Each of the them reads all of the newspapers, in a particular order and for a specified amount of time. We also know that Algy gets up at 8.30, Bertie and Charlie at 8.15, and Digby at 9.30, what is the earliest time they can all set off for college?

The order in which each student reads the papers, and the amount of time he spends reading each one is in Table 1.4.

	First	Second	Third	Fourth
Algy	FT, 60m	G, 30m	DE, 2m	Sun, 5m
Bert	G, 75m	DE, 3m	FT, 25m	Sun, 10m
Charles	DE, 5m	G, 15m	FT, 10m	Sun, 30m
Digby	Sun, 90m	FT, 1m	G, 1m	DE, 1m

Table 1.4: The newspaper problem

One of the optimal orders is shown in Table 1.5, and its GANTT chart is shown in Figure 1.2, where A, B, C, D denote Algy, Bertie, Charles and Digby respectively. Thus Charles has the Financial Times first, before it passes to Algy, then to Bertie, and finally to Digby. Similarly the Guardian passes between Charles, Bertie, Algy, and Digby in that order. Furthermore, we also can see from Figure 1.2, the optimal schedule is not necessarily to begin at 8:30 which is the earliest getting up time among them. Instead, the earliest start time for this optimal schedule is at 8:45 when Charles gets up. We can also see from the GANTT chart that the time when all of them finish reading is at 11:30. In other words, the earliest complete time when the last of them left is at 11:30 and no other schedules can have the finish time earlier than 11:30.

In this thesis, we will also see how a Genetic Algorithm solves such scheduling problems.

1.4 Applying GAs to Timetabling/Scheduling Problems

Timetabling/scheduling problems are highly computationally complex problems which often have highly domain-dependent features (that is: a particular timetabling/scheduling

	First	Second	Third	Fourth
F.T.	C	A	B	D
G.	C	B	A	D
D.E.	C	B	A	D
Sun	C	D	B	A

Table 1.5: A possible reading schedule

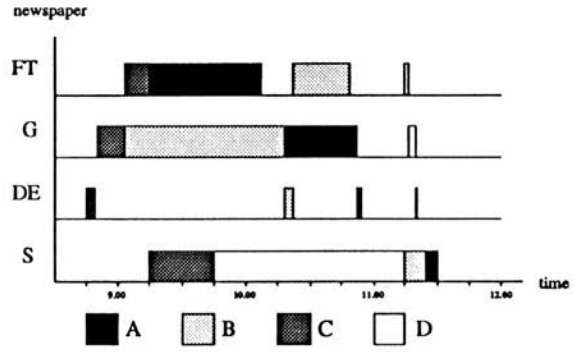

Figure 1.2: GANTT chart for a simple scheduling problem.

problem will be complicated by details of the particular domain). They thus tend to need highly domain specific optimisation algorithms. Genetic Algorithms, however, provide a way of separating the optimisation algorithm from domain knowledge which has led to promising and useful results in the areas of timetabling/scheduling, as well as several other areas. The trick Genetic Algorithms use is to separate a roughly domain independent and very powerful optimisation technique (mutation and crossover operators applied to populations of chromosomes) from the domain specific aspects of a problem (the evaluation function for the chromosomes).

1.5 Summary of Contributions

The main contributions of this thesis can be summarised as follows:

- **Timetabling:**

  - *A framework is presented for GAs in modular timetabling problems.* The approach involves three components : *declaring problem-specific constraints*, *constructing a problem-specific evaluation function*, and *using a problem-independent Genetic Algorithm*. Successful results are demonstrated and comparison with human-produced actual timetables is conducted. Furthermore, the basic approach can handle a wide variety of timetabling constraints (for example, precedence/preference constraints) or considering other allocations (for example, room allocation) simultaneously by introducing more elaborate chromosomes.
  - *The framework is extended to let our system deal with variable non-uniform-length lectures.* That is, without changing the basic representation, we can schedule lectures with different lengths by just considering each lecture's duration when checking the violation of various constraints.
  - *Smart (directed) mutation operators are used to improve the solution quality and speed for timetabling problems.* Among them, for example, *stochastical violation-directed mutation* (SVDM) chooses an event according to the proportion of 'violation score', and randomly alters its assigned time, place, or other details. *Stochastical event-freeing mutation* (SEFM) behaves like SVDM but gives it a new time, place, or agent which will maximally reduce this violation score.

    The framework is shown to be successful on several real problems of 'University Department size', and so it seems justifiable to expect it to work very well on most other problems of similar size and nature. Much work remains to do to see how it scales to larger and different kinds of timetabling problems. There are no clear reasons to expect performance to degrade significantly with *size* of problem, but very *tightly* constrained problems may of course provide problems for this framework.

- **Scheduling:**

  - *A framework is presented for GAs in job-shop/open-shop scheduling problems.* The representation used is a variant of one known to work only moder-

ately well for the travelling salesman problem, namely [Grefenstette *et al.* 85]'s ordinal representation. It has the considerable merit that domain-independent crossover and mutation will always produce legal schedules when applied to job-shop scheduling problems. Furthermore, this representation promises to effectively address the dynamic scheduling problem, the job-shop rescheduling problem and open-shop scheduling/rescheduling problems in an uniform way. Both job-shop and open-shop scheduling problems use a schedule builder to decode the chromosome and put the operations into the earliest legal places it finds.

- *Some methods are introduced to enhance the performance in job-shop scheduling problems.* A *gene-variance based operator targeting* (GVOT) method lets genetic operators concentrate on the most 'needed' parts. An *evolving idle time within gaps* (EIWG) method lets the same sequence represent different schedules in a limited search space. A *delaying circularity using virtual genes* (DCVG) method postpones and reduces the redundancy in the representation by introducing some extra virtual genes.
- *Some methods are introduced to enhance the performance in open-shop scheduling problems.* A *fixed heuristic* (FH) method improves the solution quality of the basic *job and operation* combination chromosomes (JOB+OP) by using a *fixed heuristic* rule to resolve ambiguities faced by the schedule builder. An *evolving heuristic choice* (EHC) method learns the best job-rule combination by specifying in the chromosome itself which heuristic rule the schedule builder should apply.
- The framework for scheduling is successful on a wide variety of benchmark problems of varying size and type. There is hence promise for its use as the basis of an approach to real problem instances, but this is yet to be tested. It seems safe to say that it will perform well on problems similar in size and nature of constraints to the various benchmarks. Different kinds of scheduling problems may need significant changes or additions to the framework, but many real problems will need little change.

1.6 Outline of the Dissertation

The contents of the dissertation have been arranged to be read chapter by chapter. However if the reader is already familiar with timetabling/scheduling and/or GAs, he or she can skip chapter 2 and/or chapter 3 and go to chapter 4 directly. The contents of each chapter are as follows:

- Chapter 1 gives a general introduction of this dissertation.
- Chapter 2 presents a review of relevant and related literature on the timetabling and scheduling problems, especially using GAs approach.
- Chapter 3 introduces GAs and describes the theoretical foundations.
- Chapter 4 describes a basic framework for using GAs to solve timetabling problems.
- Chapter 5 applies the basic framework described in chapter 4 to a modular exam timetabling problem (METP) and a modular lecture/tutorial timetabling problem (MLTP).
- Chapter 6 describes a basic framework for applying GAs to job-shop scheduling problems (JSSP).
- Chapter 7 describes extensions of the framework presented in chapter 6 to dynamic (stochastic) job-shop scheduling, job-shop rescheduling problems (JSRP) and open-shop scheduling/rescheduling problems (OSSP/OSRP).
- Chapter 8 provides conclusions and suggestions for further research.

Chapter 2

Literature Review in Timetabling/Scheduling

2.1 Random and/or Exhaustive Search Approaches

The performance of random search methods on timetabling/scheduling problems will typically be a function of what proportion of the space of solutions are actually *good* solutions; typically, this ratio is often very low, because such problems often have very tight constraints; for example, lots of exams must be scheduled in a short time and every extra constraint decreases this ratio. Therefore, looking randomly for a good timetable/schedule is like looking for a needle in a haystack. Classical heuristic search-based techniques often perform adequately on small timetabling/scheduling problems, but in larger problems the size of the search space is such that we can expect classical heuristic search-based techniques to be very time consuming. Classical heuristic search can be seen as very similar to the method used by human experts. In the same way, it can be expected to find local minima often. We also know exhaustive search is infeasible for NP-Complete problems due to their immense search space. Some of the constraint-satisfaction approaches treat any solution as a good solution; they use no separate measure of quality in addition to the set of hard constraints. However, there exist some exact algorithms for small timetabling/scheduling problems, for instance, [Korman 79] presents some exact graph-colouring algorithms which can be used to timetable small problems. Later in [Cangalovie & Schreuder 91], a graph-colouring algorithm is designed to solve timetabling problems with lectures of different lengths on a relatively small scale due to its exponential time complexity. In scheduling,

[Giffler & Thompson 60] provides an algorithm for the generation of all active schedules, which are processing sequences that no operation can be started earlier, for a job shop problem, and later researchers, for example [Conway *et al.* 67] and [Baker 74], present a modification of this algorithm to generate all nondelay schedules, which are schedules such that no machine is kept idle if some operation is schedulable. Nonetheless, such approaches rapidly becomes computationally infeasible as the problem size grows. There are also many other algorithms in the literature. Due to vast search space traversed by exhaustive search and the limitations of exact algorithms, the implicitly enumerative methods or more efficient heuristic methods are usually employed to solve timetabling/scheduling problems. These are described in the following sections.

2.2 Operations Research Approaches

2.2.1 Enumerative Search

- *Mathematical Programming:* Mathematical programming is a family of techniques for optimising a function constrained by independent variables. However, it is only suitable for small timetabling/scheduling problems. Several such approaches in timetabling/scheduling exist, for example, linear and integer programming [Tripathy 84], or Lagrangean relaxation[1] [Tripathy 80, Arani *et al.* 88]; a good collection of applied mathematical programming examples can be found in the book [Ciriani & Leachman 93].

- *Dynamic Programming:* Dynamic Programming is an implicit enumerative search method which can be seen as a kind of divide and conquer technique. In order to solve a large problem, we can first decompose it into several small independent subproblems. When we do not know which subproblems we should solve first, we can solve all the small subproblems and keep them for later use. For large timetabling/scheduling problems, this approach is ineffective, see for example, [Held & Karp 62].

- *Branch and Bound:* Branch and Bound search is also an implicit enumerative method. This approach consists of two fundamental procedures. Branching is

[1] It can also be used to compute the lower bound [Beasley 93].

the process of partitioning a large problem into two or more subproblems, and bounding is the process of calculating a lower bound on the optimal solution of a given subproblem. The various procedures differ primarily with respect to the branching strategy and the generation of bounds. Examples of this approach are [Balas 69, McMahon & Florian 75, Baker & McMahon 85, Carlier & Pinson 89]. For large timetabling/scheduling problems, this approach is also ineffective.

2.2.2 Heuristic Search

- *General Heuristic Rules:* Since most timetabling and job-shop scheduling problems are NP-complete, as pointed out in the appendix of [Garey & Johnson 79], it is likely to be impossible to use enumerative search to get the solution in polynomial time. Therefore, we must use some "rules of thumb" to decide when a machine is to be released or a job arrives, though this does not guarantee an optimal solution. [Werra 85] surveyed many of the graph theoretical and network models, and provided an introductory tutorial on the formulation of simple timetabling problems. He mentioned that generally real timetabling problems cannot be formulated with these simple models, however, many of the heuristic methods are often derived and adapted from exact methods developed for the simple cases. [Carter 86] reviewed the graph colouring heuristics that have been applied to practical timetabling problems; for example, largest degree first, largest degree first recursive, smallest degree last recursive. [Panwalkar & Iskander 77] classified more than 100 scheduling rules and tried to analyse the general idea behind different rules. [Blackstone Jr *et al.* 82] made a survey of heuristic rules for manufacturing job shop operations and compared several of the heuristics using the results of published studies. [Kiran 84a, Kiran 84b] also did a simulation study in job shop scheduling and compared the performance of heuristic rules based on different kinds of criteria. Some of the most famous heuristic rules are shortest processing time first (SPT), largest processing time first (LPT) and earliest due-date first (EDD).

- *Simulated Annealing:* The recent interest of simulated annealing (SA) began with the work of [Kirkpatrick *et al.* 83]. Annealing is a thermal process for obtaining

low energy states of a solid in a heat bath. The process contains two steps: first, increase the temperature of the heat bath to a maximum value at which the solid melts; second, decrease carefully the temperature of the heat bath until the particles arrange themselves in the ground state of the solid. Basically SA is a local search method, in which one wishes to choose a direction to move. Rather than always choosing the direction of best improvement, which gives steepest-ascent hill-climbing, SA initially chooses random or semi random direction but over time comes to prefer any direction of best improvement. Thus the direction-selection process is controlled by some sort of temporal parameter, which by analogy with real annealing is usually called 'temperature'.

There exist many SA applications in the Operations Research literature. For example, [Eglese & Rand 87] and [Dowsland 90] both consider timetabling problems which have no valid solution and therefore need to violate some constraints. [Abramson 91] applied simulated annealing to a school timetabling problem using both sequential and parallel machines. [Osman & Potts 89] and [Ogbu & Smith 91] both used SA to solve permutation flow shop scheduling problems, and concluded that their methods were better than previous results though they spent more computational time. [Laarhoven *et al.* 92] applied an algorithm to get good results in job shop scheduling problems involving the acceptance of cost-increasing transitions with nonzero probabilities to avoid getting stuck in local minima, but this was also very time-consuming. [Shen *et al.* 94] combined simulated annealing and simulated evolution to solve some flow shop/job-shop scheduling problems. Their algorithm guided the stochastic search of simulated annealing with an evolutionary process. They declared that their results were better than other simulated annealing approaches reported in the literature.

- *Tabu Search:* The modern form of tabu search (TS) is derived from [Glover 86]; [Hansen 86] also developed the method in a steepest ascent/mildest descent formulation. [Glover & Laguna 93] describes TS as follows:

> "Tabu search (TS) has its antecedents in methods designed to cross boundaries of feasibility or local optimality normally treated as barriers, and systematically to impose and release constraints to permit exploration of otherwise forbidden regions" ...
> ..."Tabu search scarcely involves reference to supernatural or moral considerations, but instead is concerned with imposing restrictions to guide a search

> process to negotiate otherwise difficult regions. These restrictions operate in several forms, both by direct exclusion of certain search alternatives classed as 'forbidden', and also by translation into modified evaluations and probabilities of selection."

For instance, [Hertz 91] produced a tabu-search based timetable but only deals with restricted soft constraints. Both [Widmer & Hertz 89] and [Taillard 90] applied TS to flow-shop sequencing problems. [Reeves 93] introduced a candidate list strategy to improve the solution quality and efficiency of flow shop sequencing problems. Recently, [Taillard 93] reported his results on the benchmark scheduling problems given in [Beasley 90], however, running times were very long because his purpose was to find the best results he could, using TS, regardless of time taken.

- *Bottleneck Heuristics:* In scheduling literature, there are many bottleneck heuristic methods, which are more complicated than simple heuristic rules. The most early and famous one is *Optimised Production Technology* (OPT) developed by Eliyahu Goldratt in Israel during the 1970s and commercialised within computer software in the 1980s [Goldratt 88]. The idea behind OPT (see, for example [Goldratt & Cox 84, Goldratt & Fox 86]) is to find an obvious bottleneck in the problem (for example: a particularly highly constrained task), and focus on this bottleneck. A partial solution is found in the region of the bottleneck, and the rest of the problem (now simplified) is then solved. OPT is not so useful, however, if there is no initial candidate for an obvious bottleneck. Later bottleneck heuristic methods are more or less adapted and expanded from OPT. For example, [Adams *et al.* 88] described their *Shifting Bottleneck Procedure* as follows:

  > It sequences the machines one by one, successively, taking each time the machine identified as a bottleneck among the machines not yet sequenced. Every time after a new machine is sequenced, all previously established sequences are locally reoptimized. Both the bottleneck identification and local reoptimisation procedures are based on repeatedly solving certain one-machine scheduling problems.

  [Morton & Pentico 93] described their *Bottleneck Dynamics* as follows:

  > Basically, bottleneck dynamics estimates prices (delay cost) for delaying each possible activity within a shop and, similarly, prices (delay costs) for delaying each resource in the shop. Trading off the costs of delaying the activities versus

the costs of using the resources allows calculating, for example, which activity has the highest benefit/cost ratio for being scheduled next at a resource (sequencing choice), or which resource should be chosen to do an activity to minimize the sum of resource and activity delay costs (routing choice). Thus these prices allow making local rather than global decisions with easy and intuitive cost calculations, allowing solution by advanced dispatch simulation. Decision types that can be handled include sequencing, routing, release and other timing, lot sizing, and batching.

2.3 Artificial Intelligence Approaches

2.3.1 Constraint Satisfaction Problem

A *Constraint Satisfaction Problem* (CSP) technique is defined by

- a set of variables each of which has a discrete and finite set of possible values (their domain).
- a set of constraints between these variables.

A solution to a CSP is a set of variable-value assignments which satisfy all the constraints. [Meisels *et al.* 93], for example, applies CSP algorithms to a school timetabling. [Duncan 93] describes several commercial tools for constraint satisfaction problems. Among them, three well known tools which can be used to build timetabling/scheduling applications are as follows:

- *CHIP*, originally developed at *European Computer-Industry Research Centre* (ECRC) in Munich, is a constraint logic programming language based on Prolog which combines consistency techniques (that is, arc consistency) with backtracking search [Dincbas *et al.* 88].
- *CHARME* is derived from CHIP, which is also an integrated language. Its syntax looks like C language but isn't and its main data structure is merely using arrays. Its main features are non-determinism, demons and many options for variable/value selection [Oplobedu *et al.* 89]. A good tutorial on how to use CHARME can be found in [Duncan & Johnson 91].
- *ILOG SOLVER* (previously known as PECOS) is a library but not a language. It adopts an object-oriented approach and is therefore good for modeling. Cur-

rently, it has two versions: Le-Lisp and C++. Other features are not being limited to chronological backtracking, user-defined criteria for variable/value selection and ability to define primitive constraints etc [Puget 94, Le Pape 93].

2.3.2 Knowledge-based Systems

[Beck *et al.* 91, Berry *et al.* 93] classified constructive techniques, also known as Knowledge-based or AI-based systems, as follows:

- *Order(Lot)-based Scheduling:* decisions are taken order (lot) by order (lot). All lots which require scheduling are collected and ranked in terms of their priority. An order (lot) is selected in terms of its priority, and each of its operations is then scheduled before the next order (lot) is considered. That is, highest priority order is scheduled first, for example, ISIS[2] [Fox & Smith 84].

- *Resource-based Scheduling:* decisions are taken resource by resource and the next resource is selected in terms of expected demand for its available capacity. So, most heavily loaded resources are scheduled first, for example, RESS [Lui 88].

- *Island-based (Opportunistic) Scheduling:* the heavily or over-loaded resource are identified and these resources are scheduled using resource-based scheduling. Then take a lot based approach. That is, two schedulers, one scheduling lots (orders) and the other scheduling resources, work cooperatively to produced a schedule, for example, OPIS [Smith *et al.* 86].

- *Operation-based Scheduling:* decisions involve the selection and allocation of a single operation. Therefore, bottleneck analysis is often used to select the operation, that is, the most critical period and operation to the bottleneck is scheduled first, for example, MICRO-BOSS [Sadeh 91].

[Berry *et al.* 93] also made some comments on these systems as follows (slightly adapted):

> Order-based scheduling emphasises concerns surrounding individual orders. Resource-based scheduling emphasises resource based criteria, which still

[2] This is also a frame-based expert system.

takes precedence in Opportunistic scheduling. Operation-based scheduling is more flexible and similar to the more general CSP problem. However, most of the above systems have the disadvantage that they use inefficient search strategies and are unable to deal with uncertainty.

2.3.3 Distributed Artificial Intelligence

[Huhns 87] defines *Distributed Artificial Intelligence* (DAI) as:

> the application of distributed computing resources towards the co-operative solution of a common problem. Its defining characteristic is the co-operative activity of a group of decentralised and (loosely) coupled knowledge sources (nodes) where no node has sufficient information or knowledge to solve the entire problem.

There are two approaches in DAI: *blackboard based architectures* and *multi agent systems*. For instance, [Collinot *et al.* 88] uses a control system built on a blackboard architecture to coordinate the use of various components of SONIA: a knowledge-based scheduling system, and adjust their behaviour according to the problem-solving context. An example of a multi agent system in scheduling is the *distributed asynchronous scheduler* (DAS), which decomposes the scheduling problem across a hierarchy of communicating agents where each agent exhibits the properties of opportunism, reaction and belief maintenance [Burke & Prosser 91].

2.3.4 Rule-based Expert Systems

In this approach, one uses a rule-based language to try to specify the constraints and the heuristics which human experts use to solve the problem. [Solotorevsky *et al.* 91] presented a rule based language, RAPS, for specifying resource allocation and timetabling problems which is embedded in an expert system shell – ESRA [Solotorevsky *et al.* 93]. [Solotorevsky *et al.* 93] compared the rule based and CSP approaches for solving a real-life courses timetabling problem and listed some advantages of rule-based expert systems:

- Rules are very convenient for specifying both the constraints on the problem and the heuristics used by the expert to solve it.
- A rule-based expert systems (RBS) can generate explanations easily, by following the path of rules that are fired in the reasoning process.

- The expert systems (ES) approach has the ability to specify constraints more naturally, and the ability to pursue multiple objectives rather than a single objective.

However, the syntax of this language is much more complex than the general if-then rule style. Rule-based systems are not usually used for optimisation problems.

2.3.5 Neural Networks

This approach seems promising, but there are not many results to point to at this time. A common approach is to use feed-back neural networks to find solutions to combinatorial optimisation problems, for example [Ross & Hallam 93] discusses how to apply a Hopfield net to solve the Knight's Tour problem. Different researchers have used different neural network models to deal with timetabling/scheduling problems. For instance, [Gislen *et al.* 89] demonstrated a small timetabling problem using Potts neural networks and [Gislen *et al.* 92] extended it to deal with larger timetabling problems. [Hulle 91] used a Goal Programming Network for Mixed Integer Linear Programming to solve job-shop scheduling problems, but the number of constraints increase combinatorially as problem size increases. Several neural network models have successfully been applied to the *Travelling Salesman problem* (TSP) which can be easily adapted to single machine or permutation scheduling, e.g. [Hopfield & Tank 85, Durbin & Willshaw 87, Angéniol *et al.* 88, Aarts & Korst 89, Fritzke & Wilke 91, Yokoi & Kakazu 92].

2.4 Review of GAs in Timetabling/Scheduling

2.4.1 GAs in Timetabling

[Colorni *et al.* 90] constructed a lesson timetable for an Italian high school using a Genetic Algorithm approach and first reported the successful GA timetabling application. Their representation involves a 5-tuple made up of a resource (teacher), timetable-intervals (hours), a job (lesson), a matrix (a timetable) and a function to be maximised. They used a matrix R to represent a timetable. Each row is a teacher, each column an hour and each element of R is a lesson. In their representation, row constraints are always satisfied, however column constraints may be infeasible due to over-crowded or un-covered hours. So they used *filtering* operators to help deal with these problems.

They found that their approach could satisfy all the hard constraints of their high school timetabling problem, and made some headway into minising violations of the soft constraints.

[Davis 91] presented a hybrid GA/greedy algorithm approach for solving simple graph colouring problems, which are closely related to timetabling problems. We can consider the set of events E to be the vertices of a graph and the set of times T to be colours (or labels) that can be used to paint the vertices of this graph. We can represent any constraint of the form "e_i and e_j cannot share the same time-slot" as an edge between vertices c_i and c_j. The simple timetabling problem then becomes one of colouring every vertex in such a way that no two vertices connected by an edge are of the same colour. Examination modules which are unconnected may be scheduled at the same time as there will not be any clashes.

[Abramson & Abela 91] used some algorithms to pre combine a class, teacher and room combination into a tuple and treat the tuple as an inseparable unit later. They then used a chromosome representation where, more like the actual timetable itself, positions are time slots and the values that a 'position gene' can take are sets of tuples that might occur at that time. This kind of representation however leads to a problem called the *label replacement problem* [Glover 87]; the problem is that the crossover operator can easily lead to producing timetables which leave out many of the exams that need to be scheduled. A *label replacement algorithm* must be used to then repair the chromosome to reintroduce all the necessary exams and remove duplicated exams.

[Ling 92] used hybrid Genetics Algorithms to deal with the teaching timetable of a Polytechnic in Singapore. A *Prolog* program was initially used to solve the major part of the timetable, for example some workshop sessions require several consecutive slots which must be scheduled before others. The GA was then used to schedule the rest of the timetable. The representation they used was similar to that of [Colorni *et al.* 90]. It was also a matrix R where each column is also an hour but each row a student class. They also used *filtering* [Colorni *et al.* 90] to repair the timetable. As a result, some of the over-crowded periods can be eliminated.

[Chan 94] used a similar approach to our representation [Fang 92]. He used binary encoding, and divided the evaluation function into a cost function and a penalty func-

tion which represents the penalty of soft constraints and hard constraints respectively. The total fitness function is just a linear combination of a cost function and penalty function. He then used a constraint propagation algorithm, propagating a constraint to restrict the legal value of some variables, to improve the efficiency. However, as he pointed out, "a value propagated for a previous schedule requirement may be varied in subsequent propagation and result in violating the previous schedule requirement". Therefore, he further used an iterative method to repeat the algorithm several times, hoping most constraints can be satisfied.

[Burke *et al.* 94] used a graph colouring algorithm to first produce a feasible timetable which satisfied the hard constraints, that is no student can take two exams at the same time and the rooms must be large enough than the students present at that exam. They then used a similar GA representation to ours [Fang 92] to operate on this feasible timetable. The evaluation function they used is just to control the length of the timetable, the number of second order conflicts and the number of spare seats in a room. In order to preserve the feasibility of the timetable, normal GA operators are mostly not suitable, so they further used sophisticated 'violation-healing' crossover and mutation operators to guarantee feasible timetables.

Recently, [Paechter 94, Paechter *et al.* 94] investigated the use of indirect representation in timetabling problems. In an indirect representation, the chromosome encodes *instructions* for building a timetable, rather than directly represents the timetable itself. [Paechter *et al.* 94] uses a *place-and-seek* method; in this approach, the chromosome says which event to schedule next, and where to place it; if this event cannot be placed without violating a hard constraint, then the chromosome encodes where to go next (in the list of possible times or rooms) to look for a place where the event will fit. This approach seems to search a far fitter subspace, and should achieve a near-optimum or optimum in fewer evaluations. However, since the method does not search the whole space of timetables, we cannot guarantee that it will even be possible to find a global optimum.

Our approach [Fang 92, Corne *et al.* 93, Ross *et al.* 94b] uses chromosomes in which, positions are events and the values that a 'position gene' can take are sets of start times, rooms or teachers that might occur at that event. This kind of representation

may represent timetables where not all time slots are filled: this is, however, a legal, and possibly useful timetable. Using this representation, the Genetic Algorithm does not need to be repaired with some computationally expensive algorithm like other approaches described above. We used a penalty function to evaluate the quality of the timetable and treat all the hard and soft constraints in a uniform way. The relative penalty values may be chosen to reflect intuitive judgement of the relative importance of satisfying different kinds of constraint. Further, we introduce some 'constraint violation directed' mutation operators to improve solution quality and the speed. The idea is simply to keep track, during evaluation, of the individual contributions to violations of each event; this information is then used to roughly guide mutation to where it seems most needed. More detailed information such as allocating time slots and rooms simultaneously and some stochastic violation-directed and event-freeing mutation operators have been looked into in the timetabling parts of this thesis.

2.4.2 GAs in Scheduling

[Davis 85] was the first to suggest and demonstrate the feasibility of using a GA on a simple JSSP, but his essentially *ad-hoc* set of genetic operators and memory-intensive chromosome representation left much room for improvement. [Hilliard *et al.* 87] used a classifier system to discover heuristic scheduling rules for simple job-shop scheduling problems and indicated a potential for success but not many researches in this direction are in progress.

Meanwhile, the general success of GAs on other kinds of hard scheduling problems such as the TSP [Grefenstette *et al.* 85], which is equivalent to a simplified form of the JSSP[3], started to lead to clues for more effective representations and operators for GA approaches. For example, [Whitley *et al.* 89, Whitley *et al.* 91] defined a new edge recombination operator for the TSP, but performance degraded when applied to more typical scheduling problems. [Starkweather *et al.* 91] compare six sequencing operators for TSP and scheduling problems and conclude that the results of schedule optimisation were almost the opposite of the results from the TSP. These differences

[3] [Syswerda 91, Syswerda & Palmucci 91] showed how a JSSP may be represented in a way similar to the TSP with operators to guarantee legal solutions.

can be explained by examining how these operators preserve adjacency (for the TSP) and order information (for the scheduling). [Fox & McMahon 91] also found similar results. Other Genetic approaches for the TSP are [Michalewicz 92, Tamaki *et al.* 94, Bui & Moon 94], for example.

[Cleveland & Smith 89] reported a flow-shop release algorithm which can be seen as a complex single-machine problem, where the 'machine' is actually an automated multi-stage flow line with non-standard characteristics. [Cartwright & Mott 91] reported a method to decide the suitable population sizes and crossover rate for the flow-shop scheduling problems. [Reeves 92b] reported a Genetic Algorithms for permutation flow-shop sequencing and found it outperformed a simulated annealing approach when the problem size was non-trivial. Later, [Reeves 92a] applied it to a stochastic permutation flow-shop sequencing problem and found a GA is more robust than other methods in the presence of noise. [Muller *et al.* 93] presented a single machine scheduling problem in which each operation/job has its own possible start time. Recently, [Murata & Ishibuchi 94] compared GAs with other search algorithms for permutation flow-shop scheduling problems. They found that their GAs performed a little worse than other search algorithms such as local search, tabu search and simulated annealing, however, combining GAs with local search and simulated annealing resulted in high performance.

[Nakano 91] used a conventional GA with binary chromosomes for the JSSP, but his representation did not guarantee to produce legal schedules from crossover, necessitating the use of an algorithm to repair illegal ones. His repair process first fixes local inconsistencies in the schedule for each machine, and then fixes global inconsistencies, usually producing a fairly similar but legal schedule although at significant computational cost. [Tamaki & Nishikawa 92] provided a neighbourhood model of GA where the reproduction is executed locally in a neighbourhood of each individual to avoid premature convergence and applied it to the job-shop scheduling problem. [Husbands *et al.* 90, Husbands & Mill 91] showed a model based on simulated co-evolution applied to manufacturing scheduling problems and could get very good results. Later, [Husbands 93] further presented a highly parallel GA-based 'ecosystems' model which allows the simultaneous optimisation of the process plans of a number of

components.

Recently, problem-specific knowledge has been used to augment GAs in production scheduling. [Bagchi *et al.* 91] found (using rather simplified JSSPs) that the inclusion of problem-specific information in the representation and genetic operators generally produces better results, and the results improve more as more information is included in the GA (rather than used by a heuristic search or other method hybridised with the GA). As the GA becomes more sophisticated however, the speed of finding reasonable solutions falls. Later, for example, [Uckun *et al.* 93] found that the approach used in [Bagchi *et al.* 91] (sophisticated problem-specific information in the representation and operators) produces better results in the longer term than a simpler GA enhanced with a local hill-climbing operator, but the latter method can produce *acceptable* performance much more quickly than the former. [Bruns 93] used a direct representation to represent a production schedule itself and also used domain-dependent crossover and mutation. He used a genetic algorithm simply to do the search because his represented information covered all the search space, so neither a transformation nor a scheduler is necessary.

Some hybrid methods begin to combine GAs with other existing complicated algorithms. For example, [Yamada & Nakano 92] hybridised [Giffler & Thompson 60]'s active scheduling algorithm with a GA which represented an individual directly from its operation completion times, improving on the solution quality of [Nakano 91]. Later, [Davidor *et al.* 93] further put their approach in Davidor's 'ECOlogical' framework, and found that average solution quality improved further without additional computational cost. [Croce *et al.* 92] used a multi-permutation sequence, one for each machine, and a schedule generation algorithm to specify the job priority when a machine becomes free, using the shortest-makespan criterion. [Paredis 92] explored hybrid *constrained optimisation problems* (COPs) with Genetic Algorithms in a job-shop scheduling problem and showed how COPs can be used to augment a GA with domain specific knowledge. [Dorndorf & Pesch 93] used the shifting bottleneck procedure of [Adams *et al.* 88] with a Genetic Algorithm to solve job-shop scheduling and reported better performance than pure shifting bottleneck procedure or a simulated annealing approach with the same running time. More recently, [Atlan *et al.* 93] proposed a

system based both on an original distributed scheduling model and on genetic programming for an inference of symbolic policy functions. They showed that genetic programming can be used to learn strategies for job-shop scheduling.

Our GA approach [Fang *et al.* 93, Fang *et al.* 94] uses a variant of the ordinal representation introduced in [Grefenstette *et al.* 85] and used for the TSP. This representation has the merit of producing only legal schedules under crossover and mutation. When applied to the JSSP, it produces better results than those of [Nakano 91] with pleasingly small computational effort and comparable with hybrid methods such as [Yamada & Nakano 92, Dorndorf & Pesch 93] and others. It thus provides a convenient way to handle the rescheduling, dynamic scheduling and open-shop scheduling/rescheduling problems. The rescheduling problem involves modifying a schedule in the process of execution in order to take account of changed, cancelled or new jobs. As such phenomena occur frequently in the kind of organisation that has to deal with JSSPs, it is as important to find efficient rescheduling algorithms (which ideally do not involve rebuilding the schedule from scratch) as it is to find effective algorithms for the full JSSP. The dynamic scheduling problem is a problem in which either job ready time or job operation processing time information are not all available in the beginning. However, we cannot wait until all the data is available, so we must use the incomplete information to begin to schedule and accumulate a partial schedule gradually. The open-shop problems requires each job to be processed on each machine, but there is no particular order to follow. Therefore, the schedule has to not only decide which machines process which jobs, but also the flow of the jobs between machines. The details are given in the scheduling parts of this thesis.

Chapter 3

Introduction to Genetic Algorithms

Genetic Algorithms are loosely based on the ideas underlying the theory of evolution by natural selection, and current knowledge of the basics of genetics. First we will look briefly at the basic ideas from biology which are used in GAs.

In the beginning of [Smith 89], Smith describes "Darwin's theory" as follows:

> "In *The Origin of Species*, Darwin argued that all existing organisms are the modified descendants of one or a few simple ancestors that arose on Earth" ... " He also argued that the main force driving these evolutionary changes was natural selection" ...
>
> "Darwin, then, argued that organisms do in fact multiply and vary, and that they have heredity, and that, in consequence, populations of organisms will evolve. Those organisms with characteristics most favourable for survival and reproduction will not only have more offspring, but will pass their characteristics on to those offspring. The result will be a change in the characteristics present in the population" ...
>
> "The theory of natural selection not only predicts evolutionary change: it also says something about the kind of change. In particular, it predicts that organisms will acquire characteristics that make them better able to survive and reproduce in the environment in which they live" ...

The following characteristics[1] are the essence of [Darwin 59]'s *The Origin of Species*.

1. Each individual tends to pass on its traits to its offspring
2. Nevertheless, nature produces individuals with different traits
3. The fittest individuals – those with the most favourable traits – tend to have more offspring than do those with unfavourable traits. This drives the population as a whole towards favourable traits.

[1] taken from [Winston 92]

4. Over a long period, variation can accumulate, producing entirely new species whose traits make them especially suited to particular ecological niches.

Parents which adapt better are more likely to be selected for producing children. The children of these better parents inherit genetic traits; we can identify 'genetic traits' with 'genes'; that is, the unit of inheritance is the gene. Genes in the new generation which adapt well are also selected more often for mating and reproducing. This evolutionary cycle continues from generation to generation. Therefore, 'poor' chromosomes, and 'poor' genes will tend to disappear from the environment, leaving 'better' chromosomes containing 'better genes' (and/or 'better' combinations of genes), which tend to produce 'better' children. Finally, as a result of this "survival of the fittest" activity, the population will gradually adapt optimally or near optimally to the environment.

Genes indirectly contribute to an organism's fitness by providing traits that confer some advantage in breeding. However, in biological terms fitness is usually measured in terms of breeding success, rather than in terms of some population-independent measure. That is, the fitness of a chromosome is dependent on every aspect of the external environment, which includes the particular details of the other chromosomes in the population. The terms 'poor' and 'better' are thus relative to the current population. As we shall see, this is rarely true in GAs.

3.1 What are Genetic Algorithms?

What are Genetic Algorithms? A brief answer, following closely that in [Ross *et al.* 94c], is as follows:

> A GA can be seen as an unusual kind of search strategy. In a GA, there is a set of candidate solutions to a problem; typically this set is initially filled with random possible solutions, not necessarily all distinct. Each candidate is typically (though not in all GAs) an ordered fixed-length array of values (called 'alleles') for attributes ('genes'). Each gene is regarded as atomic in what follows; the set of alleles for that gene is the set of values that the gene can possibly take. Thus, in building a GA for a specific problem the first task is to decide how to represent possible solutions. Assuming we

have thus decided on such a representation, a GA usually proceeds in the following way:

- *Initialisation*: A set of candidate solutions is randomly generated. For example, if the problem is to maximise a function of x, y and z then the initial step may be to generate a collection of random triples (x_i,y_i,z_i) if that is the chosen representation.

- Now *iterate* through the following steps, until some *termination* criterion is met (such as no improvement in the best solution so far after some specified time, or until a solution has been found whose fitness is better than a given 'adequate' value). The process alters the set repeatedly; each set is commonly called a generation.

  1. *Evaluation.* Using some predefined problem-specific measure of fitness, we evaluate every member of the current set as to how good a solution to the problem it is. The measure is called the candidate's fitness, and the idea is that fitter candidates are in some way closer to being one of the solutions being sought. However, GAs do not require that fitness is a perfect measure of quality; they can to some modest extent tolerate a fitness measure in which the fitter of some pairs of candidates is also the poorer as a solution.
  2. *Selection.* Select pairs of candidate solutions from the current generation to be used for breeding. This may be done entirely randomly, or stochastically based on fitness, or in other ways (but usually based on fitness, such that fitter individuals have more chance of being chosen).
  3. *Breeding.* Produce new individuals by using genetic operators on the individuals chosen in the selection step. There are two main kinds of operators:
     - *Recombination*: A new individual is produced by recombining features of a pair of 'parent' solutions.
     - *Mutation*: A new individual is produced by slightly altering an existing one.

     The idea of recombination is that useful components of the members of a breeding pair may combine successfully to produce an individual better

than both parents; if the offspring is poor it will just have lower chance of selection later on. In any event, features of the parents appear in different combinations in the offspring. Mutation, on the other hand, serves to allow local hill-climbing, as well introduce variation which cannot be introduced by recombination.

4. *Population update.* The set is altered, typically by choosing to remove some or all of the individuals in the existing generation (usually beginning with the least fit) and replacing these with individuals produced in the breeding step. The new population thus produced becomes the current generation.

How are Genetic Algorithms different from other optimisation and search methods? [Goldberg 89] provided an instructive answer to this question, noting the following four main differences:

- GAs work with a coding of the parameter set, not the parameters themselves.
- GAs search from a population of points, not a single point.
- GAs use payoff (objective function) information, not derivatives or other auxiliary knowledge.
- GAs use probabilistic transition rules, not deterministic rules.

In addition to the above points, an important characteristic of GAs is what [Holland 75] called *intrinsic parallelism* or *implicit parallelism*. The idea of intrinsic parallelism is that the GA effectively addresses very many related optimisation problems in parallel. Each of these individual problems is that of finding the best value (or set of values) for a particular gene (or set of genes).

3.2 Current Research of Genetic Algorithms

The initiator of research in GAs is John Holland. Holland published the book **"Adaptation in Natural and Artificial Systems"** [Holland 75], and from then on many dissertations and papers began to be published by different researchers. From 1985 the general approach began to receive wide attention. Two formal conferences on GAs: **"Proceedings of the International Conference on Genetic Algorithms and their Applications"** [Grefenstette 85, Grefenstette 87, Schaffer 89, Belew & Booker 91,

Forrest 93] are held in odd years from 1985 in United States and **"Parallel Problem Solving from Nature"** [Goos & Hartmanis 90, Manner & Manderick 92, Davidor 94] are held in even years from 1990 in Europe. In addition, the theory-oriented workshop **"Foundations of Genetic Algorithms and Classifier Systems"** [Gregory 91, Whitley 93a] are held every two years from 1990; the term classifier system denotes a simple kind of GA based machine learning system [Goldberg 89]. A well-organised *GA-List digest* newslist and another *EP-List digest* are available via e-mail, while major journals publishing GA-related papers are *Evolutionary Computation, Machine Learning, Neural Networks, Artificial Life, Adaptive Behaviour,* and other Artificial Intelligence based journals.

3.3 How Do Genetic Algorithms Work?

Genetic Algorithms solve a problem by, roughly, generating, changing and evaluating candidate solutions to that problem. A candidate solution to a problem is called a *chromosome*, and a chromosome is usually a bit string or some other encoding or representation of a solution. For example, an eight bit binary string can be a chromosome representing numbers (from 1 to 256), or two numbers in the range [73,1319], or eight distinct binary decisions, and so on.

Initially, a random population of such chromosomes is generated. Changing chromosomes is done by mutation and/or crossover operators (described below), while chromosomes (candidate solutions) are evaluated by way of a domain dependent fitness function, which first decodes the chromosome and then evaluates its optimality as a solution to the particular problem being addressed.

An outline of the flow of control in a basic Genetic algorithm is as follows:

1. Initialise and encode a random population of chromosomes. This is called the 'current population'.

2. Evaluate each chromosome's fitness in the current population.

3. Produce an intermediate generation, by stochastically selecting current popula-

tion chromosomes according to fitness. These will be parents of the next generation.

4. Apply crossover and mutation operators to pairs of and/or single chromosomes in the intermediate generation, thus producing a new generation of chromosomes. This is now the current population.

5. Repeat 2-4 until an adequate solution is found.

A Genetic Algorithm is a kind of weak search method which needs little domain knowledge and is therefore applicable to a very large number of different kinds of problem. In heuristic (or strong) search methods which need to incorporate domain-dependent knowledge, for example, what we must do for a particular problem is find a good heuristic. With GAs, the algorithm already provides a good heuristic (which may or may not be good enough) but we need to define the representation and evaluation function. The details of how Genetic Algorithms work are explained below.

3.3.1 Initialisation

There are many ways to initialise and encode the initial generation: binary or non-binary, fixed or variable length strings, and so on. Holland's original encoding method is as a binary-fixed length string, although this is not vital and often not desirable or natural. At the initial stage, the system just randomly generates valid chromosomes and evaluates each one.

3.3.2 Reproduction

3.3.2.1 Generational Reproduction

In *generational* reproduction, the whole of a population is potentially replaced at each generation [Holland 75]. The most often used procedure is to loop $N/2$ times, where N is the population size, selecting two chromosomes each time according to the current selection procedure, producing two children from those two parents, finally producing N new chromosomes (although they may not all be entirely new).

3.3.2.2 Steady State Reproduction

The *steady state* method selects two chromosomes according to the current selection procedure, performs crossover on them to obtain one or two children, perhaps applies mutation as well, and installs the result back into that population; the least fit of the population is destroyed [Whitley 89].

3.3.3 Selection

The effect of selection is to return a probabilistically selected parent. Although this selection procedure is stochastic, it does not imply Genetic Algorithms employ a directionless search. The chance of each parent being selected is in some way related to its fitness.

3.3.3.1 Fitness-based Selection

The standard, original method for parent selection is *Roulette Wheel* selection or fitness-based selection. In this kind of parent selection, each chromosome has a chance of selection that is directly proportional to its fitness. The effect of this depends strongly on the range of fitness values in the current population. For example, if fitnesses range from 5 to 10, then the fittest chromosome is twice as likely to be selected as a parent than the least fit. If, however, we add 1000 to each fitness so that the range of fitnesses is now from 1005 to 1010, then the relative likelihood of being selected for a parent is about the same for all chromosomes. In many applications this is undesirable, and leads to stagnation in the search. One way to solve this problem is to scale the fitnesses before selection; for example, see [Goldberg 89] pages 122-124 for several useful scaling techniques.

3.3.3.2 Rank-based Selection

[Baker 85] was the first to report the rank based selection method, in which selection probabilities are based on a chromosome's relative rank or position in the population, rather than absolute fitness. There are many possible forms of rank based selection,

depending on how the raw ranks are converted into objective fitnesses. [Whitley 89] used a now popular method involving a *bias* scheme to control the selection pressure. That is, the larger the bias, the stronger the selection pressure. In Whitley's scheme, if the bias is 1.0, then the fittest of N chromosomes is N times more likely to be selected than the least fit. A higher bias gives somewhat more chance to the fittest and somewhat less to the unfit.

3.3.3.3 Tournament-based Selection

The original tournament selection as described in [Brindle 81] chooses K parents at random and returns the fittest one of these. Some other forms of tournament selection exists, for example, *Boltzmann tournament selection* [Goldberg 90] evolves a Boltzmann distribution across a population and time using pairwise probabilistic acceptance and anti-acceptance mechanisms. The procedures are as follows: generate a candidate solution uniformly at random in the neighbourhood of the current solution and accept the new solution with logistic probability. Repeat this for some number of trials. *Marriage tournament selection* [Ross & Ballinger 93] chooses one parent at random, has up to K tries to find one fitter, and stops at the first of these tries which finds a fitter one. If none is better than the initial choice, then return that initial choice.

3.3.3.4 Spatially-oriented Selection

Spatially-oriented selection is a local selection method rather than a global one. That is, the selection competition is between several small neighbouring chromosomes instead of the whole population. This method is based on Wright's *shifting-balance model of evolution* [Wright 68, Wright 69, Wright 77, Wright 78], which is adapted by several authors, for example [Collins & Jefferson 91, Davidor 91, Ross & Ballinger 93]. [Whitley 93b] termed it *Cellular Genetic Algorithms*. The one we used in this thesis is adapted from [Ross & Ballinger 93], which is a two dimensional grid and maintains a roughly square population. This grid is consider *toroidal*, that is leaving one edge will wrap around to the opposite edge. Two random walks of length N are taken from a

selected starting location. The fittest found on each walk are taken as the two parents which produce one child. The child is installed at the start location if it is fitter than the one already there.

3.3.4 Other Operators

3.3.4.1 Crossover

The crossover operator is the most important operator in Genetic Algorithms. Crossover is a process yielding recombination of bit strings via an exchange of segments between pairs of chromosomes. There are many kinds of crossover, here we just illustrate the concepts of four kinds of crossover, and examples will be given in next subsection of this chapter.

- *One-point Crossover:* The procedure of one-point crossover is to randomly generate a number (less than or equal to the chromosome length) as the crossover position. Then, keep the bits before the number unchanged and swap the bits after the crossover position between the two parents.

- *Two-point Crossover:* The procedure of two-point crossover is similar to that of one-point crossover except that we must select two positions and only the bits between the two positions are swapped. This crossover method can preserve the first and last parts of a chromosome and just swap the middle part, which cannot generally be done with one-point crossover. .

- *N-point Crossover:* The procedure of n-point crossover is similar to those of one-point and two-point crossovers except that we must select n positions and only the bits between odd and even crossover positions are swapped. The bits between even and odd crossovers are unchanged.

- *Uniform Crossover:* The procedure of uniform crossover is first described in [Syswerda 89]. Each gene of the first parent has a 0.5 probability of swapping with the corresponding gene of the second parent.

3.3.4.2 Inversion

In [Holland 75], three operators are described for causing children to be different from their parents: *crossover*, *mutation*, and *inversion*. Inversion operates as a kind of reordering technique. It operates on a single chromosome and inverts the order of the elements between two randomly chosen points on the chromosome. While this operator was inspired by a biological process, it requires additional overhead. Though there are some researchers who have investigated the operator, it generally has not reported good results in Genetic Algorithms in practice.

3.3.4.3 Mutation

Mutation has the effect of ensuring that all possible chromosomes are reachable. For example, if the first position in a chromosome can be any number from one to twenty, it may happen that in the initial population there is no chromosome with "6" in the first position. With only crossover operators, it is impossible to generate such a chromosome. Even with inversion operators, the search is constrained to alleles (possible variants of genes) which exist in the initial population. The mutation operator can overcome this by simply randomly selecting any bit position in a string and changing it. This is useful since crossover and inversion may not be able to produce new alleles if they do not appear in the initial generation.

There are slight differences between different authors' definitions of this operator. For instance, in [Goldberg 89], once we decide to apply the mutation operator, the bit will unconditionally change from 0 to 1 or vice versa. However, in [Davis 91], the changed value will depend on a random bit generator, so it is possible that there will be no effective change and it could be form 0 to 0 or 1 to 1. That is to say, the actual bit changed possibility of the former mutation is two times that of the latter.

3.3.4.4 Migration

Parallel or *Island* [Whitley 93b] GAs allow us to run several populations at the same time in each processor respectively without increasing the total processing time; and

day	*AM*		*PM*	
	time1	time2	time3	time4
day1			e1	e3
day2		e5	e6	e2,e4

Table 3.1: Assigning exams to eight time slots

occasionally pass a chromosome from one populations to others, this is known as *migration* [Tanese 89, Gorges-Schleuter 90, Starkweather *et al.* 90, Ross & Ballinger 93]. At each migration, a chromosome is chosen from one population (the first population is chosen for the first migration, the second for the second and so on) according to the current selection procedure and copied into all the other populations. The populations remain fixed in size, so this insertion causes the least fit in a population to be destroyed.

3.3.5 A Simple Example to Illustrate Genetic Operators

- *Initialisation*

  In chapter 1, we introduced a simple exam timetabling problem. Now, we can use a non binary bit string representation to represent the chromosomes here, because it is easy to understand and represent. Our example has six positions representing six exams with each position's value as the time slot assigned to the exam. We can generate the population randomly to assign each exam a time slot.

  If we randomly generate six numbers 3, 8, 4, 8, 6, 7 as six time slots for e1–e6, which means e1 in slot 3, e2 in slot 8, etc, then the chromosome is *3 8 4 8 6 7* and the timetable is as in Table 3.1.

- *Reproduction and Selection*

  In Table 3.2, assuming the fitness columns of each chromosome reflects the quality of that corresponding timetable, which could be, for example, an inverse function of various violations caused. If we apply fitness-based selection and randomly generate a number 0.5 and multiply by the sum of fitnesses 0.133, then get an accumulated fitness 0.0665 which is what we expect to select among our parents. We consult Table 3.2 to get the accumulated fitness 0.0665 is between the first accumulated fitness 0.005 and the second accumulated fitness 0.067, so we select

index	*chromosome*	*fitness*	*accumulated fitness*
1	3 8 4 8 6 7	0.005	0.005
2	*7 3 7 6 1 3*	0.062	0.067*
3	5 3 5 5 5 8	0.006	0.073
4	7 6 7 7 2 2	0.020	0.093
5	*1 7 4 5 2 2*	0.040	0.133*

Table 3.2: Initial population and fitness

the second chromosome *7 3 7 6 1 3* as our first parent.

Similarly using the same method to get another accumulated fitness 0.095, we select the fifth chromosome *1 7 4 5 2 2* as our second parent.

- *Crossover*
  - One-point crossover: With the two parents selected above, we randomly generate a number 2 as the crossover position:

```
parent1:  7 3 7 6 1 3
parent2:  1 7 4 5 2 2
```

    then we get two children:

```
child1 :  7 3|4 5 2 2
child2 :  1 7|7 6 1 3
```

    where the first child has the first part from parent1 and the second part from parent2 and the second child has the first part from parent2 and the second part from parent1.
  - Two-point crossover: With the two parents above, we randomly generate two numbers 2 and 4 as crossover positions:

```
parent1:  7 3 7 6 1 3
parent2:  1 7 4 5 2 2
```

    after crossover, we get two children:

```
child1 :  7 3|4 5|1 3
child2 :  1 7|7 6|2 2
```

    N point crossover: If we randomly generate a number n equal to 3, then randomly generate 3 numbers 1, 2, 4 as crossover positions and swap even-part positions but let the odd-part positions unchanged to produce two children:

```
parent1:  7 3 7 6 1 3
parent2:  1 7 4 5 2 2
            ^ ^   ^
```

after crossover, we get two children:

```
child1 :  7|7|7 6|2 2
child2 :  1|3|4 5|1 3
```

- Uniform crossover: For each position we randomly generate a number between 0 and 1, for example 0.2, 0.7, 0.9, 0.4, 0.6, 0.1. If the number generated for a given position is less than 0.5, then child1 gets the gene from parent1, and child2 gets the gene from parent2. Otherwise, vice versa.

```
parent1:  7 *3 *7 6 *1 3
parent2:  1 *7 *4 5 *2 2
```

after crossover, we get two children and whose crossover positions are marked by a '*' after each gene respectively:

```
child1 :  7 7* 4* 6 2* 3
child2 :  1 3* 7* 5 1* 2
```

- *Inversion*

Here we just give a simple example of pure inversion:

If we randomly choose two positions 2, 5 and apply the inversion operator:

```
3 8 4 8 6 7
  ^     ^
```

then we get the new string:

```
3 6 8 4 8 7
  * * * *
```

The inversion operator only has one parent.

- *Mutation*

Assume we have already used crossover to get a new string:

```
7 3 4 5 1 3
^
```

Assume the mutation rate is 0.001 (usually a small value). Next, for the first bit 7, we randomly generate a number between 0 and 1. If the number is less than the mutation rate (0.001), then the first bit 7 needs to mutate. We generate another number between 1 and the maximum time slot (8), and get a number (for example 2). Now the first bit mutates to 2. We repeat the same procedure for the other bits. In our example, if only the first bit mutates, and the rest of the bits don't mutate, then we will get a new chromosome as below:

2 3 4 5 1 3
*

Because the mutation rate is very small, each chromosome only has a $1-(1-0.001)^6=0.005985$ possibility of mutation. It takes on average just over 167 chromosomes to have one bit mutation.

3.4 The Fundamental Theorem of Genetic Algorithms

In this section we describe the fundamental theorem of Genetic Algorithms, called the schema theorem (ST). It is not a part of this thesis to investigate this theorem as such, or do any kind of formal analysis of Genetic Algorithms. However, this thesis (like many others) demonstrates the extraordinary and surprising power of Genetic Algorithms, where simple, genetically-inspired search operators seem to quickly solve many difficult problems. The schema theorem, and associated mathematical results about Genetic Algorithms, particularly the mathematical analysis of building blocks and implicit parallelism, go a long way along the route of describing just why Genetic Algorithms are so powerful. However, after describing the ST, we will then briefly discuss some problems which make it inapplicable in general; the ST will then no longer be mentioned or used in this thesis. The work in this thesis is therefore empirically rather than theoretically based. The mathematical descriptions of Genetic Algorithms that we use in this chapter is adapted from [Holland 75] and [Goldberg 89], which is a traditional, binary-string one-point crossover and generational GA.

3.4.1 Implicit Parallelism

A *schema* [Holland 75] is a similarity template describing a subset of strings with similarities at certain string positions. Why are schemata important for Genetic Algorithms? [Goldberg 89] says:

> In some sense we are no longer interested in strings as strings alone. Since important similarities among highly fit strings can help guide a search, we question how one string can be similar to its fellow strings. Specifically we ask, in what ways is a string a representative of other string classes with similarities at certain string positions? The frame of schema provides the tool to answer these question.

A schema is a string of the following format:

 # 1 0 1 # 0

with each position a 1, 0 or #, in which # is a don't care symbol, can be 1 or 0. The more # it has, the less specific it becomes. In a binary string, if there are n '#', then a schema can describe 2^n *strings*. Equally, any string of 0s and 1s of length n is an instance of 2^n schemata. In our example there are 2 '#', therefore it can describe four strings:

0 1 0 1 *0* 0
0 1 0 1 *1* 0
1 1 0 1 *0* 0
1 1 0 1 *1* 0

The total number of different *schemata* of length n is 3^n, because each of the n positions can be 0, 1 or #. The total search space remains 2^n since the '#' symbols are not involved in the real string processing; they only serve as a tool to describe similarity templates. How many schemata are processed usefully in each generation? Despite the processing of only, say, c chromosomes each generation, Genetic Algorithms process at least c^3 schema [Holland 75]. Holland calls this *implicit parallelism*[2]

[2] The original term Holland used in 1975 was *intrinsic parallelism*.

and the observation is a major part of the explanation of the robustness of Genetic Algorithms. Even though we perform computation proportional to the size of the population, we get useful processing of at least c^3 schema in parallel with no special book keeping or memory other than the population itself. (for the detailed proof of c^3 see [Fitzpatrick & Grefenstette 88] pages 118-120 or [Goldberg 89] pages 40-41)

3.4.2 The Schema Theorem

Let us consider a simple generational GA. The frequency m(H,t) of a schema H at generation t will change at generation $t+1$ proportionally to the selection probability of strings for reproduction. There are two steps to consider. The first step is the selection step, in which we choose from the generation at time t the chromosomes that will be the parents for the generation at time $t+1$. Let $m^P(H,t)$ represent the frequency of schema H among the *parents* selected to produce generation $t+1$. In the variation step, which we will consider later, we apply crossover and mutation to these parents with each other to produce the new generation $t+1$.

Let f(H) be the average fitness of a string containing schema H in generation t and $\bar{f}$ represent the average fitness of the entire population. So, f(H) depends on t, so does $\bar{f}$. We can derive the following formula:

$$m^P(H,t) = m(H,t) * f(H)/\bar{f} \tag{3.1}$$

The following assumes we are using one-point crossover and P_c is the probability of a crossover. The defining length $\delta(H)$ is the distance between the first and last specific schema positions of H, and l is the total length of a string then the probability of a schema being disrupted because of crossover is as in equation 3.2

$$P_c * \delta(H)/(l-1) \tag{3.2}$$

The probability that an instance of schema H is crossed with a schema *different* from H is in equation 3.3:

$$1 - m^P(H,t)/c \tag{3.3}$$

where c is the number of chromosomes in the population. If we multiply the probabilities from equation 3.2 and equation 3.3, we get the probability that schema H will be

destroyed by the crossover operation. This is as in equation 3.4

$$(1 - m^P(H,t)/c) * P_c * \delta(H)/(l-1) \tag{3.4}$$

This is, however, usually a small number. Let's call this number ϵ.

The mutation rate is usually very small in normal situations. If the probability of a mutation is P_m and the order of a schema H, denoted by $o(H)$, is simply the number of fixed positions (number of 0's and 1's in binary string) present in the template, then the probability of a schema being disrupted because of mutation is as in equation 3.5

$$o(H) * P_m \tag{3.5}$$

Therefore we know that short order schemata are more likely to survive than long order schemata. Now, we can derive the following schema formula:

$$\mathbf{m(H,t+1) \geq (1-\epsilon) * (1 - o(H) * P_m) * m(H,t) * f(H)/\bar{f}} \tag{3.6}$$

The greater than-or-equal sign is there because we actually overestimate the amount of schemas H that will be destroyed. This is because it is still possible to produce instances of schema H by crossover of schema H with a chromosome which has no schema H even if the crossover position cuts the schema. Also, it is possible to produce instances of schema H even if neither of the parent chromosomes contains schema H. As we can see, since $(1 - \epsilon)$ is near to 1, this variation step does not harm the selection step too much. This means we can still be sure of having good schemas increasing in future generations, and bad ones die out, at the same time as producing new good schemas by crossover.

So, the effect of the selection step is only affected by the amount $(1 - \epsilon) * (1 - o(H) * P_m)$, which is an overestimate of how much crossover and mutation destroy instances of schema H. Now, we can conclude that highly fit, short defining length, lower order schemata are strongly propagated generation to generation. This propagation of good schemata is driven by the stochastic selection according to high fitness, and the above equation 3.6 means that we can crossover and mutate, and so produce new and possibly much better schemata, with only small change to this effect (because the amount

of destruction of good schema is usually very small). This is a very important conclusion in Genetic Algorithms and it has a special name : *the Schema Theorem* or *the Fundamental Theorem of Genetic Algorithms.*

3.5 Building Block Hypothesis

The Schema Theorem (ST) enables us to start to formulate initial and tentative suggestions and ideas with regard to why a GA works well and how we might make it more effective. The clearest point suggested by the ST is that highly fit schemata of low order and short defining length seem to be particularly important to the GA. In fact, such schemata are sufficiently important to merit their own name, and are called *building blocks* [Goldberg 89]. Genetic Algorithms appear to seek near optimal performance through the evolution of short, low-order, high-performance schemata, or building blocks. This is what Goldberg [Goldberg 89] called the *Building Block Hypothesis* (BBH). Later, Grefenstette [Grefenstette 93] observed some inadequacies with the idea of the BBH; mainly, it derives from considering (in the ST) global knowledge of the fitnesses of schemata, whereas in reality we can only estimate these fitnesses based on the small sample of their occurrence in the current population. Grefenstette therefore renames the BBH as the *Static Building Block Hypothesis* (SBBH), which makes explicit that the BBH does not consider the dynamically varying nature of the estimated fitnesses of schemata.

In the interests of finding further ideas about how and why GAs work, [Bethke 81] presents some special functions which he shows that a GA cannot optimise easily, while [Goldberg 87] used similar ideas and introduced the important notion of *Deceptive Problems*, which lead a GA to diverge from global optima by violating the building block hypothesis in the extreme. That is, a deceptive problem lures the GA into finding building blocks which *do not* constitute parts of the global optimum. A simple example is the problem of maximising the function $f(x) = x$, where x is a bitstring of length L, but where $f(0)$ is taken to be 2^{L+1} instead of 0.

Goldberg defined a simple 2-bit version of this. which he called the *minimal deceptive problem* (MDP), because it is the smallest possible deceptive problem, and showed that

a GA is so robust that it usually will find the global optimum on this anyway, even under unfavourable conditions. From then on, many other variations on deceptive problems were designed and investigated. Recently, [Grefenstette 93] criticised the Static Building Block Hypothesis (SBBH) as a description of the dynamics of GAs and showed that deception is neither necessary nor sufficient for problems to be difficult for GAs by exploiting contradictions between the SBBH and the Schema Theorem. He used *collateral convergence* to show that some functions that are highly deceptive according to the SBBH are actually very easy for GA to optimise and *high variance in sampled schema fitness* to demonstrate that some functions that have no deception at all, which should be easy for GA to optimise according to the SBBH, are nearly impossible for a GA to solve.

As a result, we know that one cannot rely on estimates of the relative fitness of a building block in the actual population, and it is a mistake to use the idea of static average fitness. In other words, it seems impossible to predict the dynamic behaviours of GAs on the basis of static average fitness. [Grefenstette & Baker 89, Mühlenbein 91, Mason 93, Altenberg 94] also expressed their dissent against the view that the Schema Theorem is the foundation for an explanation of the power of GAs. For example, [Mason 93] notes that "if we just consider the selection operator but no crossover or mutation operator, then the Schema Theorem and Intrinsic Parallelism are both satisfied". All that the ST really tells us is that crossover and mutation, under certain conditions, do not interfere with the effect of selection. What the ST fails to indicate is why the introduction of crossover really improves the performance of a GA without these operators. It is the crossover operator that makes GA different from other search algorithms, so Mason claims that "the GA community needs to adopt a more rigorous research approach less dependent upon uncertain evolutionary parallels instead of seeking to explain and understand their behaviour by considering such factors as Schema Processing and Instrinsic Parallelism".

All of above suggests that the Schema Theorem is not much use in practice. Besides, this form of the Schema Theorem is only for a single generational-based GA cycle and binary bit strings, and so is only very roughly suggestive of what goes in the GAs we use later in this thesis.

Chapter 4

GAs in Timetabling

4.1 Introduction

The simplest timetabling problems can be seen as problems of assigning v events (for example, exams, seminars, projects, meetings) to s time slots. More formally, we have the following definition of a simple timetabling problem.

Let E be the finite set of v events $\{e_1, e_2, \ldots, e_v\}$ and
let T be the finite set of s time slots $\{t_1, t_2, \ldots, t_s\}$.

We define an assignment to be an ordered pair (a,b), such that $a \in E$ and $b \in T$, with the simple interpretation that "event a occurs in time-slot b". The timetabling problem is then the problem of determining the v assignments, one per event, which make a good timetable.

Timetabling is accompanied by the presence of many constraints and objectives that should be met. These constraints and objectives can render many (or even all) of the s^v distinct timetables poor or invalid solutions. The most prominent overall constraint (central to all timetabling problems) is that there should be no *clashes*, that is: any pair of exams (or lectures, tutorials, etc ...) which are expected to share common students or teachers should not be scheduled simultaneously. The fact that a person cannot be in two places at once is central to most timetabling problems of interest; dealing with this constraint in particular is the major concern of all algorithmic research in the area, and is what we concentrate on in this thesis. We shall now discuss in turn

the various kinds of constraints that usually define a timetabling problem.

4.2 Constraints

4.2.1 Edge Constraints

We consider constraints in which two events cannot share the same time slot and term it *edge constraint*. One instructive way of thinking about this constraint is by analogy with the graph colouring problem. Under this analogy, we consider the set of events E to be the vertices of a graph and the set of times T to be the colours (or labels) that can be used to paint the vertices of this graph. The constraint under consideration can then be represented as an edge between vertices e_i and e_j, and the simple timetabling problem becomes one of colouring every vertex in such a way that no two vertices connected by an edge are of the same colour.

The *graph colouring problem* is a famous problem in graph theory, see for example [Wilson 85], and much work has been done on it. Two graph-theoretic results are worth mentioning as they can be applied to the simple timetabling problem by virtue of the direct analogy:

- **Number of colouring:** First, a graph is called *simple* if no edge starts and ends at the same vertex and at most one edge joins any pair of vertices. Then the number of ways of vertex-colouring a simple graph using s colours (time-slots) and c edges (constraints) is a polynomial $g(s)$ of degree v of the form:

  $$g(s) = s^v - cs^{v-1} + \cdots + \alpha s \tag{4.1}$$

  for some constant α, and the coefficients alternate in sign. This result is at first glance a little surprising since it suggests that at least for large values of s, a proportion of at least $(1 - c/s)$ of all timetables will be valid solutions. However, practical timetabling problems tend to involve a fairly small value of s and fairly large value of c so that not much can be said about the density of possible solutions by looking at equation 4.1.

- **Brooks' theorem** [Wilson 85]: If not every pair of vertices is joined by an edge, and if no vertex has more than g edges attached to it and there is no clique

(subgraph such that every member is connected to every other member) of size g, then the graph can be vertex-coloured using at most g colours. Thus, given an instance of the simple timetabling problem we can use Brooks' theorem to say how few times-slots are really necessary.

Graph-theoretic methods can be used to handle the simple timetabling problem. Unfortunately, real problems are hardly ever that simple. *Transversal theory* (see for example [Wilson 85] for an introduction) can be applied to analyse slightly harder timetabling problems such as finding some coherent set of assignments of events, time-slots and locations given a set of binary constraints between them. However, these techniques do not naturally extend to handling the range of non binary or non-symmetric constraints that arise in many practical timetabling problems. We give some examples of such constraints below, and in later sections we discuss how these can be reasonably handled by GA-based methods.

4.2.2 Ordering Constraints

In many cases there may be a partial ordering on the events in E. Ordering constraints occur frequently in examination timetabling, but even more frequently in, for example, class and/or lecture timetabling. In the latter, we may typically require laboratory sessions to occur later in the week than an associated lecture. Without ordering constraints, a simple timetabling problem has the property that any solution remains a solution when subjected to a permutation of the slots. That is, if the following timetable is a solution to a problem with edge constraints only:

$$((e_1, t_1), (e_2, t_1), (e_3, t_1), (e_4, t_2), (e_5, t_2))$$

Then we can permute t_1 and t_2, secure that the resulting timetable will also be a solution:

$$((e_1, t_2), (e_2, t_2), (e_3, t_2), (e_4, t_1), (e_5, t_1))$$

This is because what matters is only that slots for certain groups of events are different; the identity of the slot itself is unimportant. The introduction of ordering constraints

(and many other kinds of constraint) changes this. If in the above example time t_1 was before time t_2, and there was a constraint which preferred c_1 to occur before e_4, then the 'period permuted' solution would no longer be optimal. Note, too, that ordering constraints make the problem genuinely harder. Again, given:

$$((e_1, t_1), (e_2, t_1), (e_3, t_1), (e_4, t_2), (e_5, t_2))$$

which violates some ordering constraints, it may not be possible to find a permutation of $\{t1, t2\}$ which works. For example, imagine that there are only two slots available, and the edge constraints are such that the only possible partition of these events is into the sets e_1, e_2, e_3 and e_4, e_5. If we then also required $e_1 < e_4$ and $e_2 > e_5$ (that is, c_1 must occur before c_4 and c_5 must occur before e_2), then there would be no solution satisfying all of these constraints. Some GA timetabling work has been based on manipulating events in the fashion hinted at here; see [Abramson & Abela 91].

The main point is that the need to consider an ordering (or a partial ordering) on the set T is what makes it difficult to augment standard graph colouring based methods to cope effectively with such constraints.

4.2.3 Event-spread Constraints

It is often the case that human timetablers desire to have certain events 'spread' out across timetables in some way. For example, in exam timetabling, it is desirable that as many students as possible have their exams reasonably spread out. One possible manifestation of this desire is to stipulate that no student has more than 2 exams in one day. In lecturing timetabling, a common example is that a student should not have to sit through four lectures in a single day; rather, the lectures an individual student must attend should be evenly spaced out during the week. Also, different lectures on the same topic (for example, there may be two Prolog lectures per week) should preferably occur on different days. Another instance might be that certain laboratory sessions cannot be scheduled on the same day in order to allow time to clean up and prepare elaborate equipment for the next class.

Such event-spread constraints are important, but may be very difficult to cope with in general. Typically, they are ignored in many cases. In some cases, event-spread

constraints are handled by artificially specifying that certain named events must come between those that are to be spaced out. This reduces the problem to a more tightly constrained one involving binary ordering constraints. However, such reductions may result in a more constrained problem which is unsolvable.

4.2.4 Preset Specifications and Exclusions

At times, there may be an *a priori* requirement that a certain event must not be assigned any of a given set of times or that some assignments may be pre-specified. In some cases, such specifications or exclusions are hard constraints and so we cannot accept a timetable that doesn't satisfy them. For these cases, one possible approach is to 'remove' any event with a predefined assignment from E, and attend to the smaller problem of timetabling the remaining events. In a sense, the search space is reduced. However, in general the addition of preset specifications and exclusions results in further complication. This happens if an event e_f is fixed into a slot t_f, we probably cannot remove it from E, since other events will still be tied to it by some constraints. For example, if we require $e_f < e_g$, then it is necessary to 'transform' this constraint to "e_g must occur later than slot t_f". Similarly, all other constraints involving e_f must be transformed. Removing a specified event in this manner is fine if the event is genuinely required to occur in the given slot. However, if this is not the case, then it may be quite possible for 'better' (in some sense) timetables to emerge if we allowed some such specifications to be violated. For this reason, a more general and flexible approach seems to be to avoid this 'remove' and 'transform' approach; instead, simply associate a penalty for violating a specification constraint, making that penalty more or less harsh according to how much the specification is genuinely required.

4.2.5 Capacity Constraints

In order to check that a room's capacity isn't exceeded, we must first know the overall room capacity/availability and the students capacity for a particular lecture. In some case, for example, several exams can be held at the same time and share the same large room if there are no common students. However, several lectures/tutorials cannot use the same room at the same time even they don't share the same students. During

evaluation, the system just needs to run through this list of constraints checking each in turn and accumulating penalties for violations.

4.2.6 Hard and Soft Constraints

Making a distinction between hard constraints which must not be violated and soft constraints, or preferences, which it would be pleasing to satisfy but which can be violated if necessary, is one of the great difficulties with conventional approaches to timetabling. This is true of human timetablers, who often begin by trying to solve a particular timetabling problem assuming certain sets of hard and soft constraints. If that effort fails, they may convert one or more hard constraints into soft ones, or even delete them altogether, after re-negotiating with all the other people involved. Thus they move on to trying to solve a new problem, for which the work done on the original problem may be largely unusable.

Our GA-based approach to timetabling awards different levels of penalty for different kinds of constraint violation, so it can avoid this distinction between hard and soft constraints. Therefore, a timetable which turns out to have just one or two hard constraint violations may be treated as better than one with a large number of soft-constraint violations. The GA-based approach is thus addressing a whole spectrum of what would otherwise be regarded as differing timetabling problems, in a uniform way. This may even be what a user might really prefer -- a system that takes some liberties with the problem formulation in the first place, rather than expecting him to conduct a manual search through the space of possible timetabling problems.

4.2.7 Other Constraints

There are other kinds of constraint beside those more common ones described above. For example, 'teaching loads' constraints which can be adapted from techniques used in capacity constraints.

Many instances of timetabling problems have their own characteristics which cannot be adequately captured in a generalisation.

4.3 Basic GA Framework for Timetabling

4.3.1 Representation

When using Genetic Algorithms to solve an optimisation problem, the first important step is to choose how to represent a solution of that problem as a chromosome, which can then be submitted meaningfully to crossover/mutation operators. The choice made here is for a chromosome to be an ordered list of numbers. For example, if there are ten events to be scheduled, and eight time slots, then a chromosome in this problem is a list of length ten (each position in this list represents a particular event), where each gene can be a number from one to eight, for example, the chromosome: *3 7 2 1 7 7 4 6 5 8*, represents a timetable where event 1 is in time slot 3, event 2 is in time slot 7, etc... The motivation for this representation is that we can use any standard GA operators without any need for repair which can save us much time to focus on problem-specific constraints. Other work on timetabling problems has used a different chromosome representation where, more like the actual timetable itself, positions are time slots and the values that a 'position gene' can take are sets of events that might occur at that time. This kind of representation however leads to a problem called the *label replacement* problem [Abramson & Abela 91]; the problem is that the crossover operator can easily lead to producing timetables which leave out many of the events that need to be scheduled. An algorithm must be used to then modify the chromosome to reintroduce all the necessary events. The analogous situation in our representation is that a chromosome may represent timetables where not all time slots are filled: this is, however, a legal[1], and possibly useful timetable. Therefore, using this representation, the Genetic Algorithm does not need to be modified with a computationally expensive label replacement algorithm.

When we must allocate more than one dimension, for example time and room, in the timetabling problem simultaneously, the representation that we have successfully used is a chromosome with the length two times longer than the previous one. The first half of the chromosome is a list of numbers of length v (the number of events

[1] It might not be legal if 'legal' is defined in terms of hard constraints. However, in our GA approach, the definition of 'legal' is all events have a timetable slot assigned. So, it counts as legal in this situation.

to be scheduled), each element of which is a number between 1 and s (for example, the number of time slots available). The interpretation of such a chromosome is that if the nth number in the list is t, then event n is scheduled to occur at time t. For example, the chromosome [2,4,7,3,7] represents a candidate solution in which event 1 takes place at time 2, event 2 takes place at time 4, and so on. The second half of the chromosome is a list of numbers of length v (the number of events to be scheduled), each element of which is a number between 1 and c (for example, the number of rooms available). The interpretation of such a chromosome is that if the nth number in the list is r, then event n is scheduled to occur in room r. For example, the chromosome [3,6,5,3,1] represents a candidate solution in which event 1 takes place at room 3, event 2 takes place at room 6, and so on. Therefore, we could also think of it as one GA with operators which never violate the separation between the halves of the chromosomes of length $2v$.

Similarly, it is very easily to extend to more dimensions. For example, if it is one chromosome of length $3v$: v of them specify time-slots for the v events, v of them specify locations for the v events and the last v of them can specify the teachers or tutors for the v events etc.

4.3.2 Dealing with Constraints

4.3.2.1 Dealing with Exclusions

The size of the search space can be reduced by taking into account that some events *must be* excluded from certain time slots. The exclusion file can exclude all the unallowable time slots of each event for the purpose of minimising unallowable chromosome and as a result of minimising the total search space. In other words, the search space is reduced because the set of time-slots for those events is smaller than maximal. We can formalise above situation as follows:
Assume we have a set of events E; for each event e_i there is a set of allowable timeslots T_i. Valid timetables are a set S:

$$S = \{\{(e_1, t_1)(e_2, t_2)...(e_v, t_v)\} : t_1 \in T_1, ...t_v \in T_v\} \tag{4.2}$$

If $S_1 \in S$ and $S_2 \in S$, then the crossover of S_1 and S_2 is $\in S$, at least for one-point, two-point, N-point, uniform and all related versions of crossover which don't alter the locations of a gene within the chromosome. Mutation of $S_i \in S$ will produce another member of S provided it is defined suitably, for example, to mutate k-th gene, we need to choose a random member of T_k.

Dealing with such forced exclusions within our GA timetabling framework is straightforward; given the necessary data (for example, via an 'exclusions' file), we appropriately constrain the allele range of each gene individually. Thus, for example, a mutation of gene e_i can only result in an assignment from T_i. Similarly, exclusions for room and/or lecturer/tutor assignments can be easily accommodated.

4.3.2.2 Dealing with Capacity Constraints

One common consideration is the problem of assigning events to locations and it is usually possible (and easy) to allocate rooms for a given timetable. In addition to the sets E and T, there is a finite set L of possible locations in which the events can take place. An assignment is then a triplet (e,t,l), and the timetabling problem becomes further complicated by capacity constraints involving the placement of events into locations. For example, in lecture/tutorial timetabling it is unacceptable to have multiple events occurring at once in the same location; in examination timetabling this is usually acceptable, however there is usually a maximum number of exams and/or students who can occupy a location at a time.

Dealing with such capacity constraints within our GA timetabling framework is also straightforward. In our experience, it can be quite justifiable to consider the timetabling problem in the absence of location constraints. Human timetablers frequently break timetabling and location assignment up into separate problems anyway, solving the latter only after solving the former. Often, the location assignment problem is then found to be quite simple if there are enough locations available; however, with the increasing pressure to squeeze the most out of existing resources, the location assignment problem may well be becoming ever more troublesome in many institutions. For example, this is made difficult if the timetable includes too many exams occurring at

the same time. Therefore, it is necessary to penalise this situation in our framework and report the situation when more than one exam occurs at the same time and the total number of the students exceeds the available room capacity. Having this report, the exam organiser can either find a larger room or search for more than one room to accommodate the students. In particular, there is a specified maximum number of students who can be sitting an exam at the same time, and also a maximum number of exams that can be scheduled at the same time.

We can also encode room information in the chromosome and regard it as a hard constraint. In our framework, in order to minimise the search space, the room information should tell us which event can use which room instead of just randomly assigning a room and letting the evaluation function give punishments when the room is an unallowable one. It is useful to prune as much search space as possible before evaluation. The search space for rooms is typically smaller than for timeslots because each event just has a few fixed number of allowable rooms in comparison with a large number of allowable time slots.

We can formalise the above situation in a similar way to exclusion time slots:
Assume we have a set of events E; for each event e_i there is a set of allowable roomslots L_i. Valid timetables are a set S:

$$S = \{\{(e_1, l_1)(e_2, l_2)...(e_v, l_v)\} : l_1 \in L_1, ...l_v \in L_v\} \tag{4.3}$$

If $S_1 \in$ S and $S_2 \in$ S, then the crossover of S_1 and S_2 is $\in$ S, at least for one-point, two-point, N-point, uniform and all related versions of crossover which don't alter the locations of a gene within the chromosome. Mutation of $S_i \in$ S will produce another member of S provided it is defined suitably, for example, to mutate k-th gene, we need to choose a random member of L_k.

4.3.2.3 Dealing with Specifications

In our framework, we can let the user pre-set an event at a specific time slot by just filling the preset file with that specific time slot or room in the corresponding event position. Our 'presets' are just big exclusions, for example, e_k is pre-set if T_k is of size 1. However, for practical reasons, it makes sense to treat these cases separately, rather

than saying for example, e_k must not take place in

$$\{t_1, ...t_{j-1}, t_{j+1}, ...t_s\} \tag{4.4}$$

because this is very cumbersome to say. In other words, if T is the set of all slots, we want to say either that e_k must be confined to some subset A $\in$ T or to the complement of some subset A $\in$ T.

Dealing with such preset specifications within our GA timetabling framework is as follows; given the necessary data (for example, via a 'specifications' file), we appropriately copy the predefined value to each gene individually. Thus, the specifications and exclusions will conflict if these data sets conflict themselves. In this case, the specifications have higher priority than exclusions. However, a penalty for exclusions arises simultaneously.

4.3.2.4 Dealing with Other Constraints

So far we have noted how genuinely hard exclusions and specifications are handled in our framework. All other constraints are handled by the GA itself. That is, we allow violations of these constraints to occur, and expect (or hope) that the operation of the GA will lead to the appearance of chromosomes encoding timetables with fewer and fewer violations.

Note, however, that this means we allow very many genuinely hard constraints (the edge constraints) to be broken. This is necessary, because if indeed there was a way to automatically construct timetables which satisfied them, then there would be no need to use the GA!

4.3.3 Initialisation

Initialisation randomly chooses $t_i \subset T_i$ (and $l_i \subset L_i$ if rooms too) which helps to prune the search space for us. After initialisation, the genes in the chromosome are all valid, the GA just needs to do selection, breeding (crossover, mutation) and evaluation from generation to generation.

4.3.4 Recombination

The choice of crossover operator to use in breeding can be quite difficult. Let us consider recombination first. In a timetable, there are various possibilities as to what may constitute a good building block. Clearly any edge constraint (e_i, e_j) constrains the values assigned to the two events somewhat. If there exist many constraints between the members of some small subset of the set of events then there will be relatively few mutually compatible assignments for the members of the subset. Any such coherent set of assignments for these events might therefore be a useful building block, and it would be sensible to arrange for it also to be a short building block by arranging the chromosome so that those events occupied some contiguous slice of it. The shortness of building block matters for one-point, two-point and even N-point crossover, but not for uniform crossover. For example, the building: #a######b# has long defining length, so one-point and two-point will have trouble with it, but uniform crossover has no trouble. However, if a building block has high order, for example: #abc#de###, then even uniform crossover is likely to disrupt it.

Making such potential building blocks be short would seem to be particularly useful when employing operators such as one-point and two-point crossover, since a short building block will be less likely to be cut and hence broken by the operator than a long building block. On the other hand this argument only applies when the building block already exists, either by chance in the initial population or by being created through the previous action of some breeding operator. If the initial population is not large this suggests that one-point or two-point crossover may take some while to form that good short building block in the first place.

Uniform crossover does not exhibit this dichotomy between production and destruction of good short building blocks. It explores the set of values that might compose a building block faster than one-point or two-point crossover. It also destroys a building block faster *if that building block is present in one but not both parents.* This suggests that uniform crossover is a good choice when the representation and fitness are such that short building blocks contribute an amount to the fitness which is significantly disproportionate to their size, so that selection and population update spread the building blocks through the population usefully faster than uniform crossover can

destroy them. Uniform crossover is also a good choice when there are going to be good but long building blocks, or if we cannot easily arrange the chromosome to ensure that many good building blocks will be short.

4.3.5 Fitness

Given a space P of candidate solutions to a problem, there are three desirable properties for the fitness function $f(p)$ ($p \in P$). We would like

- $f(p)$ to be a normally increasing function of the quality of p as a candidate solution of the problem, so that optimal solutions lie at the global maxima of f.
- f to be a reasonably well-behaved function, so that its value conveys some information about the quality of p as a solution in most parts of the space.
- $f(p)$ to change in some way that reflects this as p gets closer to being an optimal solution.

The quality of a solution p may not vary smoothly as the genes comprising p vary. For example, in scheduling applications [Fang *et al.* 93, Fang *et al.* 94], the chromosome might not directly represent a solution; instead it might represent a sequence of instructions for building a solution. A trivial change to an early member of the sequence might result in the construction of a significantly different candidate solution. The genetic operators such as crossover and mutation do not vary the gene values smoothly either. This also correctly suggests that it is wise to discourage later breeding between a parent and its child, since they will lie in some shared projective subspace and their offspring will be confined to that subspace too if mutation does not shift them out of it a little, for example, see [Eshelman & Schaffer 91].

In timetabling, we know that any timetable which satisfies all the constraints is an optimal timetable. Further, it seems reasonable to distinguish between timetables in terms of fitness based on the numbers and kinds of different constraints violated. One might choose to use something inversely proportional to the number of violated constraints. For instance, if $V(p)$ is the number of violated constraints in candidate p, one could choose:

(4.5) $$f(p) = 1/(1 + V(p))$$

so that the theoretical maximum of f is 1 if $V(p)$ is 0. In other words, it is a perfect timetable. However, this function treats all constraints equally and conveys no information about the extent to which individual constraints are violated.

An answer to this is to penalise the different types of constraint violations in accordance with their relative importance. That is, constraints which are more important (or hard constraints) will have heavy weight, whereas, constraints which are less important (or soft constraints) will have light weight. If we have n kinds of constraint, the penalty associated with constraint-type i is w_i, p is a timetable, and $c_i(p)$ is the number of violations of constraints of type i in p, then the fitness function becomes:

(4.6) $$\mathbf{f(p) = 1/(1 + \sum_{i=1}^{n} w_i c_i(p))}$$

The main advantage of this general function is that we can incorporate any constraint into the fitness function, along with an appropriate penalty measure for it, and we can then expect the GA to take this constraint into account during its optimisation. The relative penalty values may be chosen to reflect intuitive judgement of the relative importance of satisfying different kinds of constraint. In general, however, a penalty function as above, using a rough choice of penalty settings derived from the course organisers' notion of relative importance of different constraints, appears adequately robust for many problems.

Another aspect of this function is that evaluation should be fairly fast, which means here that it be computationally quick to check how many constraints of each type are violated in a given timetable chromosome. Fortunately, this is true for the kinds of constraints we have considered so far, which are indeed the most commonly occurring in timetabling problems. As for setting penalty values, the basic idea is that higher penalty settings for a constraint increase the artificial evolutionary pressure to remove that constraint from the population. Hence, we penalise the hard constraints more heavily than the soft constraints, since we *must* remove the hard constraints in order to get a feasible timetable, but can and must live with violations of the soft constraints.

However, the sum of penalties of several soft constraints may be larger than that of a single hard constraints. Therefore, the GA should be able to make tradeoff between different kinds of constraint violation based on relative importance.

4.3.6 Smart Mutation

Choice of a mutation operator is straightforward. One simple mutation operator would be to randomly change the alleles of a small number of genes; that is, for each gene, change to a random new allele with some small probability. Hereinafter, this is called 'gene mutation'. Another possibility is 'order mutation' where we might swap the alleles of two randomly chosen genes. However, experience with GAs has shown that *smart* or *directed* mutation, somehow using 'inside-information' about the problem, can aid evolution greatly. Four such operators for use on timetabling problems are considered here. All involve the concept of the 'violation-score of an event': this is simply the sum, weighted by the appropriate penalty values, of the constraint violations involving that event.

Violation-directed Mutation (VDM): choose an event with a maximal violation score, and randomly alter its assigned time.

Event-freeing Mutation (EFM): choose an event with a maximal violation score. Then, give it a new time which will maximally reduce this score.

Stochastic Violation-directed Mutation (SVDM): stochastically select an event, bias toward those with higher violation scores, and randomly alter its assigned time.

Stochastic Event-freeing Mutation (SEFM): stochastically select an event, bias toward those with higher violation scores, then stochastically select a new time for this event, bias towards times which will maximally reduce the event's violation score.

The set of events with a maximal violation score can easily be determined during fitness evaluation. All these mutations promise benefits via directing mutation to where it is most needed. Applying VDM some of the time enables the GA to continually

explore new timeslots for the currently most troublesome event or events. Applying EFM promises to directly find the most promising new timeslots for these events. It is not necessarily true, though, that concentrating on the most troublesome events, and/or finding the most promising new time for them, is a good idea on problems of realistic difficulty. The effect of VDM or EFM is analogous to that of a 'greedy' algorithm, performing intensive local improvement without reference to the global picture. SVDM and SEFM, on the other hand, may eliminate this aspect, at little or no extra computational expense.

Obviously, these smart mutation operators can be directly and straightforwardly defined for 'room', 'teacher' and/or 'agent' genes if necessary. In general, these and similar operators can be defined for any application, as long as it is possible to accurately apportion 'blame' to different parts of the chromosome. Since the representation used in our GA/timetabling framework is direct, and since fitness is composed from multiple contributions each pertaining to a small number (usually two) of genes, smart mutation operators can be readily constructed.

In the next chapter, we will apply the basic framework described in this chapter to two timetable problems: exam and lecture/tutorial timetabling.

Chapter 5

Experiments with GAs in Timetabling

5.1 GAs in Exam Timetabling

In this chapter we first examine what we call the METP (*Modular Exam Timetabling Problem*). This typically arises in universities running large modular degree schemes, in which each student takes an individual selection of exams from a wide inter-departmental pool of modules, many outside their own department. The events are exams, the time slots are possible start-times for those exams, the hard constraints are that no student should take more than one exam at a time, that is edge constraints, and the objectives are to generally minimise pressure on students, so that as few as possible have multiple exams in a day, consecutive exams, and so on.

Typical METPs are *NP-complete* [Garey & Johnson 79], and strewn with local minima which make it particularly difficult to address by way of heuristic search or hill-climbing techniques. METP complexity is also illustrated by the size of the solution space. For example: if there are s possible start times, and v exams, then there are s^v candidate schedules. In the particular METP which occurs within the **EDAI**[1] in 1991/1992, having 44 exams, 28 timeslots (4 per day over 7 days), this was 28^{44}, or c. 5×10^{63}. This number represents the complete space of possibilities, very many of which would not be considered by, for example, a human timetabler. Calculating the complexity from a more realistic point of view, suppose we only consider timetables with at most

[1] The University of Edinburgh, Department of Artificial Intelligence

two exams per slot and at least one per slot. Then there are 44!/(44-28)! $\approx 1.27 \times 10^{41}$ choices of 'first' exam in each slot; and 28!/(2x28-44)! $\approx 6.36 \times 10^{20}$ ways of arranging 16 exams and 12 blanks to be the 'second' exam in each slot, if we treat blanks as indistinguishable. In the 16 cases where there are 2 exams per slot, we don't care which we call 'first'. The total number of such timetables is thus $1.27 \times 6.36 \times 10^{61}/2^{16}$ $\approx 1.232 \times 10^{57}$ timetables. Since we may not care about labelling of days either, we can further reduce this to $1.232 \times 10^{57}/7! \approx 2.444 \times 10^{53}$ cases to consider. If we just allocate exams to days, and treat the order of days as irrelevant and the ordering of exams in any days as an essentially trivial problem, we still have $44!/(7! \times 7!^2 \times 6!^5)$ or c. 1.486×10^{26} cases, which remains a very large size of search space.

When an METP is tackled, typically in a university or college department although very similar problems often occur in other cases, it is usually addressed by hand (for example: by an exam organiser). This involves producing an initial draft timetable, followed by perhaps weeks of redrafting after student feedback complaining about the unfairness of the latest draft. In some cases the exam timetable can be produced directly from the lecture timetable, but this is rarely helpful in those cases involving modules from many departments where several students attend lectures at more than one department. Even in cases where the lecture timetable is a reasonable start, the process is complicated by the different criteria relating to examinations. For example, it may be acceptable for students to attend four lectures every Monday, but it is rarely acceptable for them to have to sit four exams in a day; indeed, examinations will typically be longer than their associated lectures.

5.1.1 Problem Description

A general timetabling problem is one where events (e1,e2...) have to be performed at specific times (t1,t2...). In a particular problem, there are constraints on whether certain events can appear at the same time, close together, etc...The EDAI continually faces a hard MSc exam timetabling problem from a large multi departmental modular degree scheme, involving the scheduling of exams from mostly the Artificial Intelligence and Computer Science Departments, and some from the Cognitive Science and Meteorology departments. This is referred to as the AI/CS MSc exam timetabling problem.

TIMETABLE	AM		PM	
	09:30-11:00	11:30-13:00	14:00-15:30	16:00-17:30
Friday	0	1	2	3
WEEKEND				
Monday	4	5	6	7
Tuesday	8	9	10	11
Wednesday	12	13	14	15
Thursday	16	17	18	19
Friday	20	21	22	23
WEEKEND				
Monday	24	25	26	27

Table 5.1: Time slots allocation for examination

In the AI/CS MSc exam timetabling problem, the events are exams, and the times are 24 or 28 separate time slots (four per day for six days in 1990/1991 and seven days in 1991/1992). The number of exams and students was 38 and 47 in 1990/1991 and 44 and 93 in 1991/1992 respectively. Exams are held at two different sites; some exams are held centrally (C) and others are held in Kings Building (KB) which is located about 2 miles away from the city centre.

In the rest of this section we explore the applications of GAs to this problem.

5.1.2 The AI/CS MSc Exam Timetabling Problem

There are five days in a week for the exams and each day has four time slots, two in the morning and two in the afternoon. All exams are the same length. The interval within the same half day is half an hour, however the interval between morning and afternoon is one hour. If there are seven days for the exams, then the number of time slots will be 28 and they must occupy at least two different weeks (for example, if the first day is Friday of the first week, then the last day should be Monday of the third week). Table 5.1 illustrates an example timetable, similar to timetables used for the AI/CS MSc exam timetable.

The constraints involved are as follows:

Strong or Hard Constraints:

1. The same student cannot take two different exams at the same time. In other words, the exams cannot clash for any student. (Edge Constraint)

Weak or Soft Constraints:

1. Very strongly prefer no more than 2 exams per day for a given student. (Event-spread Constraint)
2. Strongly prefer not to have exams consecutive on the same half day for the same student. (Event-spread Constraint)
3. Prefer not to have exams consecutive on different half days of the same day for the same student. (Event-spread Constraint)
4. Try not to put too many exams in the same time slot, to avoid overcrowding the examination hall.[2] (Capacity Constraint)
5. Some lecturers specify that some exams may not be scheduled on specified days or afternoons when they are unavailable to invigilate. (Time Exclusions)
6. To avoid marking pressure, lecturers may wish to exclude heavily subscribed exams from later in this exam periods. (Time Exclusions)

Furthermore, the AI MSc exams are also available to CS MSc students and AI/CS undergraduate (UG) students. There are also Meteorology and other departments students. Therefore, it is a multi department exam timetabling problem. Hence, constraints must be taken into account which arise from other commitments of these non-AI MSc students. The following subsections will fully describe how these constraints are handled.

[2] The AI/CS MSc exam timetabling problem is simplified by the fact that a large examination hall exists which can accommodate several exams at once. Assignment of exams to particular rooms or halls therefore need not be taken into account.

5.1.3 Applying the Basic GA Framework to Exam Timetabling

5.1.3.1 Representation

The representation we choose is described in the previous chapter. The chromosome is simply a list of numbers of length e (the number of exams to be scheduled), each element of which is a number between 1 and s (the number of timeslots available). The interpretation of such a chromosome is that if the nth number in the list is t, then exam n is scheduled to occur at time t.

5.1.3.2 Exclusion Time Slots

Exclusion constraints are kept in a text file. Using this information in the exclusion file, all chromosomes are forced to have only valid slots for each gene. One way to produce valid timetables in the initialisation is to keep on generating initial pool solutions until it does not violate these exclusion constraints. The way used by us to produce allowable time slots more intelligently and efficiently are reading the exclusion file and recording the allowable time slots for each event, for later use, which do not violate these constraints. Then, just randomly generate instances from the set of allowable time slots each time. In other words, the first pass finds the allowable time slots for each event and the second pass only chooses among these for each instance. Crossover guarantee the allowable time slots because it just recombines two parents with allowable timeslots and certainly produces two children with allowable timeslots. However, mutation also needs the second pass to guarantee each gene not violating the time slots set in the exclusion information.

The exclusions information is a handy way to deal with constraints such as:

- Some lecturers specify that some exams may not be scheduled on specified days or afternoons when they are unavailable to invigilate.
- To avoid marking pressure, lecturers may wish to exclude heavily subscribed exams from later in this exam periods.

A part of the exclusion file[3] for the 1992/1993 AI/CS MSc exam timetabling problem

[3] The complete data file is listed in appendix A.

is as follows:

```
conc : 0 1 4 5 6 7 8 9 20 21 24 25 28 29 30 31 32 33 34 35
cvis : 0 1 4 5 8 9 20 21 24 25 28 29 30 31 32 33 34 35
...
```

'conc' and 'cvis' are abbreviations of Connectionist Computing and Computational Vision respectively. Each number represents a time slot, 0-3 represents the first day's 4 time slots and 4-7 represent the second day's 4 time slots etc...For example, conc cannot be put in the mornings of the first three days, in the afternoons of the 2th, 8th and 9th days and in any day from the 6th to 9th days etc....

5.1.3.3 Room Considerations

The room allocation aspect of the EDAI AI/CS MSc exam timetabling problem is made easier by the fact that two large rooms are available which can hold several examinations at a time. The only constraint we need to consider to deal with this is to disallow more than a given number of exams from occurring in the same timeslot. In this problem, we can easily handle this constraint without resorting to extending the representation to incorporate 'room genes'. Instead, the fitness function (see later), simply attaches a penalty to any instance of more than one exam occurring in the same timeslot and the total capacity larger than the largest room size.

5.1.3.4 Preset Time slots

In the AI/CS MSc exam timetable, we can let the user pre-set an exam in a specific time slot by specifying the information in a file. In this file, a positive integer denotes a slot in which that exam must happen.

A part of the preset file[4] for the 1992/1993 AI/CS MSc exam timetabling problem is as follows:

```
aied : 2
conc : 26
...
```

[4] The complete data file is listed in appendix A.

For example, aied and conc are pre-set at time slots 2 and 26 respectively. The system will penalise and report any conflicts between the preset time information and exclusion time information.

5.1.4 Specific Framework for Exam Timetabling

5.1.4.1 Student-Exam Data

The student-exam data is simply a collection of lists, where each list is the set of exams to be taken by a particular student. It contains all the information about all the students' exams, one student per line. Normal full time MSc students in AI or CS department each take eight exams. There are also part time MSc students, who take some of the eight exams each year. These student-exam information constraints are kept in a text file.

A part of the student-exam file[5] for the 1992/1993 AI/CS MSc exam timetabling problem is as follows:

```
student1 : aied conc cvis ias isc
student2 : lisp prol kri1 kri2 mvis nlp mthr cvis
student3 : lisp prol kri1 kri2 mvis os1 isc nlp
...
```

For example, student1 needs to take aied, conc, cvis, ias and isc, and student2 needs to take lisp, prol, kri1, kri2, mvis, nlp, mthr and cvis. This information is very important for exam timetabling because it is used to calculate occurrences of various constraint violations.

5.1.5 Evaluations

For a general METP, the evaluation function must take a chromosome, along with the *student data*, and return a *punishment* value for that chromosome. Chromosome fitness can then be taken as the inverse of punishment (or, to be precise, 1/(1+punishment) — to avoid division by zero). The evaluation function may comprise a weighted set of individual functions, each 'punishing' the chromosome in terms of a particular punish-

[5] The complete data file is listed in appendix A.

able 'offence'. In the METP we experimented with, the components of the evaluation function are functions which respectively count the number of instances of the following 'offences':

- **Hard constraints:**
  - clash: student can't take two exams at the same time
- **Soft constraints:**
  - consec_d: two exams per day (different half day)
  - consec: two exams per day (same half day)
  - three: three exams per day
  - four: four exams per day
  - slot: an exam in an excluded slot
  - large: more than one exam in a single timeslot and the total capacity larger than the largest room size

The overall quality of the timetable will be evaluated by the evaluation function. The evaluation function adds up all violation of all constraints. Each constraint has an associated 'weight' or 'penalty'. For example, a single clash has a penalty of 300, n clashes have a penalty of $n \times 300$. Absolute values of penalties do matter in the fitness selection case, because fitness is inversely proportional to 1 plus sum of weighted offences described in the previous chapter; with rank or tournament selection absolute value will not matter. Changes in *relative* penalty settings might make a difference, however. Experience has found, though, that overall performance is not significantly sensitive to these relative values. Therefore, it seems that intuitive choices of these relative settings will be adequate. For example, in [Fang 92] we used a set of much smaller penalty sizes and found that the overall performance is not much different. On larger or different kinds of problems, however, this may no longer be true; more research is needed to look at this general point.

Here, the relative penalty values may be chosen to reflect intuitive judgement of the relative importance of satisfying different kinds of constraint. The penalties for each constraint are recorded in the following file:

```
#define PEN_CLASH          300
#define PEN_FOUR           100
#define PEN_THREE           30
#define PEN_CONSEC          10
#define PEN_SLOT            10
#define PEN_LARGE            3
#define PEN_CONSEC_D         3
```

An instance of four is also two instances of three, but is only penalised as being the one instance of four. In general, only one of consec, consec_d, three or four is awarded, not all of them. In our exam timetabling problem, there are four slots per day but the slots are not temporally adjacent. There is always at lease 30 minutes between them. This means we don't have to worry about student travel times between exams; and in fact nearly all exams take place on one site, where travel time is small. The penalty for consecutive exams exists just to let students have more time to rest between exams which is different from two exams per day because the interval between the latter might be very large, for example one exam in the first and the other exam in last timeslots on the same day. Therefore, we consider consecutive exams instead of two exams per day. In other METPs, different sets of component functions and weightings (the above were chosen intuitively) may be more appropriate, for example: components which treat very early exams, or perhaps occurrences of four exams in two days, as separate offences. In general, the fitness function for an METP, where p is a chromosome, $\{c_1,...,c_n\}$ is a set of functions which each record the number of instances of a particular 'offence', and $\{w_1, ...,w_n\}$ are the weights attached to these offences, is:

$$f(p) = 1/(1 + \sum_{i=1}^{n} w_i c_i(p)) \tag{5.1}$$

Evaluation is generally the computational bottleneck of a GA. With the METP, and 'offence-counting' modules of the type we have described, it is easy to see that the time to evaluate a chromosome increases linearly with the number of students, modules and constraints and can involve many computations depending on the kinds of punishment being looked for.

YEAR	EXAM	STUDENT	DAY
1990/1991	38	47	6
1991/1992	44	93	7
1992/1993	44	84	9
1993/1994	59	200	9

Table 5.2: Exam problem information

In the next subsection, we present a clear comparison between the actual timetables produced by human experts and the timetables produced by the system.

5.1.6 Experiments

5.1.6.1 Actual Timetable

Table 5.2 lists the actual data and informations on the EDAI AI/CS MSc 90/91, 91/92, 92/93 and MSc/UG 93/94 exam timetable problems. The EXAM column represents the number of exams held in that particular year and the STUDENT column represents the number of students taking the exams in that year. In addition, DAY is the number of days between the first and the last exam in that year.

The actual timetables used in the first two years were produced by course organisers and the last two years were produced by GA (used by the course organiser), we summarise them in Table 5.3. The timetables used in the last two years are clearly much better than those of the first two years. That is, using the GA-produced timetable can save course organisers much effort, but more important it is more fair to the students because fewer students need to take consecutive or three exams in one day. Also, the timetables of the first two years was the result of a first draft which took about an hour to produce, followed by repeated iterative feedback, and more drafts, as students complained and the timetable was revised over several weeks.

- *Actual Timetable 90/91:* The actual human-produced timetable for AI/CS MSc 90/91 is shown in Table 5.4 with punishment=122 (consec_d=4, consec=11, three=0, four=0, large=0). In other words, there are 4 occurrences of two consecutive exams on different half days of the same day and 11 occurrences of two

YEAR	PENALTY	CONSEC_D	CONSEC	THREE	FOUR	LARGE
1990/1991	122	4	11	0	0	0
1991/1992	328	33	16	2	0	3
1992/1993	0	0	0	0	0	0
1993/1994	16	0	1	0	0	2

Table 5.3: Actual exam timetable penalties

	09:30 - 11:00	11:30 - 13:00	14:00 - 15:30	16:00 - 17:30
Monday	lis (6) aaic (4)	lac (16) cpin (2)	clg1 (3) anet (12)	mvis (12) cafr (5)
Tuesday		prol (30) ccomp (2)	spd (4)	as (20) acar (13)
Wednesday	lisp (35) bsv (3)	cad (1)	lng2 (3) ds (11)	robs (5) grph (5)
Thursday	awl (21) algc (1)	mthr (11) lng1 (3)		nlp (14) os (8) fprg (3)
Friday	kril (35) ai (4)		cvis (8) clg2 (3) db (4)	cays (4)
Monday	kri2 (26) ccom (10)	ias (10)	awt (12) aps (3)	

Table 5.4: Human-produced actual exam timetable in MSc 90/91

consecutive exams on the same half day of the same day.

- *Actual Timetable 91/92:* The actual human-produced timetable for AI/CS MSc 91/92 is shown in Table 5.5 with punishment=328 (consec_d=33, consec=16, three=2, four=0, large=3). In other words, there are 33 occurrences of two consecutive exams on the different half days of the same day and 16 occurrences of two consecutive exams on the same half days of the same day. In addition, there are 2 occurrences of three exams on the same day and 3 occurrences of an overcrowded time slot.

- *Actual Timetable 92/93:* The actual GA-produced timetable for AI/CS MSc 91/92 is shown in Table 5.6 with punishment=0. In other words, it is perfect with respect to the imposed constraints.

- *Actual Timetable 93/94:* The actual GA-produced timetable for AI/CS MSc/UG

	09:30 - 11:00	11:30 - 13:00	14:00 - 15:30	16:00 - 17:30
Monday	cca (21)	kri1 (62) ccomp (1)	clg1 (11) spd (5)	mvis (26) fprg (8)
Tuesday	cvis (21) fsm1 (2) asc (14)	cad (5)	lisp (61)	algc (8)
Wednesday	isc (21) lng1 (11) cp (1)	grph (15)	kri2 (53) cafr (10)	ccom (21)
Thursday	aied (20) clg2 (11)	da (25)	db (21) rs1 (1)	prol (49) lng2 (2) sp1 (1)
Friday	es (38) sp2 (1)	asic (6)	os (12) cpin (7) rs2 (1)	ias (12) bav (6)
Monday	swl (28)	nlp (21)		nnet (33)
Tuesday	essp (2) aps (14)	cc (11)	mthr (24)	ai (11)

Table 5.5: Human-produced actual exam timetable in MSc 91/92

	09:30 - 11:00	11:30 - 13:00	14:00 - 15:30	16:00 - 17:30
Monday	lisp (37) ip2 (9)	cc (13) os (17)	aied (28) gis (6)	ci (5) asc (21)
Tuesday	es1 (25)	cad (29) cafr (5)		prol (35) ln2 (3)
Wednesday	kri1 (44)		nlp (15) lfs1 (4)	bav (9) rs2 (9)
Thursday	swp (32) cl2 (6)	isc (40)		mvis (23) ppc (16)
Friday	cvis (26) aps (11)	es2 (15)		mthr (11) db (14) nnet (17)
Monday	cl1 (6) ip1 (9)	da (47)	ias (31)	gr (20) ln1 (5)
Tuesday	kri2 (34)	asic (17)	conc (57)	cca (23) rs1 (9)
Wednesday	ai (10)			com (21)
Thursday		cmc (18)		fpls (6)

Table 5.6: GA-produced actual exam timetable in MSc 92/93

	09:30 - 11:00	11:30 - 13:00	14:00 - 15:30	16:00 - 17:30
Monday	c4sp (40) c4cg (61) swp (25) lp (30)	kri1 (47) gis (2)	os (12)	lisp (40) ral (8)
Tuesday	os2 (23)	fpla (5) cp1 (3)	cca (17) lo1 (30)	cvis (28)
Wednesday	kri2 (43) pl (6)	ds (46)		isc (36) ci (10)
Thursday	prol (51) hci (54)	rs2 (8)	si (4)	conc (29) tnl2 (3) ip2 (9)
Friday	ca (12) gr (11) nnet (17)		aied (23) cafr (6) lnl (1)	
Monday	mvis (24) ppc (11)	bsv (6) ct (5) smt (33)	es1 (34)	ppi (13) aps (9)
Tuesday	cc (17)	asic (34)	ip1 (9)	ml (6) mthr (22)
Wednesday	pa (33)	cmc (11)	las (25)	atnl (4) com (17) le2 (16)
Thursday	tnl1 (3) db (19)	cad (23) des (8)	cl1 (31)	inlp (13) sec (16)

Table 5.7: GA-produced actual exam timetable in MSc/UG 93/94

93/94 is shown in Table 5.7 with punishment=16 (consec_d=0, consec=1, three=0, four=0, large=2). In other words, there is 1 occurrence of two consecutive exams on the same half day. In addition, there are two occurrences of overcrowded time slots.

By the way, the numbers inside the brackets below each exam in Tables 5.4, 5.5, 5.6 and 5.7 give the number of students sitting that exam.

5.1.6.2 Comparisons

In this subsection, we outline the results of several experiments with the GA every ten runs using fitness-based (FIT), rank-based (RANK), spatial-oriented (SPAT) and

tournament-based (TOUR) selection schemes on the AI/CS MSc 90/91, 91/92, 92/93 and MSc/UG 93/94 exam timetabling problems. Steady-state reproduction was employed, in which the following occurred at each reproduction cycle: with probability P_c, two parents are selected, crossover applied, a temporary child produced, with probability P_m, mutation applied to the temporary child, and the resulting child inserted back into the population to replace the least fit member. The combination of operators used are one-point (1-pt), two-point (2-pt) and uniform crossover (UX) all with random mutation, and uniform crossover with smart mutation. All the crossovers guarantee to get allowable time slots because the system already excluded all the unallowable time slots in the initial generation. The random mutation we use is to choose a gene randomly and just change the value of the gene (time slot) to another value which is allowable for that gene. In other words, it mutates to a value which does not violated the constraints set in the exclusion file. The smart mutation operators involved are violation-directed mutation (VDM), stochastic violation-directed mutation (SVDM), event-freeing mutation (EFM) and stochastic event-freeing mutation (SEFM). All runs use initial crossover rate 0.8, decreasing gradually until 0.6, and initial mutation rate 0.003, increasing gradually until 0.02. The bias of rank-based selection is 1.5 and both the size of tournament based and length of spatial-oriented selections are 2. In addition, the population size is 50 and each generation produced one children to replace the worst parent and if more than 1000 evaluations cannot improve the best solution found so far, then we regard it converged and stop running. Otherwise, the maximum number of evaluations is 10000.

Tables 5.8, 5.9, 5.10 and 5.11 each contain four parts. The upper left part records the best/worst results out of ten trials run for a given configuration. For example, Table 5.8 shows that the best timetable found using spatial-oriented selection and stochastic violation-directed mutation (SVDM) had a penalty of 0, occurred in 5 of the ten trials with this configuration, and the poorest of these ten SPAT/SVDM trials resulted in a best penalty score of 40. Looking at the upper right part of the table, we can see that the fastest of these occurred in 2899 evaluations (numbers are only recorded in this part when a perfect timetable was found). In the lower left half of the table we can see that the standard deviation of the penalty of these ten SPAT/SVDM trials is 13.3. Finally, the lower right entry shows that the average penalty score from these

ten trials was 20.0, and the average evaluations to find the best in each trial was 4359 which may be either the perfect timetable was found or cannot improve solution in 1000 continuous evaluations, or the maximum evaluation (10000) is up.

Several observations can be made from these results. First, 93/94 problem is the most difficult one, 91/92 problem is a close second and 90/91 problem is the easiest one. Second, the benefit to timetables of using the GA is clear. In the 90/91 and 91/92 cases (where human-produced timetable was available for comparison), the average GA scores are better than the human-produced scores. In the 91/92 case, even the worst GA performance was better than the human one. The best GA scores reflect markedly better timetables than the human-produced ones.

Comments can also be made about the choice of operators and selection scheme. The best overall combination of those trials, in terms of number of perfect timetables found, seems to be rank-based selection coupled with SEFM, but it is clear that the choice of operator has far more of an effect on performance than the choice of selection scheme. By counting the number of trials in which a perfect timetable was found (over all problems), we can rank the operators as follows: SEFM(33), EFM(27), VDM(15), SVDM(14), 2-pt(4), UX(3) and 1-pt(0). This rough guide suggests SEFM as best, EFM next best, VDM and SVDM next best (but no significant difference between them), while the three standard operators are the worst. A similar rough ranking of the selection schemes gives: RANK(31), TOUR(24), SPAT(23) and FIT(18).

In terms of speed, tournament-based selection tends to find its optima more quickly, while fitness-based selection seemed slowest. The 'fastest' operator is clearly SEFM, which at least partially makes up for the extra complexity involved with this operator. Though VDM and SVDM are computationally cheaper than EFM and SEFM respectively, EFM and SEFM allow more significant speedup in terms of the number of function evaluations until convergence (or best timetable found). Further, SVDM seems to perform better than VDM and SEFM seems to perform better than EFM in terms of average solution quality and evaluations needed. For example, SVDM/TOUR is better than VDM/TOUR and SEFM/TOUR is better than EFM/TOUR in the 90/91 exam timetabling problem in terms of solution quality with a confidence larger than 80% and 85% respectively according to Student's t-test. Especially, SEFM can

90/91	BEST(times)/WORST				LEAST EVALS for perfect timetable			
	FIT	RANK	SPAT	TOUR	FIT	RANK	SPAT	TOUR
1-pt	10/124	10/129	6/155	10/157	-	-	-	-
2-pt	0(2)/60	0(1)/109	10/79	0(1)/50	4069	3616	-	4606
UX	10/107	10/134	0(2)/110	0(1)/80	-	-	4468	3932
VDM	0(1)/55	0(4)/140	0(2)/84	0(4)/81	6618	3310	3084	2841
SVDM	10/50	0(4)/50	0(5)/40	0(2)/50	-	3904	2899	2491
EFM	0(3)/20	0(4)/46	0(5)/39	0(4)/103	1244	1370	943	924
SEFM	0(3)/20	0(6)/40	0(4)/40	0(5)/40	1198	1500	729	1221
	STANDARD DEVIATION				AVERAGE/EVALS			
	FIT	RANK	SPAT	TOUR	FIT	RANK	SPAT	TOUR
1-pt	38.1	39.4	48.4	49.9	43.0/6412	50.5/5702	44.3/5716	67.7/4886
2-pt	23.6	35.9	22.3	15.9	31.2/6295	42.5/5754	41.3/5530	22.4/5835
UX	35.6	43.8	36.4	28.8	43.8/6181	45.1/5525	42.7/5860	34.5/4858
VDM	17.9	53.8	31.2	31.2	24.2/5530	39.9/5102	35.1/4831	25.2/4072
SVDM	13.2	16.4	13.3	14.5	18.0/5911	14.3/5291	10.0/4359	14.3/4489
EFM	8.3	17.2	12.6	34.5	10.5/2441	14.2/2442	8.9/2109	24.4/2064
SEFM	8.2	13.2	15.8	13.7	10.0/2361	8.0/2485	15.0/1887	11.0/1921

Table 5.8: Comparisons for GA-produced 90/91 exam timetable

91/92	BEST(times)/WORST				LEAST EVALS for perfect timetable			
	FIT	RANK	SPAT	TOUR	FIT	RANK	SPAT	TOUR
1-pt	13/239	39/253	43/119	62/203	-	-	-	-
2-pt	26/140	39/182	49/103	26/226	-	-	-	-
UX	26/124	16/160	36/187	29/154	-	-	-	-
VDM	52/146	16/161	9/149	29/210	-	-	-	-
SVDM	23/84	10/80	20/79	23/83	-	-	-	-
EFM	16/119	6/98	19/92	19/138	-	-	-	-
SEFM	10/63	0(2)/43	10/69	0(1)/66	-	2220	-	1175
	STANDARD DEVIATION				AVERAGE/EVALS			
	FIT	RANK	SPAT	TOUR	FIT	RANK	SPAT	TOUR
1-pt	69.7	78.4	27.0	41.9	104.0/7174	110.1/6961	72.9/6955	118.1/5228
2-pt	34.7	40.3	16.7	65.9	73.4/7816	85.6/6862	72.2/6924	100.3/5914
UX	28.8	46.0	53.6	36.7	57.8/7655	90.7/6316	113.9/5858	81.9/6230
VDM	29.0	45.8	43.7	49.8	79.0/7539	72.7/6122	67.0/6182	86.9/5290
SVDM	20.6	41.6	14.7	22.0	52.2/8023	41.6/6819	47.8/6893	42.3/6177
EFM	33.3	33.5	23.9	41.5	50.4/4364	46.2/3678	46.9/3626	65.6/4018
SEFM	18.4	16.7	18.8	20.4	29.7/3724	18.6/3429	34.7/2943	30.3/2852

Table 5.9: Comparisons for GA-produced 91/92 exam timetable

92/93	*BEST(times)/WORST*				*LEAST EVALS for perfect timetable*			
	FIT	RANK	SPAT	TOUR	FIT	RANK	SPAT	TOUR
1-pt	10/144	20/94	10/78	23/132	-	-	-	-
2-pt	20/80	13/97	3/102	19/92	-	-	-	-
UX	16/99	9/89	16/116	16/106	-	-	-	-
VDM	0(1)/57	0(2)/100	6/59	0(1)/82	4715	4281	-	4141
SVDM	0(1)/50	10/66	0(1)/64	0(1)/49	4683	-	4493	4434
EFM	0(4)/23	0(3)/39	0(1)/46	0(3)/82	1405	1372	1595	1166
SEFM	0(3)/69	0(5)/23	0(3)/36	0(1)/30	1304	2072	1480	1356
	STANDARD DEVIATION				*AVERAGE/EVALS*			
	FIT	RANK	SPAT	TOUR	FIT	RANK	SPAT	TOUR
1-pt	45.9	22.1	21.4	31.9	62.7/6104	45.3/6004	51.6/5326	53.8/5648
2-pt	20.3	28.2	32.6	22.7	40.0/6273	42.4/6316	46.5/5696	40.6/5165
UX	30.0	25.6	29.5	27.3	53.7/5806	41.6/6088	44.3/5663	45.6/4811
VDM	22.7	32.2	14.7	25.5	31.8/5531	37.1/5105	37.2/4455	34.6/3965
SVDM	17.2	21.4	17.6	15.0	25.9/5430	39.6/5061	23.6/4790	27.6/5051
EFM	9.5	15.7	13.2	25.7	8.8/2980	17.0/2805	18.5/2531	22.7/2919
SEFM	21.9	9.8	12.8	9.0	16.9/2721	7.6/2676	15.2/2335	13.2/2413

Table 5.10: Comparisons for GA-produced 92/93 exam timetable

93/94	*BEST(times)/WORST*				*LEAST EVALS for perfect timetable*			
	FIT	RANK	SPAT	TOUR	FIT	RANK	SPAT	TOUR
1-pt	25/118	25/279	29/182	42/217	-	-	-	-
2-pt	42/209	52/169	52/245	55/149	-	-	-	-
UX	35/122	26/139	35/145	58/225	-	-	-	-
VDM	22/65	22/78	29/85	29/139	-	-	-	-
SVDM	16/82	35/102	19/159	29/79	-	-	-	-
EFM	29/92	28/205	22/292	34/132	-	-	-	-
SEFM	13/56	16/63	13/86	13/126	-	-	-	-
	STANDARD DEVIATION				*AVERAGE/EVALS*			
	FIT	RANK	SPAT	TOUR	FIT	RANK	SPAT	TOUR
1-pt	27.2	85.9	54.0	54.3	68.5/8259	118.3/7548	80.4/6622	95.2/6684
2-pt	57.2	42.4	60.2	35.4	89.6/7831	96.4/6703	92.3/6388	97.4/6419
UX	28.5	30.5	39.0	54.0	65.2/7308	76.9/7045	76.5/6469	107.4/5468
VDM	14.7	17.0	19.8	28.4	45.6/6403	49.7/6368	57.9/7293	67.5/5444
SVDM	17.5	22.4	40.0	14.6	45.5/7108	55.8/5999	53.7/5557	46.2/5183
EFM	21.5	49.8	79.2	31.1	56.0/4139	67.8/4605	84.2/4858	71.1/4119
SEFM	11.9	14.4	23.7	32.4	32.7/3993	37.8/3860	36.9/3121	46.2/3666

Table 5.11: Comparisons for GA-produced 93/94 exam timetable

produce optimal solutions in three occurrences in 91/92 while all other methods cannot find even one instance of optimal solutions.

5.1.7 Discussion

Although the system can schedule timetables very well, many improvements and extensions are possible, for example:

- *Adding Classroom Considerations*

  Though there are large classrooms, more than two exams can be held at the same place and at the same time if the classroom is large enough and no students are taking these exams simultaneously. Rather than requiring the user to allocate rooms once the timetable has been generated, it should be possible to get the GA to handle room allocation at the same time as time-slot allocation.

- *Ordering Constraints*

  In many cases there is a partial ordering over modules. For example, module1 should be before module2, while module3 must take place after module4. In the real world, such situations often happen as follows: exam A prefers/requires be to be set before exam B because exam A involves more students which needs to be held early in order to let the teacher have enough time to mark it, or both the exams are taught by the same teacher and he is expected to appear on both exams timeslots. Currently we just use the preset or exclusion file to let the GA deal with such constraints, so we need a more general method to cope with such kinds of ordering constraints.

- *Different Length Exams*

  The Genetic Algorithms Timetabling system can be used for a very wide range of exam timetabling problems by simply appropriately changing the data files; however, the framework as described so far assumes that all of the exams are of the same length. In some cases, students might have a mixture of 90 minute exams and two-hour exams, for example. We can easily modify the system to allow for this by introducing the notion of *duration* for each module. For example, chromosomes will still be the same, but alleles will represent the start time of

each module rather than the time-slot. By using input data concerning each module's duration, checking for violation of an edge constraint simply amounts to checking if the two events involved overlap in time.

In the next section, we will extend the exam timetable system to deal with the more complicated lecture/tutorial timetabling problem.

5.2 GAs in Lecture/Tutorial Timetabling

The aim of this section is to extend the METP method to handle the more general *Modular Lecture/tutorial Timetabling Problem* - MLTP. The current GA-based exam timetabler cannot deal adequately with this, because it takes no account of different sets of rooms in which exams may occur. This is not critically important given the current size of the problem in the Department of Artificial Intelligence, because there is a very large room which can handle most groups of exams which the timetable may have occurring at the same time. This may change in future, though; also, lecture timetabling is critically sensitive to the available rooms and their capacities. Furthermore, lectures are not all the same length and there can be various ordering (precedence/preference) constraints which increase the complexity of the problem. Also, with exam timetabling, particularly if we are seeking to ensure that students have just one or two exams per day, time to travel from one exam to another is not very significant in the problem. In lecture/tutorial timetabling, students will have several lectures per day at least, and traveling becomes important. Besides, two lectures can never occupy the same room and the same time if they share the same students. The purpose of this section, then, will be to extend our previous exam timetabler to handle the more general problem in which events must be allocated a particular room, as well as a particular time. In addition, each event may occupy more than one timeslot and have various kinds of ordering and event-spread constraints.

When an MLTP is tackled, typically in a university or college department although very similar problems often occur in industry, it is usually addressed by hand (for example, by a course organiser). This involves producing an initial draft timetable,

followed by perhaps weeks of redrafting after coordinating with other department about the latest draft. The initial draft is often based on merging different departments' teaching/course timetables; but in large modular degree schemes, the fact that many students from one department typically take courses in others, and the fact that there are different lecture timetables for different terms, makes this a recipe for finding local minima, which then typically need extensive repair if better solutions are to be found (and hence better solutions are often not found).

5.2.1 Problem Description

The *MSc in Information Technology (IT): Knowledge Based Systems* in the department of Artificial Intelligence of Edinburgh University is a twelve month course and retrains graduates from other disciplines so that they can become active practitioners in the field of Knowledge Based Systems. The emphasis is on practical techniques for the design and construction of Knowledge Based Systems, enabling graduates from this course to apply their skills in a variety of settings. Students also have some opportunity to gain a working knowledge of conventional computer science, by taking appropriate modules.

The course in MSc in IT in EDAI is organised into four 'themes' namely: *Expert Systems (ES)*, *Foundation of Artificial Intelligence (AI)*, *Intelligent Robotics (IR)* and *Natural Language (NL)*. Each theme involves a particular combination of 8 modules, of which some are compulsory and some are optional. Details are given in Table 5.12.

5.2.2 The AI/CS MSc Lecture/Tutorial Timetabling Problem

The lecture/tutorial timetable must occupy a five-day week; each day has eight time slots, four in the morning and four in the afternoon.

Each time slot in this problem lasts one hour. The total number of time slots is 40 from Monday to Friday. Table 5.13 illustrates this, with the time slots numbered. The constraints involved in this MLTP are as follows:

1. Computer Science (CS) lectures should be in the morning, Artificial Intelligence (AI) lectures should be in the afternoon (a departmental agreement). Tutorials can be in either time period. (Time Exclusions)

	Modules	
	Compulsory	*Optional*
ES	Knowledge Representation and Inference 1 Knowledge Representation and Inference 2 Programming in Lisp Programming in Prolog Expert systems 1 Expert systems 2	AI and Education Connectionist Computing Database Systems Natural Language Processing Neural Networks
AI	Knowledge Representation and Inference 1 Knowledge Representation and Inference 2 Programming in Lisp Programming in Prolog	Computational Vision Connectionist Computing Mathematical reasoning Natural Language Processing Neural Networks
IR	Knowledge Representation and Inference 1 Machine Vision Programming in Lisp Intelligent Assembly Systems Intelligent Sensing and Control	Computational Vision Connectionist Computing Neural Networks Programming in Prolog Software for Parallel Computers
NL	Knowledge Representation and Inference 1 Knowledge Representation and Inference 2 Programming in Lisp Linguistics 1 Computational Linguistics 1 Computational Linguistics 2	Logic and Formal Semantics 1 Logic and Formal Semantics 2 AI and Education Linguistics 2

Table 5.12: MSc in IT 1992/1993

2. Lectures from 1pm to 2pm should be avoided if possible because it is the time when most people would prefer to have lunch. (Punishment for Noon)

3. A student should have at least 1/2 hour to get from a class or tutorial at KB, where all CS events happen, to the centre where all AI events happen and vice-versa. (Site/Room Considerations)

4. Each module may have a variable number (1-4) of lectures a week, which every student taking that module must attend. (Duration Consideration and Event-spread Constraints)

5. Each module should have 1 tutorial group for every N students (N=12 for example), and a student need attend only 1 instance of these per week. There may be multiple tutorials scheduled at the same time. (Tutorial Considerations)

6. A class can start on a 1/2 hour boundary and last for 0.5, 1.0, 1.5, 2.0 or 2.5

	9-10	10-11	11-12	12-13	13-14	14-15	15-16	16-17
Monday	0	1	2	3	4	5	6	7
Tuesday	8	9	10	11	12	13	14	15
Wednesday	16	17	18	19	20	21	22	23
Thursday	24	25	26	27	28	29	30	31
Friday	32	33	34	35	36	37	38	39

Table 5.13: Time slots allocation for lecture/tutorial

hours[6]. (Duration Considerations)

7. The room must be big enough to accommodate the maximum number of students wanting to attend the lecture. (Room Inclusions)

8. Staff will not be available at all times (that is IR lectures cannot be scheduled Tues 10:30-13:00). (Time exclusions)

9. Only one class in a room at a time (unlike in the METP case). This means that the scheduler will also need a list of rooms information. (Room Considerations / Hard Constraints)

10. Lecturers prefer/request the tutorial/laboratory (seminar) to be after the 1st lecture of the week. (Ordering Constraints)

5.2.3 Applying the Basic GA Framework to Lecture/Tutorial Timetabling

5.2.3.1 Representation

The representation we use is a chromosome of length $2e$ (e is the number of events to be scheduled); e of them specify time-slots for the e events, and e of them specify rooms for the e events. The interpretation of such a chromosome is that if the nth number in the list is t and the $(n+e)$th number in the list is r, then module n is scheduled to occur at time t and room r.

The difference between this representation and METP is that we introduce the concept of *non-standard duration*. In the METP, each exam's duration is a constant which happens to occupy a timeslot. So we don't need to worry about problems caused by

[6] In actual timetables, we only use remarks to specify which modules start on the half hours because the basic interval in our timetable is one hour.

varieties of durations. However, in the MLTP, we have events of varying durations, so we need to consider many problems caused by this. For example, long lectures cover more than one time slot, and we therefore need to assure that use the same room during these periods. Other considerations include that we cannot allow long lectures to overlap the end of a day. The main difference is that checking for edge constraints and event-spread constraints must take these durations into account, rather than considering only the timeslot of an event.

5.2.3.2 External Specifications File

Specification constraints for the MLTP are more varied and complicated than we have seen so far. An example of this file for AI/CS MSc 1992/1993 is listed in appendix B which contains several sections as follows:

- *Compulsory Sections*:
  - `Modules section`
  - `Rooms section`
- *Optional Section*:
  - `Lecture-exclusions section`
  - `Tutorial-exclusions section`
  - `Precedence section`
  - `Preference section`
  - `Simultaneous section`
  - `Different-day section`
  - `Preset section`

Users can easily add more optional sections in a similar way. We describe the details of each section and their corresponding constraints next.

5.2.3.3 Module Considerations

The *modules section* contains each module's 'private' information, that is, independent of any relationship with other modules. It contains the size of the lecture, the preferred size of any tutorials or seminars associated with this module (which can then be used to work out the number of tutorials or seminars necessary), the duration of lectures, tutorials and seminars associated with this module, and any room or time slot specifications or exclusions for this module. Example lines from a module section are as follows:

```
...
aied_2 ## AI and Education
       duration 120
       room SB-F10
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
...
@aied-sem ## AI and Education-Seminars
          size 23
          unit 10
          duration 120
          room SB-A10
          exclusion 19
...
@es1-tut  ## Expert Systems 1-tutorials
          size 30
          room SB-A10 SB-C10 SB-C11 SB-C5
          exclusion 19
...
```

In the above example, the durations of aied_2 and all[7] the seminars of aied are 120 minutes whereas the duration of all the es_1 tutorials are 60 minutes (default value). The size of lecture aied is 23 and each seminar is of size 10, so we need three occurrences of aied seminars each week. The size of lecture es1 is 30 and its tutorials are of size 12 (default value), therefore, we also need three es1 tutorials per week. aied_2 can only use South Bridge room F10, and all the seminars of aied must also use South Bridge room A10. On the other hand, all the tutorials of es1 can use either room A10, C10, C11 or C5 in South Bridge. Finally, aied_2 cannot be put in mornings (slots 0-3, 8-11, etc...), or Wednesday 12:00-13:00 (slot 19). However, all seminars of aied and all tutorials of es1 can appear at any time except slot 19. Such slot exclusions deal with

[7] Modules prefixed by a @ represent multiple instances of an event (for example, tutorial, seminar or laboratory) and the size of each and the number of instances can be deduced from the size of its corresponding lecture.

constraints 1 and 8, as given in section 5.2.2.

- Computer Science (CS) lectures should be in the morning, Artificial Intelligence (AI) lectures should be in the afternoon (a departmental agreement). Tutorials can be in either time period.
- Staff will not be available at all times (that is IR lectures cannot be scheduled Tues 10:30-13:00).

When we give the durations for lectures, we just assume that any lecture consumes one or more complete one-hour slots. Most lectures are just 60 minutes, so this should not cause much trouble. It is possible to generate illegal chromosomes for some lectures whose duration exceeds 60 minutes. For example, a long lecture which may overlap the end of a day or overlap with other excluded timeslots. One way to legalise a chromosome to ensure that a long lecture does not overlap the end of a day is shifting all long lectures that overlap the end of a day back enough to avoid the overlap, or perhaps forward to the start of the next day. For each shift in turn, we must choose the best of these two. This does not necessarily find the best combination of shifts, since it is not necessarily the case that the best can be found by such local hill-climbing. Therefore, we use another much simpler way: when we first read each module and its corresponding duration, we calculate its unallowable start times according to its duration and the excluded slots given. These start times become excluded for this module. As a result, initialisation guarantees exclusion of the unallowable timeslots, and so do crossover and mutation. For example, in Table 5.14, module A's duration exceeds 60 minutes, so we automatically exclude the last slots (the upper case 'X') in each day for module A. If duration exceeds 120 minutes, then we must also exclude the penultimate slots (the lower case 'x') in each day, and so on. In other words, module A cannot start at the last few hours each day depending on its duration, so it is guaranteed not to overlap the end of a day. Another automatic exclusion example is in Table 5.15, which just tells us that module A is excluded from timeslot 19. This is no problem if module A's duration is at most 60 minutes. However, if module A's duration exceeds 60 minutes, then we also cannot allow it start at timeslot 18 (the one with upper case 'X'), because it will then clash with the pre-excluded timeslot 19 in the second half of its duration. Furthermore, if module A's duration exceeds 120 minutes, then timeslot 17 (the one with lower case 'x') also needs to be excluded and so on. In other words, module A must exclude its pre-excluded and some earlier timeslots,

	9-10	10-11	11-12	12-13	13-14	14-15	15-16	16-17
Monday							x	X
Tuesday							x	X
Wednesday							x	X
Thursday							x	X
Friday							x	X

Table 5.14: Exclude time slots which overlap the end of a day for long events

	9-10	10-11	11-12	12-13	13-14	14-15	15-16	16-17
Monday								
Tuesday								
Wednesday		x	X	E(19)				
Thursday								
Friday								

Table 5.15: Exclude time slots which overlap pre-excluded slots for long events

depending on its duration.

Using the exclusions information, the initial population of chromosomes is forced to have only valid alleles for each gene. What we need to do is to read the exclusion information and record the allowable time slots for each event, for later use, which do not violate these constraints. Then, randomly generate instances from the set of allowable time slots each time. In other words, the first pass finds the allowable time slots for each event and the second pass only chooses among these for each instance.

5.2.3.4 Room Considerations

The way we produce allowable rooms more intelligently and efficiently are similar to exclusion time slots. The first pass needs to record the allowable rooms for each event when it first reads the room inclusion information. The second pass just randomly generates instances from the set of allowable rooms for later initialisation and mu tation. Crossover guarantee the allowable room slots because it just recombines two parents with allowable room slots and certainly produces two children with allowable room slots. However, mutation also needs the second pass to guarantee each gene not violating the room slots set in the inclusion information.

Another compulsory section, the *rooms section*, contains the abbreviations and full

names of all the rooms and their location codes which is used to calculate the penalties between two consecutive modules at different sites.

An example of the location code is as follows:

- **1** – Centre of Edinburgh (C)
- **2** – King's Building (KB)

That is, in the University of Edinburgh, the Artificial Intelligence department and Cognitive Science Centre are located in the Edinburgh city centre whereas Computer Science and Meteorology departments are located in King's Building which is about 2 miles away from the city centre. Therefore, we must use different codes to tell the system that the distance between them is non-trivial.

Example lines from a rooms section are as follows:

```
SB-A10    :  1   ## south bridge room A10
...
AT-LT3    :  1   ## appleton tower lecture theatre 2
...
CSSR      :  1   ## Cog. Sci. Seminar Room at 1 Buccleuch Place
...
4310      :  2   ## JCML room 4310
...
```

Rooms SB-A10, AT-LT3 and CSSR use the same code because they belong to the same area, whereas room 4310 is located in King's Building. We can prevent modules at different sites from being consecutive if necessary by giving some punishments to modules which are consecutive but at different sites. It turns out that this problem rarely occurs because all the CS lectures are in the morning and AI lectures are in the afternoon and avoid 13:00-14:00. This can deal with the constraints:

- A student should have at least 1/2 hour to get from a class or tutorial at KB, where all CS events happen, to the centre where all AI events happen and vice-versa.

Tutorials differ from lectures in a important way. Unlike lectures, students are not assigned to tutorials until after the timetable has been formed and is being used. A reasonable possibility to solve the problem is not to care about tutorials at all and to schedule them only when the lecture timetable has been decided. It is unlikely that

a student from AI department will take more than 1-2 CS modules, so if there are several tutorials scheduled, then it is likely that one will be available. Of course, this may not be so in other universities. It is also typical to have to move tutorials at the beginning of the term because one cannot tell which tutorials will be attended by how many students. So some may not have any attendees.

The system will also penalise the situation when two or more module happen at the same time and share the same room, and of course long lectures must occupy the same room consecutively. From above, the system can express the constraints such as:

- No more students should be in a class than is possible to fit into a room.
- Only one class in a room at a time (unlike in the METP case). This means that the scheduler will also need a list of rooms information.

5.2.3.5 Preset Time and Room

In the lecture/tutorial timetable, we can pre-set either the time slot or both of the time and room slots. Presetting time/room slots is necessary in several situations:

1. Some events must only be scheduled in one time slot, because they are shared with some other students whose timetables were determined in advance.
2. Some lecturers want their event to be scheduled at a specific time or room slot.
3. It can be used to adjust an existing timetable by fixing most of the time slots, that is, re-timetabling.
4. It can be used to test the quality of a existing timetable is, for example, test a former actual timetable, by fixing all of the time or room slots.

An example of the *preset time/room section* is as follows:

```
es1_1 7
lisp_1 22 AT-LT2
...
```

For example, es1_1 is preset at time slot 7. In addition, lisp_1 is preset at time slot 22 and room AT-LT2. The system will penalise and report the conflicts between the

preset time/room information and exclusion/inclusion time/room information.

5.2.4 Specific Framework for Lecture/Tutorial Timetabling

5.2.4.1 Edge Constraints

The *lecture-exclusions section* contains the compulsory and recommended modules for each theme which is the main constraint information for lecture/tutorial timetable. The system can guarantee that compulsory and recommended modules will not clash with each other in order to let students to choose freely from these modules. Unlike METP case, at the time we schedule the lectures and tutorials timetable, we still don't know *exactly* what modules each student will take. We just can predict what modules it is *possible* to take according to the theme (lecture-exclusions) information. Therefore, this is the data for us to test the quality of the timetable.

Example lines of lecture-exclusions section are as follows:

```
lisp_1 lisp_2 kri1_1 kri1_2 es1_1 es1_2 aied_1 aied_2
lisp_1 lisp_2 kri1_1 kri1_2
lisp_1 lisp_2 kri1_1 kri1_2 ias_1 ias_2 mvis_1 mvis_2 ...
lisp_1 lisp_2 kri1_1 kri1_2 cl1_1 cl1_2 ln1_1 ln1_2 lfs1_1 lfs1_2 ...
```

For example, lisp_1, lisp_2 kri1_1, kri1_2, es1_1, es1_2, aied_1 and aied_2 are the most likely modules for students of the first group. In additions, lisp and kri1 happen in all the four groups; this means that they are more important than others when considering the edge-constraints.

Another important constraint is that each module should have 1 tutorial group for every N students (N=12 for example), and a student need attend only 1 instance of these per week. Therefore, there may be multiple tutorials scheduled at the same time. This constraint is handled by using the information in a *tutorial-exclusions section.*

The tutorial-exclusions section contains all the tutorials, laboratories and seminars. Users don't need to specify the number of these items because it's all deducible from information about the size of the lecture and its corresponding tutorial, seminar or

laboratory. Since such modules have multiple occurrences, the system only needs to guarantee that each tutorial (the one to the left of colon) will not clash with its own lectures and the compulsory lectures in its theme (those to the right of colon). In order to produce a valid timetable, all the tutorials, laboratories and seminars must not clash with their corresponding lectures or with any compulsory lectures in the same theme. Therefore, it is also a kind of edge-constraint.

An example of tutorial-exclusions section is as follows:

```
@aied-sem @aied-lab : aied_1 lisp_1 lisp_2 kri1_1 kri1_2
@es1-tut :  es1_1 es1_2 lisp_1 lisp_2 kri1_1 kri1_2
@ias-lab :  ias_1 ias_2 mvis_1 mvis_2 lisp_1 lisp_2 kri1_1 kri1_2
...
```

The tutorial, seminar or laboratory begins with a @ to tell the system it represents all the corresponding tutorials, seminars or laboratories. For example, the seminars and laboratories of AI and Education cannot clash with aied_1, lisp and kri1 and the tutorials of Expert Systems 1 cannot clash with all the es1, lisp, kri1 lectures.

There is no need to guarantee that the tutorials of the same lecture will not clash with each other because there are several identical tutorials and each student just needs to attend one of them. It is still a valid timetable even though some tutorials will clash. Users can also use different day information, if they want to, in the different day section described later to ensure that tutorials for the same course are not scheduled on the same day in order to scatter them about and let students have more different tutorial time slots to choose from.

5.2.4.2 Ordering Constraints

An example of a hard ordering constraint is that lecturers request laboratories/seminars to take place after the 1st lecture of the week. This is a kind of ordering constraints which, in our framework, is handled by the information in the *precedence section*.

An example of a precedence section is as follows:

```
aied_1 : @aied-sem @aied-lab
ias_1  : @ias-lab
```

The precedence section contains all the precedence information, for example, aied_1 must be before all its seminars and laboratories and ias_1 must be before all its laboratories. In each row of this section, the module to the left of the colon will begin before those to the right of colon. In addition, it is a kind of hard constraint, therefore we need to use a large punishment to prevent, or rather discourage, violations.

Sometimes, it is not vital to ensure some module must be before some other modules, so we can treat it as a kind of soft ordering constraint. For example, lecturers prefer rather than request the tutorial to be after the 1st lecture of the week. In such a situation, the system will do its best to satisfy this constraint. This is also a kind of ordering constraints which, in our framework, is handled by the information in the *preference section.*

The format of this section is similar to that of the precedence section. Example lines of a precedence section are as follows:

```
es1_1  : @es1-tut
kri1_1 : @kri1-tut
...
```

The preference section contains all the preference information, for example, es_1 and kri1_1 prefers to be scheduled before all their tutorials. In addition, it is a kind of soft constraint, therefore we just need to use a small punishment because it can be violated if it is unavoidable.

Another example of ordering constraints is to let all the specified modules happened at the same time slot. The *simultaneous section* contains the data which should happen at the same time because they will not share the same students or other reasons. For example, we can let aied_2 and aied-sem_1 happen at the same time because the former just happens in the 1st, 2nd and 9th week whereas the latter happens within 3th-8th week. In other words, they guarantee not to be held at the same week, so we can re-

gard them as a single one module and schedule them at the same time slot in order to reduce the size of the total modules. An example of a simultaneous section is as follows:

```
aied_2 aied-sem_1
```

The system just assigns the timeslot of the first module to the rest of the modules in the same line. In addition, the system will penalise and report the conflicts between the simultaneous information and exclusion/inclusion time/room information just like in the preset time/room case.

5.2.4.3 Event-spread Constraints

Because each lecture may have multiple occurrences each week and each student may have to attend all these lectures, we want the lecture to be held on different days to let students have enough time to prepare or digest the course. Another instance might be that certain laboratory sessions cannot be scheduled on the same day in order to allow time to clean up and prepare elaborate equipment for the next class. Therefore, we need a different constraint to spread them apart. This is a kind of event-spread constraints described in the previous chapter and is included in the *different day section*.

An example of a different-day section is as follows:

```
aied_1 aied_2
@ias_lab
...
```

From above, we know AI and Education's two lectures are held on different days and all the Intelligent Assembly System's laboratories are held on different days too.

Other event-spread constraints are not to let a student take too many modules per day. We solve this problem using various weights of penalties which is similar to the exam timetable. Because the total number of time slots is fixed - 40 for five days, a student is often obliged to take two or three lectures per day. Therefore, we don't need to punish such cases. However, we still don't want a student to take too many lectures per day, so we use various weights to punish violations of such event-spread constraints.

5.2.5 Evaluations

In exam timetabling problems, we already know the information about the students and the exams each student must take. Therefore, we can use the student/exam data to calculate the violations of edge constraints. However in lecture/tutorial timetabling problems, we must schedule the timetable before the term. In other words, at the time we scheduling the lecture/tutorial timetable, we still don't know the number of the students and the modules each student will take though we can over-estimate how many students take any course. Our approach is just to use the group themes information to classify the lecture/tutorial groups for the students and use that information to calculate the violations of edge constraints and other data to calculate other kinds of violations, for example, ordering or event-spread constraints.

The evaluation function is similar to METP but has more conditions to check than for the exam timetable case. Furthermore, each module may have different durations which complicates the evaluation function.

In general, the fitness function for an MLTP, where p is a chromosome, $\{c_1,...,c_n\}$ is a set of functions which each record the number of instances of a particular 'offence', and $\{w_1, ...,w_n\}$ are the weights attached to these offences, is:

$$f(p) = 1/(1 + \sum_{i=1}^{n} w_i c_i(p)) \tag{5.2}$$

In the MLTP we experimented with, the components of the evaluation function are functions which respectively count the number of instances of the following 'offences':

- **Hard constraints:**
  - clash: as describe in lecture and tutorial exclusions section
  - large: two modules at the same time use the same room
  - precedence: check if violates precedence constraint or not
  - differentday: check if violates different day constraint or not
- **Soft constraints:**
  - two: two modules per day

- three: three modules per day
- four: four modules per day
- five: five modules per day
- six: six modules per day
- seven: seven modules per day
- slot: check if any conflict or not
- larger than seven: more than seven modules per day
- far: two consecutive modules at different sites
- noon: between 13:00 and 14:00
- preference: check if violates preference constraint or not

The penalties for each constraint are recorded in the following file:

```
#define PEN_CLASH                500
#define PEN_DIFFERENTDAY         500
#define PEN_LARGE                300
#define PEN_PRECEDENCE           300
#define PEN_EIGHT+               200
#define PEN_SEVEN                100
#define PEN_PREFERENCE            30
#define PEN_NOON                  30
#define PEN_SIX                   20
#define PEN_SLOT                  10
#define PEN_FIVE                   5
#define PEN_FOUR                   1
#define PEN_FAR                    1
#define PEN_THREE                  0
#define PEN_TWO                    0
```

The 'large' type of penalty, could be used to deal with the following constraint.

- Only one class in a room at a time (unlike in the METP case). This means that the scheduler will also need a list of rooms information.

The system will also know how many occurrences happen at the same time and use the same room and then use the 'large' penalty to punish them in order to get rid of this situation.

In the next subsection, we present a clear comparison between the actual timetables produced by human experts and the timetables produced by the system.

YEAR-TERM	LECT	TUT	SEM	LAB	TOTAL
1992/1993-1	48	21	3	4	76
1992/1993-2	46	24	0	3	73
1993/1994-1	51	31	0	0	82
1993/1994-2	50	22	1	0	73

Table 5.16: Lecture/tutorial problem information

5.2.6 Experiments

5.2.6.1 Actual Timetable

Table 5.16 lists informations pertaining to the EDAI AI/CS MSc 92/93 and 93/94 lecture timetabling problems. Both years involve two terms, so there are altogether 4 problems. The YEAR-TERM column represents the term number of a one year course beginning in October of the year before the slash, and finishing in September of the year after the slash. LECT, TUT, SEM and LAB represent the number of lectures, tutorials, seminars and laboratories held per week in that particular term of that particular year.

The actual timetables used in these four problems were produced by course organisers; we summarise them in Table 5.17. Each one was the result of a first draft followed by repeated iterative feedback from lecturers of several different departments.

For convenience, we will discuss the 'quality' of lecture timetables by using a vector $V = \{f, d_4, d_5, e, l, s_d, s_i, c\}$. The components of V respectively denote violations of far distance constraints; cases where a 'virtual student' faced four events in a single day; cases where a 'virtual student' faced five events in a single day; slot-exclusion constraints; lunchtime constraints; *desirable* event-order constraints (preference); *important* event-order constraints (precedence); and finally, clash (edge) constraints. Violations of all other constraints mentioned above (for example: room exclusions, capacity constraints and cases where a 'virtual student' faced more than five events in a single day) are not recorded, since they were fully satisfied in all cases.

- *Actual Timetable 92/93-1:* The actual human-produced timetable for AI/CS MSc 92/93 first term is shown in Table 5.18 with V = {65,2,0,14,4,4,0,0}, total punishment = 447. In other words, there are 65 violations of far distance con-

YEAR/TERM	f	d_4	d_5	e	l	s_d	s_i	c	TOTAL
unit penalty	(1)	(1)	(5)	(10)	(30)	(30)	(300)	(500)	N/A
1992/1993-1	65	2	0	14	4	4	0	0	447
1992/1993-2	49	3	1	5	2	9	1	0	737
1993/1994-1	84	2	0	7	5	0	0	0	306
1993/1994-2	50	0	0	1	3	3	0	0	240

Table 5.17: Actual lecture/tutorial timetable penalties

straints, 2 cases where a 'virtual student' faced four events in a a single day, 14 violations of slot-exclusion constraints, 4 violations of lunchtime and preference constraints respectively.

- *Actual Timetable 92/93-2:* The actual human-produced timetable for AI/CS MSc 92/93 second term is shown in Table 5.19 with V = {49,3,1,5,2,9,1,0}, total punishment = 737. In other words, there are 49 violations of far distance constraints, 3 cases where a 'virtual student' faced four events, and another 1 case where a 'virtual student' faced five events in a single day. Further, 5 violations of slot-exclusion constraints, 2 violations of lunchtime, 9 violations of preference constraints and finally 1 violation of precedence constraint.

- *Actual Timetable 93/94-1:* The actual human-produced timetable for AI/CS MSc 93/94 first term is shown in Table 5.20 with V = {84,2,0,7,5,0,0,0}, total punishment = 306. In other words, there are 84 violations of far distance constraints, 2 cases where a 'virtual student' faced four events in a single day. In addition, 7 violations of slot-exclusion constraints, and 5 violations of lunchtime.

- *Actual Timetable 93/94-2:* The actual human-produced timetable for AI/CS MSc 93/94 second term is shown in Table 5.21 with V = {50,0,0,1,3,3,0,0}, total punishment = 240. In other words, there are 50 violations of far distance constraints, 1 violation of slot-exclusion constraints, 3 violations of lunchtime and preference constraints respectively.

The item inside the brackets below each event in Table 5.18, 5.19, 5.20 and 5.21 is the name of the room allocation for that event.

	0900 - 1000	1000 - 1100	1100 - 1200	1200 - 1300	1300 - 1400	1400 - 1500	1500 - 1600	1600 - 1700
Mon	cl1 (DHT7.01)	cl1 (DHT7.01) mvis (SB-F10) os (TH-B)	asic (3317) ln1 (DHT7.01)	ln1 (DHT7.01) cc (3317) os-tut (3315)		aied (SB-F10) si (4310) lisp-tut (SB-A10) lisp-tut (SB-C11)	ias (SB-F10) fpls (6224) kri1-tut (SB-A10) kri1-tut (SB-C5)	kri1 (AT-LT2)
Tues	lisp (AT-LT3) aps (3315) rs1 (8216)	cca (5327) ias-lab (IAS-LAB)	cp1 (DHT7.01)	cp1 (DHT7.01) ci (TH-A)	aied-sem (SB-A10)	aied-sem (SB-A10) si (4310) lfs1 (DHT7.01)	lfs1 (DHT7.01) asc (6206) aied-lab (SB-C1) ias-lab (IAS-LAB) mvis-tut (FH-B9)	es1 (AT-LT2)
Wed	mvis (SB-F10) com (6324)	cmc (3317) aied-sem (SB-A10) aps-tut (3315)	aied-sem (SB-A10) asic (3317) es1-tut (SB-C10)	scsal (SB-F10) alco (TH-A) ip1 (8216)				kri1 (AT-LT2)
Thur	ci (TH-C) ln1 (DHT7.01)	ln1 (DHT7.01) os (TH-B) mvis-tut (FH-B9)	lisp (AT-LT3) aps (3218)	cc (3317) cp1 (DHT7.01) lisp-tut (AT-2C) mvis-tut (FH-B9) ci-tut (3315)	cp1 (DHT7.01)	ias (SB-F10) lfs1 (DHT7.01) kri1-tut (SB-A10)	lfs1 (DHT7.01)	asc (6206)
Fri		es1 (AT-LT3) lisp-tut (SB-A10) lisp-tut (SB-C5) aps-tut (3315)	cmc (3317) cl1 (DHT7.01) es1-tut (SB-A10) es1-tut (SB-C5)	cl1 (DHT7.01) fpls (6224) rs1 (8216)	aied (SB-F10) aied-sem (SB-F10)	aied (SB-F10) aied-sem (SB-F10) cca (3218) ip1 (8216) aied-lab (SB-C1) kri1-tut (SB-A10)	com (6324) kri1-tut (SB-C5) kri1-tut (SB-C10)	

Table 5.18: Human-produced actual lecture/tutorial timetable in MSc 92/93-1

	0900 - 1000	1000 - 1100	1100 - 1200	1200 - 1300	1300 - 1400	1400 - 1500	1500 - 1600	1600 - 1700
Mon	db (TH-C) ln2 (CSSR)	ln2 (CSSR) bav (3315) nlp-tut (SB-C11)	gr (TH-A) db-tut (3218)	rs2 (8216)	nnet (FR-S)	cvis (AT-Rm8) conc-tut (SB-C11)	conc (AT-LT5) cafr (6311)	es2 (AT-LT2) conc-tut (SB-C11)
Tues	cl2 (Rm-17)	cl2 (Rm-17) ppc (3218) isc-lab (ISC-LAB) kri2-tut (SB-C5) kri2-tut (SB-C10)	swp (6203) es2-tut (SB-C11)	ds (3317) prol-tut (SB-C10) prol-tut (SB-C11)		isc (AT-LT5) lfs2 (CSSR) mthr-tut (SB-C10)	lfs2 (CSSR) nlp (AT-LT5)	es2 (AT-LT2) cvis-tut (SB-C10)
Wed	nnet-tut (CSSR)	cafr (3315) kri2-tut (SB-C10) kri2-tut (SB-C11)	mthr (SB-F10) ds (3317) isc-lab (ISC-LAB)	al-sw (SB-F10) cad (3317)		prol (AT-LT2)	conc-tut (SB-C11)	kri2 (AT-LT2) conc-tut (SB-C10)
Thur	swp (6203) ln2 (CSSR)	ln2 (CSSR) db (TH-C)	ds (3317) mthr-tut (SB-C11) nnet-tut (CSSR)	ip2 (8216) isc-lab (ISC-LAB) nlp-tut (SB-C11)	nnet (FR-N)	nlp (AT-LT5) lfs2 (CSSR) cvis-tut (SB-A10)	lfs2 (CSSR) conc (AT-LT5)	prol (AT-LT2)
Fri	gr (3218)	bav (3315) prol-tut (SB-C5) prol-tut (SB-C10)	ppc (3218) cl2 (Rm-14) es2-tut (SB-C5) es2-tut (SB-C10)	cl2 (Rm-14) mthr (SB-F10) cad (3317) rs2 (8216)		isc (AT-LT4) ip2 (8216)	cvis (AT-LT4)	kri2 (AT-LT2)

Table 5.19: Human-produced actual lecture/tutorial timetable in MSc 92/93-2

	0900 - 1000	1000 - 1100	1100 - 1200	1200 - 1300	1300 - 1400	1400 - 1500	1500 - 1600	1600 - 1700
Mon	ad (3317) ln1 (CSSR)	ln1 (CSSR) os (LT-B)	cc (3317) cl1 (AFBG17)	asc (3317)	aied (SB-F10)	aied (SB-F10) lfm1 (CSSR)	kri1 (AT5)	es1 (AT4)
Tues	aps (3317) cp1 (CSSR) rs1 (8216)	cp1 (CSSR) cca (6301) aied-tut (SB-C5) kri1-tut (SB-A10)	ppc (3218) es1-tut (SB-C5) kri1-tut (SB-A10)	ci (LT-A) es1-tut (SB-C5) kri1-tut (SB-A10)	inlp (SB-F10)	ias (SB-F10) aps-tut (3315)	mv (AT3) bsv (3315)	tnlp1 (SB-A10)
Wed		cmc (3317) aied-tut (SB-C5) inlp-tut (SB-A10)	ad (3317) es1-tut (SB-A10) ias-tut (SB-C5) tnlp1-tut (SB-A18)	alca (3218) ip1 (8216) es1-tut (SB-C5) inlp-tut (SB-A10)		prolog (AT4)	ind-sem (AT4)	ind-sem (AT4)
Thur	ci (LT-A) ln1 (CSSR) gis (DS107)	ln1 (CSSR) gis (DS107) os (LT-B) inlp-tut (SB-A10) prolog-tut (SB-C5)	cc (3317) cl1 (AFBG17) kri1-tut (SB-A10) prolog-tut (SB-C5)	aps (6206) prolog-tut (SB-C5) tnlp1-tut (SB-A10) ci-tut (3315)	kri1 (AT5) ppc (5326)	mv (AT3) lfm1 (CSSR) aps-tut (3315) gis-tut (DS107)	ias (SB-F10) bsv (3315)	tnlp1 (SB-A10)
Fri	asc (3317) cp1 (CSSR)	cp1 (CSSR) cca (5326) mv-tut (FH-B9) prolog-tut (SB-A10)	inlp (SB-F10) cmc (3317) ias-tut (SB-C5) kri1-tut (SB-A10)	rs1 (8216) mv-tut (FH-B9) prolog-tut (SB-A10) tnlp1-tut (SB-A18) os-tut (3315)	aied (SB-F10)	aied (SB-F10) ip1 (8216)	es1 (AT4)	prolog (AT4)

Table 5.20: Human-produced actual lecture/tutorial timetable in MSc 93/94-1

	0900 - 1000	1000 - 1100	1100 - 1200	1200 - 1300	1300 - 1400	1400 - 1500	1500 - 1600	1600 - 1700
Mon	db (3218) ln2b (CSSR)	ln2b (CSSR) gr (LT-B) cl2 (CSSR)	cl2 (CSSR) si (6206) rs2 (8216) db-tut (3218)	rs2 (8216) fpls (5325)	nnet (DHT-S)	cv (SB-F10) lfm2 (CSSR)	lfm2 (CSSR) mr (SB-F10) tnlp2 (SB-A10)	ccon (AT4)
Tues	swp (3317)	ccon-tut (SB-C5) tnlp2-tut (SB-A10)	com (LT-B) mr-tut (SB-C5)	ds (3317)	kri2 (AT4)	isc (SB-F10)	mr (SB-F10)	atnlp (SB-F10) es2 (AT4)
Wed	ln2a (CSSR)	ln2a (CSSR) cafr (3315) cv-tut (SB-C5) kri2-tut (SB-A10)	ccon-tut (SB-C5) kri2-tut (SB-A10)	cad (3317) ccon-tut (SB-C5) kri2-tut (SB-A10)		ind-sem (AT4)	ind-sem (AT4)	kri2 (AT2)
Thur	fpls (3317) ln2b (CSSR)	ln2b (CSSR) db (3218) cl2 (CSSR) lisp-tut (SB-A10)	cl2 (CSSR) ds (3317) ip2 (8216) cv-tut (SB-A18) es2-tut (SB-C5) lisp-tut (SB-A10)	ip2 (8216) gr (6206) atnlp-tut (SB-C5) lisp-tut (SB-A10)	nnet (DHT-N)	isc (SB-F10) si (6206) lfm2 (CSSR)	lfm2 (CSSR) cv (SB-F10) tnlp2 (SB-A10)	lisp (AT2)
Fri	swp (3317) ln2a (CSSR)	ln2a (CSSR) cafr (3315) lisp-tut (SB-A10) mr-tut (SB-C5)	com (LT-A) kri2-tut (SB-C5) lisp-tut (SB-A10)	cad (3317) ip2 (8216) es2-tut (SB-A10) kri2-tut (SB-C5)		lisp (AT4) rs2 (8216)	ccon (AT4)	atnlp (SB-F10) es2 (AT4)

Table 5.21: Human-produced actual lecture/tutorial timetable in MSc 93/94-2

5.2.6.2 Comparisons

Experiments were performed to assess the performance of our GA-based timetabling framework on the four EDAI AI/CS MSc 92/93 and 93/94 lecture timetabling problem. Fifteen experiments were run, involving all combinations of three different selection schemes and five different mutation operators. All runs use uniform crossover starting at the rate 0.8, decreasing adaptively until 0.6, and mutation starting at the rate 0.003, increasing adaptively until 0.02. The mutation operators we used are gene mutation (genem), violation-directed mutation (VDM), stochastic violation-directed mutation (SVDM), event-freeing mutation (EFM) and stochastic event-freeing mutation (SEFM). The population size is 50, the bias of rank-based selection is 2.0 and both the size of tournament-based and length of spatial-oriented selections are 5. Steady-state reproduction was employed and a trial ended when no improvement as the fittest chromosome was recorded in 1,000 evaluations, or when 10,000 evaluations had been made.

Tables 5.22, 5.23, 5.24 and 5.25 each contain four parts. The upper left part records the best/worst results out of ten trials run for a given configuration. For example, Table 5.22 shows that the best timetable found using rank-based selection and stochastic event-freeing mutation (SEFM) had a penalty of 0, and occurred in 4 of the ten trials with this configuration, and the poorest of these ten RANK/SEFM trials resulted in a best penalty score of 91. Looking at the upper right part of the table, we can see that the fastest of these occurred in 1855 evaluation (numbers are only recorded in this part when a perfect timetable was found). In the lower left half of the table we can see that the standard deviation of the penalty of these ten RANK/SEFM trials is 30.6. Finally, the lower right entry shows that the average penalty score from these ten trials was 28.7, and the average evaluations to find the best in each trial was 3210.

Several observations can be made from these results. Most of the worst solutions produced by the GA are better than the human-produced actual timetables. It is clear that the results here roughly tally with those found in the exam timetabling experiments in section 5.1.6. By counting the number of trials in which a perfect timetable was found (over all problems), we can rank the operators as follows: SEFM(38), SVDM(30), VDM(23), EFM(21), genem(2). In general, the four smart mutation operators can

92/93-1	BEST(times)/WORST			LEAST EVALS for perfect timetable		
	RANK	SPAT	TOUR	RANK	SPAT	TOUR
genem	1/213	1/484	1/692	-	-	-
VDM	0(1)/97	0(1)/94	32/125	5260	2663	-
SVDM	0(3)/93	1/92	0(1)/90	2982	-	2185
EFM	1/92	0(1)/66	1/365	-	5370	-
SEFM	0(4)/91	0(1)/42	0(2)/62	1855	1258	1130
	STANDARD DEVIATION			AVERAGE/EVALS		
	RANK	SPAT	TOUR	RANK	SPAT	TOUR
genem	76.6	168.7	214.5	79.7/6447	167.5/5646	200.3/5735
VDM	38.9	31.8	31.9	45.8/6091	42.7/6283	63.0/7170
SVDM	32.5	32.3	25.6	22.2/4486	49.0/4358	43.5/4236
EFM	28.6	22.7	105.9	44.2/5934	36.7/6236	83.2/6316
SEFM	30.6	14.7	20.0	28.7/3210	26.8/2946	20.9/2667

Table 5.22: Comparisons for GA-produced 92/93-1 lecture/tutorial timetable

92/93-2	BEST(times)/WORST			LEAST EVALS for perfect timetable		
	RANK	SPAT	TOUR	RANK	SPAT	TOUR
genem	0(1)/152	2/151	1/155	5260	-	-
VDM	0(7)/95	0(5)/96	0(1)/66	3738	3916	4317
SVDM	0(8)/31	0(5)/91	0(5)/60	2321	1925	1993
EFM	0(7)/33	0(3)/37	0(2)/65	1427	937	1421
SEFM	0(6)/30	0(4)/30	0(5)/33	1720	1086	898
	STANDARD DEVIATION			AVERAGE/EVALS		
	RANK	SPAT	TOUR	RANK	SPAT	TOUR
genem	49.2	50.8	46.1	52.5/6210	61.2/5343	49.9/6118
VDM	30.6	33.7	22.4	13.7/5587	22.8/5220	21.1/5126
SVDM	9.8	29.3	21.0	3.2/3701	18.6/3478	12.4/3409
EFM	15.9	17.1	25.3	9.9/4034	20.0/5159	27.5/4271
SEFM	9.3	12.4	13.2	3.6/2414	6.6/2048	7.0/1771

Table 5.23: Comparisons for GA-produced 92/93-2 lecture/tutorial timetable

93/94-1	BEST(times)/WORST			LEAST EVALS for perfect timetable		
	RANK	SPAT	TOUR	RANK	SPAT	TOUR
genem	33/304	60/272	33/183	-	-	-
VDM	30/154	73/215	32/156	-	-	-
SVDM	30/121	30/94	0(2)/124	-	-	2473
EFM	0(1)/211	32/128	0(2)/186	2107	-	4534
SEFM	30/92	0(1)/151	0(1)/122	-	1505	1172
	STANDARD DEVIATION			AVERAGE/EVALS		
	RANK	SPAT	TOUR	RANK	SPAT	TOUR
genem	75.5	64.1	47.3	133.9/7452	107.2/6982	104.8/6724
VDM	37.6	44.0	43.5	87.5/9094	122.0/7885	88.2/7345
SVDM	28.3	24.3	40.9	66.9/6411	67.4/5018	60.9/4759
EFM	68.0	35.3	58.2	93.6/6272	76.2/6884	87.0/6530
SEFM	20.3	41.2	32.0	60.7/3859	63.4/3319	60.3/3129

Table 5.24: Comparisons for GA-produced 93/94-1 lecture/tutorial timetable

93/94-2	BEST(times)/WORST			LEAST EVALS for perfect timetable		
	RANK	SPAT	TOUR	RANK	SPAT	TOUR
genem	2/120	1/212	0(1)/124	-	-	4983
VDM	0(4)/91	0(2)/66	0(2)/91	3272	3458	1566
SVDM	0(2)/91	0(2)/61	0(2)/62	3235	3554	2923
EFM	0(1)/65	0(1)/69	0(3)/182	2308	2584	2824
SEFM	0(4)/61	0(6)/60	0(4)/91	1735	1015	1078
	STANDARD DEVIATION			AVERAGE/EVALS		
	RANK	SPAT	TOUR	RANK	SPAT	TOUR
genem	42.1	58.5	36.8	61.2/6567	70.3/5819	58.3/5847
VDM	30.4	27.6	28.5	28.1/6192	29.2/5527	32.2/5492
SVDM	26.6	20.4	24.6	33.3/4643	31.0/4127	30.7/4003
EFM	19.9	27.7	59.0	38.8/6012	36.2/5208	45.5/4464
SEFM	21.1	25.3	30.3	18.2/3032	18.0/2082	27.3/2273

Table 5.25: Comparisons for GA-produced 93/94-2 lecture/tutorial timetable

produce better solution in best, worst and average solution than gene mutation. As before, SEFM appears clearly better than the other choices of mutation operator in terms of both speed and solution quality. For example, SEFM/RANK produces better solutions than EFM/RANK in the 93/94-2 lecture/tutorial timetabling problem according to Student's t-test. With a 95% confidence interval, this improvement is between 1.3 and 39.9 penalties. Also, RANK seems the best of those examined in terms of number of perfect timetables found, however, TOUR seems the fastest in terms of average number of evaluations needed. In general, however, it is very difficult to judge which method or combination of methods is 'best'. What 'best' means depends on many things. If it is very important to find the best timetable possible, but it is okay to take a long time, then we may prefer RANK over TOUR; because RANK seems slightly better than TOUR at finding the optimum, but generally seems to take longer. On the other hand, if very fast good results are needed, we might prefer TOUR. In practical use, a sensible strategy might therefore be to take the best result of 10 runs.

Further details of the best and mean penalty score from these experiments appear in Tables 5.26, 5.27, 5.28 and 5.29. For each GA configuration, these tables give the best and mean values for the components of V. For convenient comparison, each table also presents V for the appropriate human produced actual timetable. These tables provide a compelling view of the difference in timetable quality between the human-produced efforts and the GA results. The most striking aspect is the ability of the GA to successfully minimise over instances of $\mathbf{f}$, while simultaneously dealing effectively

92/93-1 unit penalty	f (1)	d_4 (1)	d_5 (5)	e (10)	l (30)	s_d (30)	s_i (300)	c (500)	TOTAL N/A
actual	65	2	0	14	4	4	0	0	447
RANK(mean)	1.1	0.6	0	0	1.4	1.2	0	0	79.7
RANK/VDM(mean)	2.8	1.0	0	0	1.4	0	0	0	45.8
RANK/SVDM(mean)	0.6	0.6	0	0	0.7	0	0	0	22.2
RANK/EFM(mean)	3.4	0.8	0	0.1	1.3	0	0	0	44.2
RANK/SEFM(mean)	0.3	0.4	0	0.1	0.9	0	0	0	28.7
SPAT(mean)	2.0	0.5	0	0	2.2	0.3	0.3	0	167.5
SPAT/VDM(mean)	3.2	0.5	0	0	1.2	0.1	0	0	42.7
SPAT/SVDM(mean)	0.7	0.3	0	0	1.6	0	0	0	49.0
SPAT/EFM(mean)	2.1	0.6	0	0.1	1.1	0	0	0	36.7
SPAT/SEFM(mean)	0.4	0.4	0	0.5	0.7	0	0	0	26.8
TOUR(mean)	2.0	0.4	0	0	2.0	0.6	0.4	0	200.3
TOUR/VDM(mean)	2.4	0.6	0	0	2.0	0	0	0	63.0
TOUR/SVDM(mean)	1.2	0.3	0	0	1.4	0	0	0	43.5
TOUR/EFM(mean)	3.8	0.4	0	0.1	1.6	0	0.1	0	83.2
TOUR/SEFM(mean)	0.1	0.8	0	0.2	0.6	0	0	0	20.9
RANK(best)	0	1	0	0	0	0	0	0	1
RANK/VDM(best)	0	0	0	0	0	0	0	0	0
RANK/SVDM(best)	0	0	0	0	0	0	0	0	0
RANK/EFM(best)	0	1	0	0	0	0	0	0	1
RANK/SEFM(best)	0	0	0	0	0	0	0	0	0
SPAT(best)	1	0	0	0	0	0	0	0	1
SPAT/VDM(best)	0	0	0	0	0	0	0	0	0
SPAT/SVDM(best)	0	1	0	0	0	0	0	0	1
SPAT/EFM(best)	0	0	0	0	0	0	0	0	0
SPAT/SEFM(best)	0	0	0	0	0	0	0	0	0
TOUR(best)	1	0	0	0	0	0	0	0	1
TOUR/VDM(best)	2	0	0	0	1	0	0	0	32
TOUR/SVDM(best)	0	0	0	0	0	0	0	0	0
TOUR/EFM(best)	1	0	0	0	0	0	0	0	1
TOUR/SEFM(best)	0	0	0	0	0	0	0	0	0

Table 5.26: Actual vs GA lecture/tutorial timetable penalties - 92/93-1

with all the other constraints, for example lunchtime, whereas each human produced timetable seem to be the result of 'giving up' on this constraint in the face of the great difficulty of handling the rest anyway. In general, the result strongly supports the suggestion that GAs can deal effectively with certain multi constraint problems. In particular, when combined with the results reported in section 5.1.6, it is clear that the GA timetabling framework presented here shows great potential for use in arbitrary timetabling problems.

In brief, the algorithms and techniques used in the system can be very easily modified to some other similar kinds of timetabling problems (for example: timetabling athletics

92/93-2 unit penalty	f (1)	d_4 (1)	d_5 (5)	e (10)	l (30)	s_d (30)	s_i (300)	c (500)	TOTAL N/A
actual	49	3	1	5	2	9	1	0	737
RANK(mean)	0.6	0.9	0	0	1.1	0.6	0	0	52.5
RANK/VDM(mean)	1.0	0.2	0.1	0	0.2	0.2	0	0	13.7
RANK/SVDM(mean)	0	0.2	0	0	0.1	0	0	0	3.2
RANK/EFM(mean)	0.4	0.5	0	0	0.3	0	0	0	9.9
RANK/SEFM(mean)	0	0.6	0	0	0.1	0	0	0	3.6
SPAT(mean)	0.5	0.7	0	0	1.4	0.6	0	0	61.2
SPAT/VDM(mean)	0.9	0.9	0	0	0.7	0	0	0	22.8
SPAT/SVDM(mean)	0.1	0.5	0	0	0.6	0	0	0	18.6
SPAT/EFM(mean)	0.3	1.7	0	0	0.6	0	0	0	20.0
SPAT/SEFM(mean)	0	0.6	0	0	0.2	0	0	0	6.6
TOUR(mean)	0.7	1.2	0	0	0.8	0.8	0	0	49.9
TOUR/VDM(mean)	1.3	1.8	0	0	0.6	0	0	0	21.1
TOUR/SVDM(mean)	0.1	0.3	0	0	0.4	0	0	0	12.4
TOUR/EFM(mean)	1.9	1.6	0	0	0.7	0.1	0	0	27.5
TOUR/SEFM(mean)	0	1.0	0	0	0.2	0	0	0	7.0
RANK(best)	0	0	0	0	0	0	0	0	0
RANK/VDM(best)	0	0	0	0	0	0	0	0	0
RANK/SVDM(best)	0	0	0	0	0	0	0	0	0
RANK/EFM(best)	0	0	0	0	0	0	0	0	0
RANK/SEFM(best)	0	0	0	0	0	0	0	0	0
SPAT(best)	1	1	0	0	0	0	0	0	2
SPAT/VDM(best)	0	0	0	0	0	0	0	0	0
SPAT/SVDM(best)	0	0	0	0	0	0	0	0	0
SPAT/EFM(best)	0	0	0	0	0	0	0	0	0
SPAT/SEFM(best)	0	0	0	0	0	0	0	0	0
TOUR(best)	0	1	0	0	0	0	0	0	1
TOUR/VDM(best)	0	0	0	0	0	0	0	0	0
TOUR/SVDM(best)	0	0	0	0	0	0	0	0	0
TOUR/EFM(best)	0	0	0	0	0	0	0	0	0
TOUR/SEFM(best)	0	0	0	0	0	0	0	0	0

Table 5.27: Actual vs GA lecture/tutorial timetable penalties - 92/93-2

events in a stadium, or conference timetabling).

5.2.6.3 Experiments with an Island GA

We also experimented with an *Island* GA [Whitley 93b], in which the population was divided into five sub-populations each having 50 chromosomes. In such an island model, evolution occurs as usual in each island (sub-populations), as if each were a separate single-population GA. At intervals, for example, every k evaluations, *migration* occurs between populations. Migration, in the model we used for experiments, consists of

93/94-1 unit penalty	f (1)	d_4 (1)	d_5 (5)	e (10)	l (30)	s_d (30)	s_i (300)	c (500)	TOTAL N/A
actual	84	2	0	7	5	0	0	0	306
RANK(mean)	1.7	0.2	0	0	2.2	2.2	0	0	133.9
RANK/VDM(mean)	3.3	0.2	0	0	2.1	0.7	0	0	87.5
RANK/SVDM(mean)	0.9	0	0	0	1.9	0.3	0	0	66.9
RANK/EFM(mean)	3.4	0.2	0	0	2.3	0.7	0	0	93.6
RANK/SEFM(mean)	0.5	0.2	0	0	1.9	0.1	0	0	60.7
SPAT(mean)	1.5	0.7	0	0	2.5	1.0	0	0	107.2
SPAT/VDM(mean)	3.9	1.1	0	0	2.9	1.0	0	0	122.0
SPAT/SVDM(mean)	1.1	0.3	0	0	2.1	0.1	0	0	67.4
SPAT/EFM(mean)	3.4	0.8	0	0	2.0	0.4	0	0	76.2
SPAT/SEFM(mean)	0.3	0.1	0	0	1.6	0.5	0	0	63.4
TOUR(mean)	2.2	0.6	0	0	2.2	1.2	0	0	104.8
TOUR/VDM(mean)	3.6	0.6	0	0	2.2	0.6	0	0	88.2
TOUR/SVDM(mean)	0.5	0.4	0	0	1.6	0.4	0	0	60.9
TOUR/EFM(mean)	2.6	0.4	0	0	2.0	0.8	0	0	87.0
TOUR/SEFM(mean)	0.3	0	0	0	1.8	0.2	0	0	60.3
RANK(best)	3	0	0	0	1	0	0	0	33
RANK/VDM(best)	0	0	0	0	1	0	0	0	30
RANK/SVDM(best)	0	0	0	0	1	0	0	0	30
RANK/EFM(best)	0	0	0	0	0	0	0	0	0
RANK/SEFM(best)	0	0	0	0	1	0	0	0	30
SPAT(best)	0	0	0	0	0	2	0	0	60
SPAT/VDM(best)	13	0	0	0	2	0	0	0	73
SPAT/SVDM(best)	0	0	0	0	1	0	0	0	30
SPAT/EFM(best)	1	1	0	0	1	0	0	0	32
SPAT/SEFM(best)	0	0	0	0	0	0	0	0	0
TOUR(best)	2	1	0	0	1	0	0	0	33
TOUR/VDM(best)	2	0	0	0	1	0	0	0	32
TOUR/SVDM(best)	0	0	0	0	0	0	0	0	0
TOUR/EFM(best)	0	0	0	0	0	0	0	0	0
TOUR/SEFM(best)	0	0	0	0	0	0	0	0	0

Table 5.28: Actual vs GA lecture/tutorial timetable penalties - 93/94-1

selecting a chromosome from one of the sub-populations (using the selection scheme in use for that GA) and placing a copy of it into all of the other subpopulations (replacing the least fit in those subpopulation). The choice of subpopulation to migrate from is cyclic, and the choice of subpopulation to migrate *to* is just excluding the sub-population being migrated from. The migration interval (k) we use is 500, and we experiment with SVDM and SEFM paired with rank-based and tournament-based selection methods. The other parameters are the same as in previous experiments.

Tables 5.30, 5.31, 5.32 and 5.33 show the results of these experiments. Columns headed 'S50' and 'S250' refer to a normal single population GA with population size 50 and

93/94-2 unit penalty	f (1)	d_4 (1)	d_5 (5)	e (10)	l (30)	s_d (30)	s_i (300)	c (500)	TOTAL N/A
actual	50	0	0	1	3	3	0	0	240
RANK(mean)	0.9	0.3	0	0	1.2	0.8	0	0	61.2
RANK/VDM(mean)	0.6	0.5	0	0	0.9	0	0	0	28.1
RANK/SVDM(mean)	0	0.3	0	0	1.1	0	0	0	33.3
RANK/EFM(mean)	2.1	0.7	0	0	1.1	0.1	0	0	38.8
RANK/SEFM(mean)	0	0.2	0	0	0.6	0	0	0	18.2
SPAT(mean)	0.9	0.4	0	0	1.6	0.7	0	0	70.3
SPAT/VDM(mean)	1.8	0.4	0	0	0.9	0	0	0	29.2
SPAT/SVDM(mean)	0.5	0.5	0	0	1.0	0	0	0	31.0
SPAT/EFM(mean)	2.6	0.6	0	0	0.9	0.2	0	0	36.2
SPAT/SEFM(mean)	0	0	0	0	0.6	0	0	0	18.0
TOUR(mean)	0.8	0.5	0	0	1.2	0.7	0	0	58.3
TOUR/VDM(mean)	1.8	0.4	0	0	0.9	0.1	0	0	32.2
TOUR/SVDM(mean)	0.4	0.3	0	0	0.9	0.1	0	0	30.7
TOUR/EFM(mean)	2.7	0.8	0	0	1.1	0.3	0	0	45.5
TOUR/SEFM(mean)	0.1	0.2	0	0	0.9	0	0	0	27.3
RANK(best)	1	1	0	0	0	0	0	0	2
RANK/VDM(best)	0	0	0	0	0	0	0	0	0
RANK/SVDM(best)	0	0	0	0	0	0	0	0	0
RANK/EFM(best)	0	0	0	0	0	0	0	0	0
RANK/SEFM(best)	0	0	0	0	0	0	0	0	0
SPAT(best)	1	0	0	0	0	0	0	0	1
SPAT/VDM(best)	0	0	0	0	0	0	0	0	0
SPAT/SVDM(best)	0	0	0	0	0	0	0	0	0
SPAT/EFM(best)	0	0	0	0	0	0	0	0	0
SPAT/SEFM(best)	0	0	0	0	0	0	0	0	0
TOUR(best)	0	0	0	0	0	0	0	0	0
TOUR/VDM(best)	0	0	0	0	0	0	0	0	0
TOUR/SVDM(best)	0	0	0	0	0	0	0	0	0
TOUR/EFM(best)	0	0	0	0	0	0	0	0	0
TOUR/SEFM(best)	0	0	0	0	0	0	0	0	0

Table 5.29: Actual vs GA lecture/tutorial timetable penalties - 93/94-2

250 respectively, while columns headed 'I' refer to the particular island model described above. From the results reported in Tables 5.30, 5.31, 5.32 and 5.33, it appears that the Island model can improve both solution quality and speed compared with a normal single population GA. In addition, 'S250' can get better solution quaility than 'S50' by using tournament-based selection, but it also needs more evaluations. However, 'S50' gets better solutions and needs fewer evaluations than 'S250' when using rank-based selection. In fact, the rank-based selection results for 'S250' are particularly poor. This is probably due to a combination of two main factors: first, the selection pressure of rank-based selection effectively *increases* as we increase the population

size, since the relative chance of selecting the best and the worst is increasing as function of population size. Hence, higher pressure in 'S250' compared with 'S50' means we effectively lose the greater diversity which a bigger population gives, and there is consequently undue pressure on the fittest chromosomes. Secondly, larger populations naturally need longer to converge, because there are generally fewer trials for each population member. In particular, the stopping criterion of 'no improvement after 1,000 successive evaluations' is much more restrictive on a population of 250 than a population of 50. Note also that, in comparison, tournament-based selection fares better in the larger population. The difference is mainly that tournament-based selection pressure (since the same tournament size is used) is *lower* in the 'S250' case than in the 'S50' case. So, tournament based selection takes advantage of the extra diversity introduced. It also seems clear that smart mutation methods just need to use small population sizes. Further, we once more see that SVDM and SEFM are very powerful operators. The speed of island GA is reflected in the average evaluations needed, for example, it only needs less than 1500 evaluations in 3 out of four problems using SEFM and tournament based selection method. In addition, the frequency of island GA successes reflects from the times of optimal timetable found, for example, sometimes 10 out of 10 runs using rank-based selection method.

Table 5.34 shows standard deviations of the penalty and confidence intervals derived from Student's t-test for the 'S50', 'S250' and 'I' (tournament-based selections) experiments. On average, the standard deviation in descending order is 'S50', 'S250' and 'I'. The t-test involves a 95% confidence interval[8] for the average solution in each case. The narrow and small intervals also show a great improvement by using Island GA which roughly tallies with those found by using average and standard deviation.

5.2.7 Discussion

The GA/timetabling framework presented appears successful and useful on a range of medium sized real world timetabling problem. The key elements in the success of the approach are direct chromosome representation used, coupled with stochastic event-freeing mutation, which takes advantage of the directness of the representation.

[8] The real lower bound is not less than 0.

92/93-1	*BEST(times)*			*WORST*			*AVERAGE*			*AVE-EVALS*		
	S50	S250	I	S50	S250	I	S50	S250	I	S50	S250	I
SVDM(rank)	0(3)	11	0(9)	93	3282	2	22.2	793.3	0.2	4486	6288	4069
SVDM(tour)	0(1)	0(2)	0(7)	90	92	32	43.5	37.3	3.4	4236	7380	3128
SEFM(rank)	0(4)	4	0(6)	91	3921	30	28.7	1255.6	3.3	3201	6281	3174
SEFM(tour)	0(2)	0(2)	0(8)	62	61	1	20.9	17.1	0.2	2667	5563	1488

Table 5.30: Panmictic vs Island models for 92/93-1 lecture/tutorial timetable

92/93-2	*BEST(times)*			*WORST*			*AVERAGE*			*AVE-EVALS*		
	S50	S250	I	S50	S250	I	S50	S250	I	S50	S250	I
SVDM(rank)	0(8)	131	0(10)	31	2852	0	3.2	1338.0	0.0	3701	4236	2810
SVDM(tour)	0(5)	0(7)	0(8)	60	30	31	12.4	6.1	3.2	3409	6691	2577
SEFM(rank)	0(6)	17	0(10)	30	2153	0	3.6	985.4	0.0	2414	4665	1870
SEFM(tour)	0(5)	0(9)	0(9)	33	2	1	7.0	0.2	0.1	1771	4369	1288

Table 5.31: Panmictic vs Island models for 92/93-2 lecture/tutorial timetable

93/94-1	*BEST(times)*			*WORST*			*AVERAGE*			*AVE-EVALS*		
	S50	S250	I	S50	S250	I	S50	S250	I	S50	S250	I
SVDM(rank)	30	98	0(4)	121	3218	60	66.9	1209.8	21.5	6411	6527	5568
SVDM(tour)	0(2)	30	0(3)	124	94	61	60.9	52.1	33.1	4759	9473	4296
SEFM(rank)	30	17	0(2)	92	3050	60	60.7	855.6	24.1	3859	7769	3939
SEFM(tour)	0(1)	0(1)	0(1)	122	90	61	60.3	42.2	33.2	3129	6445	3303

Table 5.32: Panmictic vs Island models for 93/94-1 lecture/tutorial timetable

93/94-2	*BEST(times)*			*WORST*			*AVERAGE*			*AVE-EVALS*		
	S50	S250	I	S50	S250	I	S50	S250	I	S50	S250	I
SVDM(rank)	0(2)	23	0(6)	91	2725	30	33.3	1103.7	12.0	4643	6265	4254
SVDM(tour)	0(2)	0(6)	0(6)	62	32	30	30.7	12.2	12.0	4003	7347	2915
SEFM(rank)	0(4)	0(2)	0(10)	61	1796	0	18.2	653.0	0.0	3032	5939	1823
SEFM(tour)	0(4)	0(8)	0(9)	91	30	30	27.3	6.0	3.0	2273	4850	1362

Table 5.33: Panmictic vs Island models for 93/94-2 lecture/tutorial timetable

92/93-1	STD-DEVIATION			95% CONFIDENCE INTERVAL		
	S50	S250	I	S50	S250	I
SVDM	25.6	31.7	10.1	(25.2,61.8)	(14.6,60.0)	(-3.8,10.6)
SEFM	20.0	22.3	0.4	(6.6,35.2)	(1.1,33.1)	(-0.1,0.5)
92/93-2	STD-DEVIATION			95% CONFIDENCE INTERVAL		
	S50	S250	I	S50	S250	I
SVDM	21.0	12.6	9.8	(-2.7,27.5)	(-2.9,15.1)	(-3.8,10.2)
SEFM	13.2	0.6	0.3	(-2.5,16.5)	(-0.3,0.7)	(-0.1,0.3)
93/94-1	STD-DEVIATION			95% CONFIDENCE INTERVAL		
	S50	S250	I	S50	S250	I
SVDM	40.9	20.5	26.4	(31.6,90.2)	(37.4,66.8)	(14.2,52.0)
SEFM	32.0	25.3	17.2	(37.4,83.2)	(24.1,60.30)	(20.9,45.5)
93/94-2	STD-DEVIATION			95% CONFIDENCE INTERVAL		
	S50	S250	I	S50	S250	I
SVDM	24.6	15.8	15.5	(13.1,48.3)	(0.9,23.5)	(0.9,23.1)
SEFM	30.3	12.6	9.5	(5.6,49.0)	(-3.0,15.0)	(-3.8,9.8)

Table 5.34: Standard deviation and t-test for lecture/tutorial timetable

There are certain reasons, however, for supposing that the framework used so far will not extend satisfactorily to other kinds of scheduling problem. In timetabling, we are almost always given (especially in lecture timetabling) a fixed set of timeslots in which to place events. This allows us to set a fixed allele range for each event gene. Also, each individual constraint between a pair of events, or on a single event, tends to exclude only a small subset of the possible pairs of assignments for those events. For example, if we have 30 slots and there is an edge constraint between e_1 and e_2, then this edge constraint excludes just 30 of the possible 900 pairs of assignment for e_1 and e_2. In job-shop scheduling and other scheduling problems, however, the structure of the set T and the nature of the constraints which commonly apply are importantly different. Time is treated more 'continuously'; there are more slots, the interval between slots is smaller, and there is no fixed number of them. Constraints are far more likely to be hard precedence constraints than edge constraints, which considerably cuts down on the number of possible co-assignments of any pair of constrained events. For example, if there are 30 slots, and there is a constraint that event e_1 should be before event e_2, then 465 of the possible 900 pairs of assignments for e_1 and e_2 are excluded.

The combination of a less fixed slot structure and more 'constraining' constraints calls for an alternative from the GA/timetabling framework we have looked at so far. In the next two chapters we will investigate such an alternative.

Chapter 6

GAs in Scheduling

In the previous two chapters, we showed how we can use a direct representation in GA to solve exam and lecture/tutorial timetabling problems. In this chapter and the next, we will show how to use an indirect representation in GA to solve job-shop (flow-shop) and open-shop scheduling problems.

In timetabling problems, if the number of time-slots s is large enough and the number of constraints c remains fixed, in other words if c/s is small enough according to equation 4.1, a very large proportion of possible timetables will be valid timetables. In scheduling, events are often tasks or operations which must be processed by a particular processor or machine. More than in timetabling, events have a wide variation in length and precedence/ordering constraints are much more common. Furthermore, there are usually more slots, in the sense of possible start-time. The number of slots isn't too important but the key point is that events are typically very long in relation to the time-gaps between slots. We can also represent scheduling problems in the same way as timetabling problems. But then, some difficulties will arise, for example, we will have to choose an arbitrary maximum number of slots, or event-length might vary from very small to very large, which will increase the complexity of evaluations.

In these circumstances it ought to be possible to construct solutions directly. This suggests an alternative approach for scheduling problems: consider the space of schedules that require some precedence constraints between the j operations (jobs) and have a fixed number of machines, and get a GA to try to minimise the time used. In this kind of approach a chromosome might be taken to represent some permutation of the

set of operations (jobs). Given such a chromosome, it can be turned into a schedule by assigning times to the operations (jobs) in the order given by the chromosome. A schedule builder is used to decide how to allocate the time as early as possible but can't let the operations on the same job overlap or different jobs use the same machine at the same time. The quality of the schedule is just the criteria of the schedule, for example the shorter the better for minimising complete time, because a schedule builder should guarantee to produce a legal schedule for us. Therefore, we don't need to use various weighting punishment functions described in the last two chapters to get rid of the illegal schedule or to evaluate quality of the schedule.

6.1 Introduction

6.1.1 Static vs Dynamic Scheduling

In job-shop scheduling problems, based on the arrival time of jobs, we can classify them as static or dynamic scheduling. In *static* scheduling, all the jobs that are to be processed in the shop are ready for use simultaneously. In other words, no more jobs will arrive later. In *dynamic* scheduling, jobs arrival times are not fixed at a single point. In other words, jobs can arrive at some known or unknown future times.

6.1.2 Deterministic vs Stochastic Scheduling

A *deterministic* job shop has all related information known beforehand or with certainty. In other words, machines and jobs are either all available in the beginning (static) or arrive at known future (dynamic). Furthermore, the processing times must be known when the jobs are available. In a *stochastic* job shop, either job ready times or operation processing times are random variables described by a known statistical form or probabilitic distribution. In this situation, the job shop becomes a queuing system. The most common way to solve the queuing system is to use simulation method to choose the predetermined priority rules to assign the job to machine. In other words, simulation is probably the technique most used to investigated dynamic and stochastic job-shops in the job-shop scheduling literature.

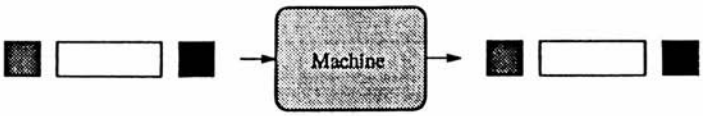

Figure 6.1: Single Machine Shop

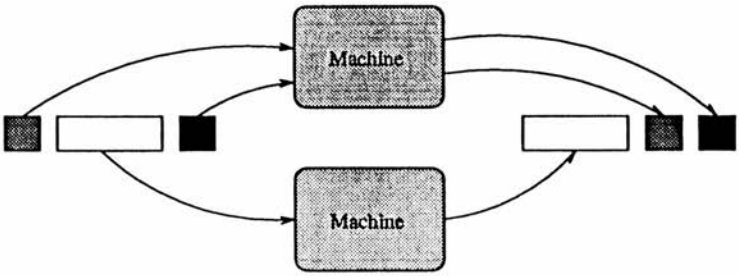

Figure 6.2: Parallel Machine Shop

6.2 Classifications of Scheduling Problems

There are four classes of Job Shop Scheduling Problems:

6.2.1 Single Machine Shop

Each job has a single operation that is to be performed on the single machine existing in the shop. Scheduling in this context means finding a sequence in which the jobs have to be processed (see Figure 6.1).

6.2.2 Parallel Machine Shop

Each job has a single operation and there are m machines that work in parallel and the function of these machines is the same. In general, a given job can be processed on several of the m machines, possibly on any of them (see Figure 6.2).

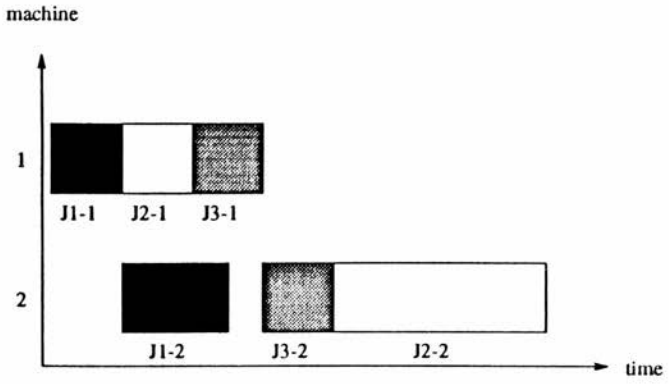

Figure 6.3: Flow-Shop

6.2.3 Flow-Shop

The flow shop contains m machines, and each job consists of a strictly ordered sequence of operations. All movement between machines within the shop must be in a uniform direction. In other words, the machine order for all jobs is the same (see Figure 6.3). A special case of flow shop is called *permutation* flow shop or scheduling, in which not only the machine order is the same, but we also restrict the search to schedules in which the job order is the same for each machine (see Figure 6.4).

6.2.4 General Job-Shop

There is no such restriction in the general job-shop problem. In other words, there are m machines and j jobs. Each job has its own processing order and this may bear no relation to the processing order of any other job (see Table 6.1 and Figure 7.1). The open-shop can be considered as a extension of general job shop in which there is no flow pattern within each job (see Table 7.23 and Figure 7.4), and the flow-shop is a special case of general job shop which has identical flow pattern within each job and permutation flow-shop further restricts things to identical job order on all machines.

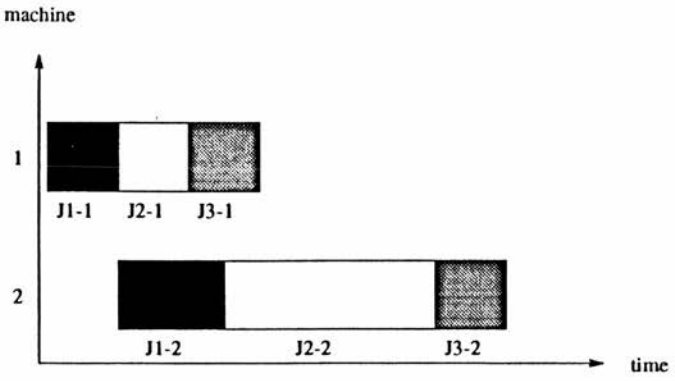

Figure 6.4: Permutation Flow-Shop

Single/parallel machine and (permutation) flow-shops are easier than in general job-shop scheduling problems. From now on, the thesis will use the term job-shop as meaning the general job-shop. In this chapter, we will first introduce the framework we used in job-shop scheduling and later in the next chapter, we will introduce the applications of this framework to some benchmarks of flow-shop, job-shop and open-shop scheduling problems.

6.3 Assumptions in the General Job-Shop Scheduling Problems

In this thesis, we investigate the general job-shop scheduling problems which are mostly based on the following assumptions taken from [Conway *et al.* 67].

- Each machine is continuously available for assignment, without significant division of the time scale into shifts or days, and without consideration of temporary unavailability for causes such as breakdown or maintenance.
- Jobs are strictly-ordered sequences of operations, without assembly or partition.
- Each operation can be performed by only one machine in the shop.
- There is only one machine of each type in the shop.

- Preemption is not allowed – once an operation is started on a machine, it must be completed before another operation can begin on that machine.
- The processing-times of successive operations of a particular job may not be overlapped. A job can be in process on at most one operation at a time.
- Each machine can handle at most one operation at a time.
- The simple job-shop process has only one limiting resource.

Other assumptions exist, for example [Bellman *et al.* 82] divided the assumptions into based on jobs, based on machine and based on processing times. In real world problems, we usually need to relax some assumptions to fit the requirements. We don't discuss these possible relaxations in this thesis to meet various of special cases, however, our framework can deal with job-shop rescheduling as well as dynamic job-shop scheduling.

6.4 Basic GA Framework for Scheduling

6.4.1 Representation

In this thesis, a novel approach to JSSP and related problems is presented, in which the chromosome is a fixed length string, encoding a schedule in such a way that crossover and mutation always produce valid schedules. The idea of the representation is as follows: The genotype for a $j \times m$ problem we use is a string containing $j \times m$ chunks (genes), each chunk being large enough to hold the largest job number (j). It provides instructions for building a legal schedule as follows; the string of chunks $abc\cdots$ means: put the first untackled task of the a th uncompleted job into the earliest place where it will fit in the developing schedule, then put the first untackled task of the b th uncompleted job into the earliest place where it will fit in the developing schedule, and so on. The task of constructing an actual schedule is handled by a schedule builder described later which maintains a circular list of uncompleted jobs and a list of untackled tasks for each such job. Thus the notion of "a-th uncompleted job" is taken modulo the length of the circular list to find the actual uncompleted job. For example, Table 6.1 shows the standard 6×6 benchmark problem (that is, $j = 6, m = 6$) taken from [Fisher & Thompson 63]. In this example, job 1 has to go to machine 3 first, for 1 unit of time; then to machine 1 for 3 units of time; and so on. The task is to generate a good schedule showing what each machine is to do when. We encode an answer as a

	(m,t)	(m,t)	(m,t)	(m,t)	(m,t)	(m,t)
Job 1:	3,1	1,3	2,6	4,7	6,3	5,6
Job 2:	2,8	3,5	5,10	6,10	1,10	4,4
Job 3:	3,5	4,4	6,8	1,9	2,1	5,7
Job 4:	2,5	1,5	3,5	4,3	5,8	6,9
Job 5:	3,9	2,3	5,5	6,4	1,3	4,1
Job 6:	2,3	4,3	6,9	1,10	5,4	3,1

Table 6.1: The 6x6 benchmark JSSP

36-chunk string, each chunk large enough to hold the largest job number (1-6 in above case). so that (say) 111111163... means

```
job 1 -> machine 3 (3 being first untackled task)
job 1 -> machine 1 (1 now being first untackled task for job 1)
job 1 -> machine 2
job 1 -> machine 4
job 1 -> machine 6
job 1 -> machine 5 (job 1 now complete)
job 2 -> machine 2 (2 being first untackled task; job 2 being number 1
                    in uncompleted list)
job 2 -> machine 3 (uncompleted job list is [2,3,4,5,6] so the 6th in
                    this list, treating it as circular, is job 2
                    again; and machine 3 is job 2's first uncompleted
                    task)
job 4 -> machine 2 (job 4 is number 3 in uncompleted list)
  .
  .
  .
```

The similar representation used in the travelling salesman problem (TSP) called *ordinal representation* [Grefenstette *et al.* 85] has the following disadvantage: in each generation, the bits after the crossover point will change their meaning. In other words, because the representation is indirect, there is a non-trivial mapping between jobs (or cities, in the TSP), and the alleles of the ith gene. In a direct representation, for example in our timetabling representation, this is the identity map (allele k always means job k, for any gene i). In the ordinal TSP representation and our job-shop scheduling representation, this mapping is unstable. It is different for each i, and depends on the alleles at $j < i$, and so is different for each distinct chromosome. Now, the more diversity in the population before point i in each chromosome, the more diverse the mapping after point i. Hence, the more random the job referred to by an allele after i

chosen from that population. However, our representation in Job-Shop Scheduling is different from ordinal representation in two ways:

- We treat the list as *circular*, so we don't need to restrict the range value of each bit but in [Grefenstette *et al.* 85] a circular list is not used and the i-th chunk can only range from 1 to $N - i + 1$ in value; this complicates the whole process unduly. Ours is a more general way to represent the problem since it allows the same operation to be done on each gene without regard to its range of values.
- The tail randomness does not happen after the first crossover point as in *ordinal representation*, instead it depends on when one of the jobs has completed. The more operations jobs have, the later randomness the tail chromosome sets in.

For example, consider the following two tours for TSP taken from [Grefenstette *et al.* 85]:

```
ordinal tours          path tours
-------------          ----------
(1 2 3 2 1)            (a c e d b)
(2 4 1 1 1)            (b e a c d)
```

Suppose we crossover between the second and the third positions. Then we get the following two children:

```
ordinal tours          path tours
-------------          ----------
                             e->
(1 2 1 1 1)            (a c b d e)
(2 4 3 2 1)            (b e d c a)
                             e->
```

The path tours to the left of the crossover points do not change, but the path tours to the right of the crossover points are completely changed. We can also see from the example, the earlier the crossover point, the greater the disruption of the children's path tour. As to our JSSP representation, we want to see whether the crossover point will affect the tail randomness or not as in [Grefenstette *et al.* 85]'s TSP. For example, assume we have a j=3, m=3 JSSP as follows:

```
chromosome                job sequence
----------                ------------
(1 2 3 3 2 2 2 3 1)       (1 2 3 3 2 2 3 1 1)
(1 1 2 3 3 1 2 1 1)       (1 1 2 3 3 1 3 2 2)
```

Case 1: suppose we crossover the parents between the second and the third positions. Then we get the following two children:

```
chromosome                job sequence
----------                ------------
                                    *->
(1 2 2 3 3 1 2 1 1)       (1 2 2 3 3 1 2 1 3)
(1 1 3 3 2 2 2 3 1)       (1 1 3 3 2 2 2 1 3)
                                    *->
```

Case 2: suppose we crossover the original parents between the fifth and the sixth positions. Then we get the following two children:

```
chromosome                job sequence
----------                ------------
                                    *->
(1 2 3 3 2 1 2 1 1)       (1 2 3 3 2 1 2 1 3)
(1 1 2 3 3 2 2 3 1)       (1 1 2 3 3 2 2 1 3)
                                    *->
```

Comparing these two cases, we see the 'tail randomness' in our representation is independent of the crossover point. The simple example tells us that the tail randomness happens only after position seven which is the first circular position no matter whether the crossover point falls between the early or later positions. In other words, the randomness of initial generation will depend on the random number generator and seed used, and all other generations will depend on the previous generation. The next section looks at the chances of this happening.

6.4.2 Algorithm for Probability of Preserving a Gene's Full Meaning

Suppose there is a J jobs and M machines JSSP and we want to know the n-th gene's probability of preserving its full meaning. We say that a gene *has its full meaning* if the j different alleles that could be there refer to j distinct tasks. More precisely, suppose that the chromosome before the n-th gene remains fixed but the n-th gene

has any of its j possible values. When these j partial chromosomes are decoded up to the n-th gene, j different partial schedules result. If less than j result, we say that the n-th gene *has lost some of its full meaning.* The cause is of course that all the tasks of some job have been included in the schedule by that point, so that the circular list of jobs that have unscheduled tasks remaining is no longer of length j. It guarantees to preserve its meaning when $n \leq M$ because circularity doesn't happen even when the first M bits are all the same values. But in this extreme case, the $M+1$th gene can be expressed in two ways. So, we don't need to consider the first M bits in this algorithm and regard probability of preserving their meaning as 1.0. Similarly, ambiguity must appear after the $J \times (M-1)+1$th gene. If all the jobs appear $M-1$ times exactly in the first $J \times (M-1)$ genes, then the $J \times (M-1)+1$th gene will complete one of the jobs. That is the $J \times (M-1)+2$th gene refers to one of $j-1$ jobs, although it has j possible values. So, we also don't need to consider the last $M-1$ bits in this algorithm and regard probability of preserving their meaning as 0.0.

Now the algorithm for calculating the probability of preserving a gene's meaning, for a gene from gene $M+1$ to $J \times (M-1)+1$ is as follows:

- *First*, find all the possible partitions (P) before the nth gene.
- *Second*, find each partition's permutation (E) respectively.
- *Third*, find each partition's possible value sequences (S) respectively.
- *Fourth*, calculate the total search space (T) before gene n.
- *Finally*, the probability of preserving the nth gene's meaning (PPM) is

$$\mathbf{PPM(n) = P(n) \times E(n) \times S(n) \ / \ T(n)} \tag{6.1}$$

For example, if want to know what is the probability of preserving the 10th gene's full meaning in a 6 jobs and 6 machines JSSP. We must first find all the legal partitions P (at most occurs 5 bit for a single bit value) in position 9 as follows:

```
* 9
* 8  1
* 7  2
* 7  1  1
* 6  3
* 6  2  1
* 6  1  1  1
  5  4
  5  3  1
  5  2  2
  5  2  1  1
  5  1  1  1  1
  4  4  1
  4  3  2
  4  3  1  1
  4  2  2  1
  4  2  1  1  1
  4  1  1  1  1  1
  3  3  3
  3  3  2  1
  3  3  1  1  1
  3  2  2  2
  3  2  2  1  1
  3  2  1  1  1  1
* 3  1  1  1  1  1  1
  2  2  2  2  1
  2  2  2  1  1  1
* 2  2  1  1  1  1  1
* 2  1  1  1  1  1  1  1
* 1  1  1  1  1  1  1  1  1
```

All the above partitions are legal except those with leading '*'. The reason is not hard to understand. For instance, partition (5 4) is legal, which means 5 bits having the same value and another 4 bits having another same value; some possible combinations are 222225555, 111144444 or 662662222. Whereas, partition (8 1) is illegal, for example 333333334, there are eight bits having the same value which is not longer considered in our legal partition because our legal partition can just allow at most five bits of the same value. Partition (3 1 1 1 1 1 1) is also an illegal one because it implies that we have seven different kinds of partitions or values, for example 111234567, but 7 is illegal because the bit value in this example can just be between 1 and 6, that is only six jobs exist.

After getting all the legal partitions, we then find each partition's permutation (E). For example, the number of permutations of partition (5 4) is 9!/(5!x4!), and the number of permutations of partition (4 2 2 1) is 9!/(4!x2!x2!x1!).

position	1-6	7	8	9	10	11	12	13	14
probability	1.0	0.9999	0.9992	0.9974	0.9932	0.9854	0.9723	0.9525	0.9241
position	15	16	17	18	19	20	21	22	23
probability	0.8858	0.8367	0.7763	0.7051	0.6245	0.5370	0.4459	0.3553	0.2697
position	24	25	26	27	28	29	30	31	32-36
probability	0.1933	0.1291	0.0792	0.0436	0.0209	0.0082	0.0024	0.0004	0.0

Table 6.2: Probability of preserving full meaning for 6x6 JSSP

Then, we can find each partition's possible value sequence (S) based on the partitions we found. For example, the number of value sequences of partition (5 4) is C(6,2)x2!=(6x5)/(1x2)x2=30 and the number of value sequences of partition (4 2 2 1) is C(6,4)x4!/2!=180.

Also, the total search space (T) before position 10 is 6^9.

Finally, the probability of preserving its meaning at position 10, PPM(10), is P(10) x C(10) x S(10) / T(10) $\approx$ 0.9932.

That is to say, if no bit occurs more than 5 times by position 9, then we can say position 10 is still keeping its full meaning. However, the probability that a certain job appears more than 5 times is very small, and for the 6x6 example we know that bit 10 has 99% probability to preserve its full meaning. Furthermore, even if position 10 contains the sixth occurrence of any bit, we can just say that position 11 begins circular, but position 10 still preserves its meaning. However the bits after that can lose some of their meaning. In Table 6.2, we list the probability of preserving full meaning for 6x6 JSSP problem. The probability cannot guarantee to preserve its meaning from the 7th gene and does so with probability less than 0.5 after 20th gene and so on.

6.4.3 Fitness

Many criteria exist to measure the quality of a schedule, the list below is an annotated adaptation of the criteria list of [French 82], who divided the main criteria into several branches.

6.4.3.1 Based on Complete Time

- *Maximum Complete Time* (C_{max}): C_{max} is also called *total production time* or *makespan*. Minimising C_{max} implies that the cost of a schedule depends on how long the processing system is devoted to the entire set of jobs.
- *Mean Complete Time* ($\bar{C}$): Minimising $\bar{C}$ implies that a schedule's cost is directly related to the average time it takes to finish a single job.
- *Maximum Flow Time* (F_{max}): Minimising F_{max} implies that a schedule's cost is directly related to its longest job.
- *Mean Flow Time* ($\bar{F}$): Minimising $\bar{F}$ implies that a schedule's cost is directly related to the average time it takes to process a single job.

6.4.3.2 Based on Due Date

- *Maximum Lateness* (L_{max}): Minimising L_{max} implies that a schedule's cost (reward) is directly related to its latest job.
- *Mean Lateness* ($\bar{L}$): Minimising $\bar{L}$ implies that a schedule's cost is directly related to the average difference between complete times and due-dates for all the jobs. Early jobs will have negative differences, in effect contributing a reward.
- *Maximum Tardiness* (T_{max}): Minimising T_{max} implies that a schedule's cost is directly related to its latest job that completes after its due-date.
- *Mean Tardiness* ($\bar{T}$): Minimising $\bar{T}$ implies that a schedule's cost is directly related to the average late time for all the jobs, where early jobs are considered to have a late time of zero.
- *Number of Tardy Jobs* (NT): Minimising NT implies that a schedule's cost depends on the number of jobs that complete after their due-dates.
- *Maximum Earliness* (E_{max}): Minimising E_{max} implies that a schedule's cost is directly related to its earliest job that completes before its due-date. It happens in the situation that a schedule's inventory cost is more expensive than it's work-in-process cost because customer will not accept the product until the due-date.

Some other criteria exist, for example criteria based on the inventory and utilisation costs; examples are the mean number of jobs waiting for machines, the mean number of unfinished jobs or the mean number of complete jobs and so on.

All the criteria described above except E_{max} belong to the class of *regular* measures of performance. According to [Hax & Candea 84], a performance is regular if:

- it is a function of the job completion times
- its value has to be minimized
- its value increases only if at least one of the completion times in the schedule increases

The *non regular* measure performance is less known than the regular ones in the scheduling literature, however, our GA approach can deal with the non regular criteria in the same manner as those of the regular ones. Our GA representation and schedule builder described later can be applied to any of the criteria or combinations of them. The only thing to change is the fitness function. For example, in achieving the objective of minimising makespan (C_{max}) and assuming all the job's ready times are zero, then we just need to record all the machine's last free times and choose the largest one as our makespan. In other words, we need a function,

$$f(m) = C_{max} = max(c_1, c_2 ... c_m) \tag{6.2}$$

where 1..m represent the machine number, then to minimise this function.

Another method to calculate the makespan is based on each job's finish time which includes the job's ready time plus the waiting and the processing time of all the operations. Then choose the one which has the latest finish time as makespan.

$$f(j) = C_{max} = max(c_1, c_2 ... c_j) \tag{6.3}$$

where 1..j represent the job number, then to minimise this function. In addition, if we want to minimise mean complete time ($\bar{C}$), the function we need is

$$f(j) = \bar{C} = \sum_{i=1}^{j} (C_i)/j \tag{6.4}$$

Furthermore, if want to achieve the goal of mean tardiness ($\bar{T}$), we need first know all the due-dates (d_j) of each job. Then we can calculate the lateness (L_j) of each job by deducting d_j from each C_j, where C_j is the complete time of each job which can be recorded by the schedule builder described later. The tardiness of each job can be calculated: $T_j = \max(L_j, 0)$ because tardiness only considers the late work. Finally we can get the average tardiness ($\bar{T}$) as follows:

$$f(j) = \bar{T} = \sum_{i=1}^{j}(T_i)/j \tag{6.5}$$

where 1..j represent the job number.

Similarly, the earliness of each job can be calculated: $E_j = \max(0, -L_j)$ because earliness only considers the early work.

6.4.4 Schedule Builder

When generating a schedule, the most common way is to choose an operation from a set of *schedulable operations* one at a time and assign a start-time to each. A schedulable operation is an operation all of whose preceding operations have been completed. The problem is how to choose an operation from the set of schedulable operations and how to assign the start time to this operation. The process is a schedule generation procedure.

There are four cases for any feasible schedule; inadmissible, semiactive, active, and nondelay schedules. The number of *inadmissible schedules* or *schedule with excess idle time* is infinite and of no interest in schedule generation because it is useless under any measure of performance. A *semiactive schedule* contains no excess idle time. In other words, we can obtain a semiactive schedule by forward-shifting a schedule until no such excess idle time exists. However, a semiactive schedule can also be improved if we can shift it by skipping some operations to the front without causing other operations to start later than the original schedule. An *active schedule* allows no such shift to be made in a semiactive schedule. The optimal schedule is guaranteed to fall within the active schedules. The active schedule is also a superset of nondelay schedules. In a *nondelay schedule*, a machine is never kept idle if some operation is able to be processed. The best schedule is not necessarily nondelay, but the nondelay is easier

to generate than active schedules and may be a very near optimal schedule even if it is not an optimal one. The active schedules are generally the smallest dominant set in the job shop problem and the best schedule is necessarily an active schedule. The nondelay schedules are a subset of active schedules, so the number of them are smaller than those of active schedules, however they are not dominant though the average solution quality may be better than that of active schedules.

Table 6.3 illustrates these four kinds of schedule, assuming job1 must go to machine2 first, then machine1, machine3 and job2 must go to machine3 first, then machine1, machine2.

```
Excess idle time (inadmissible) schedule:
     mc1:    111       22222
     mc2:1111                  22
     mc3:       1111112
Semiactive schedule:
     mc1:    111       22222
     mc2:1111               22  (shifting excess idle time)
     mc3:       1111112
Active schedule:
     mc1:    11122222
     mc2:1111       22
     mc3:2      1111111  (skipping operation)
Nondelay schedule:
     mc1: 22222111  (keeping machine busy)
     mc2:1111  22
     mc3:2        111111
```

Table 6.3: Four kinds of schedule

From above, we know the nondelay schedule generates the schedule one unit time longer than active schedule. In other words, nondelay schedules cannot guarantee to get the optimal schedule.

The relationship between these types of schedule is in Figure 6.5.

All active schedules can be generated by active-schedule generation developed by [Giffler & Thompson 60], any of the schedulable operations which can start earlier than the earliest complete time among the schedulable operations, so that the procedure might be called active-schedule dispatching. Similarly, all nondelay schedules

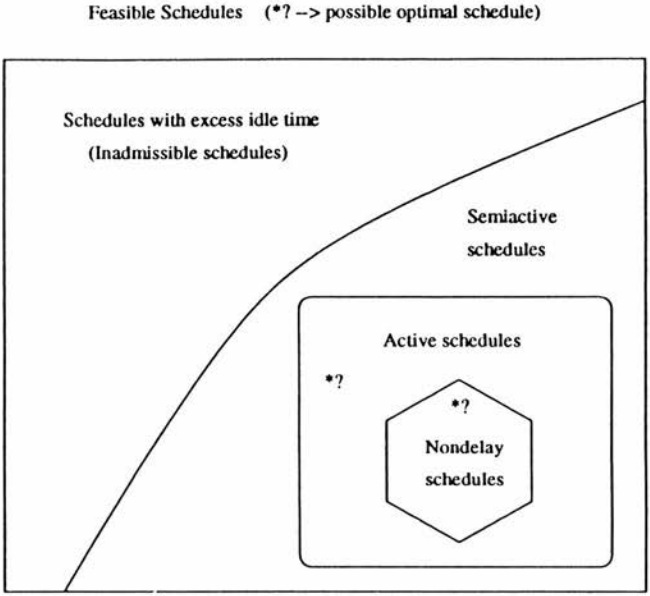

Figure 6.5: Types of feasible schedule

can be generated by nondelay-schedule generation (see for example [Conway *et al.* 67]) which is a modification of [Giffler & Thompson 60], choosing the schedulable operation which can start earliest or any of them if several ones can start as early as the earliest one, so that the procedure might be called nondelay-schedule dispatching. These procedures guarantee that no idle time can be scheduled on a machine and there is no operation which could be left-shifted into the interval of idle-time.

A *dispatching procedure* is one in which the actual decisions affecting a given machine can be implemented in the same order that they are made. Any active schedule may be obtained by a generation procedure which selects an operation from the potential schedulable operations and schedules it at the soonest possible time consistent with the precedence constraints and the existing schedule for the machine which the operation requires. This procedure is, however, not a dispatching procedure since an operation can be inserted before another previously scheduled operation. Our schedule is a kind

of active schedule and our GA-based schedule builder is straightforward, belonging to this kind of non dispatching procedure, but our scheduling report sorted by the start time and finish time of operations is a dispatching one which will not insert before any previously scheduled operation. Our schedule builder must consider four cases when slotting a task into a developing schedule. For example, in Table 6.4, suppose it is asked to slot job 1 into machine 2, with processing time 2. If there is a suitable gap in the schedule for machine 2, it may be possible to fit the task in there with or without *compulsory idle time*; and there may be no suitable gap, so that task has to be added to the end of the machine's schedule with or without compulsory idle time. The ## shows where the schedule builder would place the task in each case.

```
With suitable gap, idle time needed:
     mc1:....... 111333
     mc2:.... 222   ## 33333...

With suitable gap, idle time not needed:
     mc1:.... 11144444
     mc2:..... 222##  44...

No suitable gap, idle time needed:
     mc1:.... 555111
     mc2:.... 2225  ##

No suitable gap, idle time not needed:
     mc1:....... 666111
     mc2:...... 222 6666##
```

Table 6.4: Schedule builder choices of task placement

Schedule building is a computationally easy and cheap process. Using our representation, there are j^{jm} distinct genotypes, and in general each legal schedule can be encoded in very many ways. The tail of a genotype will exhibit a high degree of redundancy because the circular list of jobs gets shorter as we pass along the chromosome building the schedule, and so several possible values for a chunk will map to the same uncompleted job, for instance, the last gene is totally redundant which represents the only unfinished operation. So far, we have not found this to be a problem. The degree of redundancy (seemingly the price to pay for a representation scheme which avoids illegal schedules) is not so high as to render crossover useless (*that is*, continually re-

sulting in new representations of already examined schedules), and low enough to allow impressive results from genetic search, especially when we used several possible strategies to vary operator rates, taking account of the different degrees of redundancy and the (related) different convergence rates in different parts of the chromosome.

6.4.5 A Simple Example to Build a Schedule

The representation is conceptually quite simple, but the details about how to decode and schedule the jobs and tasks are potentially confusing, so we will use a very simple example to illustrate the representation: Imagine a 2 jobs by 3 machines job shop scheduling problem as follows:

```
Job1: 1 (10)   3 (7)   2 (20)
Job2: 2 (15)   1 (2)   3 (8)
```

So, job1 consists of a task (or operation) on machine 1 which takes 10 units of processing time, followed by a task on machine 3 which takes 7 units, followed by a task on machine 2 which takes 20 units. We represent a schedule for such a problem by using 6 genes (the number of genes is the same as the total number of tasks from all of the jobs), each of which can take a value from 1 to 2 (from 1 to j, where j is the number of jobs). We illustrate the representation and schedule builder by stepping through its interpretation of the chromosome:

```
1 2 2 2 2 1
```

The interpretation of this chromosome is as follows:

```
                              *
schedule  the next task of the  1st unfinished job,
schedule  the next task of the  2nd unfinished job,
schedule  the next task of the  2nd unfinished job,
schedule  the next task of the  2nd unfinished job,
schedule  the next task of the  2nd unfinished job,
schedule  the next task of the  1st unfinished job,
                              *
```

Notice how the "1 2 2 2 2 1" chromosome is reflected in the column between the '*'s. Now, to run through what this actually means, interpretation proceeds like this in the

beginning:

```
circular list = [ 1=(1,3,2), 2=(2,1,3) ]
chromosome = [ 1 2 2 2 2 1 ]
```

The first gene is 1, so take a task off the first member of list to get:

```
circular list = [ 1=(3,2), 2=(2,1,3) ]
chromosome = [ 2 2 2 2 1 ]
schedule is:
       machine1: 1111111111
       machine2:
       machine3:
```

Next gene is 2, so take a task off the second member to get:

```
circular list = [ 1=(3,2), 2=(1,3) ]
chromosome = [ 2 2 2 1 ]
schedule is:
      machine1: 1111111111
      machine2: 222222222222222
      machine3:
```

Next gene is 2, so take a task off the second member again and place it as early as possible to get:

```
circular list = [ 1=(3,2), 2=(3) ]
chromosome = [ 2 2 1 ]
schedule is:
      machine1: 1111111111     22
      machine2: 222222222222222
      machine3:
```

Next gene is 2, so take a task off the second member again and place it as early as possible to get:

```
circular list = [ 1=(3,2), 2=() ]
chromosome = [ 2 1 ]
schedule is:
     machine1: 1111111111     22
     machine2: 222222222222222
     machine3:                   22222222
```

Next gene is 2, treating the list of unfinished jobs as a circular list, the second unfinished job is job 1 because job 2 is already finished, so take a task off the first member and place it as early as possible to get:

```
circular list = [ 1=(2), 2=() ]
chromosome = [ 1 ]
schedule is:
     machine1: 1111111111      22
     machine2: 222222222222222
     machine3:            111111122222222
```

Finally, no matter what the gene left, it always represent the first job in this example because there is only one operation left, so place it as early as possible to get:

```
circular list = [ 1=(), 2=() ]
chromosome = [ ]
schedule is:
     machine1: 1111111111      22
     machine2: 222222222222222  1111111111111111111
     machine3:            111111122222222
```

The list of unfinished jobs is now empty, so we have finished and the makespan is 37.

6.5 Heuristics for Dynamic Job-Shop Scheduling

In the dynamic job-shop scheduling literature, the most often used heuristic is the scheduling rule. The scheduling rule can be used synonymously with a dispatch rule, priority rule or heuristic [Panwalkar & Iskander 77]. This says that a decision must be made when a machine becomes free. That is to decide which job to be scheduled next. In the survey paper by [Panwalkar & Iskander 77], which listed more than one hundred scheduling rules, their definition and application is given. Later, [Blackstone Jr *et al.* 82] did a state-of-the-art survey of dispatching rules in manufacturing job-shop operations and compared several of dispatching rules. [Kiran 84a, Kiran 84b] also reported their survey about simulation studies and performance of priority rules based on different criteria in job-shop scheduling.

Two kinds of schedule generation method are often used, non-delay schedule and active schedule, which we have described in the previous section. The scheduling rule is then applied to any tie of each decision rather than to generate all the possible schedules. So far, a lot of simulation has been done in the past to learn which scheduling rule is effective for which criteria and problem. Here we list some of the most important scheduling rules and their brief description then in the next chapter we implement

these rules listed here to compare their use in our GA approach in dynamic scheduling problem.

Assume t is current time, w_i is the weight, r_i is the ready time or arrival time, d_i is the due-date, R_i is the remaining processing time, T_i is the total processing time, and n_i is the number of the remaining operations of i_{th} job. Further, p_{ij} is the processing time, od_{ij} is the operational due date, mod_{ij} is the modified operational due-date, and s_{ij} is the slack time of j_{th} operation of i_{th} job. Now we can define the schedule rules:

- *RND* (randomly) : select the schedulable operation with equal possibility.
- *FASFS* (first arrive shop first serve) : select the schedulable operation with the smallest r_i.
- *SPT* (shortest processing time) : select the schedulable operation with the smallest p_{ij}.
- *WSPT* (weighted shortest processing time) : select the schedulable operation with the largest w_i/p_{ij}.
- *LPT* (largest processing time) : select the schedulable operation with the largest p_{ij}.
- *LWKR* (least work remaining) : select the schedulable operation with the smallest R_i.
- *WLWKR* (weighted least work remaining) : select the schedulable operation with the largest w_i/R_i.
- *LTWK* (least total work) : select the schedulable operation with the smallest T_i.
- *WTWORK* (weighted least total work) : select the schedulable operation with the largest w_i/T_i.
- *EGD* (earliest global due-date) : select the schedulable operation with the smallest d_i.
- *EOD* (earliest operational due-date) : select the schedulable operation with the smallest operational due-date, $od_{ij} = r_i + (d_i - r_i) * R_i / T_i$.

- *EMOD* (earliest modified operational due-date) : select the schedulable operation with the smallest modified operational due-date, mod_{ij} = larger (od_i, t + p_{ij}).
- *MST* (minimum slack time) : select the schedulable operation with the smallest slack time, $s_{ij} = d_i - R_i - t$.
- *S/OP* (slack per operation) : select the schedulable operation with the smallest s_{ij}/n_i.
- *CR* (critical ratio) : select the schedulable operation with the largest $R_i/(d_i-t)$.

Experiments using each of these are reported in the next chapter.

Chapter 7

Experiments with GAs in Scheduling

7.1 GAs in Job-Shop Scheduling/Rescheduling Problems

7.1.1 Problem Description

7.1.1.1 Job-Shop Scheduling Problem (JSSP)

A description of a simple form of *Job Shop Scheduling Problem* (JSSP) goes as follows. There is a set of m machines $M = \{m_1, m_2, \ldots, m_m\}$, and a collection of j jobs $J = \{j_1, j_2, \ldots, j_j\}$, each of which comprises a collection of operations (sometimes called tasks). An operation is an ordered pair (a, b), in which a is the machine on which the operation must be performed, and b is the time it will take to process this operation on machine a. Each job consists of a collection of such operations in a given *job-dependent* order. A feasible schedule for the JSSP is one which assigns a start time to each operation, satisfying the constraint that a machine can only process one operation at a time, and that two or more operations from the same job cannot be processed at the same time. The definition of 'good schedule' can vary considerably; examples are the criteria described in the previous chapter. For example, a criterion of looking for the shortest makespan – the shortest time between the start time of the first scheduled task and end time of the last task to finish. For the particular 6x6 problem in Table 6.1, the minimum makespan is known to be 55 as in, for example, Figure 7.1.

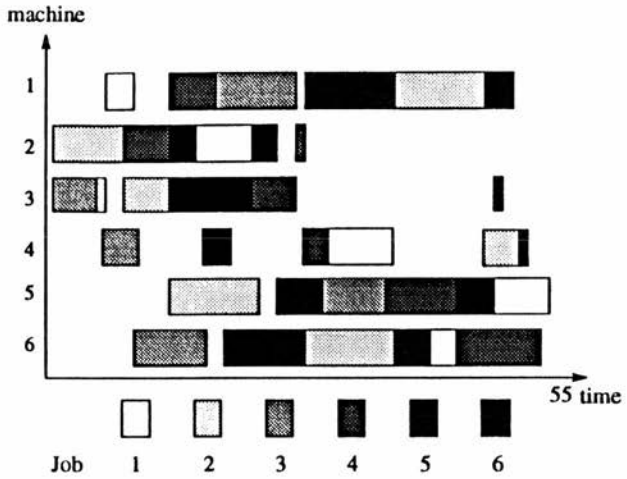

Figure 7.1: Makespan for the 6x6 JSSP

There are two similar benchmarks, of sizes 10×10 and 20×5, for the makespan criterion [Fisher & Thompson 63]. Some branch & bound methods produce good makespan results but may take considerable computer time because of the significant amount of schedule generation implicit in these methods. The shifting bottleneck method [Adams *et al.* 88] also produces good makespan results but is quite criterion-dependent, so it may be not so robust with other criteria. Furthermore, an industrial survey paper by [Panwalkar *et al.* 73] found that due-date based criteria are the most important ones in most companies, and that the most popular makespan criterion in the literature is inapplicable to the dynamic scheduling situation, because incoming jobs normally undermine previous efforts to minimise makespan with regard to earlier jobs. Therefore, in this chapter we will investigate how to use an indirect representation in GA to produce good results for static job-shop scheduling problems and show how they can be adapted to dynamic deterministic/stochastic scheduling for various criteria besides makespan without much effort.

7.1.1.2 Job-Shop Rescheduling Problem (JSRP)

Job-Shop Rescheduling Problems (JSRPs) arise from the continual need to alter previously worked out schedules in the light of problems which arise. This typically means revising the expected processing time for some job in the schedule, or revising (typically delaying) the start time for a particular task. Rescheduling is more common in scheduling than in timetabling because of the different kinds of problems. In timetabling, most constraints are known in advance, and unlikely to change, unless, for example, an exam room blows up during an exam. Furthermore, in our timetabling representation, retimetabling is simply to preset time and room by fixing most of the time or room slots, because each allele represents a slot or room directly. In scheduling, it is much more possible for things to go wrong – more jobs may be introduced at short notice (the equivalent is very rare is timetabling), machines may break down (more likely than an exam hall catching fire), and operations may take longer than expected (but exams or lectures do not change their length) or be delayed and so on. Also, in our scheduling representation, it is hard for people to preset the allele directly because of the indirect representation of chromosome and it is impossible for a person to predict which operation should happen at which time. There is thus a need for efficient methods of rescheduling.

To summarise, rescheduling is made necessary by changes that occur in the environment. A system can respond in three ways:

1. *schedule again from scratch:* Throw away the previous schedule, and simply run the program again with the altered data/constraints. This will take as long as the previous run and is usually unsuitable for two reasons. First, it simply seems unduly wasteful to start again from scratch, especially on a large and difficult problem. Second, the previous schedule may already be in operation, placing further constraints on possible new schedules.

2. *repair the schedule where the changes occur:* This begins with a complete schedule of unacceptable quality and iteratively modifies it until its quality is found to be satisfactory. The disadvantage is that repair methods could suffer from local minima in the sense that they can cycle indefinitely through a set of un-

satisfactory solutions. Another problem is that repair methods are usually not complete and therefore not guaranteed to encounter the best possible solution [Zweben *et al.* 92].

3. *remove some tasks from previous near optimal schedule and reschedule from an intermediate state:* In this approach, much of the previous schedule is retained, while a new, smaller, scheduling problem is considered which involves only those jobs affected by the change or changes which lead to the need to reschedule.

Rescheduling from scratch is obviously to be avoided in the light of the large processing time required for large problems and the frequency of the need to reschedule. Our representation and schedule builder, however, lend themselves naturally to the third method, in which we make a smaller scheduling problem, via a simple *dependency analysis* which finds out which tasks in the existing schedule are affected by the changes. For rescheduling a schedule which is already under way but has been undermined by some change in a job yet to be processed, we begin with that part of the current schedule as yet unperformed.

Two kinds of situation are usually dealt with: a change in the *processing time* of some task (which includes the case of removing a task entirely), and a change in the *start time* of some task (if, for example, a task must be delayed because of problems with a machine or delays in obtaining resources). Input to the reschedule builder is simply the genome representing the schedule which must be altered. The user then enters the required modification (to the processing time and/or start time of one or more tasks).

7.1.2 Applying the Basic GA Framework to JSSP/JSRP

7.1.2.1 Representation

The representation we choose is the same as we described in the previous chapter. The genotype for a $j \times m$ problem is a string containing $j \times m$ chunks, each chunk being large enough to hold the largest job number (j). It provides instructions for building a legal schedule as follows; the string of chunks $abc\cdots$ means: put the first untackled task of the a-th uncompleted job into the earliest place where it will fit in

the developing schedule, then put the first untackled task of the b-th uncompleted job into the earliest place where it will fit in the developing schedule, and so on. The task of constructing an actual schedule is handled by a schedule builder which maintains a circular list of uncompleted jobs and a list of untackled tasks for each such job. Thus the notion of "a-th uncompleted job" is taken modulo the length of the circular list to find the actual uncompleted job.

7.1.2.2 Fitness

The most common fitness function used is to achieve the objective of minimising makespan or maximum complete time (C_{max}). However, in real world practice, if each job's importance is different, we can assign a corresponding weight to each job and then incorporate this within the basic criteria described in section 6.4.3. For example, suppose $i = 1..j$ represents the job number, r_i is the ready time, d_i is the due-date, W_i is the weight and C_i is the completion time of job i. Let $F_i = C_i - r_i$, $L_i = C_i - d_i$, $T_i = \max(0,L_i)$, $E_i = \max(0,-L_i)$, then some of the weighted criteria are as follows:

- *Weighted Flow Time* (F_{weight}):

(7.1) $$F_{weight} = \sum_{i=1}^{j} W_i * F_i$$

- *Weighted Mean Flow Time* ($\bar{F}_{weight}$):

(7.2) $$\bar{F}_{weight} = \sum_{i=1}^{j} (W_i * F_i) / \sum_{i=1}^{j} W_i$$

- *Weighted Lateness* (L_{weight}):

(7.3) $$L_{weight} = \sum_{i=1}^{j} W_i * L_i$$

- *Weighted Tardiness* (T_{weight}):

(7.4) $$T_{weight} = \sum_{i=1}^{j} W_i * T_i$$

- *Weighted Number of Tardy Jobs* (NT_{weight}):

(7.5) $$NT_{weight} = \sum_{i=1}^{j} W_i * n_i$$

if T_i=0 then n_i=0 else n_i=1.

- *Weighted Earliness* (E_{weight}):

$$E_{weight} = \sum_{i=1}^{j} W_i * E_i \tag{7.6}$$

In addition, combinations of the criteria are also possible, for example:

- *Weighted Flow Time plus Weighted Tardiness* (FT_{weight}):

$$FT_{weight} = \sum_{i=1}^{j} (W_i * F_i + W_i * T_i) \tag{7.7}$$

- *Weighted Earliness plus Weighted Tardiness* (ET_{weight}):

$$ET_{weight} = \sum_{i=1}^{j} (W_i * E_i + W_i * T_i) \tag{7.8}$$

In the case of FT_{weight} or ET_{weight}, or any criterion involving combinations of different component criteria, the job weighting W_i used for each component will usually be different. For example, it may be very undesirable for job i to be late, but we may be indifferent to it being early. The earliness/tardiness criterion is a non-regular criteria which often happens when we do not want jobs completed too early or too late. In other words, we want jobs to be finished just in time (JIT). This is a difficult but important criterion in practice, but few heuristic rules can satisfy this requirement currently.

We will test both unweighted and weighted criteria later in this section.

7.1.2.3 Schedule Builder

The schedule builder is the same as we described in the previous chapter which is straightforward, and must consider four cases when slotting a task into a developing schedule. That is, the combinations of suitable gap or not and idle time needed or not. If there is a suitable gap in the schedule, it may be possible to fit the task in there with or without compulsory idle time; and there may be no suitable gap, so that task has to be added to the end of the machine's schedule with or without compulsory idle time. These cases are as described in section 6.4.4.

7.1.3 Specific Framework for JSRP

7.1.3.1 Dependency Analysis

Rescheduling uses a previously developed schedule, plus the previous problem information, and the change specifications. The input schedule is possibly already in use, but for some reason (for example, an operation is delayed) needs to be changed. The rescheduling handles two distinct types of alteration of the original data. The first one is *changing processing time:* this happens when we need to change the processing time of some operations, or we want to remove some operations entirely (by changing its processing time to zero). The second is *shifting operation:* this happens when, for example, operations are delayed, so that the current schedule cannot be used. First of all, in either case, we find the first gene which is directly affected by the change. That is, we find the gene which is interpreted, by the schedule builder, as relating to the earliest operation which must be rescheduled as a result of the change. If we are altering the processing time of an operation, then this is just the gene which is interpreted to directly relate to that operation. If, however, we are moving the start time of an operation, then there are three cases to consider. For example, suppose we are requested to shift job 1 of machine 1 to another time of the original schedule, the ^^^ shows where the shift is required to happen in each case.

```
if the shifted start time is later than the original start time:
      mc1:.....44  222111333.....
                           ^^^
then the first affected gene is the one with the original start
   time, in our case it is job1's operation 111.


if the shifted start time is earlier than the original start time, and
   falls within some other gene's start and finish time:
      mc1:.....44  222111333.....
                    ^^^
then the first affected gene is the one which is interrupted by
   the shifted start time, here it is job2's operation 222.


if the shifted start time is earlier than the original start time, and
   does not fall within some other gene's start and finish time:
      mc1:.....44  222111333.....
                  ^^^
then the first affected gene is the first one after the shifted
   start time, in our case it is also job2's operation 222.
```

In each case we then do *dependency analysis* once and shift all the unaffected genes and the user-shifted genes if they exist to the front of the gene string and shift all the affected genes to the rear of the gene string. The dependency analysis just marks the gene where the first change happens as the current gene and marks the following genes which are affected by current gene either because of their precedence constraint for the same job or capacity constraint for the same machine and then steps forward to the next marked gene, sets it as current gene and repeats the same procedure until there are no more left. In short, we find all of the operations, and thereby all of the genes, directly or indirectly affected by the initial change. An example to illustrate dependency analysis is as follows:

Index	...	24	25	26	27	28	29	30	31	32	33	34	35	36
job	...	3	1	4	5	2	1	6	4	5	2	6	1	5
machine	...	5	4	5	6	1	6	5	6	1	4	3	5	4
start	...	31	31	38	39	39	43	46	46	49	49	50	50	53
finish	...	37	37	45	42	48	45	49	54	51	52	50	55	53

The above figure indicates part of a chromosome for a 6x6 JSSP. 'Index' refers to the position (gene) in the chromosome, while the other rows indicate the operation referred to by the allele at this position (as interpreted by the schedule builder), given by 'job' and 'machine', and the start and finish times assigned to this operation. Consider a case in which the duration of the fourth operation in job 1 (the 25th gene) is changed; call this the 'current operation'. This will affect the 29th and 35th genes because they are the successors of current operation in the same job, and the 33rd and 36th genes because they will use the same machine as the current operation; recursively, any change to the 29th gene will affect the 31st gene similarly, and so on. After finishing the dependency analysis, we can shift the (in this case) six affected genes (25th, 29th, 31st, 33rd, 35th, and 36th) to the rear part of the gene string so that their indices are now from 31 to 36. The GA then applies its crossover and mutation operations only to the last six genes, keeping the front 30 genes unchanged. The system also keeps all the necessary information (for example, how many times each job already appears, and the latest finish times of each job in the front part) applying them to the rear part of all the other schedules.

This method does not guarantee an optimal new schedule; the GA, of course, never guarantees optimality anyway, but the point is that the retention of a fixed (unaffected) portion of the previous (near) optimal schedule might preclude the discovery of an optimal schedule which might otherwise be possible to find by rescheduling from scratch. The strength of this rescheduling method, however, lies in its speed. There is thus a tradeoff between the speed in which a good new schedule can be found via retaining parts of the previous schedule, and the potential advantage of rescheduling from scratch with the (probably low) possibility of evolving a significantly better schedule.

7.1.3.2 Reschedule Builder

After finishing the dependency analysis, we have shifted all the unaffected genes or the predefined shift genes to the front of the gene string. Now our gene string has two parts:

- *Front Part:* Includes all the unaffected genes and the shift duration genes if they exist.
- *Rear Part:* Includes all the affected genes and the change processing time genes if they exist.

The algorithm used by the reschedule builder is the same as that used by the schedule builder, but it operates only on the rear part genes, using known information from the front part. Because all the schedules' front parts are considered the same, we do not need to duplicate our effort to schedule the front part each time. In other words, the reschedule builder does partial scheduling instead of full scheduling; this will save us much time.

7.1.4 Performance Enhancement

On hard problems like the JSSP, GA researchers routinely need to use either problem-specific or problem-type specific performance enhancements to improve performance. These enhancements are interesting because of the light they shed on the dynamics of the GA approach and the aspects of problems which make it hard or easy for GAs

to solve them. For example, [Nakano 91]'s representation is highly redundant (with $2^{mj(j-1)/2}$ genomes representing approximately $j!^m$ distinct schedules) and so leads to the possibility of false competition among genotypes, in which different representations of the same schedule compete against one another, possibly to yield inferior descendants which combine aspects of their parents' representations which do not translate into good building blocks. Nakano combats this with *forcing*, in which he replaces illegal genotypes in the pool with their 'nearest' legal matches. This forces a one-to-one genotype/schedule mapping in a gene pool, eliminating false competition in the selection step (although still typically resulting in illegal schedules after crossover). Nakano hence uses a highly redundant representation with a complex evaluation technique for the basic GA, and then significantly improves performance by using forcing to reduce false competition.

Our approach does not require forcing, since the representation always encodes legal schedules, but there is high redundancy (though less high than Nakano's), and we similarly need a way of countering false competition. One determiner of the extent of this is the ratio p/s, where s is the number of distinct legal schedules and p is the population size; this is a measure of the degree to which the same schedule might occur (independently of whether or not it is multiply represented) in a given population. Normally, this is far too small to be significant, but because our representation is highly context sensitive (because it is a sequence of instructions for the schedule builder) the front parts of the genotype converge more quickly than later parts. This means that s is effectively reduced so much with time and 'lateness' in the genome (early-part stabilisation leads to a smaller subspace being explored in the later part) that p/s becomes significant. This effect is increased by the greater redundancy of the tail of the genome; since the circular list of jobs gets shorter as we traverse the genotype, there will be many more ways to represent the same job. We can visualise the effect of this in Figure 7.2, in which we can clearly see gradually decreasing convergence speed as we traverse the chromosome from left to right. This figure shows a plot of the *variance* of each chunk of the genotype within the pool (size 500) with its position in the genotype, and with generation as the GA operates, for 300 generations of a run on the 10×10 benchmark problem.

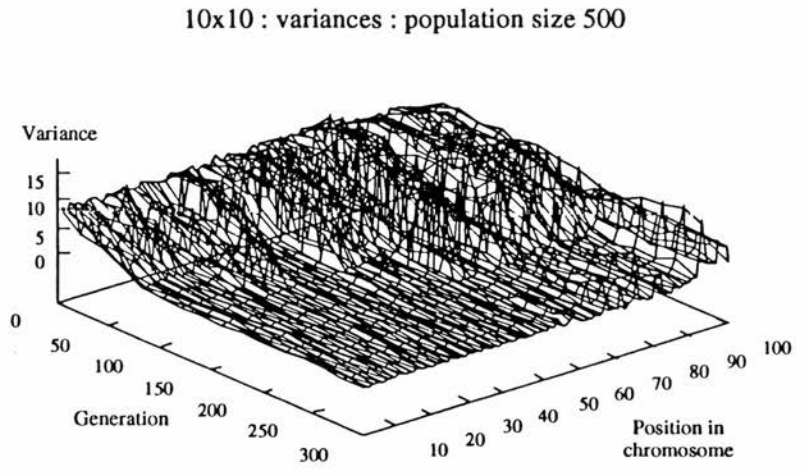

Figure 7.2: Plot of variance of chunk using 1-point crossover, 10x10 JSSP

7.1.4.1 Gene-Variance Based Operator Targeting (GVOT)

As Figure 7.2 shows, gene convergence rates fall fairly smoothly as a function of position in the genotype. This kind of behaviour should be typical of GA problems where the chromosome representation is context sensitive in the sense that there is a variation in 'significance' across the chromosome. In the JSSP case, in which large scale problems not only cost significant computational time, but in which the solutions produced might significantly affect profits and/or product quality, we should be able to exploit this effect by using it to inform ways of increasing overall convergence speed and/or solution quality. In [Fang *et al.* 93], we describe a gene-variance based operator targeting strategy, which is a principled first attempt at doing just this, by making sure that genetic operators are concentrated where and when they seem to be most 'needed'. The situation exhibited in Figure 7.2 suggests a strategy to improve solution quality.

- The faster stabilisation of early parts of the genome suggest premature convergence. Increasing mutation probability at fast converging sites may thus im-

prove performance; also, this measure should obviate 'wasted' mutation in later, slow-converging parts of the schedule which are still in relatively early stages of exploration.

- We can expect crossover at early, more stable positions to have little or no effect on the genotype pool, since it is likely we are to produce children which are simply replicas of the parents. So, encouraging crossover at later, less stable positions should lead to more effective exploitation.

For instance, if we have five chromosomes as follows, we would like to encourage mutation on front genes (for example, gene 1 and 2) because they converge fast, and crossover on rear genes (for example, gene 8 and 9) because they converge slowly.

```
gene position : 1 2 3 4 5 6 7 8 9
-----------------------------------
 chromosome-1 ( 2 3 1 1 3 1 2 3 1 )
 chromosome-2 ( 2 2 2 2 1 1 1 2 1 )
 chromosome-3 ( 2 3 1 1 3 3 3 1 2 )
 chromosome-4 ( 2 3 2 3 2 2 3 1 3 )
 chromosome-5 ( 2 3 2 1 3 3 2 2 3 )
-----------------------------------
```

A principled way of implementing these effects is what we term *gene-variance based operator targeting* (GVOT). This works by sampling the variances of genes at each position of the genotype in a pool, and choosing the actual point of crossover or mutation based on these variances. Crossover position is selected probabilistically but according to allele variance, while order-based mutation positions are selected according to the inverse of allele variance. Hence, high variance sections are more likely to be chosen for crossover; low variance sections for mutation.

This can be seen as a specific instance of an idea which should be of more general use in GA performance enhancement on hard problems. In many other kinds of problem we can't expect smooth changes in variance across the genotype; this would not occur in the JSSP, for instance if (unusually) task processing times were to grow as a function of advancing position in the job sequence so that the tail of the genome would then be much more significant in determining the overall makespan. However, whenever significant variation in convergence rate does occur (smooth or not), the GVOT strategy,

```
Insert 1 unit idle time:
     mc1:    111       22222
     mc2:1111                ^22
     mc3:       1111112
Insert 2 units idle time:
     mc1:    111       22222
     mc2:1111                ^^22
     mc3:       1111112
Insert 3 units idle time:
     mc1:    111       22222
     mc2:1111                ^^^22
     mc3:       1111112
...
...
...
```

Table 7.1: Idle time in the tail of schedules

targeting operators solely on the basis of dynamically sampled variance, should work just as well.

7.1.4.2 Evolving Idle Time Within Gaps (EIWG)

In chapter 6, we noted that the inclusion of excess (non-compulsory) idle time leads to inadmissible schedules. In particular, when the operation being delayed by such idle time is at the tail of a partial schedule, we might insert arbitrarily large amounts of idle time. For example in Table 7.1 we can insert any amount of idle time before job2's last operation in machine 2 because it is in the tail of a schedule. However, if the operation to be scheduled falls within a gap (for example, see Table 7.2), and if there is more than enough room for the operation in this gap, then there are a finite number of possible choices of idle time we might add to delay the operation within this gap. We name such kinds of excess idle time *optional idle time* from now on, to distinguish them from the compulsory idle time mentioned in the previous chapter.

In scheduling theory, optional idle time is not necessary because it belongs to the domain of inadmissible schedules and we know that the optimal solution is guaranteed to fall within the domain of active schedules. However, in our representation or in some other GAs, it is sometimes very easy to get stuck in a local minimum and converge

```
No optional idle time: (idle time = 0, 5, 10 ...)
     mc1:....... 111333333
     mc2:.... 222   ##    33333...
1 optional idle time: (idle time = 1, 6, 11 ...)
     mc1:....... 111333333
     mc2:.... 222    ##   33333...
2 optional idle time: (idle time = 2, 7, 12 ...)
     mc1:....... 111333333
     mc2:.... 222     ##  33333...
3 optional idle time: (idle time = 3, 8, 13 ...)
     mc1:....... 111333333
     mc2:.... 222      ## 33333...
4 optional idle time: (idle time = 4, 9, 14 ...)
     mc1:....... 111333333
     mc2:.... 222       ##33333...
```

Table 7.2: Optional idle time within gaps

quickly. The insertion of optional idle time can lessen the convergence speed because more schedule alternatives exist. In a sense, this also gives the GA more room to manoeuvre, occasionally making it easier for operators to lead to better schedules. For example, assume A and B are two genes (operations) whose processing times are 10 and 5 respectively and further assume that B before A can produce a better schedule than A before B. In the simplest case (without optional idle time), we must evolve schemata which lead to B being scheduled before A; this may well be particularly hard if schemata which lead to the reverse order, A before B, tend to be comparably fit. Via *genetic drift* [Harvey 93], the population may well converge to 'A before B' schedules. This will be bad luck, because it may require several fortunate simultaneous mutations of different genes to recover any 'B before A' schemata. However, if we also encode genes for varying amounts of optional idle time, then an 'A before B' schema can be changed to encode a 'B before A' schedule simply by making A's optional idle time 5 or more units longer than B's optional idle time; when B is scheduled, it will then fit into the gap before A.

In the representation we have used so far, only the sequence of the genes decides the quality of the schedule and how quickly it converges according to the variance graph. Inserting optional idle time provides an alternative to produce a similar effect to the sequencing of genes, in other words both can affect the quality of the schedule. Because

inserting optional idle time into the tail of a partial schedule may lead to infinitely many schedules, we only consider inserting optional idle time within gaps, so that there is a finite search space. If we further treat the idle time circularly, then we need not worry about conflicts between the idle time encoded for an operation and the size of the relevant gap (if any).

We include optional idle time in a chromosome by simply adding an extra gene for each operation. The optional idle time genes evolve with the job-operation genes simultaneously, but are only used when the current operation is not to be appended to the end of the partial schedule. Otherwise, it is ignored. In other words, the representation now uses two parts $ABCD...$ and $abcd...$ meaning: decide which operation of the Ath uncompleted job to place next and put it into the earliest place it will fit in the developing schedule, with or without a units of optional idle time, decide which operations of the Bth uncompleted job to place next and put it into the the earliest place it will fit in the developing schedule, with or without b units of optional idle time, and so on. In a large enough gap, for example if the size of the gap is larger than the sum of the current operation's processing time and its accompanying idle time, the size of the idle time is not a problem because it will not overlap with later operations; however if the gap is not large enough or the idle time is too large, then it is a different story. The user still can specify a large number as the largest possible idle time in the beginning, in practice the largest processing time can be used because it is large enough to let any later operation be inserted before this operation in this gap. The choice of how much optional idle time to use in constructing an actual schedule is handled by the schedule builder which maintains a circular list of largest idle time within each gap. Therefore, even if the user specifies a very large number as the largest possible idle time, the search space is still finite because of the circularity of the idle time list. Thus the notion of "a units of idle time" is taken modulo the length of the maximum possible idle time in the appropriate gap to find the actual idle time needed.

For example, in Table 7.2 if the next operation is job1 in machine 2 with 2 unit processing time, **##**, then the amount of the optional idle time in this operation's associated idle time gene decides which position to put **##** in machine 2. As we can see, the maximum possible idle time that can be used in this gap is 4, so whatever the

value is provided by the idle time gene for this operation, it is interpreted modulo 4.

7.1.4.3 Delaying Circularity Using Virtual Genes (DCVG)

In chapter 6, we mentioned about [Grefenstette *et al.* 85]'s TSP *ordinal representation* having the following disadvantage: in each generation, alleles after the crossover point will definitely change their meaning. We also pointed out that the 'tail randomness' of our representation is not dependent on the crossover point but on the first job which finished all its operations. Therefore, the more operations the jobs have, the later randomness starts to affect the chromosome. This gives us a hint; if we can encode some 'virtual' genes in the tail of the chromosome, we will delay this loss of meaning, hence allowing the GA to perform more effectively.

We repeat the same example as in section 6.4.1 here for illustration:

```
chromosome                job sequence
----------                ------------
                                    *->
(1 2 3 3 2 2 2 3 1)       (1 2 3 3 2 2 3 1 1)
(1 1 2 3 3 1 2 1 1)       (1 1 2 3 3 1 3 2 2)
                                    *->
```

First, let's just interpret the chromosome itself. Both chromosomes become circular after the third last gene. That is, the last three genes may lose some of their meaning in both chromosomes.

Now we can introduce six virtual genes by giving each job two extra operations of zero length. Consider the following two longer chromosomes:

```
chromosome                        job sequence
----------                        ------------
                                                *->
(1 2 3 3 2 2 2 3 1 1 3 1 2 3 3)   (1 2 3 3 2 2 2 3 1 1 3 1 2 1 3)
(1 1 2 3 3 1 2 1 1 3 1 2 2 3 1)   (1 1 2 3 3 1 2 1 1 2 2 3 3 2 3)
                                            *->
```

The first chromosome becomes circular after the second last gene and the second chromosome becomes circular after the sixth last gene. However, not all of the genes are

position	1-3	4	5	6	7	8-9
probability	1.0	0.8889	0.6667	0.3704	0.1235	0.0

Table 7.3: Probability of preserving full meaning for 3x3 JSSP

actually used by the schedule builder because some of them are virtual genes (that is, they get interpreted as zero-length operations). We can think of the virtual gene (? - question mark shown below) as a virtual machine with zero processing time in the tail of each job, so that it will not affect the criteria we want to optimise.

```
chromosome                          job sequence
----------                          ------------
                                                          *->
(1 2 3 3 2 2 2 3 1 1 3 1 2 3 3)     (1 2 3 3 2 2 ? 3 1 1 ? ? ? ? ?)
(1 1 2 3 3 1 2 1 1 3 1 2 2 3 1)     (1 1 2 3 3 1 2 ? ? 2 ? 3 ? ? ?)
                                                    *->
```

Now, no 'real' gene in the first chromosome falls within the last two circular genes and only two real genes in the second chromosome fall within the last six circular genes. That is, no real genes lose any of their meaning in the first chromosome and only two real genes may lose some of their meaning in the second chromosome. So, both of the longer chromosomes contain fewer circular genes than those chromosomes without using virtual genes.

We can also roughly calculate the probability of preserving each gene's full meaning using equation 6.1 for these two cases. Comparing Tables 7.3 and 7.4, the probability of preserving its full meaning becomes less than 0.5 for the sixth gene in the former one and the 11th gene in the latter one. Also, the latter cannot guarantee to be legal from the 6th gene but the former only from the 4th. Since virtual operations are added to the end of a job's operations; and hence are to be 'scheduled' later than any of their job's real operations, they will tend to occur mostly towards the tail of the chromosome. As a result, 'real gene' will tend to avoid the tail, strengthening the chances of preserving their meaning. A further example involving crossover will support this observation.

Suppose we crossover the parents in our continuing example using two-point crossover at the points marked below, giving the following two children:

position	1-5	6	7	8	9	10	11	12	13	14-15
probability	1.0	0.9877	0.9465	0.8642	0.7362	0.5655	0.3734	0.1956	0.0652	0.0

Table 7.4: Probability of preserving full meaning for 3x(3+2) JSSP

```
chromosome                    job sequence
----------                    ------------
                                        *->
(1 2 3 3 3 1 2 3 1)           (1 2 3 3 3 1 2 1 2)
(1 1 2 3 2 2 2 1 1)           (1 1 2 3 2 2 3 1 3)
      ^   ^                             *->
```

The genes in these children begin to lose full meaning after the seventh gene in the first chromosome and eighth gene in the second chromosome compared with the meaning of their parent chromosomes. That is, two and three genes may lose some of their meaning in the first and second chromosomes respectively.

Suppose, we do the same with the 'virtual genes' example, again using crossover points one third and two thirds of the way along the chromosomes. Then we get the following child chromosomes and job sequences:

```
chromosome                          job sequence
----------                          ------------
                                              *->
(1 2 3 3 2 1 2 1 1 3 3 1 2 3 3)     (1 2 3 3 2 1 2 1 1 3 3 1 3 2 2)
(1 1 2 3 3 2 2 3 1 1 1 2 2 3 1)     (1 1 2 3 3 2 2 3 1 1 1 3 3 2 2)
          ^           ^                       *->
```

The first chromosome loses full meaning after the 10th gene and the second chromosome after the 11th gene. Replacing virtual genes by '?' below:

```
chromosome                          job sequence
----------                          ------------
                                              *->
(1 2 3 3 2 1 2 1 1 3 3 1 2 3 3)     (1 2 3 3 2 1 2 1 ? 3 ? ? ? ? ?)
(1 1 2 3 3 2 2 3 1 1 1 2 2 3 1)     (1 1 2 3 3 2 2 3 1 ? ? ? ? ? ?)
          ^           ^                       *->
```

We see that just one real gene in the first chromosome loses its meaning and none in

the second chromosome. As shown above, most of the genes which may change their meaning are virtual genes which are not used by the schedule builder.

In short, the introduction of virtual genes can help delay the randomness of the chromosome. Most of the randomness happens at virtual genes, which will not affect the schedule quality. The cost to pay is the extra length needed to encode the virtual genes and possibly some useless crossover and mutation. However, we will show that this can improve the solution quality on average.

This method is especially useful when the number of operations is small, for example in the 20x5 problem of [Fisher & Thompson 63], we can get the optimal solution 1165 in 484 generations by looking on it as a 20x20 problem, and the average solution quality is also improved.

7.1.5 Experiments

7.1.5.1 Static Scheduling

The most famous criteria in static scheduling literature is [Fisher & Thompson 63]'s benchmark problems in makespan: C_{max}. Previous research in these problems is shown in Table 7.5.

Paper/Report	Algorithms	6 x 6	10 x 10	20 x 5
Balas 69	Branch & Bound	55	1177	1231
McMahon & Florian 75	Branch & Bound	55	972	1165
Baker & McMahon 85	Branch & Bound	55	960	1303
Adams *et al* 88	Shifting Bottleneck	55	930	1178
Carlier & Pinson 89	Branch & Bound	55	930	1165
Nakano 91	GA	55	965	1215
Yamada & Nakano 92	GA	55	930	1184
Croce *et al* 92	GA	55	946	1178
Dorndorf & Pesch 93	GA	55	938	1178
Atlan *et al* 93	GA	55	943	1182

Table 7.5: Published makespan benchmark results

The 6x6 JSSP is the simplest one, in which the optimal solution can be found by most techniques. We introduce some methods we have used here, and test on 10x10 and 20x5 benchmark problems. The GA combinations we used are as follows:

- *UX*: using uniform crossover instead of 1-point or 2-point crossover.
- *GVOT I*: 50% GVOT and 50% uniform crossover.
- *GVOT II*: 80% GVOT and 20% uniform crossover.
- *EIWG I*: maximum idle time equal to one fifth of the largest processing time.
- *EIWG II*: maximum idle time equal to the largest processing time.
- *DCVG I*: add m virtual genes to each job.
- *DCVG II*: add $j - m$ virtual genes to each job.
- *MIXED I*: DCVG I plus 100% GVOT.
- *MIXED II*: DCVG II plus 100% GVOT.

The results in Table 7.6 and Table 7.7 all involved 20 runs, population sizes of 200, elitist generational-reproduction, and marriage tournament selection [Ross & Ballinger 93] with tournament size 5. Crossover rate began at 0.8, decreasing by 0.0005 each generation and stopped at 0.3. Order mutation rate was 0.5 in the 1-point, 2-point, UX, and EIWG experiments, and 1.0 in the GVOT, DCVG, and MIXED experiments. In the latter case, the higher mutation rate was chosen to balance the effect of having a longer chromosome in the DCVG case and/or less disruptive crossover in the GVOT case.[1] Runs continued for at most 1,000 generations, but were stopped immediately if there was no improvement in the best chromosome after 500 successive generations. We did not attempt to optimise speed or make use of any problem dependent knowledge such as using a very good seed in our initial population. However, our method is very robust and it can be adjusted to many criteria easily.

First, Figure 7.3, calculated from equation 6.1, tells us that the number of job j has a negative effect and the number of machine (operation) m has a positive effect on preserving full meaning. Further, from the results in Tables 7.6 and 7.7, we also find that performance has some relationship with when the gene is circular. In other words, if we have more jobs, then there is more chance that circularity appears soon.

[1] When using GVOT, the chance of a gene to be swapped between parents is proportional to its variance in the population; except for early on in the GA run, this is usually less than 0.5.

Method	Worst	Best	Average	Std-Dev	95% Confidence Interval
1-p crossover	1084	985	1019.8	26.5	(1007.4,1032.3)
2-p crossover	1051	966	1001.8	22.9	(991.0,1012.5)
UX	1008	953	978.4	16.9	(970.5,986.3)
GVOT I	986	948	968.5	12.7	(962.5,974.4)
GVOT II	993	940	970.4	15.1	(963.3,977.5)
EIWG I	1009	939	980.1	18.7	(971.3,988.7)
EIWG II	1031	957	989.2	21.2	(979.3,999.1)
DCVG I	1006	955	980.8	13.1	(974.6,986.9)
MIXED I	1001	953	973.3	11.9	(967.7,978.8)

Table 7.6: 10x10 Benchmark Problem (optimal solution = 930)

Method	Worst	Best	Average	Std-Dev	95% Confidence Interval
1-p crossover	1300	1211	1257.8	27.7	(1244.8,1270.8)
2-p crossover	1306	1223	1257.3	24.1	(1246.0,1268.6)
UX	1237	1189	1210.3	14.4	(1203.5,1217.0)
GVOT I	1216	1186	1202.8	6.5	(1199.8,1205.9)
GVOT II	1239	1191	1208.0	11.3	(1202.7,1213.2)
EIWG I	1239	1190	1209.3	13.0	(1203.2,1215.3)
EIWG II	1254	1184	1208.0	17.4	(1199.8,1216.1)
DCVG I	1233	1173	1197.7	15.0	(1190.7,1204.7)
DCVG II	1233	1165*	1194.4	14.3	(1187.7,1201.1)
MIXED I	1216	1173	1192.3	9.7	(1187.7,1196.8)
MIXED II	1207	1175	1190.7	11.1	(1185.5,1195.8)

Table 7.7: 20x5 Benchmark Problem (optimal solution = 1165)

Whereas, if we have more operations, then there is less chance of early circularity. On the 10x10 problem, GVOT I and GVOT II give the best average performance and the 95% confidence interval derived from Student's t test. Neither DCVG nor EIWG seems to have any beneficial effect. On the 20x5 problem, however, DCVG gives a particularly clear improvement in best and average over the 20 trials, while the most effective method on average is MIXED II, combining GVOT and DCVG. The 95% confidence intervals derived from Student's t test back up these observations. Note, for example that the interval for MIXED II is further to the left than all of the others. It is not hard to see why DCVG is more useful in 20x5 than in 10x10, for the former has more jobs and less operations per job than the latter. In other words, the genes will

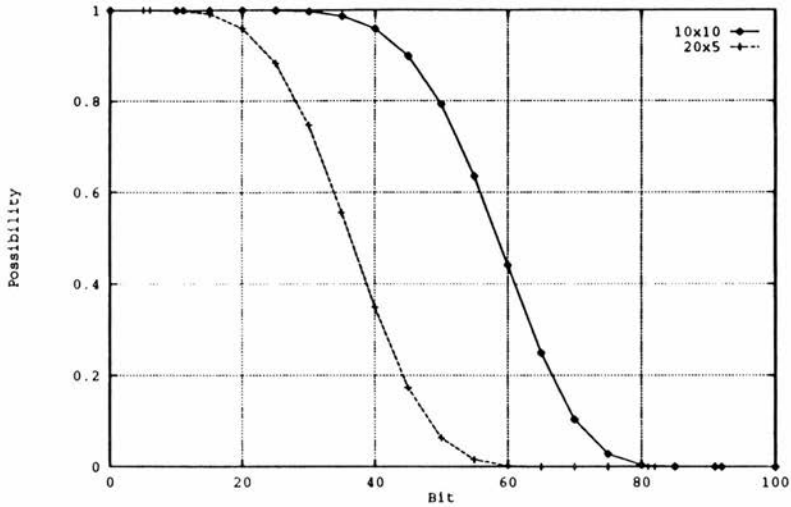

Figure 7.3: Probability of preserving full meaning for 10x10 vs 20x5 JSSPs

lose their meaning earlier in the former than the latter. The motivation behind DCVG is to delay this effect, so it is no surprise that it works best when the loss of meaning is more prominent to start with, which is the case when the number of operations per job decreases. GVOT tries to mutate more on the front part because it converges quickly and crossover more on the rear part because it converges slowly. The variance change in 10x10 is smoother than 20x5, therefore it is not hard to see why GVOT performs better in 10x10 than in 20x5. In addition, a mixture of DCVG and GVOT can improve the solution more than just using DCVG because the latter part of the chromosome in DCVG contains more virtual genes, so it is less meaningful. The mixed method gives us the advantage of both DCVG and GVOT. Finally, EIWG improves on UX a little, but is not a great success on these problems. EIWG seems more effective on average on the 20x5 problem, although it hit upon the best of all trials on the 10x10 problem.

Though GVOT, EIWG and DCVG are all ideas to, via different methods, discourage premature convergence, all need to pay extra computational cost to gain a small

percentage improvement and none seem able to guarantee to improve solution quality very much. The increased average solution quality may not be enough to warrant the extra time spent by these methods. For example:

- *GVOT*: needs to calculate gene variances and choose the position according the variance.

While with DCVG and EIWG, the schedule builder itself also takes longer than with UX. However, it should also be pointed out that these methods might show more effective improvement over UX if allowed more time. For example:

- *EIWG*: needs to introduce the extra genes to represent the idle time, and hence needs more time to search the increased number of combinations.
- *DCVG*: needs to introduce virtual genes to delay the circularity, again lengthening the chromosome.

Therefore, if we have enough time left and we know all the required data in advance (for example, with static scheduling), we can try these three in order to improve the solution quality. However, most real world scheduling problems are dynamic and possibly stochastic, and rescheduling may often be necessary. In other words, it is less meaningful to try to obtain small percentage improvement because the data itself is probably inaccurate. This is because it may be estimated (stochastic scheduling) and/or may be changed (rescheduling). What we need most is to be able to get a near optimal solution with least computational time. In this consideration, we found that uniform crossover with marriage tournament selection has a similar effect to GVOT, EIWG and DCVG. Therefore, in the dynamic case which we study next, we mostly use marriage-tournament selection with uniform crossover as our basic GA.

7.1.5.2 Dynamic Deterministic Scheduling

The dynamic scheduling reported here involved non-zero ready time and different weights for each job; the benchmark data set of dynamic scheduling problems is taken

Problem	BENCHMARK	RND	FASFS	SPT	EQD	EOD	EMOD	MST	S/OP	GA(ux)
JB1	17	17	20	20	20	17	20	17	17	17*
JB2	13	13	21	40	13	22	22	13	13	13*
JB4	63	63	63	63	63	63	63	63	63	47**
JB9	54	105	114	182	54	54	182	73	73	40**
JB11	5	7	15	27	9	5	5	13	13	0**
JB12	69	69	69	69	69	69	69	69	69	58**
LJB1	55	56	96	66	77	55	55	77	77	48**
LJB2	115	194	164	195	123	118	243	124	124	112**
LJB7	141	225	213	110*	179	161	110*	152	152	136
LJB9	222	445	392	684	240	219	688	214*	214*	323
LJB10	92	161	154	112	92	84*	104	107	107	94
LJB12	88	213	244	190	88	84*	189	85	85	120
FL1	16	48	60	31	20	28	31	16	16	16*
FL3	40	40	40	40	40	40	40	40	40	40*
FL5	51	63	113	101	64	71	101	47	47	45**
FL7	71	110	127	163	71	83	149	72	72	71*
FL9	117	153	159	166	122	128	166	120	120	114**
FL11	39	56	82	142	42	57	86	42	42	11**
LFL1	84	113	131	107	96	92	107	94	94	74**
LFL3	108	132	141	176	108	108	176	120	120	108*
LFL5	147	161	199	246	147	147	147	147	147	147*
LFL7	77	82	105	124	77	82	105	77	77	77*
LFL9	169*	269	249	644	169*	263	497	169*	169*	232
LFL11	114	120	140	140	114	120	140	115	115	107**

Table 7.8: T_{max} for FSSP and JSSP

	BEST		*WORST*		*AVERAGE*		*STD-DEVI*	
Problem	UX	MIX	UX	MIX	UX	MIX	UX	MIX
Ljb7	135	96**	240	186	168	139	35.2	28.8
Ljb9	323	328	456	407	396	370	40.3	24.0
Ljb10	94	87	137	137	117	106	14.5	14.6
Ljb12	120	107	182	172	141	131	18.2	18.1
Lfl9	232	230	318	327	280	273	29.2	33.6

Table 7.9: Comparisons of UX/MIX for T_{max}

from [Morton & Pentico 93]'s accompanied scheduling software *PARSIFAL*. The problems labeled JB? and LJB? are small and large job-shop scheduling problems and FL? and LFL? are small and large flow-shop scheduling problems respectively. The flow-shop problems are odd-numbered problems and the job-shop problems are non-reentrant ones. The heuristic rules R&M, WCOVERT, WMOORE and BENCHMARK are also taken from *PARSIFAL* which modified the original unweighted rules to run in the dynamic case using their special iterated and pricing method. The other heuristic rules are implemented by ourselves for the sake of comparison with our GA methods. We used nondelay schedule generation for each heuristic involved and chose the best of ten runs for comparison. For the GA, the results reported all involve 10 runs, population sizes of 100, and stop running if more than 20 generations cannot improve the solution quality. The other parameters are the same as the static case.

In Tables 7.8 to 7.22, '**' means the best solution the GA found was better than those of all other heuristics listed and '*' means the best solution the GA found was as good

Problem	EMOD	RND	FASFS	SPT	EGD	EOD	MST	S/OP	GA(ux)
JB1	3.4	3.4	3.4	3.4	3.4	3.9	4.4	4.4	3.0**
JB2	2.2	2.2	3.6	4	1.7	2.2	1.7	1.7	1.7*
JB4	16	16	16	16	16	16	16	16	16*
JB9	23	29	40	28	16	17	20	20	12**
JB11	0.3	4.0	2.5	2.1	0.9	0.5	1.1	1.1	0**
JB12	12	12	18	12	12	12	12	12	8.3**
LJB1	16	18	21	16	18	16	20	20	13**
LJB2	26	37	42	28	27	26	29	29	24**
LJB7	18	25	30	19	16	22	16*	17	17
LJB9	68*	106	122	74	119	109	111	111	84
LJB10	25	41	43	30	40	31	46	46	24**
LJB12	25	44	52	31	31	32	35	35	25*
FL1	7.2	6.2	19	7.2	3.8	5.3	3.8	3.8	3.4**
FL3	12	12	14	12	12	12	12	12	11**
FL5	19	18	42	19	23	20	18	18	14**
FL7	25	37	39	30	25	28	28	28	20**
FL9	40	51	61	41	44	45	46	46	31**
FL11	9.8	10	11	11	3.8	5.5	3.0	3.0	1.5**
LFL1	18	24	26	18	20	20	22	22	18*
LFL3	30	34	34	31	33	33	36	36	23**
LFL5	40	42	47	45	39	39	41	41	32**
LFL7	12	13	12	12	12	12	13	13	8.2**
LFL9	80*	70	68	58	53	55	55	55	65
LFL11	34	34	34	34	34	34	35	35	31**

Table 7.10: $\bar{T}$ for FSSP and JSSP

	BEST		*WORST*		*AVERAGE*		*STD-DEVI*	
Problem	UX	MIX	UX	MIX	UX	MIX	UX	MIX
Ljb7	17	13**	27	27	22	19	3.7	3.8
Ljb9	84	85	123	107	101	96	12.6	8.2
Lfl9	65	60	73	72	70	65	2.6	4.1

Table 7.11: Comparisons of UX/MIX for $\bar{T}$

as the best found by other heuristics. If the GA cannot find better solutions than or at least the same quality as the best heuristic rule, then '*' means the best solution found among all other heuristic rules. Furthermore we also applied MIXED I (simply termed MIX hereafter) on those problems when GA(UX) did not perform best. For instance: Table 7.8 shows that GA(UX) can get results at least as good as any other heuristic rules in 19 out of 24 problems. 11 of them are better than those produced by any of the other heuristic rules. We also tried GA(MIX) with the remaining 5 problems on which GA(UX) didn't perform best. In Table 7.9, all the average solutions of MIX are better than those of UX and only 1 out of 5 produced by MIX is worse than UX in best and worse solutions respectively. In addition, the last two columns shows the standard deviations of T_{max} over the trials of these two methods. Further, MIX can produce better results for Ljb7 than any other heuristic involved, so we also marked it using '**'. In short, Table 7.9 suggests that overall MIX is a better choice than UX, since MIX's average is typically of the order of 1/2–1 standard deviation better than UX's and their best are very similar. The criteria, and associated heuristic rules used

Problem	BENCHMARK	RND	FASFS	WSPT	WLWKR	WTWORK	GA(ux)
JB1	716	716	731	716	716	716	716*
JB2	1211	1211	1301	1211	1213	1213	1206**
JB4	701	701	701	701	701	701	698**
JB9	1.20E4	1.54E4	1.48E4	1.20E4	1.44E4	1.54E4	1.17E4**
JB11	8356	8194	8116	8181	7968	8020	7966**
JB12	1.10E4	1.11E4	1.10E4	1.12E4	1.10E4	1.12E4	1.10E4*
LJB1	1195	1206	1215	1201	1193	1199	1138**
LJB2	1.20E4	1.20E4	1.25E4	1.19E4	1.18E4	1.20E4	1.13E4**
LJB7	1.46E4*	1.59E4	1.57E4	1.51E4	1.60E4	1.61E4	1.48E4
LJB9	1.37E4*	1.57E4	1.66E4	1.39E4	1.37E4*	1.55E4	1.51E4
LJB10	3328	3702	3606	3309*	3721	3961	3318
LJB12	4462	4811	4978	4480	4605	4866	4436**
FL1	4841	4848	4944	5020	5044	5030	4754**
FL3	3642	3642	3642	3701	3642	3642	3642*
FL5	7330	8227	9009	7330	7843	7843	7234**
FL7	9138*	1.01E4	1.07E4	9315	9630	9672	9732
FL9	7018	7216	7935	7155	7381	7397	6879**
FL11	1.92E4	1.90E4	1.97E4	1.95E4	1.98E4	2.01E4	1.87E4**
LFL1	6501	6503	6731	6479	6488	6491	6351**
LFL3	1.34E4	1.44E4	1.45E4	1.40E4	1.35E4	1.36E4	1.32E4**
LFL5	2.79E4	2.83E4	2.89E4	2.79E4	2.79E4	2.85E4	2.63E4**
LFL7	3.42E4	3.42E4	3.46E4	3.43E4	3.46E4	3.45E4	3.29E4**
LFL9	4.24E4*	4.62E4	4.60E4	4.43E4	4.52E4	4.66E4	4.50E4
LFL11	4.06E4	4.06E4	4.08E4	4.06E4	4.08E4	4.08E4	4.06E4*

Table 7.12: F_{weight} for FSSP and JSSP

	BEST		*WORST*		*AVERAGE*		*STD-DEVI*	
Problem	UX	MIX	UX	MIX	UX	MIX	UX	MIX
Ljb7	14775	14716	15579	15595	15161	15156	299.1	280.0
Ljb9	15095	14660	17269	15710	15891	15357	733.3	355.6
Ljb10	3318	3265**	3472	3416	3399	3342	56.1	54.5
fl7	9732	9151	10048	9970	9826	9630	88.4	284.2
Lfl9	44996	43223	48650	46451	46395	44499	1091.8	1173.9

Table 7.13: Comparisons of UX/MIX for F_{weight}

for comparison are as follows:

- Maximum Tardiness (T_{max}):

  In [Baker 74] theorem 2.6 mentioned that maximum job tardiness and maximum job lateness are minimised by earliest due-date. So we compare GA with the earliest due-date rule and some other due-date related rules. The result is in Table 7.8 and Table 7.9.

- Mean Tardiness ($\bar{T}$):

  According to [Anderson 94], earliest modified operation due-date (EMOD) performs quite well in minimising mean tardiness. So in this experiment, we compare GA with EMOD and some other due-date related scheduling rules. The result is in Table 7.10 and Table 7.11.

- Weighted Flow time (F_{weight}):

Problem	BENCHMARK	RND	FASFS	WSPT	WLWKR	WTWORK	GA(ux)
JB1	-96	-97	-81	-97	-97	-97	-97*
JB2	-556	-556	-466	-556	-555	-555	-560**
JB4	311	311	311	311	311	311	308**
JB9	-173	395	2624	774	2261	3254	-483**
JB11	-2516	-2534	-2757	-2691	-2904	-2853	-2907**
JB12	-851	-883	-867	-880	-883	-880	-894**
LJB1	-97	-85	-77	-81	-99	-94	-155**
LJB2	507	233	1021	432	315	458	-266**
LJB7	-3803*	-3308	-2713	-3315	-2370	-2239	-3614
LJB9	2917	4704	5875	3117	2893*	4697	4333
LJB10	-118	208	158	-139*	274	513	-119
LJB12	484	900	999	501	627	887	458**
FL1	-492	-462	-389	-428	-289	-303	-579**
FL3	-97	-98	-98	-39	-98	-98	-98*
FL5	-357	154	1321	-358*	154	155	-278
FL7	-591*	211	998	-415	-100	-58	2
FL9	883	1181	1800	1020	1246	1262	744**
FL11	-3611	-3736	-3096	-3280	-2971	-2673	-4119**
LFL1	459	368	689	437	446	449	309**
LFL3	1794	2259	2919	2448	1949	2078	1683**
LFL5	4118	3811	5084	4155	4106	4723	2492**
LFL7	-8080	-7871	-7734	-8006	-7720	-7837	-9397**
LFL9	5500*	8473	9161	7402	8361	9700	8091
LFL11	6239	6298	6426	6189*	6461	6461	6297

Table 7.14: L_{weight} for FSSP and JSSP

	BEST		*WORST*		*AVERAGE*		*STD-DEVI*	
Problem	UX	MIX	UX	MIX	UX	MIX	UX	MIX
Ljb7	-3614	-3673	-2809	-2794	-3228	-3233	299.2	279.9
Ljb9	4333	3869	6506	4948	5153	4566	720.1	370.8
Ljb10	-119	-182**	65	23	-42	-90	61.2	68.1
fl5	-278	-306	323	187	-116	-141	164.4	145.7
fl7	2	-579	318	252	96	-100	88.3	284.2
Lfl9	8091	6355	11783	9584	9501	7631	1126.8	1174.0
Lfl11	6297	6107**	7309	7484	6757	6706	319.2	395.7

Table 7.15: Comparisons of UX/MIX for L_{weight}

[Conway *et al.* 67] investigated the criteria of mean flow time and found that SPT performs best, and also in [Baker 74] theorem 2.4 mentioned that weighted mean flow time is minimised by WSPT sequence for single machine. So we compare GA with WSPT - the weighted version of SPT and some other processing time related scheduling rules. The result is in Table 7.12 and Table 7.13.

- Weighted Lateness (L_{weight}):

  Weighted lateness and weighted flow time are similar criteria. Minimising weighted lateness can be achieved by setting arrival time equal to due-date and minimising weighted flow time. Furthermore, in [Baker 74] theorem 2.5 mentioned that mean lateness is minimised by SPT sequencing. So we use the same scheduling rule set as in weighted flow time to compare with GA. The result is in Table 7.14 and Table 7.15.

Problem	(R&M)	RAN	WSPT	EGD	EOD	MST	S/OP	(WCOVERT)	GA(ux)
JB1	104	104	104	128	104	113	113	104	95**
JB2	59	59	133	59	59	59	59	59	59*
JB4	351	351	351	351	351	351	351	351	350**
JB9	1230*	2018	3526	1842	1980	1945	1945	1230*	1251
JB11	20	270	387	84	34	81	81	68	0**
JB12	1911	2193	2478	1911	1955	1911	1911	1981	1219**
LJB1	229	238	230	284	253	291	291	233	171**
LJB2	3234	3746	3646	2911	2850	2968	2968	2859	2841**
LJB7	920*	1361	2310	1161	1525	964	1064	949	1127
LJB9	3947*	6217	4810	7128	6523	6575	6575	4563	5928
LJB10	954	1367	1086	1401	1030	1531	1531	889	860**
LJB12	1019	1521	1465	1114	1162	1213	1213	1031	991**
FL1	236	389	655	226	188	158*	158*	188	197
FL3	778	778	778	778	778	778	778	778	738**
FL5	886	1213	957	1378	958	1031	1031	1054	760**
FL7	1723	2064	2284	2045	1870	2081	2081	1722	1589**
FL9	1663	2202	2133	2215	2059	2180	2180	1756	1632**
FL11	740	1326	3655	584	829	441	441	480	235**
LFL1	1718	1930	1754	1852	1735	1793	1793	1709	1561**
LFL3	4126	4254	4565	4905	4706	5042	5042	3862	3599**
LFL5	8861	8996	9738	8319	8396	8404	8404	8366	7179**
LFL7	4318	4757	4637	4273	4383	4475	4475	4377	3259**
LFL9	9958*	1.58E4	1.54E4	1.16E4	1.12E4	1.15E4	1.15E4	1.01E4	1.46E4
LFL11	1.08E4	1.07E4	1.09E4	1.11E4	1.11E4	1.13E4	1.13E4	1.07E4	1.00E4**

Table 7.16: T_{weight} for FSSP and JSSP

	BEST		*WORST*		*AVERAGE*		*STD-DEVI*	
Problem	UX	MIX	UX	MIX	UX	MIX	UX	MIX
jb9	1251	1173**	1941	1762	1510	1464	217.8	228.6
Ljb7	1127	976	2099	1561	1518	1227	329.8	231.0
Ljb9	5928	5414	7398	7191	6941	6003	462.4	573.9
fl1	197	158*	270	244	230	204	24.3	32.1
Lfl9	14625	13198	17889	15014	16025	14327	1223.3	787.8

Table 7.17: Comparisons of UX/MIX for T_{weight}

- Weighted Tardiness (T_{weight}):

  [Carroll 65] found his slack-based heristic (COVERT) performs very well with mean tardiness criteria. [Vepsalainen & Morton 87, Vepsalainen & Morton 88] showed their R&M scheduling rule with their lead time estimates performed better than other scheduling rules. Here we compare the weighted versions of these two rules which are modified and implemented by [Morton & Pentico 93], and some other slack-based and due-date based scheduling rules with our GA. The result is in Table 7.16 and Table 7.17.

- Weighted Number of Tardy Jobs (NT_{weight}):

  In the scheduling literature, Moore's algorithm or Hodgson's algorithm [Moore 68] performs best in minimising the number of tardy jobs for a single machine shop. [Morton & Pentico 93] also modified and implemented a version of this algorithm for the weighted case. We take WMOORE, WCOVERT and R&M from

Problem	(WMOORE)	RND	WSPT	EGD	EOD	MST	S/OP	(WCOVERT)	(R&M)	GA(ux)
JB1	11	10	11	12	11	13	13	11	11	6.9**
JB2	2.7	2.7	3.3	6.5	2.7	6.5	6.5	2.7	2.7	2.7*
JB4	13	13	13	13	13	13	13	13	13	13*
JB9	43	53	45	79	74	79	79	51	45	25**
JB11	4.4*	6.6	20	19	9.4	16	16	12	6.6	5.7
JB12	51	53	61	51	51	51	51	53	51	51*
LJB1	6.0	6.0	6.6	8.5	8.5	8.5	8.5	8.3	6.0	5.0**
LJB2	28	32	41	46	46	46	46	48	40	25**
LJB7	15	19	26	33	28	20	21	22	15	12**
LJB9	19	36	34	50	47	47	47	45	35	18**
LJB10	17	20	19	24	24	25	25	25	21	14**
LJB12	13	18	16	25	26	26	26	27	19	9.7**
FL1	12	17	17	17	17	17	17	17	17	6.4**
FL3	36	36	42	42	42	42	42	42	36	26**
FL5	20	20	20	40	25	40	40	31	20	19**
FL7	21	28	25	50	43	46	46	47	26	17**
FL9	16	21	23	34	36	34	34	30	20	16*
FL11	25	29	34	33	42	23*	23*	29	30	25
LFL1	32	35	36	42	40	41	41	40	33	26**
LFL3	75	80	86	107	103	107	107	91	79	47**
LFL5	115	118	115	126	126	126	126	124	112	103**
LFL7	92	97	105	123	123	128	128	114	105	79**
LFL9	44*	105	81	156	150	164	164	146	96	55
LFL11	179	179	179	192	184	192	192	184	179	167**

Table 7.18: NT_{weight} for FSSP and JSSP

	BEST		*WORST*		*AVERAGE*		*STD-DEVI*	
Problem	UX	MIX	UX	MIX	UX	MIX	UX	MIX
jb11	5.7	4.4**	11.6	11.0	8.4	7.6	2.2	2.2
fl11	25	15**	35	34	30	25	4.1	7.0
Lfl9	55	51	71	72	65	61	5.5	6.7

Table 7.19: Comparisons of UX/MIX for NT_{weight}

Problem	RND	WSPT	WLWKR	WMWKR	EGO	EOO	S/OP	CR	GA(ux)
JB1	201	201	201	209	209	201	201	201	154**
JB2	568	689	613	571	559	613	559	549	339**
JB4	40	40	40	40	40	40	40	40	0**
JB9	1032	2753	2676	2655	795	995	795	795	0**
JB11	2329	3079	3055	2247	1912	2402	1815	1806	7.2**
JB12	2517	3158	3031	2722	2505	2362	2505	2505	158**
LJB1	305	321	348	298	295	295	295	295	19**
LJB2	2155	3214	3153	2508	2645	2553	2645	2669	544**
LJB7	4351	5625	5685	5401	3470	3908	4089	4123	310**
LJB9	974	1693	2228	1790	642	618	549	564	117**
LJB10	815	1225	1308	1067	872	748	803	868	18**
LJB12	441	964	783	620	320	316	225	343	0**
FL1	832	967	1339	1204	546	638	518	638	111**
FL3	817	817	1000	817	817	817	817	817	552**
FL5	681	1315	743	1032	473	1102	473	473	0**
FL7	1362	2699	2583	1728	879	1303	942	906	290**
FL9	747	1113	1725	830	485	600	485	485	138**
FL11	4321	6936	6910	6233	3811	3928	4009	3937	516**
LFL1	1268	1318	1303	1503	1257	1248	1300	1300	579**
LFL3	1811	2117	2459	1995	1699	1841	1699	1699	24**
LFL5	4451	5583	5351	4794	4263	4395	4142	4138	656**
LFL7	1.21E4	1.24E4	1.27E4	1.26E4	1.20E4	1.20E4	1.19E4	1.19E4	7241**
LFL9	5405	8036	9026	8374	2512	3174	2472	2470	422**
LFL11	4496	4688	4551	4655	4496	4529	4496	4496	716**

Table 7.20: E_{weight} for FSSP and JSSP

Problem	RND	WSPT	WLWKR	WMWKR	EGO	EOO	S/OP	CR	GA(ux)
JB1	306	306	306	343	337	306	314	314	276**
JB2	624	822	672	694	618	672	618	634	618**
JB4	391	391	391	391	391	391	391	391	390**
JB9	3158	6279	7612	7643	2637	2976	2740	2637	2540**
JB11	2809	3466	3205	2964	1995	2436	1896	1907	1166**
JB12	4318	5636	5179	5120	4416	4318	4416	4416	2159**
LJB1	579	551	597	612	579	548	587	579	256**
LJB2	5723	6860	6621	8718	5556	5403	5613	5608	3913**
LJB7	6541	7934	9000	8160	4631	5433	5153	5314	3318**
LJB9	7231	6504*	7349	1.19E4	7771	7141	7125	6533	6767
LJB10	2470	2311	2889	2690	2274	1778	2334	2203	1173**
LJB12	2023	2428	2192	2971	1435	1478	1439	1691	1260**
FL1	1315	1315	1404	1336	1316	1315	676	826	592**
FL3	1466	1668	1694	2034	1563	1466	1595	1595	1343**
FL5	4504	4506	4506	4693	4504	4504	1505	1505	1050**
FL7	3797	4556	4556	4317	4074	4115	3023	3013	2144**
FL9	4185	7965	7649	5067	2952	3803	2665	2565	2174**
FL11	4090	4433	4433	4495	4230	4568	4450	4303	2545**
LFL1	3050	3072	3052	3868	3109	2964	3093	3094	2595**
LFL3	6210	6682	6867	8189	6604	6547	6741	6604	4379**
LFL5	1.27E4	1.53E4	1.48E4	1.53E4	1.26E4	1.28E4	1.25E4	1.27E4	8789**
LFL7	1.66E4	1.69E4	1.78E4	1.82E4	1.63E4	1.64E4	1.64E4	1.64E4	1.27E4**
LFL9	1.89E4	2.35E4	2.64E4	3.22E4	1.43E4	1.44E4	1.40E4*	1.44E4	1.75E4
LFL11	1.54E4	1.56E4	1.56E4	1.61E4	1.56E4	1.56E4	1.58E4	1.56E4	1.19E4**

Table 7.21: ET_{weight} for FSSP and JSSP

	BEST		*WORST*		*AVERAGE*		*STD-DEVI*	
Problem	UX	MIX	UX	MIX	UX	MIX	UX	MIX
Ljb9	6767	6492**	9234	8140	7828	7341	718.5	585.8
Lfl9	17501	15491	21574	19047	19300	17297	1522.3	1301.6

Table 7.22: Comparisons of UX/MIX for ET_{weight}

[Morton & Pentico 93] and compared with our GA and other slack and due-date based scheduling rules. The result is in Table 7.18 and Table 7.19.

- Weighted Earliness (E_{weight}):

  The non-regular criteria is less covered in the literature. We compare some of the processing time, due-date and slack based heuristics with our GA. The result is in Table 7.20.

- Weighted Earliness plus Weighted Tardiness (ET_{weight}):

  Combined criteria are harder to optimise than single criteria, especially the combination of regular with non-regular criteria. Again we compare some of the processing time, due-date and slack based heuristics with our GA. The result is in Table 7.21 and Table 7.22.

In short, GA(UX) can find solutions in most problems tested at least as good as those found by the best heuristic rules compared, especially for the last two non-regular criteria. The non-regular criteria are the most difficult ones for existing heuristic rules,

however our GA approach can handle them just like other regular criteria and seem to perform much better compared with other heuristic rules. In general, GA(MIX) can further improve the quality of GA(UX) among all the criteria.

7.1.5.3 Dynamic Stochastic Scheduling

In real job shop scheduling, the arrival of a job (operation) and its processing time are usually unknown beforehand. Therefore, at certain times, there is only incomplete data available. If production must begin, then we need to schedule the partial data immediately. Our approach can handle this kind of situation by inputting the partial data so far and outputting a partial schedule which can be used immediately. This is a dynamic stochastic scheduling problem.

For example, suppose we just have the following incomplete data for the previous 6x6 JSSP in the beginning of a day and we may need to produce our production immediately based on these incomplete data. In this example, each row represents a job, and each pair a machine and processing time needed. That is, data available for job1 is machine3 for 1 unit time then machine1 for 3 units time. The only data available for job2 is just machine2 for 8 units processing time and so on. Other data will arrive later, but we don't know when and how much. Our representation and schedule builder can solve this problem dynamically.

```
Initial Data set:
1: 3 1     1 3
2 :2 8
3 :
4: 2 5     1 5
5: 3 9     2 3     5 5
6:
```

We can get a partial schedule according to the data list above; its GANTT chart and the output sequence sorted by start time and finish time are shown below:

```
Partial GANTT Chart so far:
    01234567890123456789O1
------------------------
mc 1: 111 44444
mc 2:4444422222222555
mc 3:1555555555
```

```
mc 4:
mc 5:                    55555
-------------------------
    01234567890123456789O1

Partial schedule output so far:
JOB          1  4  1  5  4  2  5  5
MACHINE      3  2  1  3  1  2  2  5
START        1  1  2  2  6  6 14 17
FINISH       1  5  4 10 10 13 16 21
```

Later, assume some new data arrives at the shop but the whole data set is still incomplete. At this time, based on the new set of data and the partial schedule list above, the user just needs to indicate in which stage of the old partial schedule to insert the new data to produce a new partial schedule. Recursively, we can extend each partial schedule to have a more and more complete schedule. At the same time the schedule can be used without waiting for the arrival of other data. For example, new data (operations to be scheduled) may arrive before the job4 machine1 operation starts in the current schedule. Hence, new data arrives before time '6'. In this case, operations already underway are fixed, and we run the GA with the new data combined with those operations already tentatively scheduled but not yet started. That is, we can extend the partially completed schedule to cater for the new data. The data set, GANTT chart and schedule output so far might be as follows:

```
Data set so far:
1: 3 1     1 3     2 6
2: 2 8     3 5
3: 3 5
4: 2 5     1 5
5: 3 9     2 3     5 5
6: 2 3     4 3     6 9     1 10

Partial GANTT Chart so far:
    012345678901234567890123456789O
-----------------------------------
mc 1: 111 44444           6666666666
mc 2:444446661111115552222222
mc 3:155555555533333           22222
mc 4:         666
mc 5:                  55555
mc 6:            666666666
-----------------------------------
    012345678901234567890123456789O
```

```
Partial schedule output so far:
JOB        1  4  1  5  6  4  6  1  3  6  5  5  2  6  2
MACHINE    3  2  1  3  2  1  4  2  3  6  2  5  2  1  3
START      1  1  2  2  6  6  9  9 11 12 15 18 18 21 26
FINISH     1  5  4 10  8 10 11 14 15 20 17 22 25 30 30
```

Finally, if all the data arrives, based on the previous partial schedule and the complete data, we can have the complete schedule produced in the same manner described above. For example, if the user indicates that the last 6 operations in the partial schedule output are to be rescheduled along with the new data, then the complete GANTT Chart and final schedule output can be achieved as follows:

```
Final data set:
1: 3 1     1 3     2 6     4 7     6 3     5 6
2: 2 8     3 5     5 10    6 10    1 10    4 4
3: 3 5     4 4     6 8     1 9     2 1     5 7
4: 2 5     1 5     3 5     4 3     5 8     6 9
5: 3 9     2 3     5 5     6 4     1 3     4 1
6: 2 3     4 3     6 9     1 10    5 4     3 1
```

```
Final GANTT Chart:
    012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
-------------------------------------------------------------------------------------------
mc 1: 111 44444          6666666666555   333333333                  2222222222
mc 2:444446661111115552222222                    3
mc 3:15555555553333344444     22222    6
mc 4:         666   11111113333444     5                                              2222
mc 5:               55555  1111116666444444442222222222333333
mc 6:           666666666 111555533333333      444444444 2222222222
-------------------------------------------------------------------------------------------
    012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
```

```
Final schedule output:
J.. 6  5  1  4  5  2  6  1  3  5  1  4  2  3  5  6  5  6  4  3  4  2  3  3  2  2  2
M.. 6  2  4  3  5  2  1  6  4  6  5  4  3  6  1  5  4  3  5  1  6  5  2  5  6  1  4
S..12 15 15 16 18 18 21 22 22 25 25 26 26 29 31 31 34 35 35 37 43 43 46 53 53 63 73
F..20 17 21 20 22 25 30 24 25 28 30 28 30 36 33 34 34 35 42 45 51 52 46 59 62 72 76
```

The approach cannot guarantee to get a better solution than in the static or dynamic deterministic case, when all the data is known in advance. However, this is a more practical approach because in the real job shop much data is possibly unknown in the beginning but we must begin production before all the data arrives. In this case, the static and dynamic deterministic scheduling approaches are inadequate because they assume we know all the data in advance. One way to handle such uncertainty is to

use the dynamic stochastic accumulated schedule described here – schedule a partial schedule based on the known data so far, use the partial schedule immediately and later extend the partial schedules to a more and more complete schedule. Another way is to apply a rescheduling technique, which uses approximate data and schedules a complete schedule in advance, and when actual data arrives, just reschedules the inaccurate part of the estimated data without changing most of the original schedule. Now the problem is how to just reschedule the changed part of a complete schedule. We give more details later in this chapter.

7.1.6 Discussion

[Anderson 94] considers reasons to use simple heuristic rules in dynamic scheduling:

> If a near optimal loading rule could be found for a particular job shop, we could be sure that it would be highly complicated, taking account of huge amounts of information as to where all the other jobs were in the job shop and their various characteristics. The challenge of the job shop scheduling area is to achieve success in scheduling and at the same time use only a simple rule. This may seem obtuse: if spending more time on calculation can achieve a better result, why not do it?

He further pointed out:

> For the vast majority of factories it would be a mistake to treat the future as predictable, and consequently trying to achieve a very tightly defined near optimal schedule will be disastrous. Moreover no one has yet demonstrated a method of this type which works in a dynamic job shop of realistic size. For these reasons it is better to use simple rules which make minimal assumptions about what will happen in the future, but nevertheless control the job shop in roughly the way we would like.

Our GA scheduling approach can work very well in comparison with the best heuristic rules in several ways:

- Though the GA spends more time to search than simple heuristic rules, we use *no* domain-dependent knowledge and our approach is very robust across many criteria and on average outperforms the heuristic rules. If spending a reasonable amount of extra time on calculation can achieve a better result, why not do it?
- Our *dynamic stochastic scheduling* approach allows us to build partial schedule using known information without predicting any further uncertain information. When new information does arrive, we can schedule it as the tail of previous

partial schedule and grow a larger schedule and so on. So we don't need to make predictions about what will happen in the future, or to carry out a simple 'what-if' simulation to test out different possible scheduling decisions.

- Our *rescheduling* approach using dependency analysis allows us to reschedule quickly when some unpredictable change happens. This is especially useful when a schedule is already running, so that we cannot reschedule from scratch or we already have a near optimal schedule and no time left to reschedule from scratch. In other words, we can achieve a very tightly defined near optimal schedule if we have enough time and when changes happen we can reschedule in a short time.

In the next section, we will extend our job-shop scheduling/rescheduling techniques to handle the open-shop scheduling/rescheduling problems.

7.2 GAs in Open-Shop Scheduling/Rescheduling Problems

The *Open-Shop Scheduling Problem* (OSSP) is a complex scheduling problem which occurs often in industry [Gonzalez & Sahni 76]. OSSPs arise in an environment where there is a collection of tasks to perform on one or more machines. Efficient production and manufacturing demands effective methods to optimise various aspects of a schedule, usually focusing on the total time taken to process all of the operations. The OSSP is similar to the JSSP, with the exception that there is no *a priori* ordering on the tasks within a job. The OSSP has a considerably larger search space than the JSSP and is possibly harder than job-shop scheduling problems of the same size if the number of jobs and the number of machines is very close [Taillard 93]. It seems to be less heavily addressed in the literature, although it is an important and ubiquitous problem, occurring in any job-shop situation in which tasks for a particular job may be carried out in (almost) any order. The common illustration of this kind of problem is that of an automotive repair shop [Gonzalez & Sahni 76]. In such a shop, a typical job might involve the operations 'spray-paint', 'fix-brakes', and 'change-tyres' to be performed on the same vehicle. These operations cannot usually be performed

concurrently (especially if the stations at which these operations are performed are in different places, for instance), but can be performed in any order. Also it is trivially clear that different stations (that is: 'machines') can concurrently process operations from different jobs (for example: involving different vehicles). If the operations in a job must be performed in some fixed order, then this becomes a 'Job-Shop Scheduling Problem' (JSSP). If each job has the same fixed order, then it further becomes a 'Flow-Shop Scheduling Problem'. Another example for OSSP is upgrades/repairs (tasks) for PCs (jobs).

A commonly used simplification of the OSSP is to specify that each given operation can only be processed on a given specified machine. In real problems it is often the case that an operation can be processed in a number of alternative ways, any of which may involve more than one machine. There may also be due-dates and machine setup times to consider. In the following however we will concentrate on a simplified form of the general problem; this is done mainly because the benchmark problems on which we test the performance of our GA approach are thus simplified. We will later discuss simple amendments to our approach which promise to successfully cope with the more general problem.

7.2.1 Problem Description

7.2.1.1 Open-Shop Scheduling Problem (OSSP)

A description of a simple form of OSSP goes as follows. There is a set of m distinct machines $M = \{m_1, m_2, \ldots, m_m\}$, and a collection of j jobs $J = \{j_1, j_2, \ldots, j_j\}$, each of which comprises a collection of operations (sometimes called tasks). An operation is an ordered pair (a, b), in which a is the machine on which the operation must be performed, and b is the time it will take to process this operation on machine a. Each job consists of a collection of such operations in a *job-independent* order. A feasible schedule for the OSSP is one which assigns a start time to each operation, satisfying the constraint that a machine can only process one operation at a time, and that two or more operations from the same job cannot be processed at the same time. The main objective is usually to generate a schedule with a makespan as short as possible; the

makespan is simply the total elapsed time in the schedule, that is, the time between the start of the first operation of the first job to begin, and the end of the last operation of the last job to finish. More complex objectives often arise in practice, where due dates and machine set up times must also be taken into account, for example. Table 7.23 shows one of the standard 5×5 benchmark OSSPs (that is, $j = 5, m = 5$) taken from [Beasley 90].

	(m,t)	(m,t)	(m,t)	(m,t)	(m,t)
Job 1:	4,85	1,64	3,31	5,44	2,66
Job 2:	1,7	4,14	2,69	5,18	3,68
Job 3:	4,1	1,74	2,70	5,90	3,60
Job 4:	2,45	4,76	5,13	3,98	1,54
Job 5:	1,80	4,15	2,45	5,91	3,10

Table 7.23: An 5x5 benchmark OSSP

In this example, operation 1 of job 1 must go to machine 4, for 85 units of time; and operation 2 of job 1 must go to machine 1 for 64 units of time, and so on, and no restrictions are placed on the order in which the operations for any job are to be processed. The task is to generate a good schedule showing what each machine is to do and when. The definition of 'good schedule' can vary considerably; for simplicity we use here the standard criterion of looking for the shortest makespan. For this particular problem the minimum makespan is known to be 300, as in, for example, the schedule in Figure 7.4. Here, the number inside each bar represents the operation *index* of each job: that is, the *5th* operation of job1 goes to machine2 first for 66 units of time; then *2nd* operation of job1 goes to machine1 for 64 units of time, and so on.

7.2.1.2 Open-Shop Rescheduling Problem (OSRP)

Open-Shop Rescheduling Problems (OSRPs) are similar to Job-Shop Rescheduling Problems, being beset by the continual need to alter previously worked out schedules in the light of problems which arise. This typically means revising the expected processing time for some job in the schedule, or revising (typically delaying) the start time for a particular task. If work has not yet begun on the current schedule, then an obvious and simple approach to rescheduling would be to rerun the schedule-finding program

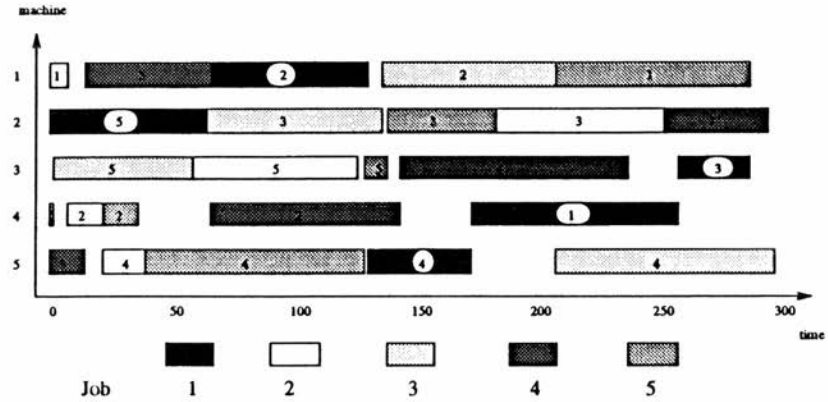

Figure 7.4: Makespan for the 5x5 the OSSP

(for example: in this case, a GA) from scratch on the changed data. Strict rescheduling, however, means *not* scheduling the entire problem from scratch; rescheduling is thus strictly necessary when either there is not enough time to be able to schedule from scratch, or when part of the current schedule is already in progress. A proper rescheduling method would be to re-use some of the work already done in finding the previous schedule. This might involve augmenting the previous schedule with the new change, and iteratively modifying it until it is acceptable. Another method would be to recover a new, smaller scheduling problem made up from all and only those parts of the previous schedule that are affected by the change.

If the schedule hasn't been used, then we don't need to worry about the time horizontal. Whereas, if the schedule is already half way through, for example, time horizontal is now at time T, then we can neither change the durations of the operations which start before time T nor can we shift operation before time T because both cases are illegally trying to update or preempt a completed or in-process operation. In both the JSRP and OSRP we can shift an operation backward to any place if the operation has not begun and the system can tell us which operations will be affected and need to be rescheduled.

An important difference between JSRP and OSRP is that in the JSRP we cannot

shift an operation to before a previous operation of the same job because of the job-dependent sequence for operations; that is, operations must obey the precedence constraints. However, in the OSRP we can shift an operation to any place because there are no precedence constraints. In other words, in the JSRP we just need to consider not preempting those operations of the same machine which are under way. Therefore, we don't need to worry about its preempting previous completed operations of the same job. Whereas, in the OSRP we must consider not only the operations of the same machine but also operations of the same job which are under way because we can let it shift to any place it wants. For example, assume the time horizontal is now at time T, an operation has started at time $T - t$ and is expected to finish at time $T + t$, where T and t are non-negative values. Then this operation is regarded as under way because it has already begun but not yet finished, so it cannot be preempted by any change.

7.2.2 Applying the Basic GA Framework to OSSP/OSRP

7.2.2.1 Representation

The basic extension of the representation to the OSSP involves two separate parts of the genome $ABCD...$ and $abcd...$ meaning: put the ath untackled task of the Ath uncompleted job into the earliest place it will fit in the developing schedule, put the bth untackled task of the Bth uncompleted job into the earliest place it will fit in the developing schedule, and so on. Both untackled task and uncompleted job are taken from a circular list in this case. Whereas previously, for the JSSP, the 'first untackled task' for any particular job was always predetermined owing to the *a priori* ordering on tasks, in the OSSP case we need to incorporate an extra factor for each job to encode which of the remaining tasks for a job to choose (since with no predetermined ordering, any may be chosen). This representation has the disadvantage that once the list becomes circular, both the genes, for example (Ath, ath) pair, will affect each other's meaning. Therefore, it will lose its original meaning completely after circularity becomes dominant. We refer to this representation with the abbreviation JOB | OP.

7.2.2.2 Fitness

The fitness function we use in the OSSP is to achieve the objective of minimising makespan or maximum completion time (C_{max}) which is also the most common criterion used in open-shop scheduling benchmark problems. That is, we again use the function,

$$f(j) = C_{max} = max(c_1, c_2 ... c_j) \tag{7.9}$$

where 1..j represent the job number, and minimise this function.

7.2.2.3 Schedule Builder

The open-shop schedule builder is similar to the job-shop schedule builder but doesn't need to consider precedence constraints. In other words, any operation of a given job can be scheduled anywhere, as long as there is a big enough gap. That is, a current operation can be scheduled before any scheduled operations if it can be inserted before them. Furthermore, it also needs to consider whether to insert compulsory idle time or not within each gap or tail of the partial schedule as in the JSSP.

Another difference is the *overlapping* problem. In the JSSP, because of the precedence constraints, we do not need to worry too much that operations of the same job will overlap each other because later operations must be serviced after previous operations. Therefore, the schedule builder just needs to avoid overlapping with the task immediately before the current task of the same job. In the OSSP, the schedule builder must pay attention to all possible overlapping situations (either head-overlap or tail-overlap) among all the scheduled tasks of the same job because the current operation can be inserted at any place in the schedule so far. In other words, the schedule builder in the JSSP just needs to look at the tail of the partial schedule. In the OSSP schedule builder, it must look everywhere in the schedule for a possible gap.

7.2.3 Specific Framework for OSRP

7.2.3.1 Dependency Analysis

Currently, two kinds of situation need to be rescheduled in our system. They are:

- *Changing Processing Time:* If some operation needs to change the processing time or if we want to remove some operation (set processing time 0).

- *Shifting Operation:* When we don't satisfy the current schedule or some operation is delayed, we could shift the operation to any place after the current time.

In both situations we must do the *dependency analysis* once and shift all the unaffected genes and the user-shifted genes to the front of the gene string and shift all the affected genes to the rear of the gene string. The former case will not affect the bits before the current changed bit because our input schedule is sorted by the start time, however, the latter may affect all the bits after the current time T. Dependency analysis will mark either the gene where the first change happens or the gene first affected by the shifting operation and so on, recursively.

A difference between JSRP and OSRP dependency analysis is that the range of choices in OSRP is much larger than JSRP. For example, in Figure 7.5, assume current time is T. If we want to shift 5 units of job 1 in machine 1 to other places, then '!' represents the possible positions to shift to and '?' represents illegal positions to shift to.

7.2.3.2 Reschedule Builder

Because all the schedules' front parts are the same, we do not need to duplicate effort to schedule the front part each time. The front part is almost a copy of part of our previous schedule (except for the user shifted operations), in other words, it is already a near optimal schedule if the input schedule is a near optimal one. So, we can expect that after rescheduling, the whole new schedule is also likely to be a near optimal schedule.

The algorithm for rescheduling the rear part is the same as that of the schedule builder but instead of scheduling all the genes, it just reschedules the rear part (affected genes) using information from the front part. The GA then just applies its crossover and mutation operators to the rear part and keeps the front part unchanged. This will save time compared to rescheduling from scratch.

```
Job Shop Rescheduling:
======================
input schedule:
   mc1:       T       333333        11111 2222
   mc2:333333333333333111112222222222222222
   mc3:22222222222222      111111111
       (can only shift to positions both after T and its
        immediately previous operation of the same job.)
dependency analysis:
   mc1:????????T??????????????????????????!!!!!!!!!!...
   mc2:333333333333333111112222222222222222
   mc3:22222222222222      111111111
       (just shifted operation itself and the operation
        of job 2 in machine 1 need to be rescheduled.)

Open Shop Rescheduling:
=======================
input schedule:
   mc1:       T       333333        11111 2222
   mc2:333333333333333111112222222222222222
   mc3:22222222222222      111111111
       (can shift to any position after T.)
dependency analysis:
   mc1:????????T!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!...
   mc2:333333333333333111112222222222222222
   mc3:22222222222222      111111111
       (all the operations may need to be rescheduled
        except the operation of job 2 in machine 3 and
        the operation of job 3 in machine 2.)
```

Figure 7.5: JSRP vs OSRP dependency analysis

As to which gap we can use, for example, consider the changing processing time case. Assume current time is T, and we want to change the processing time of job 1 in machine 1; '!' represents the possible places to put it and ? represents illegal positions to put it. The difference between the JSRP and OSRP reschedule builders is shown in Figure 7.6. Here, changing the processing time of job 1 in machine 1 only affects the changed operation itself and the operation of job 2 in machine 1. It will not destroy the schedule before the changed operation. However, it may be inserted before the previous operation if there is a suitable gap. Therefore, the OSRP reschedule builder needs to consider such possible gaps, which will not happen in the JSSP schedule or reschedule builder.

```
Job Shop Rescheduling:
======================
input schedule:
   mc1:                333333         11111 2222
   mc2:333333333333333111112222222222222222
   mc3:22222222222222      111111111
possible gaps:
   mc1:????????T???????333333????????!!!!!!!!!!...
   mc2:333333333333333111112222222222222222
   mc3:22222222222222      111111111
       (only consider to reschedule the gaps after both
        T and its previous operation of the same job)

Open Shop Rescheduling:
=======================
input schedule:
   mc1:                333333         11111 2222
   mc2:333333333333333111112222222222222222
   mc3:22222222222222      111111111
possible gaps:
   mc1:????????T!!!!!!!333333????????!!!!!!!!!!...
   mc2:333333333333333111112222222222222222
   mc3:22222222222222      111111111
       (need to consider all the gaps after T)
```

Figure 7.6: JSRP vs OSRP reschedule builder

7.2.4 Performance Enhancement

A key aspect of a GA-based approach to a problem is the choice of a chromosome representation for the problem. Following this decision there is then sometimes a choice in how to define the mapping from the space of possible chromosomes to the space of possible solutions. For example, in deciding how to represent a set of parameters for some real parameter optimisation problem, two of the many possible choices are to use a bitstring representation, and to use a representation consisting simply of a list of genes whose alleles can take real values (one per parameter). Given the representational choice, we can interpret a chromosome as a potential solution to the problem in several ways. For instance, a common choice in the 'real-valued gene' case is just for the alleles to be interpreted directly as parameter values. An alternative is for the list of values in the chromosome to be the input for a local hill-climbing operator. In the latter case the GA can be seen as searching a space of distinct regions of the solution space, where the solutions in a region all map, via the hill-climbing operator, to the same point.

The above illustrates a common general technique for hybridising a GA with a heuristic search or heuristic rule based method. That is, use the GA to search a space of *abstractions* of solutions, and employ a heuristic or some other method to convert the points delivered by the GA into candidate solutions. Such hybridisation is one way of avoiding the often highly complicated problem of representing a complete solution as a chromosome in a way that facilitates effective GA-based search; it is usually easier to represent abstract regions of the solution space, and have these abstractions converted into (that is: interpreted as) solutions by some other technique.

7.2.4.1 Applying Heuristic Rules

The heuristic rules used in the makespan for OSSP described here are mostly largest-processing-time (LPT) oriented. However, for very small problems, shortest-processing-time (SPT) oriented or random choice is also possible and may produce better solutions than those produced by *LPT-oriented* rules. Let's define some heuristic rules as follows. Say there are k operations, o_1, o_2, o_3...o_k. Depending on the partially built schedule, each o_i has an earliest time it can start, given by o_{it} and its corresponding processing time, given by o_{ip}. Also, each has a flag o_{ig}, so that if o_{ig}=1, then it fits in a gap, but if o_{ig}=0, it fits onto the end of the schedule. Given the o_{it}, o_{ip} and o_{ig} for each i, then we can define some heuristic rules as follows:

- *Shortest Processing Time* (**SPT**): It always schedules the one with the shortest o_{ip}. In other words, let the shortest operation be processed as early as possible. The SPT heuristic rule is inspired by not letting operations wait too long to schedule if we schedule the shortest one first.

- *Longest Processing Time* (**LPT**): It always schedules the one with the longest o_{ip}. In other words, let the larger operation be processed as early as possible. The LPT heuristic rule is inspired by avoiding the situation of leaving the larger operations until the last minute.

- *Earliest First - Shortest Processing Time* (**EF-SPT**): If there is a smallest o_{it}, schedule o_i. If there is a set of smallest o_{it}, schedule the one with the shortest o_{ip}. In other words, when several operations can start at the same time, then

always choose the one with the shortest processing time. That is to say, let the shorter operation be processed as early as possible if it can start at least as early as other operations.

- *Earliest First - Longest Processing Time* (**EF-LPT**): If there is a smallest o_{it}, schedule o_i. If there is a set of smallest o_{it}, schedule the one with the longest o_{ip}. In other words, when several operations can start at the same time, then always choose the one with the longest processing time. That is to say, let the larger operation be processed as early as possible if it can start at least as early as other operations.

- *Earliest First - Break Tie Randomly* (**EF-BTR**): If there is a smallest o_{it}, schedule o_i. If there is a set of smallest o_{it}, schedule either one with the same possibility. In other words, when several operations can start at the same time, then randomly choose an operation to be scheduled next.

- *Suitable Gap - Longest Processing Time* (**SG-LPT**): If all o_{ig}=0, then as LPT. If not, then as LPT, but considering only the o_i for which o_{ig}=1. In other words, see if any of the tasks can be placed in a gap in the schedule. If so, choose one of these tasks with maximal processing time. If no such gap exists, then process as in LPT.

- *Shortest Remaining Gap* (**SRG**): If all o_{ig}=0, then as LPT. If not, then considering only the o_i for which o_{ig}=1, choose the one with the shortest remaining gap left. That is, choose the operation which leaves minimal time left in its gap.

- *Longest Remaining Gap* (**LRG**): If all o_{ig}=0, then as LPT. If not, then considering only the o_i for which o_{ig}=1 and choosing the one with the longest remaining gap left. That is, choose the operation which leaves maximal time left in its gap.

We use some examples to illustrate how to apply the heuristic rule with the schedule builder. In each case we illustrate below, assuming we want to find job1's next operation and the possible candidate operations are represented by #...#.

7.2.4.1.1 SPT/LPT Heuristic Rules A cheap and simple way without using global information is just to apply the SPT or LPT heuristic rules. An example

illustrating SPT and LPT is in Figure 7.7.

```
m/c 1: 111111111...
    2: 5 222222 ##  4444444...
    3: 42  33333333####55555...
    4:  44444444444##########...
    5: 333355555555555555####...
```

Figure 7.7: SPT vs LPT heuristic rules

In Figure 7.7, job1's machine2 operation has the shortest processing time among job1's next operations. So machine2 would be chosen by SPT. If we apply LPT, then the machine4 operation has the largest processing time, and is therefore the LPT choice.

7.2.4.1.2 Earliest First Heuristic Rules The first part of this type of heuristic $EF-$ is used as follows: if the current job is job x, then we consider x's currently unscheduled operations, and choose the one with the *earliest* possible start time. When several operations of the same job can start early at the same time, for example when several gaps can be used for many of several operations at the same start time, then we apply the second part of these heuristic rules to make a decision. That is to say, if the $EF-$ part tells us that more than one equally early start times are available, then a simple heuristic rule is used to choose which task to actually place in which gap. Three simple heuristic rules are useful in our approach: (a) choose the task with the shortest processing time (-SPT); (b) choose the task with the largest processing time (-LPT); (c) choose randomly from the available tasks (-BTR). The more detailed OSSP schedule builder for these heuristic rules is in Figures 7.8 and 7.9.

```
m/c 1: 111111111...
    2: 5 222222 ##  4444444...
    3: 42  33333333####55555...
    4:  44444444444######...
    5: 333355555555555555#########...
```

Figure 7.8: Earliest first oriented heuristic rules - 1

In Figure 7.8, if #...# represent job1's possible next operations, then just applying the $EF-$ part is enough, because machine2 has an earlier start time than machine3,

machine4 or machine5 for job1's next operation. Therefore, we choose machine2 to service the next operation of job1.

```
m/c 1: 111111111...
    2: 5 222222 ##  4444444...
    3: 42  3333 ####  5555555...
    4:  444444444444######...
    5: 33335555555555#########...
```

Figure 7.9: Earliest first oriented heuristic rules - 2

In Figure 7.9, if #...# represent job1's possible next operations, then machine2, machine3, machine4 and machine5 all have the chance to service job1. The *EF-* part finds that machine2 and machine3 have an earlier start time than machine4 or machine5 for job1, hence one of these will be chosen. The second part of the heuristic rule is now to decide which one to choose. In *EF-SPT* heuristic rule, we choose machine2 to service job1 next because its processing time is shortest, in *EF-LPT* heuristic rule, we will choose machine3 next, and with *EF BTR*, machine2 and machine3 have an equal chance to be chosen.

7.2.4.1.3 Other Heuristic Rules The last three heuristic rules we used are also LPT-oriented. In SG-LPT, we must look for a *suitable gap* to let the large processing time operation be allocated first. A suitable gap means it is large enough and does not overlap with other operations of the same job. *SRG* checks if any of the tasks can be placed in a gap in the schedule. If there are several such gaps, then choose the one with the shortest remaining gap left after slotting the operation in. The idea behind this rule is to fill up a gap as much as possible. A similar heuristic rule is *LRG*, which is the same as SRG but chooses the one with the longest remaining gap left instead. The idea behind this rule is to leave as much gap as possible for other operations to use later.

If no suitable gap exists in the above three heuristic rules, then we just apply the *LPT* heuristic rule. That is, choose the task with the largest processing time and append it to the end of the machine's partial schedule with or without compulsory idle time just as in the JSSP schedule builder. Examples of these heuristic rules with or without

finding gaps are in Figures 7.10 and 7.11.

```
m/c 1: 11111111...
    2: 5 22222 ##   4444444...
    3: 42  33333#### 5555555...
    4:  444444444444######...
    5: 333355555555555#########...
```

Figure 7.10: Other heuristic rules (with gap)

In Figure 7.10, if we apply the *SG-* part, we find both machine2 and machine3 have suitable gaps to allocate job1's unfinished operations. Then the second part, *-LPT* heuristic rule, is used to decide which task (in machine2 or machine3) to choose. Because job1's processing time in machine3 is larger than machine2, we choose machine3 next. In addition, we find that there are 3 and 2 units processing time left if we fill in the gaps of machine2 and machine3 respectively. Therefore, in *SRG*, we choose the operation in machine3 as our next candidate because it uses the gap more completely. However, in *LRG*, we must choose the operation in machine2 as the next candidate because it leaves more space for later use.

```
m/c 1: 1111111111...
    2: 5 222222 444444###
    3: 42  3333  5555555####
    4:  44444444 #####
    5: 3333555552222#########
```

Figure 7.11: Other heuristic rules (without gap)

In Figure 7.11, we cannot find any machine which has gaps large enough to allocate job1's unfinished operations. So, the *LPT* heuristic rule is now to decide which machine (machine2, machine3, machine4 or machine5) to choose in order to append job1's next operation to the tail of that machine. Because job1's processing time in machine5 is larger than the others, we choose machine5's next operations according to the *LPT* heuristic rule in all of the *SG-LPT*, *SRG* and *LRG* cases.

In a nutshell, there are two cases to consider in these three gap-related heuristic rules: first, look for the gaps which can accommodate remaining operations, and if several such gaps exist, then choose one of them according to the heuristic rules pre-specified.

Otherwise, if no such gap exists, just apply the *LPT* heuristic rule to each case.

7.2.4.2 Fixed Heuristic (FH)

In this method, an alternative representation is to use precisely the same representation as for the JSSP's basic one, but change the interpretation of *abc*... to: "heuristically choose an untackled task from the *a*th uncompleted job and place it in the earliest place it will fit in the developing schedule, heuristically choose an untackled task from the *b*th uncompleted job and place it in the earliest place it will fit in the developing schedule", and so on. In this case, at each step the schedule builder applies a certain one of the heuristic rules described above. For example, at any given point, the schedule builder must schedule one of the tasks from the given job. The partially built schedule has candidate gaps for each of these tasks. The heuristic rule decides which one to schedule, based on the processing times, the start-time and/or other rules for the tasks (this is not a problem in JSSP because only one operation of a given job can start at a certain time). When decoding an OSSP chromosome, at any given stage, there is a partially built schedule, and a job (operation) to add to it. The task is to decide which of the unfinished operations of that job to choose. We term this method '**FH**', for '*Fixed Heuristic*'. In later experiments, we use, for example FH(LPT), to refer to the fixed-heuristic hybrid method, with LPT being the heuristic rule used in this case.

7.2.4.3 Evolving Heuristic Choice (EHC)

In a GA application which involves implicit serially interpreted encodings of solutions, as in the JSSP, it is often clear that alleles read earlier during the interpretation process have a greater effect on the finished interpreted solution. In the approach described above, for example, the last gene in a chromosome is actually redundant. Whatever has happened previously during the interpretation process, there is only one remaining unfinished job by the time we reach the final gene, and hence the particular identity of the allele in the final position has no effect. This is sometimes true of the penultimate gene too, depending on whether there are one or two unfinished jobs remaining before we schedule the final two operations. Even if the penultimate gene does represent a

choice between two different jobs, notice that half of its possible alleles will indicate one of these jobs, and the rest will indicate the other. Alleles in the first few genes to be interpreted, however, always uniquely specify distinct jobs. Also, it is intuitively reasonable to expect that operations scheduled earlier in a schedule have a greater effect on the overall quality of the schedule than operations scheduled later. Hence, due to both increasing allele redundancy and decreasing significance as regards overall schedule quality, alleles interpreted later in the serial schedule building process become less and less significant in terms of fitness. The result of these observations is that the genes exhibit different convergence rates. As is further discussed and illustrated in section 7.1.4.1, high-significance genes (interpreted earlier by the schedule builder) will converge more quickly than lower-significance genes (interpreted later). This suggests premature convergence of the high significance genes, since allele choices in these positions may converge before having been adequately sampled in the context of alleles in more slowly converging positions.

To fight this source of premature convergence in the OSSP, we introduce the concept of evolving heuristic rules. There is no good reason to rely on a fixed heuristic for each choice of operation while building a schedule. Indeed, it is quite easy to see that varying the choice of heuristic according to the particular job being processed, and also according to the particular stage in the schedule building process, may make more sense. It is hard to find some principled *a priori* method for making these varied choices, but we can implement a simple adaptive strategy by extending our basic chromosome representation as follows. Instead of using a single heuristic rule through the whole run, which is likely to converge quickly, we encode a further component of the chromosome to represent the heuristic rule we use in each gene and let the heuristic rule evolve with the job simultaneously. So, the representation now uses two components $ABCD...$ and $abcd...$ meaning: "apply heuristic rule a to decide which operation of the Ath uncompleted job to use and put it into the earliest place it will fit in the developing schedule, apply heuristic rule b to decide which operation of the Bth uncompleted job to use and put it into the earliest place it will fit in the developing schedule", and so on. We term this method '**EHC**', for '*Evolving Heuristic Choice*'. Alleles of genes which are interpreted as heuristic choices range through the number of available heuristics described above. In this way, each gene can choose the better heuristic rule

used dynamically. Furthermore, if we know that a certain heuristic rule tends to do well on a particular type of problem, then there are various ways we could bias the process towards favouring this rule. One simple such method we use is to arrange for the initial population to have a given percentage of its 'heuristic choice genes' preset. In this approach, even if we set 100% at a certain heuristic rule in the initial generation because we think it is the best rule for that particular problem or criterion, it is still possible to mutate to other heuristic rules during evolution. In later experiments, we use, for example EHC(LPT-1.0) or EHC(EF-SPT-0.5), to refer to the evolving-heuristic hybrid method, with 100% of the initial generation's 'heuristic genes' set at LPT or 50% of the initial generation's 'heuristic genes' set at of EF SPT respectively.

7.2.5 Experiments

7.2.5.1 Scheduling

The OSSP results involved fitness based selection with scaling (by taking the current value minus optimal value or lower bound) with elitism and maximum runs of 1,000 generations but stopping if more than 500 generations pass without improvement. The population size we used was 200 and the reproduction scheme was generational.

We found that results did not vary significantly across changes in crossover rate and adaptation regime. The OSSP experiments used adaptive crossover (starting with p_c at 0.8, falling by 0.0005 per generation, with a limit of 0.3) and fixed mutation at 0.5 per individual. Typically, order-based mutation (swap alleles between two randomly chosen genes) was used. For the JOB+OP and EHC, the mutation rate was 0.5 for the probability of mutating the first component; and 0.25 for the possibility of mutating the second component, otherwise mutate both simultaneously. For the 4x4 problem, we run 10 times and for the others run 20 times. For each size of problem, there are ten different instances. For example, the ten 4x4 problems are called 4x4.0 to 4x4.9.

In Table 7.24 to Table 7.27, the figure before the colon is the mean makespan, the figure after is the standard deviation of the makespan, and the figure after the slash is the best value or smallest makespan over these trials. For all the problems, the 'BK' column shows the optimal solution if it is followed by a '*', and it is the best known

OSSP	BK(LB)	JOB+OP	FH(EF-SPT)	FH(EF-LPT)	FH(EF-BTR)
4×4.0	193*	194.4:1.3/193*	193.4:0.8/193*	211.0:0.0/211	195.6:0.5/195
4×4.1	236*	240.8:2.9/236*	239.5:0.8/239	239.0:0.0/239	240.5:1.4/239
4×4.2	271*	271.8:0.4/271*	271.0:0.0/271*	271.2:0.4/271*	271.4:0.5/271*
4×4.3	250*	252.9:2.5/250*	252.2:0.4/252	255.4:2.1/253	252.0:0.0/252
4×4.4	295*	297.5:3.0/295*	295.4:1.3/295*	303.7:1.5/302	303.2:2.9/298
4×4.5	189*	191.8:2.7/189*	191.6:3.5/189*	189.0:0.0/189	193.9:1.9/193
4×4.6	201*	203.0:0.8/201*	201.0:0.0/201*	203.0:0.0/203	203.7:0.5/203
4×4.7	217*	218.6:1.7/217*	219.8:1.0/217*	220.0:0.0/220	219.9:2.2/217*
4×4.8	261*	263.4:3.1/261*	268.8:0.0/268	268.0:0.0/268	266.8:2.4/261*
4×4.9	217*	221.2:2.7/217*	220.5:3.7/217*	225.0:0.0/225	223.8:0.6/222
5×5.0	300*	308.3:5.5/301	306.9:6.7/301	312.4:2.0/305	305.9:2.8/301
5×5.1	262*	269.5:4.9/262*	268.4:3.0/265	267.9:2.5/266	268.6:3.5/262*
5×5.2	323*	340.8:4.4/331	340.9:5.6/331	340.4:5.1/329	341.1:3.2/335
5×5.3	310*	324.6:6.7/312	324.7:3.0/318	323.7:3.9/320	322.4:4.3/312
5×5.4	326*	341.6:5.8/330	337.1:5.5/326*	338.3:3.4/331	336.9:3.7/333
5×5.5	312*	325.1:4.7/312*	325.9:4.2/318	318.6:3.8/312*	321.1:4.1/312*
5×5.6	303*	317.2:6.5/308	313.0:4.0/310	312.9:2.8/310	312.6:4.1/307
5×5.7	300*	309.6:4.3/301	308.5:3.3/305	308.0:1.6/306	308.1:3.6/304
5×5.8	353*	362.6:4.6/355	362.6:1.2/362	360.9:1.2/358	366.0:4.5/359
5×5.9	326*	338.7:4.9/330	336.4:5.5/330	336.3:3.2/333	338.6:3.7/334
7×7.0	438?(435)	452.6:6.9/441	450.4:5.4/440	449.3:4.7/441	450.1:5.5/436#
7×7.1	449?(443)	468.8:6.7/456	470.0:6.8/455	467.3:7.4/450	466.3:7.0/455
7×7.2	479?(468)	499.1:7.3/488	504.6:7.7/492	493.2:6.6/482	498.5:7.0/483
7×7.3	467?(463)	482.8:7.5/467?	483.5:5.6/472	479.3:4.4/472	484.3:6.1/470
7×7.4	419?(416)	436.1:7.2/423	436.1:4.9/429	434.6:5.6/423	433.7:6.3/422
7×7.5	460?(451)	483.6:9.6/465	485.0:5.9/472	472.7:5.9/464	480.4:6.0/470
7×7.6	435?(422)	459.1:7.8/447	457.0:7.6/447	450.3:5.8/438	452.5:6.5/442
7×7.7	426?(424)	444.0:7.4/430	437.6:4.6/432	436.6:5.0/430	437.6:6.1/427
7×7.8	460?(458)	474.9:6.9/460?	470.8:6.3/462	467.6:5.2/458*	474.3:5.3/461
7×7.9	400?(398)	415.3:7.7/404	418.6:5.9/406	415.1:5.9/404	418.1:5.3/408

Table 7.24: Results for OSSPs benchmark using JOB+OP and FH(EF-X)

solution found by any method so far if it is followed by a '?'. If our GA solution is better than the best known solution found otherwise so far, it is followed by a '#'.

Our initial results using job and operation (JOB+OP) representation for a set of small benchmark open shop scheduling problems and using *earliest* first (EF-) heuristic rules alone on average produced better solutions than other heuristic rules for the three smaller benchmark open shop scheduling problems shown in Table 7.24.

From Table 7.24, we know that FH methods usually produced better solutions than JOB+OP combinations except when the problem size was very small, for example 4×4 problems. EF-SPT can get the optimal values frequently for 4×4 problems, however, if we look at EF-LPT, we see that it very easily gets stuck in local minima. This gives

OSSP	BK(LB)	EHC(EF-SPT-0.5)	EHC(EF-LPT-0.5)	EHC(EF-BTR-0.5)
4×4.0	193*	193.0:0.0/193*	194.3:1.4/193*	194.1:1.4/193*
4×4.1	236*	239.8:1.1/239	239.0:0.0/239	239.8:1.3/239
4×4.2	271*	271.3:0.5/271*	271.8:0.4/271*	271.5:1.0/271*
4×4.3	250*	251.1:1.2/250*	252.5:1.0/250*	251.5:1.1/250*
4×4.4	295*	297.7:4.4/295*	296.3:2.4/295*	297.4:3.3/295*
4×4.5	189*	189.1:0.3/189*	190.0:1.6/189*	191.0:2.8/189*
4×4.6	201*	203.3:0.5/203	203.0:0.0/203	203.0:0.0/203
4×4.7	217*	219.5:1.9/217*	219.8:1.0/217*	220.8:1.3/220
4×4.8	261*	265.0:3.5/261*	266.7:2.1/261*	265.4:3.1/261*
4×4.9	217*	217.7:2.2/217*	220.9:4.1/217*	217.0:0.0/217*
5×5.0	300*	306.2:4.5/300*	305.9:3.3/302	305.1:3.7/300*
5×5.1	262*	268.4:3.6/262*	269.1:3.2/262*	268.4:4.1/262*
5×5.2	323*	339.3:3.9/331	336.6:4.1/331	338.7:4.7/332
5×5.3	310*	324.4:5.1/316	325.8:3.6/317	322.6:5.7/310*
5×5.4	326*	337.6:5.3/326*	336.5:3.6/331	336.3:3.3/329
5×5.5	312*	321.6:1.9/320	322.4:6.2/312*	319.9:4.4/312*
5×5.6	303*	312.1:4.2/308	314.4:4.2/308	312.8:4.4/308
5×5.7	300*	307.3:4.1/301	306.9:2.5/303	305.8:1.6/303
5×5.8	353*	362.0:6.1/353*	359.1:1.3/358	358.4:3.5/353*
5×5.9	326*	337.6:4.3/329	336.6:4.5/330	337.0:3.2/330
7×7.0	438?(435)	450.8:6.5/442	444.6:4.0/435#*	447.1:5.6/435#*
7×7.1	449?(443)	470.1:6.9/455	462.4:8.1/447#	466.8:7.8/454
7×7.2	479?(468)	498.9:7.2/488	490.9:6.9/472#	494.6:7.1/481
7×7.3	467?(463)	485.4:5.5/475	476.1:4.3/469	478.9:6.9/466#
7×7.4	419?(416)	438.0:7.8/423	432.8:4.4/427	433.4:7.2/417#
7×7.5	460?(451)	483.6:7.1/469	470.1:5.8/460?	477.2:9.1/459#
7×7.6	435?(422)	452.9:6.2/443	447.3:7.1/435?	450.4:6.1/441
7×7.7	426?(424)	441.6:5.7/429	437.2:7.7/424*	436.7:5.8/427
7×7.8	460?(458)	473.9:6.0/461	467.5:5.3/459#	472.6:5.8/463
7×7.9	400?(398)	413.7:6.2/407	410.5:5.7/398#*	413.1:7.2/399#

Table 7.25: Results on OSSPs benchmark using EHC(EF-X-0.5)

us a hint that apply evolving heuristic rules may be a good idea. The disadvantage is that evolving heuristic rules will take a little more time than fixed heuristic rule.

We experimented with EHC(EF-X-Y) for X = SPT, LPT, and BTR, and Y = 0.5 and 1.0 on all of the 4 × 4, 5 × 5 and 7 × 7 benchmark OSSPs. For example, if 50% were initialised to EF-SPT, this is shown as EHC(EF-SPT-0.5). The results are in Table 7.25 and Table 7.26 respectively.

Now, the average performance is better than using FH, even if we choose the wrong rule in the beginning. For example, FH(EF-LPT) is the worst of the fixed heuristic methods tried earlier (see Table 7.24), but if we allow the heuristic rules choice to individually evolve after fixing them at EH-LPT only in the initial generation, that

OSSP	BK(LB)	EHC(EF-SPT-1.0)	EHC(EF-LPT-1.0)	EHC(EF-BTR-1.0)
4×4.0	193*	193.4:0.8/193*	195.0:1.2/193*	194.0:1.1/193*
4×4.1	236*	239.4:0.8/239	239.0:0.0/239	238.1:1.4/236*
4×4.2	271*	271.1:0.3/271*	271.1:0.3/271*	271.1:0.3/271*
4×4.3	250*	251.5:2.5/250*	251.7:1.3/250*	251.8:0.6/250*
4×4.4	295*	295.4:1.3/295*	295.7:1.5/295*	299.7:3.9/295*
4×4.5	189*	189.0:0.0/189*	189.3:0.5/189*	191.4:2.1/189*
4×4.6	201*	201.2:0.6/201*	203.0:0.0/203	203.4:0.5/203
4×4.7	217*	218.5:1.6/217*	220.0:0.0/220	220.6:1.6/217*
4×4.8	261*	267.7:0.5/267	267.6:0.5/267	264.6:3.1/261*
4×4.9	217*	220.5:3.7/217*	221.7:4.1/217*	217.7:2.2/217*
5×5.0	300*	304.5:3.2/301	307.7:4.3/302	304.3:4.3/300
5×5.1	262*	267.2:3.6/263	267.1:2.7/262*	269.1:3.2/265
5×5.2	323*	339.9:4.9/330	337.6:4.1/331	339.1:4.6/329
5×5.3	310*	324.3:4.8/314	324.6:3.7/316	321.8:5.9/312
5×5.4	326*	339.8:5.7/326*	334.1:3.0/330	337.4:3.8/332
5×5.5	312*	321.8:3.8/314	319.9:5.3/312*	319.7:3.8/312*
5×5.6	303*	311.9:4.1/307	314.8:2.4/310	310.6:2.8/308
5×5.7	300*	307.9:3.1/304	308.1:1.9/304	306.9:3.0/304
5×5.8	353*	364.3:2.9/361	359.6:1.6/358	364.1:5.0/353*
5×5.9	326*	337.2:4.0/329	336.4:4.0/330	337.7:3.7/331
7×7.0	438?(435)	452.4:6.2/441	446.9:5.4/435#*	449.0:7.2/435#*
7×7.1	449?(443)	466.7:6.4/454	460.4:8.5/446#	468.8:6.8/457
7×7.2	479?(468)	501.6:6.9/488	490.3:5.1/479?	497.1:7.6/481
7×7.3	467?(463)	488.6:5.4/478	475.5:6.8/466#	481.6:5.2/472
7×7.4	419?(416)	436.3:5.4/428	431.9:7.5/421	435.4:6.3/424
7×7.5	460?(451)	479.2:7.1/468	468.8:6.1/458#	475.0:7.9/464
7×7.6	435?(422)	455.0:7.7/441	448.7:6.1/434#	451.9:7.7/434#
7×7.7	426?(424)	441.9:6.5/427	434.2:4.5/424#*	438.2:5.9/429
7×7.8	460?(458)	472.9:6.8/461	468.1:6.2/458#*	470.8:6.5/458#*
7×7.9	400?(398)	420.1:6.8/403	415.4:7.2/405	414.5:6.0/404

Table 7.26: Results on OSSPs benchmark using EHC(EF-X-1.0)

is: use EHC(EF-LPT-1.0), then we can see how EHC enables us to escape from the initially poor heuristic choices. However, improvements are most marked on the harder 5×5 and 7×7 problems. In particular, on the 7×7 problems, EHC methods in our experiments have beaten the previous best known results.

Similar tests were performed on 30 larger benchmarks: 10×10, 15×15 and 20×20. The results appear in Table 7.27.

On these larger problems, the benefits of EHC and FH over JOB+OP, show up more clearly: JOB+OP never matched the previous best known or optimum on any of these 30 problems. However, FH found optima on 12 of these problems while beating previous best known results on a further 4, while EHC found optima on 11 of these

OSSP	BK(LB)	JOB+OP	FH(LPT)	EHC(LPT-1.0)
10×10.0	645?(637)	690.7:10.8/668	662.7:6.0/646	660.9:8.6/641#
10×10.1	588*	618.8:10.6/596	604.4:3.7/598	601.5:5.5/590
10×10.2	611?(598)	634.0:10.9/618	623.7:3.4/615	621.0:5.0/614
10×10.3	577*	599.8:10.2/583	587.9:3.9/579	587.7:4.5/577*
10×10.4	641?(640)	677.6:13.1/659	663.1:2.2/659	660.0:4.1/654
10×10.5	538*	566.9:12.9/548	546.5:3.6/540	544.7:3.1/541
10×10.6	623?(616)	648.1:12.2/634	633.4:5.3/621#	632.5:6.0/619#
10×10.7	596?(595)	630.3:6.5/622	610.0:3.7/602	610.8:4.1/605
10×10.8	595*	630.1:9.4/615	610.5:5.3/600	609.4:4.3/597
10×10.9	602?(596)	627.5:9.3/611	610.1:4.5/600#	611.7:5.4/597#
15×15.0	937*	968.1:11.9/946	943.5:3.0/937*	942.6:4.6/937*
15×15.1	918*	992.2:7.6/974	931.0:6.0/920	931.3:5.1/919
15×15.2	871*	925.5:11.6/905	873.6:3.6/871*	873.1:2.7/871*
15×15.3	934*	964.0:10.8/950	938.8:3.4/934*	938.3:4.3/934*
15×15.4	950?(946)	1028.0:12.2/991	967.4:4.4/960	965.8:5.7/954
15×15.5	933*	976.3:16.8/943	937.9:4.7/933*	936.7:3.6/933*
15×15.6	891*	960.5:14.9/933	898.3:2.6/893	900.2:2.4/895
15×15.7	893*	938.4:14.9/909	898.5:4.3/893*	899.1:3.5/893*
15×15.8	908?(899)	977.5:13.3/955	916.6:6.0/903#	918.0:8.6/904#
15×15.9	902*	963.9:14.7/946	914.3:3.7/907	915.4:5.5/904
20×20.0	1155*	1247.0:12.0/1224	1164.7:5.6/1155*	1167.4:8.1/1156
20×20.1	1244?(1241)	1372.4:10.0/1351	1270.8:8.8/1255	1269.5:9.4/1252
20×20.2	1257*	1316.1:17.4/1287	1260.3:3.3/1257*	1260.6:3.1/1257*
20×20.3	1248*	1356.9:13.0/1335	1263.1:5.8/1255	1262.8:6.2/1253
20×20.4	1256*	1327.8:12.6/1306	1260.3:2.2/1256*	1260.3:3.0/1256*
20×20.5	1209?(1204)	1276.9:16.6/1236	1209.8:3.8/1204#*	1210.7:5.0/1204#*
20×20.6	1294*	1384.0:18.5/1338	1302.0:4.2/1294*	1303.7:4.9/1295
20×20.7	1173?(1169)	1309.2:13.4/1287	1192.7:5.8/1184	1192.7:7.2/1181
20×20.8	1289*	1351.2:17.6/1319	1289.8:2.1/1289*	1290.1:1.1/1289*
20×20.9	1241*	1323.8:20.4/1278	1241.3:0.6/1241*	1243.2:2.9/1241*

Table 7.27: Results on benchmark using JOB+OP vs FH(LPT) vs EHC(LPT-1.0) for large OSSPs

problems and beat previous best known results on a further 5. EHC seems to just have the edge over FH on these larger problems, since it finds better solutions than FH on 14 of the 30, and does worse on just 6.

We summarise the upper bound (UB), lower bound (LB) and best solutions found by our GA in Table 7.28. In each case, again '*' represents the optimal solution and '#' represents solutions found by our GA which are better than the best solutions found so far. In other words, they are to be regarded as new upper bounds. The results are very promising. We found better solutions in all of the ten 7×7 problems than those produced by other methods so far. Among them, 4 are optimal solutions. None of the benchmark methods can find these optima despite using lengthy procedures to

produce these previous best results [Taillard 93].

SMALL					*LARGE*				
OSSP	UB	LB	GA	%	OSSP	UB	LB	GA	%
4x4.0	193*	186	193*	0	10x10.0	645	637	641#	-0.6
4x4.1	236*	229	236*	0	10x10.1	588*	588	590	0.3
4x4.2	271*	262	271*	0	10x10.2	611	598	614	0.5
4x4.3	250*	245	250*	0	10x10.3	577*	577	577*	0
4x4.4	295*	287	295*	0	10x10.4	641	640	654	2.2
4x4.5	189*	185	189*	0	10x10.5	538*	538	540	0.4
4x4.6	201*	197	201*	0	10x10.6	623	616	619#	-0.6
4x4.7	217*	212	217*	0	10x10.7	596	595	602	1.0
4x4.8	261*	258	261*	0	10x10.8	595*	595	597	0.3
4x4.9	217*	213	217*	0	10x10.9	602	596	597#	-0.8
5x5.0	300*	295	300*	0	15x15.0	937*	937	937*	0
5x5.1	262*	255	262*	0	15x15.1	918*	918	919	0.1
5x5.2	323*	321	323*	0	15x15.2	871*	871	871*	0
5x5.3	310*	306	310*	0	15x15.3	934*	934	934*	0
5x5.4	326*	321	326*	0	15x15.4	950	946	954	0.4
5x5.5	312*	307	312*	0	15x15.5	933*	933	933*	0
5x5.6	303*	298	307	1.3	15x15.6	891*	891	893	0.2
5x5.7	300*	292	301	0.3	15x15.7	893*	893	893*	0
5x5.8	353*	349	353*	0	15x15.8	908	899	903#	-0.6
5x5.9	326*	321	329	0.9	15x15.9	902*	902	904	0.2
7x7.0	438	435	435#*	-0.7	20x20.0	1155*	1155	1155*	0
7x7.1	449	443	446#	-0.7	20x20.1	1244	1241	1252	0.6
7x7.2	479	468	472#	-1.5	20x20.2	1257*	1257	1257*	0
7x7.3	467	463	466#	-0.2	20x20.3	1248*	1248	1253	0.4
7x7.4	419	416	417#	-0.5	20x20.4	1256*	1256	1256*	0
7x7.5	460	451	458#	-0.4	20x20.5	1209	1204	1204#*	-0.4
7x7.6	435	422	434#	-0.2	20x20.6	1294*	1294	1294*	0
7x7.7	426	424	424#*	-0.5	20x20.7	1173	1169	1181	0.7
7x7.8	460	458	458#*	-0.4	20x20.8	1289*	1289	1289*	0
7x7.9	400	398	398#*	-0.5	20x20.9	1241*	1241	1241*	0

Table 7.28: Summary of OSSPs, Benchmark vs GA

From the above experiments, we can draw some conclusions as follows.

1. Using heuristic rules achieves a 'best' result close to the problem benchmark in almost all cases tested than the GA without heuristic rules.

2. EF-oriented heuristic rules, particularly EF-SPT, are useful in small problems, especially to decide which operation is to be scheduled in the beginning of the partial schedule when several operations can begin at the same time. LPT-oriented rules are good heuristic rules for larger problems because we don't want to delay large operations, otherwise we may miss suitable gap, and have to append

large operations to the end of the schedule.

3. It is clear from the above tables that the overall approach is successful on these difficult problems, coming mostly within well under 1 per cent of the best known solution on most trials in the heuristic rule cases.

4. EHC, with an appropriate initial 'preset' choice, seems to be the best method overall.

5. Our approach can beat the previous best solutions in many of the benchmark problems. We cannot say whether or not they are optimal solutions in those cases where they are still larger than lower bounds unless we use exhaustive search to compare. However, we can say that these provide some new upper bounds for many of the benchmark OSSPs.

7.2.5.2 Rescheduling

To examine our rescheduling methods, we compare lower bound (LB) with *dependency analysis rescheduling* (DP), *non-dependency analysis rescheduling* (NDP) and *rescheduling from scratch* (SCR) for the first problem in each size to gain some feeling for how time taken and solution quality vary with the degree to which we re-use the previous schedule. In each run we input an optimal or near optimal schedule and make some change by doubling the processing time of the gene one sixth, two sixths, three sixths, four sixths and five sixths of the way along the chromosome respectively to produce the rescheduling problems used here. Tournament selection and uniform crossover are used in this experiment and the population size we used is 200, stopping after 20 successive generations failed to improve best solution quality.

Plots of the average makespan, the number of bits that need to reschedule, and the time taken are shown in Figures 7.12 to 7.16 respectively; the best results and the average time taken (in seconds) over five runs are in Table 7.29.

We can draw short conclusions as follows:

- *speed consideration:* The speed of rescheduling depends heavily on how many genes need to be rescheduled. The more genes are affected, the more genes need

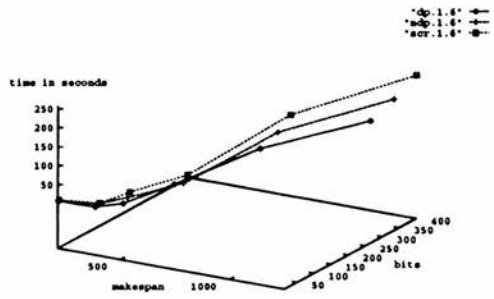

Figure 7.12: Change at one sixths gene for OSRPs

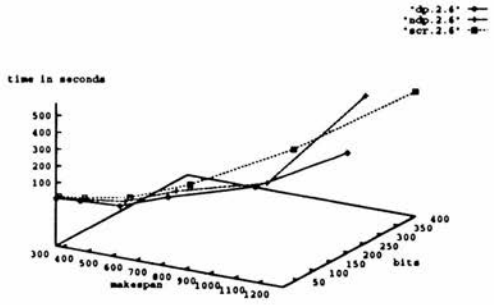

Figure 7.13: Change at two sixths gene for OSRPs

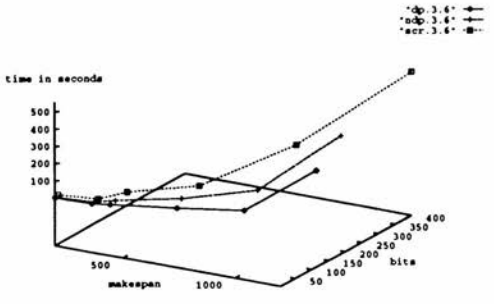

Figure 7.14: Change at three sixths gene for OSRPs

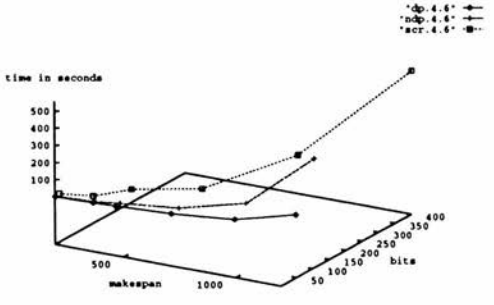

Figure 7.15: Change at four sixths gene for OSRPs

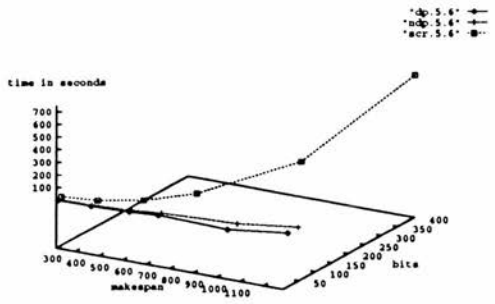

Figure 7.16: Change at five sixths gene for OSRPs

to be rescheduled and therefore the longer it takes. If all the genes need to be rescheduled (in fact this is impossible in our rescheduling case), then it is the same as rescheduling from scratch. If not, then we know that rescheduling is much faster than scheduling normally. In other words, the number of bits needing to be rescheduled in ascending order is DP, NDP, SCR, so the time spent in ascending order is also DP, NDP, SCR. The difference is more and more evident when the changed gene approaches the tail. In this case, DP and NDP schedule fewer and fewer genes whereas SCR still needs to schedule all of them. Also, DP needs to schedule fewer genes than NDP because of using dependency analysis. In short, *DP involves least change, SCR the most, so DP is fastest and SCR is slowest.*

- *quality consideration:* The quality of the solution depends heavily on the input schedule. If the input schedule is a very good one, then the solution after rescheduling must also be a good one. Two related reasons can explain this observation: first, the meaning of the rear part is dependent on the front part in our representation and second, because we adopt a good front part (assuming the input schedule is a good one) and apply GA to the rear part. This way, we can preserve much of the quality of the original schedule. In other words, results

problem	position	LB	DP/Time	NDP/Time	Scratch/Time
4x4.0	1 / 6	216	216*/6.1	216*/6.8	216*/8.0
	2 / 6	259	273/4.4	273/6.3	273/7.5
	3 / 6	193	193*/0.21	193*/0.26	193*/7.3
	4 / 6	193	193*/0.16	193*/0.21	193*/4.7
	5 / 6	220	227/3.0	227/3.7	220*/9.1
5x5.0	1 / 6	385	385*/7.9	385*/11.6	385*/12.6
	2 / 6	364	364*/7.6	364*/9.9	364*/12.6
	3 / 6	362	362*/5.4	362*/7.8	362*/12.7
	4 / 6	335	365/4.6	365/6.2	335*/15.6
	5 / 6	361	366/3.6	366/4.6	361*/13.1
7x7.0	1 / 6	486	487/19.9	487/30.0	486*/40.9
	2 / 6	517	518/13.1	518/22.5	517*/30.9
	3 / 6	435	435*/10.2	435*/22.6	449/48.4
	4 / 6	455	464/7.5	464/12.1	464/60.5
	5 / 6	522	525/6.5	525/9.1	522*/31.5
10x10.0	1 / 6	637	656/71.6	670/67.2	669/81.4
	2 / 6	637	658/65.0	666/91.6	684/98.0
	3 / 6	680	692/31.4	692/65.0	696/89.4
	4 / 6	691	702/14.2	702/23.0	704/68.8
	5 / 6	637	641/12.6	641/16.0	669/65.8
15x15.0	1 / 6	937	940/160.8	939/179.2	945/207.4
	2 / 6	937	937*/139.0	937*/118.0	937*/241.6
	3 / 6	937	937*/28.0	938/106.6	943/255.4
	4 / 6	937	937*/3.2	937*/63.2	951/192.6
	5 / 6	937	937*/1.0	937*/3.6	942/230.8
20x20.0	1 / 6	1250	1250*/203.2	1250*/226.8	1250*/257.6
	2 / 6	1155	1171/295.2	1175/575.6	1198/463.2
	3 / 6	1155	1175/260.4	1180389.2	1201/555.2
	4 / 6	1155	1155*/43.2	1158/316.2	1199/557.0
	5 / 6	1155	1155*/6.4	1155*/8.2	1164/749.0

Table 7.29: Results on OSRPs

in the DP and NDP cases will depend on the quality of the input schedule but SCR retains no relationship with the input schedule. For small problems, the solution quality is very near in all three cases, but *DP outperforms the other two if the problem size is large and the running time is limited.*

7.2.6 Discussion

The approach we describe provides excellent results on difficult benchmark problems. Although this is far from a guarantee that the approach will generalise successfully to real problems, and/or perform just as well on different and larger benchmarks, it is clearly a promising enough basis for continued research along these lines.

It is particularly encouraging that even the best results were achieved with essentially simple heuristics, improvements on which can be readily imagined. This augments a continuing theme in GA research literature, which shows that GAs begin to compete closely with or outperform other known methods on some problems when successfully hybridised with a well-chosen heuristic, for example: [Ling 92, Reeves 94].

Considering the interplay between the GA and the heuristics, these results appear counter to findings like those of [Bagchi *et al.* 91, Uckun *et al.* 93], which suggest that the more search done directly by the GA at the expense of the heuristic, the better in terms of final solution quality, though probably at the expense of time. Our results, and those of other authors in other applications, tend to show the opposite: better quality results arrive through hybridisation with heuristic methods, with little extra time cost. Reconciliation of such counter observations are readily found however, by recognising that these observations arise from a necessarily limited amount of experiments. Better solution quality may well have arrived here through a 'pure' GA approach (such as JOB+OP) but only at a rather extreme cost in time; for example: for JOB+OP to compete in terms of solution quality with EHC may be possible, but perhaps only if we use much larger population sizes and consequently wait far longer for convergence. This also explains the observation that JOB+OP can produce good results in the 4×4 problems, but its performance becomes poorer when the problem size increases. Bagchi *et al*'s notion makes sense if we consider that it allows the GA full rein over the space of possible solutions, rather than searching a smaller space as is effectively done in most hybrid methods. However, this 'expansion' of the space that we allow the GA to survey carries with it the need for more extensive sampling and hence much larger population sizes. A hybrid GA/heuristic method thus tends to seem the better practical choice, offering a better tradeoff in terms of speed *vs* quality on most problems.

The OSSP problems referred to can be obtained via [Beasley 90]. The *OR library* referred to in the latter article is an electronic library from which may be obtained benchmarks for a wide range of OR problems.

Chapter 8

Conclusions and Future Research

8.1 Conclusions

8.1.1 GAs in Timetabling

We have shown that a Genetic Algorithm approach is very effective and useful on the AI/CS MSc exam and lecture/tutorial timetabling problems. Using the methods we have described shows great potential for leading to timetables in future which are fairer to students (and so may even produce better academic results in the end), and also give the human expert timetablers more time for other pressing jobs.

Our GA/timetabling framework seems directly applicable (perhaps with various minor extensions/changes) to a very wide variety of other timetabling problems. For example, experimental results show that a key aspect towards its success is the employment of the smart mutation operators described, especially SEFM. SEFM makes much use of problem specific constraint information, but this information is directly available for any timetabling-like problem. As we have shown, extensions of the basic technique to handle room constraints, and deal with events of different lengths, have been successful. This shows promise for similar extensions that may be needed to deal with other sundry kinds of constraints that may arise in other problems. There are many types of problem, not necessarily arising in educational institutions , involving mainly edge and/or event spread constraints, which seem able to be addressed by our GA/Timetabling framework. For example, conference timetabling, timetabling events

at an athletics meeting, airline timetabling, court proceedings timetabling, and so on.

The GA/timetabling framework has been shown to be successful on several real problems of 'University Department size', and so it seems we can justify the expectation for it to work very well on most other problems of similar size and nature. That is, there is no reason to suspect that there is anything particularly easy about the problems it was tested on, in comparison to other real problems. Much work remains to do to see how performance scales to larger and otherwise different kinds of timetabling problems. There are no clear reasons to expect performance to degrade significantly simply if we try problems with more events; what matters is how tightly constrained the problem is; this is a complicated function of the combination of the number of events, number of slots, and numbers and kinds of constraints. It may be the case that performance will degrade significantly on extremely tightly constrained problems. However, in a sense this does not matter very much in real applications; either suboptimal solutions will be acceptable anyway, or the administrators will be prepared to relax the problem (making one more day available for timetabling can make a very great difference); experience suggests that very tightly constrained real timetabling problems are usually relaxable in this way.

In summary, we have shown that our GA/timetabling framework, involving a direct chromosome representation backed up with smart mutation operators, seems robust and very successful over a range of real timetabling problems which have arisen at the EDAI. Comparing GA results with independently produced efforts by human timetablers on some of these problems, the benefits of the GA are clearly apparent.

8.1.2 GAs in Scheduling

We have described a promising new GA approach to the JSSP. The GA representation we describe seems to provide reasonably good solutions to JSSPs quickly, and also permits fast, good quality dynamic job-shop scheduling, and rescheduling. In particular, we look at various extensions to the basic approach. In attempt to counteract the confounding effects of the context sensitivity of different positions in the chromosome, whereby genes interpreted earlier converge more quickly, we describe GVOT. This was shown to be mildly successful. An alternative idea for improving performance was the

idea of evolving optional idle time within gaps in partial schedules (EIWG), thus allowing more freedom and possibilities for the GA's operators. This also offered improved overall performance on some problems. Thirdly, we described an idea to alleviate the possible problems caused to the GA by the fact that, with the basic representation we use, genes tend to lose their full meaning the later they are interpreted; by using DCVG, introducing 'virtual', or 'dummy' genes to delay the onset of this loss of meaning, we find good overall improvement on some JSSP benchmarks. In particular, it seems most useful, as might be expected, when the number of operations to schedule falls in proportion to the number of jobs. The general approach used extends easily to dynamic job-shop scheduling environments, but in particular we have demonstrated that it shows very strong general robustness over a wide range of different schedule quality criteria. This last point is particularly interesting, since it seems that our JSSP framework to be a powerful *general* job-shop scheduling tool. For example, rather than going to the trouble of designing and testing specialised heuristic rules for a given JSSP-type problem with its own idiosyncratic requirements for a schedule quality criterion, our results suggest that direct application of our JSSP framework is likely to lead to as good as if not better results than any such specialised rule.

We have also presented an approach to the OSSP/OSRP which performs very promisingly on some simple benchmark OSSPs, outperforming previous reported benchmark attempts we know of in many cases. We notice that the obvious extension to our JSSP approach for the OSSP(JOB+OP) is not particularly impressive. This seems to be because the GA is searching a much larger space in which there are far more possible places to schedule operations at any stage during schedule building. We therefore attempted more thoughtful methods of choosing the operations to schedule. The first method was to directly incorporate a fixed heuristic (FH) for choosing which operation to schedule next into the schedule builder. This generally worked better than JOB+OP, but performance on a given size of problem was sensitive to the choice of heuristic. A further method was tried in which we allow the possibility of any of a number of heuristics to be chosen each time the schedule builder needs to schedule an operation. These heuristic choices are incorporated in the chromosome, and hence evolve along with the rest of the chromosome (which encodes the choice of job to schedule at each stage). This method, EHC, was found to be very successful indeed. Using

EHC we have been able to find new best results on several benchmark OSSPs.

The framework for scheduling is successful on a wide variety of benchmark problems of varying size and type. There is hence promise for its use as the basis of an approach to real problem instances, but this is yet to be tested. For example, by using the *dependency analysis* technique described, we can achieve good fast rescheduling. It is reasonable to say that it will perform well on problems similar in size and nature of constraints to the various benchmarks. For instance, many real job shop scheduling problems may have only a partial ordering imposed on the tasks within a job, and/or may allow a choice of different machines for many of the tasks. In most such cases, it is easy to see how the scheduling framework can be easily extended to deal with it. It remains to be seen if good performance still arises, but there seems no clear reason to expect such changes to the framework to undermine its success. Different kinds of scheduling problems, for example vehicle routing, production planning, project scheduling, personnel scheduling, line-balancing problem, may need significant changes or additions to the framework, but many real problems will need little change. In a nutshell, the simplicity of the approach, its apparent success, and the evident potential for much further improvement and extension, seem to render it a promising method warranting further research.

8.2 Future Research

8.2.1 GAs in Timetabling

Further work is under way to more thoroughly test the performance of our technique on a wider variety of timetabling problems. An example is the problem of timetabling paper presentations at conferences with parallel sessions; if each delegate was given the opportunity to provide the organisers with their individual preferences regarding the papers to be presented, a GA based timetabling system very similar to the one we describe could arrange for parallel sessions to be organised such that delegates are collectively satisfied as far as possible, in terms of minimising the degree to which two presentations a delegate wishes to see are scheduled to occur at the same time.

Comparison of our GA/timetabling framework with alternative GA-based approaches

is also necessary, and in preparation. In particular, an alternative worth considering is one which uses an indirect chromosome representation, similar to the methods we use for the JSSP and OSSP. Whereas the direct representation searches the complete space of valid timetables, the indirect approach (for example: [Paechter 94, Paechter *et al.* 94]) searches a subspace much more dense in 'feasible' points. This suggests that solutions may be found more quickly in terms of number of evaluations. However, chromosome interpretation, and hence the time taken for an evaluation, will typically be much slowed down in such an approach. Also, the use of a direct representation makes it much easier for us to trace and analyse the problems involved with individual timetables [Ross *et al.* 94a, Corne *et al.* 94b], which therefore makes it possible to design directed mutation operators such as SEFM. Further work is warranted to investigate the pros and cons of these two approaches and perform comparisons in terms of speed and quality between them [Corne *et al.* 94a].

Several other methods for addressing timetabling problems exist. For example, simulated annealing, tabu search, graph colouring, and the constraint logic programming. Though some of these methods may be faster in terms of speed to find a good solution, some show obvious problems. For example, constraint satisfaction based approaches tend to only be able to find a solution if one exists which satisfies all of the constraints, otherwise they are more difficult to deal with soft constraints. However, the GA approach manages this by handling hard and soft constraints in a uniform way. Also, simulated annealing, tabu search, and heuristic search tend to lead to a single result, while with the GA it is typically possible to produce multiple distinct solutions in a single run (without increasing the number of generations or the population size); methods such as *sharing* [Goldberg & Richardson 87, Deb & Goldberg 89] would enhance this effect. Each method tends to have its unique advantages and disadvantages, and so they are difficult to compare. However, such comparison is needed in order to discover any important differences in performance.

Considering details of the GA/timetabling framework itself, several thoughts have been gleaned from the course organiser[1] on possible useful improvements to the event timetabling process, although these are not necessarily GA issues. The reason why not

[1] Dr Robert Fisher

is that so far we have used the GA mainly to resolve important stated constraints, but not for aesthetic and similar issues which often concern human timetablers. So, there might be some value in a post-processing module that can:

- move events from one slot into an adjacent slot, subject to not violating any additional constraints, with the goal of packing the rooms more completely so as to reduce room bookings and invigilation requirements.
- if a set of events is at 16:00 and there are no events at 14:00, then shift all events into that time period, subject to not violating any additional constraints. Similarly for shifting from 9:30 to 11:30. The goal is to try to keep events near the middle of the day, which keeps everyone happier.
- assess the average spread of the events in some way, that is, if we have 2 timetables that satisfy all of the constraints, the one with the larger average event spread is the better. This gives the students more time between events. How to quantify this is unclear - we would not want a set of events over an 11 day timetable with 1 on the 1st day, 1 on the last day and the rest on 3 consecutive days in the middle, for example.

Such criteria, or approximations to them, could be built into the evaluation function, but as it stands they are very much secondary criteria compared to those we have dealt with already.

8.2.2 GAs in Scheduling

Further work is under way to more thoroughly test the performance of our technique on the benchmark JSSPs/OSSPs and on a set of real world scheduling problems. Since such work involves indirect chromosome interpretation, and as a result is particularly time consuming, we are also planning to implement these experiments on transputer based parallel processing systems [Fogarty & Huang 90]. Such further work is necessary to assess properly how our methods scale up to larger and/or more realistic problems.

Having shown that our GA based scheduling method performs very strongly in comparison with a wide range of heuristic rules in the context of several different schedule quality criteria, it seems justifiable to expect reasonable success on such larger and more realistic problems.

The more general statement of a job shop problem is a more complex matter. For example, an operation may have a collection of alternative process plans (it can be done on different machines), rather than the single process plan of being specified to be done on a particular machine. Let each job j_i have p_i alternative process plans. Each such plan is a distinct set of machines and associated processing times, each representing an alternative way of discharging the job. We might extend our representation to incorporate such alternatives as follows: a schedule $abcd\cdots$ means: "choose an untackled operation from the the a th uncompleted job, using the b th valid process plan for this job, and place it into the earliest place where it will fit in the developing schedule, choose an untackled operation from the the c-th uncompleted job, using the d th valid process plan for this job, and place ..." and so on. Here, when the schedule builder identifies the job currently referred to in the chromosome (via the heuristic choice), earlier choices in the schedule constrain the set of valid alternative process plans that are still 'live' for this job. The valid set is treated as circular, and chosen from as directed by the chromosome. There are of course several other possibilities. For example, we could use essentially the same representation as used for the simpler OSSP, but change the interpretation to: "heuristically choose a valid process plan from the a th uncompleted job and then heuristically choose an operation from this plan, and place it into the earliest place where it will fit in the developing schedule, heuristically choose a process plan from the b th uncompleted job and then ..." and so on. This involves the addition of a heuristic to choose the process plan as well as choose an operation. Possibilities for the heuristic which chooses the process plan are easily imagined. For example, we might choose the plan for which the total processing time remaining is smallest.

Finally, various possibilities are apparent for extending the approach to deal with more general problems. Such has been reported, for example, in the context of highly generalised manufacturing scheduling problems [Husbands & Mill 91], although this did

not report on the hybridisation of the GA with simple heuristics (chromosomes were much more direct representations of schedules). Our main point here is to note how our approach readily allows for extensions which will allow it to cope with problems of the fully general kind found in real machine shop environments, while still retaining its basic flavour, and hence retaining the presumed source of its success. It may not be immediately apparent that the success we demonstrate on simplified benchmark JSSPs/OSSPs will carry over to effective performance on more complex problems in an extended approach, but there is no apparent reason to be too skeptical of this possibility either.

In conclusion, this thesis has reported on a wide collection of techniques and experiments which show generally strong potential for the application of GAs to real timetabling and scheduling problems. Rather than in-depth study on small aspects of applications, we have performed extensive general experimentation using a wide range of candidate techniques on a wide range of problems, in order to obtain a general flavour for the potential of GAs in these areas as a whole. The results have been very promising, and have led to many ideas for further work and continued study.

Bibliography

[Aarts & Korst 89] Emile H.L. Aarts and Jan H.M. Korst. Boltzmann machine for travelling salesman problems. *European Journal of Operations Research*, 39:79–95, 1989.

[Abramson & Abela 91] D. Abramson and J. Abela. A parallel genetic algorithm for solving the school timetabling problem. Technical report, Division of Information Technology, C.S.I.R.O., April 1991.

[Abramson 91] D. Abramson. Constructing school timetables using simulated annealing: sequential and parallel algorithms. *Management Science*, 37(1):98–113, January 1991.

[Adams *et al.* 88] Joseph Adams, Egon Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, March 1988.

[Altenberg 94] Lee Altenberg. The Schema Theorem and Price's Theorem. Technical report, Institute of Statistics and Decision Sciences, Duke University, 1994.

[Anderson 94] Edward J. Anderson. *The Management of Manufacturing: Model and Analysis.* Addison-Wesley, 1994.

[Angéniol *et al.* 88] Bernard Angéniol, Gaël De La Croix Vaubois, and Jean-Yves Le Texier. Self-organizing feature maps and the travelling salesman problem. *Neural Networks*, 1:289–293, 1988.

[Arani *et al.* 88] Taghi Arani, Mark Karwan, and Vahid Lotfi. A lagrangean relaxation approach to solve the second phase of the exam scheduling problem. *European Journal of Operations Research*, 34:372–383, 1988.

[Atlan *et al.* 93] Laurent Atlan, Jerome Bonnet, and Martine Naillon. Learning distributed reactive strategies by genetic programming for the general job shop problem. In *Proceedings of the Seventh Annual Florida Artificial Intelligence Research Symposium*, 1993.

[Bagchi *et al.* 91] Sugato Bagchi, Serdar Uckun, Yutaka Miyabe, and Kazuhiko Kawamura. Exploring problem-specific recombination operators for job shop scheduling. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 10–17. San Mateo: Morgan Kaufmann, 1991.

[Baker & McMahon 85] J. R. Baker and G. B. McMahon. Scheduling the general job-shop. *Management Science*, 31(5):594–598, 1985.

[Baker 74] Kenneth R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley and Sons, Inc., 1974.

[Baker 85] James Edward Baker. Adaptive selection methods for genetic algorithms. In J. J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 101–111. San Mateo: Morgan Kaufmann, 1985.

[Balas 69] E. Balas. Machine sequencing via disjunctive graphs: An implicit enumeration algorithm. *Operations Research*, 17:941–957, 1969.

[Beasley 90] J.E. Beasley. OR-library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41:1069–1072, 1990.

[Beasley 93] John E Beasley. Chapter 6: Lagrangean relaxation. In Colin R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Publications, 1993.

[Beck *et al.* 91] Howard Beck, Brian Drabble, and Richard Kirby. *Course Notes on Planning and Scheduling.* Artificial Intelligence Applications Institute, University of Edinburgh, 1991.

[Belew & Booker 91] R.K. Belew and L.B. Booker, editors. *Proceedings of the Fourth International Conference on Genetic Algorithms.* San Mateo: Morgan Kaufmann, 1991.

[Bellman *et al.* 82] R. Bellman, A. O. Esogube, and I. Nabeshima. *Mathematical Aspects of Scheduling & Applications.* Oxford: Pergamon Press, 1982.

[Berry *et al.* 93] Pauline M. Berry, Howard Beck, and Tim Duncan. *Course Notes on Advanced Scheduling Methods.* Artificial Intelligence Applications Institute, University of Edinburgh, 1993.

[Bethke 81] A. D. Bethke. *Genetic Algorithms as Function Optimizers.* Unpublished PhD thesis, University of Michigan, Ann Arbor, 1981.

[Blackstone Jr *et al.* 82] J.H. Blackstone Jr, D.T. Philips, and G.L. Hogg. A state-of-art survey of dispatching rules for manufacturing job shop operations. *International Journal of Production Research*, 20(1):27–45, 1982.

[Brindle 81] A. Brindle. *Genetic Algorithms for Function Optimization.* Unpublished PhD thesis, University of Alberta, Edmonton, Canada, 1981.

[Bruns 93] Ralf Bruns. Direct chromosome representation and advanced genetic operators for production scheduling. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 352–359. San Mateo: Morgan Kaufmann, 1993.

[Bui & Moon 94] T. Bui and B. Moon. A new genetic approach for the travelling salesman problem. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 7–12, 1994.

[Burke & Prosser 91] P. Burke and P. Prosser. A distributed asynchronous system for predictive and reactive scheduling. *Artificial Intelligence in Engineering*, 6(3):106–124, 1991.

[Burke *et al.* 94] Edmund Burke, David Elliman, and Rupert Weare. A genetic algorithm for university timetabling. In *AISB Workshop on Evolutionary Computation.* Workshop Notes, 1994.

[Cangalovie & Schreuder 91] Mirjana Cangalovie and Jan A.M. Schreuder. Exact colouring algorithm for weighted graphs applied to timetabling problems with lectures of different lengths. *European Journal of Operations Research*, 51:248–258, 1991.

[Carlier & Pinson 89] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, February 1989.

[Carroll 65] D. C. Carroll. *Heuristic Sequence of Single and Multiple Component Jobs.* Unpublished PhD thesis, Sloan School of Management, M.I.T., Cambridge, MA, 1965.

[Carter 86] Michael W. Carter. A survey of practical applications of examination timetabling algorithms. *Operations Research*, 34(2):193–202, March-April 1986.

[Cartwright & Mott 91] Hugh M. Cartwright and Gregory F. Mott. Looking around: using clues from the data space to guide genetic algorithm searches. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 108–114. San Mateo: Morgan Kaufmann, 1991.

[Chan 94] Yam Ling Chan. A genetic algorithm (GA) shell for iterative timetabling. Unpublished M.Sc. thesis, Department of Computer Science, RMIT, 1994.

[Ciriani & Leachman 93] Tito A. Ciriani and Robert C. Leachman, editors. *Optimization in Industry: Mathematical Programming and Modeling Techniques in Practice.* John Wiley & Sons Ltd, 1993.

[Cleveland & Smith 89] Gary A. Cleveland and Stephen F. Smith. Using genetic algorithms to schedule flow shop releases. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 160–169. San Mateo: Morgan Kaufmann, 1989.

[Collinot *et al.* 88] A. Collinot, C. LePape, and G. Pinoteau. SONIA: A knowledge-based scheduling system. *artificial intelligence in engineering*, 3(2), 1988.

[Collins & Jefferson 91] Robert J. Collins and David R. Jefferson. Selection in massively parallel genetic algorithms. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 249–256. San Mateo: Morgan Kaufmann, 1991.

[Colorni *et al.* 90] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Genetic algorithms and highly constrained problems: The time-table case. In G. Goos and J. Hartmanis, editors, *Parallel Problem Solving from Nature*, pages 55–59. Springer-Verlag, 1990.

[Conway *et al.* 67] Richard W. Conway, William L. Maxwell, and Louis W. Miller. *Theory of Scheduling.* Addison Wesley Publishing Company, 1967.

[Corne *et al.* 93] Dave Corne, Hsiao-Lan Fang, and Chris Mellish. Solving the module exam scheduling problem with genetic algorithms. In Paul W.H. Chung, Gillian Lovegrove, and Moonis Ali, editors, *Proceedings of the Sixth International Conference in Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 370–373. Gordon and Breach Science Publishers, 1993.

[Corne *et al.* 94a] Dave Corne, Peter Ross, and Hsiao-Lan Fang. Evolutionary timetabling: Practice, prospects and work in progress. In *The 13th UK Planning and Scheduling Special Interest Group*, 1994.

[Corne *et al.* 94b] Dave Corne, Peter Ross, and Hsiao-Lan Fang. Fast practical evolutionary timetabling. In Terence C. Fogarty, editor, *Proceedings of the AISB Workshop on Evolutionary Computation*. Springer-Verlag, 1994.

[Croce *et al.* 92] Della Croce, R. Tadei F., and G. Volta. A genetic algorithm for the job shop problem. Technical report, D.A.I., Politecnico di Torino, Italy, 1992.

[Darwin 59] Charles Darwin. *The Origin of Species*. John Murray, 1859.

[Davidor 91] Yuval Davidor. A naturally occurring niche & species phenomenon: The model and first results. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 257–263. San Mateo: Morgan Kaufmann, 1991.

[Davidor 94] Y. Davidor, editor. *Parallel Problem Solving from Nature III*. Springer-Verlag, 1994.

[Davidor *et al.* 93] Yuval Davidor, Takeshi Yamada, and Ryohei Nakano. The ECOlogical framework II: Improving GA performance at virtually zero cost. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 171–176. San Mateo: Morgan Kaufmann, 1993.

[Davis 85] L. Davis. Job shop scheduling with genetic algorithms. In J. J. Grefenstette, editor, *Proceedings of the International Conference on Genetic Algorithms and their Applications*, pages 136–140. San Mateo: Morgan Kaufmann, 1985.

[Davis 91] L. Davis, editor. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold, 1991.

[Deb & Goldberg 89] Kalyanmoy Deb and David E. Goldberg. An investigation of niche and species formation in genetic function optimization. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 42–50. San Mateo: Morgan Kaufmann, 1989.

[Dincbas *et al.* 88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic

Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-88), ICOT, Tokyo*, November 1988.

[Dorndorf & Pesch 93] Utrich Dorndorf and Etwin Pesch. *Evolution Based Learning in a Job Shop Scheduling Environment.* INFORM, 1993.

[Dowsland 90] K. A. Dowsland. A timetabling problem in which clashes are inevitable. *Journal of Operations Research society*, 41:907–918, 1990.

[Duncan & Johnson 91] Tim Duncan and Ken Johnson. *Course Notes on Reasoning with Constraints.* Artificial Intelligence Applications Institute, University of Edinburgh, 1991.

[Duncan 93] Tim Duncan. A Review of Commercially Available Constraint Programming Tools. Technical Report AIAI-TR-149, Artificial Intelligence Applications Institute, University of Edinburgh, 1993.

[Durbin & Willshaw 87] R. Durbin and D. J. Willshaw. An analogue approach to the travelling salesman problem using an elastic net method. *Nature*, 326:689–691, 1987.

[Eglese & Rand 87] R. W. Eglese and G. K. Rand. Conference seminar timetabling. *Journal of Operations Research society*, 38:591–598, 1987.

[Eshelman & Schaffer 91] Larry J. Eshelman and J. David Schaffer. Preventing premature convergence in genetic algorithms by preventing incest. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 115–122. San Mateo: Morgan Kaufmann, 1991.

[Fang 92] Hsiao-Lan Fang. Investigating Genetic Algorithms in scheduling. Technical Paper 12, Department of Artificial Intelligence, University of Edinburgh, 1992.

[Fang *et al.* 93] Hsiao-Lan Fang, Peter Ross, and Dave Corne. A promising Genetic Algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 375–382. San Mateo: Morgan Kaufmann, 1993.

[Fang *et al.* 94] Hsiao-Lan Fang, Peter Ross, and Dave Corne. A promising hybrid GA/heuristic approach for open-shop scheduling problems. In A. Cohn, editor, *Proceedings of the 11th European Conference on Artificial*

Intelligence, pages 590–594. John Wiley & Sons, Ltd., 1994.

[Fisher & Thompson 63] H. Fisher and G. L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In J. F. Muth and G. L. Thompson, editors, *Industrial Scheduling*, pages 225–251. Prentice Hall, Englewood Cliffs, New Jersey, 1963.

[Fitzpatrick & Grefenstette 88] J.M. Fitzpatrick and J.J. Grefenstette. Genetic algorithms in noisy environments. *Machine Learning*, 3(2/3):101–120, 1988.

[Fogarty & Huang 90] Terence C. Fogarty and Runhe Huang. Implementing the Genetic Algorithm on transputer based parallel processing systems. In G. Goos and J. Hartmanis, editors, *Parallel Problem Solving from Nature*, pages 145–149. Springer-Verlag, 1990.

[Forrest 93] Stephanie Forrest, editor. *Proceedings of the Fifth International Conference on Genetic Algorithms.* San Mateo: Morgan Kaufmann, 1993.

[Fox & McMahon 91] B.R. Fox and M.B. McMahon. Genetic operators for sequencing problems. In J. E. Gregory, editor, *Foundations of Genetic Algorithms*, pages 284–300. San Mateo: Morgan Kaufmann, 1991.

[Fox & Smith 84] M. Fox and S. Smith. ISIS: A knowledge-based system for factory scheduling. *Expert Systems*, 1, July 1984.

[French 82] S. French. *Sequencing and Scheduling : an introduction to the mathematics of the job-shop.* Ellis Horwood Limited, 1982.

[Fritzke & Wilke 91] Bernd Fritzke and Peter Wilke. Flexmap - a neural network for the travelling salesman problem with linear time and space complexity. Technical report, Universität Erlangen-Nürnberg, 1991.

[Garey & Johnson 79] Michael R. Garey and David S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, 1979.

[Giffler & Thompson 60] B. Giffler and G. L. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8(4):487–503, 1960.

[Gislen *et al.* 89] Lars Gislen, Carsten Peterson, and Bo Soderberg. 'Teachers and Classes' with neural networks. *International Journal of Neural Systems*, 1(2):167–176, 1989.

[Gislen *et al.* 92] Lars Gislen, Carsten Peterson, and Bo Soderberg. Complex scheduling with Potts neural networks. *Neural Computation*, 4:805–831, 1992.

[Glover & Laguna 93] Fred Glover and Manuel Laguna. Chapter 3: Tabu search. In Colin R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, 1993.

[Glover 86] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Ops. Res.*, 5:533–549, 1986.

[Glover 87] David E. Glover. Solving a complex keyboard configuration problem through generalized adaptive search. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 12–31. San Mateo: Morgan Kaufmann, 1987.

[Goldberg & Richardson 87] David E. Goldberg and Jon Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 41–49. San Mateo: Morgan Kaufmann, 1987.

[Goldberg 87] D. E. Goldberg. Simple genetic algorithms and the minimal, deceptive problem. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 74–88. San Mateo: Morgan Kaufmann, 1987.

[Goldberg 89] David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Reading: Addison Wesley, 1989.

[Goldberg 90] David E. Goldberg. A note on Boltzmann tournament selection for genetic algorithms and population-oriented simulated annealing. *Complex Systems*, 4:445–460, 1990.

[Goldratt & Cox 84] E. Goldratt and J. Cox. *The Goal*. Aldershot, UK: Gower, 1984.

[Goldratt & Fox 86] E. Goldratt and R. Fox. *The Race*. North River Press Inc., 1986.

[Goldratt 88] Eliyahu M. Goldratt. Computerized shop floor scheduling. *International Journal of Production Research*, 26(3):443–455, 1988.

[Gonzalez & Sahni 76] Teofilo Gonzalez and Sartaj Sahni. Open shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery*, 23(4):665–679, October 1976.

[Goos & Hartmanis 90] G. Goos and J. Hartmanis, editors. *Parallel Problem Solving from Nature*. Springer-Verlag, 1990.

[Gorges-Schleuter 90] M. Gorges-Schleuter. Explicit parallelism of genetic algorithms through population structures. In G. Goos and J. Hartmanis, editors, *Parallel Problem Solving from Nature*, pages 150–159. Springer-Verlag, 1990.

[Grefenstette & Baker 89] John J. Grefenstette and James E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 20–27. San Mateo: Morgan Kaufmann, 1989.

[Grefenstette 85] J. J. Grefenstette, editor. *Proceedings of the First International Conference on Genetic Algorithms and their Applications*. San Mateo: Morgan Kaufmann, 1985.

[Grefenstette 87] J. J. Grefenstette, editor. *Proceedings of the Second International Conference on Genetic Algorithms and their Applications*. San Mateo: Morgan Kaufmann, 1987.

[Grefenstette 93] J. J. Grefenstette. Deception considered harmful. In L. Darrell Whitely, editor, *Foundations of Genetic Algorithms 2*, pages 75–91. San Mateo: Morgan Kaufmann, 1993.

[Grefenstette *et al.* 85] J. J. Grefenstette, Rajeev Gopal, Brain Rosmaita, and Dirk Van Gucht. Genetic algorithms for the travelling salesman problem. In J. J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 160–168. San Mateo: Morgan Kaufmann, 1985.

[Gregory 91] J. E. Gregory, editor. *Foundations of Genetic Algorithms*. San Mateo: Morgan Kaufmann, 1991.

[Hansen 86] P. Hansen. *Congress on Numerical Methods in Combinatorical Optimization*. Capri, italy, 1986.

[Harvey 93] Inman Harvey. The puzzle of the persistent question marks: A case study of genetic drift. In Stephanie Forrest, editor, *Proceedings of the Fifth International*

Conference on Genetic Algorithms, pages 15–22. San Mateo: Morgan Kaufmann, 1993.

[Hax & Candea 84] Arnoldo C. Hax and Dan Candea. *Production and Inventory Management.* Prentice-Hall, 1984.

[Held & Karp 62] M. Held and R. M. Karp. A dynamic programming approach to sequencing problem. *SIAM*, 1962.

[Hertz 91] A. Hertz. Tabu search for large scale timetabling problems. *European Journal of Operations Research*, 54:39–47, 1991.

[Hilliard *et al.* 87] M.R. Hilliard, G.E. Liepins, Mark Palmer, Michael Morrow, and Jon Richardson. A classifier-based system for discovering scheduling heuristics. In *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 231–235. San Mateo: Morgan Kaufmann, 1987.

[Holland 75] John H. Holland. *Adaptation in Natural and Artificial Systems.* Ann Arbor: The University of Michigan Press, 1975.

[Hopfield & Tank 85] J. J. Hopfield and D. Tank. 'neural' computation of decisions in optimization problems. *Biological Cybernetics*, 5:141–152, 1985.

[Huhns 87] M.H. (ed) Huhns. *Distributed Artificial Intelligence 1.* Pitman, London, 1987.

[Hulle 91] M. M. Van Hulle. A goal programming network for mixed integer linear programming: A case study for the job-shop scheduling problem. *International Journal of Neural Systems*, 2(3):201–209, 1991.

[Husbands & Mill 91] P. Husbands and F. Mill. Simulated co-evolution as the mechanism for emergent planning and scheduling. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 264–270. San Mateo: Morgan Kaufmann, 1991.

[Husbands 93] P. Husbands. An ecosystems model for integrated production planning. *Inter. J. Compt. Integrated Manufacturing*, 1993.

[Husbands *et al.* 90] P. Husbands, F. Mill, and S. Warrington. Genetic algorithms, production planning optimisation and scheduling. In G. Goos and J. Hartmanis, editors, *Parallel Problem Solving from Nature*, pages 80–84. Springer-Verlag, 1990.

[Kiran 84a] Ali S. Kiran. Simulation studies in job shop scheduling - I : A survey. *Computers & Industrial Engineering*, 8:87–93, 1984.

[Kiran 84b] Ali S. Kiran. Simulation studies in job shop scheduling - II : Performance of priority rules. *Computers & Industrial Engineering*, 8:95–105, 1984.

[Kirkpatrick *et al.* 83] S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Korman 79] S. M. Korman. Chapter 8: The graph-colouring problem. In N. Christofides, A. Mingozzi, P. Toth, and C. Sandi, editors, *Combinatorial Optimization*. Wiely, Chichester, 1979.

[Laarhoven *et al.* 92] Peter J.M. Van Laarhoven, Emile H.L. Aarts, and Jan Karel Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40(1):113–125, Jan-Feb 1992.

[Le Pape 93] Claude Le Pape. Using Object-Oriented Constraint Programming Tools to Implement Flexible "Easy-to-use" Scheduling Systems. In *Proceedings of the NSF Workshop on Intelligent, Dynamic Scheduling for Manufacturing*, Cocoa Beach, Florida, 1993.

[Ling 92] Si-Eng Ling. Intergating genetic algorithms with a Prolog assignment problem as a hybrid solution for a polytechnic timetable problem. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature II*, pages 321–329. Elsevier Science Publisher B.V., 1992.

[Lui 88] B. Lui. *Reinforcement planning for Resource allocation and constraint satisfaction.* Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1988.

[Manner & Manderick 92] R. Manner and B. Manderick, editors. *Parallel Problem Solving from Nature II.* Elsevier Science Publisher B.V., 1992.

[Mason 93] Andrew J. Mason. Crossover non-linearity ratios and the genetic algorithm: Escaping the blinkers of Schema processing and Instrinsic Parallelism. Technical report, University of Auckland, New Zealand, September 1993.

[McMahon & Florian 75] G. McMahon and M. Florian. On scheduling with ready times and due dates to minimize maximum

lateness. *Operations Research*, 23(3):475–482, March-April 1975.

[Meisels *et al.* 93] Amnon Meisels, Jihad Ell-sana', and Ehud Gudes. Comments on CSP algorithms applied to timetabling. Technical report, Department of Mathematics and Computer Science, Ben-Gurion University, Israel, 1993.

[Michalewicz 92] Zbigniew Michalewicz. Chapter 8: The travelling salesman problem. In *Genetic Algorithms + Data §tructures = Evolution Programs*. Springer-Verlag, 1992.

[Moore 68] J. M. Moore. Sequencing n jobs on one machine to minimize the number of late jobs. *Management Science*, 15, 1968.

[Morton & Pentico 93] T.E. Morton and D.W. Pentico. *Heuristic Scheduling Systems*. John Wiley, 1993.

[Mühlenbein 91] H. Mühlenbein. Evolution in time and space - the parallel genetic algorithm. In J. E. Gregory, editor, *Foundations of Genetic Algorithms*, pages 316–338. San Mateo: Morgan Kaufmann, 1991.

[Muller *et al.* 93] C. Muller, E.H. Magill, and D.G. Smith. Distributed genetic algorithms for resource allocation. In J. Dorn and K.A. (eds) Froeschl, editors, *Scheduling of Production Processes*. Ellis Horwood Limited, 1993.

[Murata & Ishibuchi 94] Tadahiko Murata and Hisao Ishibuchi. Performance evaluation of genetic algorithms for flowshop scheduling problems. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 812–817, 1994.

[Nakano 91] Ryohei Nakano. Conventional genetic algorithms for job shop problems. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 474–479. San Mateo: Morgan Kaufmann, 1991.

[Ogbu & Smith 91] F.A. Ogbu and D.K. Smith. Simulated annealing for the permutation flow-shop problem. *OMEGA*, 19:64–67, 1991.

[Oplobedu *et al.* 89] A. Oplobedu, J. Marcovitch, and Y. Tourbier. Charme: Un Langage Industriel de Programmation

par Contraintes, Illustré par une Application chez Renault. In *Ninth International Workshop on Expert Systems and their Applications: General Conference, Volume 1*, pages 55–70, Avignon, May 29th – June 2nd 1989. EC2.

[Osman & Potts 89] I. H. Osman and C. N. Potts. Simulated annealing for permutation flow-shop scheduling. *OMEGA*, 17:551–557, 1989.

[Paechter 94] Ben Paechter. Optimising a presentation timetable using evolutionary algorithms. In Terence C. Fogarty, editor, *Proceedings of the AISB Workshop on Evolutionary Computation*. Springer-Verlag, 1994.

[Paechter *et al.* 94] B. Paechter, H. Luchian, A. Cumming, and M.Petruic. Two solutions to the general timetable problem using evolutionary methods. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 300–305, 1994.

[Panwalkar & Iskander 77] S. S. Panwalkar and Wafik Iskander. A survey of scheduling rules. *Operations Research*, 25(1):45–61, 1977.

[Panwalkar *et al.* 73] S. S. Panwalkar, R. A. Dudek, and M. L. Smith. Sequencing research and the industrial scheduling problem. In S. E. Elmaghraby, editor, *Symposium on the Theory of Scheduling and Its Applications*, pages 29–38. Springer-Verlag, Berlin, 1973.

[Paredis 92] Jan Paredis. Exploiting constraints as background knowledge for genetic algorithms: a case-study for scheduling. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature II*, pages 229–238. Elsevier Science Publisher B.V., 1992.

[Puget 94] Jean-François Puget. A C++ Implementation of CLP. In *Ilog Solver Collected papers*. Ilog SA, 12 avenue Raspail, BP-7, 94251 Gentilly Cedex, France, 1994.

[Reeves 92a] C. R. Reeves. A genetic algorithm approach to stochastic flowshop sequencing. *Proc. IEEE Colloquium on Genetic Algorithms for Control and Systems Engineering*, 1992.

[Reeves 92b] C. R. Reeves. A genetic algorithm for flowshop sequencing. *Computers & Ops. Res.*, 1992.

[Reeves 93] C. R. Reeves. Improving the efficiency of tabu search for machine sequencing problems. *Journal of the Operational Research Society*, 1993.

[Reeves 94] C. R. Reeves. Hybrid genetic algorithms for bin-packing and related problems. Technical report, School of Mathematical and Information Sciences, Coventry University, 1994.

[Ross & Ballinger 93] Peter Ross and Geoffrey H. Ballinger. *PGA - Parallel Genetic Algorithm Testbed.* Department of Artificial Intelligence, University of Edinburgh, 1993.

[Ross & Hallam 93] Peter Ross and John Hallam. *Lecture Notes on Connectionist Computing.* Department of Artificial Intelligence, University of Edinburgh, 1993.

[Ross *et al.* 94a] Peter Ross, Dave Corne, and Hsiao-Lan Fang. Improving evolutionary timetabling with delta evaluation and directed mutation. In Y. Davidor, editor, *Parallel Problem Solving from Nature III.* Springer-Verlag, 1994.

[Ross *et al.* 94b] Peter Ross, Dave Corne, and Hsiao-Lan Fang. Successful lecture timetabling with evolutionary algorithms. In *Proceedings of the 11th ECAI Workshop.* Springer-Verlag, 1994.

[Ross *et al.* 94c] Peter Ross, Dave Corne, and Hsiao-Lan Fang. Timetabling by Genetic Algorithms: Issues and approaches. Technical Report AIGA-006-94, Department of Artificial Intelligence, University of Edinburgh, 1994. revised version to appear in Applied Intelligence.

[Sadeh 91] N. Sadeh. *Look-ahead techniques for Micro-opportunistic job-shop scheduling.* Unpublished PhD thesis, Carnegie-Mellon University, 1991.

[Schaffer 89] J. D. Schaffer, editor. *Proceedings of the Third International Conference on Genetic Algorithms and their Applications.* San Mateo: Morgan Kaufmann, 1989.

[Shen *et al.* 94] Chang Yun Shen, Yoh-Han Pao, and Percy Yip. Scheduling multiple job problems with guided evolutionary simulated annealing approach. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 702–706, 1994.

[Smith 89] John Maynard Smith. *Evolutionary Genetics.* Oxford University Press, New York, 1989.

[Smith *et al.* 86] S. Smith, M. Fox, and P. S. Ow. Constructing and maintaining detailed production plans: investigations into the development of knowledge-based factory scheduling systems. *A.I. Magazine*, 7(4), 1986.

[Solotorevsky *et al.* 91] Gadi Solotorevsky, Ehud Gudes, and Amnon Meisels. Raps - a rule-based language for specifying resource allocation and time-tabling problems. Technical report, Department of Mathematics and Computer Science, Ben-Gurion University, Israel, 1991.

[Solotorevsky *et al.* 93] Gadi Solotorevsky, Nimrod Gal-oz, Ehud Gudes, and Amnon Meisels. Comparing the rule-based and CSP approaches for solving a real-life course scheduling problem. Technical report, Department of Mathematics and Computer Science, Ben-Gurion University, Israel, 1993.

[Starkweather *et al.* 90] T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In G. Goos and J. Hartmanis, editors, *Parallel Problem Solving from Nature*, pages 176–185. Springer-Verlag, 1990.

[Starkweather *et al.* 91] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. A comparison of genetic sequencing operators. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 69–76. San Mateo: Morgan Kaufmann, 1991.

[Syswerda & Palmucci 91] G. Syswerda and J. Palmucci. The application of genetic algorithms to resource scheduling. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 502–508. San Mateo: Morgan Kaufmann, 1991.

[Syswerda 89] G. Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 2–9. San Mateo: Morgan Kaufmann, 1989.

[Syswerda 91] G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*, pages 332–349. New York: Van Nostrand Reinhold, 1991.

[Taillard 90] E. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operations Research*, 47:65–74, 1990.

[Taillard 93] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[Tamaki & Nishikawa 92] Hisashi Tamaki and Yoshikazu Nishikawa. A parallel genetic algorithms based on a neighborhood model and its application to jobshop scheduling. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature II*, pages 573–582. Elsevier Science Publisher B.V., 1992.

[Tamaki *et al.* 94] H. Tamaki, H. Kita, N. Shimizu, K. Maekawa, and Y. Nishikawa. A comparison study of genetic coding for the travelling salesman problem. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 1–6, 1994.

[Tanese 89] Reiko Tanese. *Distributed Genetic Algorithms for Function Optimization.* Unpublished PhD thesis, Department of Computer Science and Engineering, University of Michigan, 1989.

[Tripathy 80] Arabinda Tripathy. A lagrangean relaxation approach to course timetabling. *Journal of the Operational Research Society*, 31:599–603, 1980.

[Tripathy 84] Arabinda Tripathy. School timetabling – a case in large binary integer linear programming. *Management Science*, 30(12):1473–1489, December 1984.

[Uckun *et al.* 93] S. Uckun, S. Bagchi, K. Kawamura, and Y. Miyabe. Managing genetic search in job shop scheduling. *IEEE Expert*, pages 15–24, October 1993.

[Vepsalainen & Morton 87] A. Vepsalainen and T. E. Morton. Priority rules and leadtime estimation for job shop scheduling with weighted tardiness costs. *Management Science*, 33:1036–1047, 1987.

[Vepsalainen & Morton 88] A. Vepsalainen and T. E. Morton. Improving local priority rules with global leadtime estimates. *Journal of Manufacturing and Operations Management*, 1:102–118, 1988.

[Werra 85] D.de Werra. An introduction to timetabling. *European Journal of Operations Research*, 19:151–162, 1985.

[Whitley 89] Darrell Whitley. The GENITOR algorithm and selection pressure. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. San Mateo: Morgan Kaufmann, 1989.

[Whitley 93a] Darrell Whitley, editor. *Foundations of Genetic Algorithms 2.* San Mateo: Morgan Kaufmann, 1993.

[Whitley 93b] Darrell Whitley. A genetic algorithm tutorial. Technical report, Colorado State University, March 1993.

[Whitley *et al.* 89] Darrell Whitley, Timothy Starkweather, and D'Ann Fuquay. Scheduling problems and travelling salesmen: The genetic edge recombination operator. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 133–140. San Mateo: Morgan Kaufmann, 1989.

[Whitley *et al.* 91] D. Whitley, T. Starkweather, and D. Shaner. travelling salesman and sequence scheduling: Quality solutions using genetic edge recombination. In L. Davis, editor, *Handbook of Genetic Algorithms*, pages 350–372. New York: Van Nostrand Reinhold, 1991.

[Widmer & Hertz 89] M. Widmer and A. Hertz. A new heuristic method for the flow shop sequencing problem. *European Journal of Operations Research*, 41:186–193, 1989.

[Wilson 85] Robin James Wilson. *Introduction to Graph Theory.* Longman, London, 3nd edition, 1985.

[Winston 92] Patrick Winston. *Artificial Intelligence, 3rd edition.* Addison Wesley Publishing Company, 1992.

[Wright 68] S. Wright. *Evolution and the Genetics of Populations, Volume 1: Genetic and Biometric Foundations.* University of Chicago Press, 1968.

[Wright 69] S. Wright. *Evolution and the Genetics of Populations, Volume 2: The theory.* University of Chicago Press, 1969.

[Wright 77] S. Wright. *Evolution and the Genetics of Populations, Volume 3: Experimental Results and Evolutionary Deductions.* University of Chicago Press, 1977.

[Wright 78] S. Wright. *Evolution and the Genetics of Populations, Volume 4: Variability within and among Natural Populations.* University of Chicago Press, 1978.

[Yamada & Nakano 92] Takeshi Yamada and Ryohei Nakano. A genetic algorithm application to large-scale job-shop problems. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature II*, pages 281–290. Elsevier Science Publisher B.V., 1992.

[Yokoi & Kakazu 92] Hiroshi Yokoi and Yukinori Kakazu. An approach to the travelling salesman problem by a bionic model. *Heuristic: the journal of knowledge engineering*, 5:13–27, 1992.

[Zweben *et al.* 92] Monte Zweben, Eugene Davis, Brian Daun, and Michael Deale. Rescheduling with iterative repair. In *Practical Approaches to Scheduling and Planning*, pages 92–96. AAAI-1992 Spring Symposium Series working notes, 1992.

Appendix A

Sample Data for Exam Timetable

The exclusion file for the 1992/1993 AI/CS MSc exam:

```
aied : 0 1 4 5 6 8 9 10 14 15 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
conc : 0 1 4 5 6 7 8 9 20 21 24 25 28 29 30 31 32 33 34 35
cvis : 0 1 4 5 8 9 20 21 24 25 28 29 30 31 32 33 34 35
es1 : 6 7 28 29 30 31 32 33 34 35
es2 : 28 29 30 31 32 33 34 35
iss : 0 1 4 5 8 9 20 21 24 25 28 29 30 31 32 33 34 35
isc : 0 1 4 5 8 9 20 21 24 25 28 29 30 31 32 33 34 35
kri1 : 28 29 30 31 32 33 34 35
kri2 : 2 3 6 7 10 11 12 13 14 15 18 19 22 23 28 29 30 31 32 33 34 35
lisp : 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
mvis : 0 1 2 3 4 5 6 7 16 17 28 29 30 31 32 33 34 35
mthr : 10 11 28 29 30 31 32 33 34 35
nlp : 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
prol : 28 29 30 31 32 33 34 35
ln1 : 26 27 28 29 30 31 32 33 34 35
ln2 : 26 27 28 29 30 31 32 33 34 35
cl1 : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 26 27  28 29 30 31 32 33 34 35
cl2 : 16 17 18 19 26 27 28 29 30 31 32 33 34 35
lfs1 : 0 1 2 3 26 27 28 29 30 31 32 33 34 35
nnet: 26 27 28 29 30 31 32 33 34 35
asic : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 26 27 28 29 30 31 32 33 34 35
cad  : 0 1 2 3 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
cc   : 0 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
ci   : 28 29 30 31 32 33 34 35
cmc  : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 34 35
ds   : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 22 23 24 25 26 27 28 29 30 31 32 33 34 35
gr   : 28 29 30 31 32 33 34 35
ppc  : 28 29 30 31 32 33 34 35
sec  : 28 29 30 31 32 33 34 35
gis  : 4 5 12 13 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
re1 : 28 29 30 31 32 33 34 35
re2 : 28 29 30 31 32 33 34 35
ip1 : 28 29 30 31 32 33 34 35
ip2 : 28 29 30 31 32 33 34 35
```

The student file for the 1992/1993 AI/CS MSc exam:

```
student1 : aied conc cvis iss isc
student2 : lisp prol kri1 kri2 mvis nlp mthr cvis
student3 : lisp prol kri1 kri2 mvis es1 isc nlp
student4 : prol kri1 mvis isc iss cvis cad sec
student5 : lisp prol kri1 kri2 mvis isc cvis conc
student6 : lisp prol kri1 kri2 es1 mthr conc sec
student7 : prol kri1 iss isc conc sec
student8 : prol kri1 kri2 mvis es1 aied es2 mthr
student9 : lisp kri1 kri2 mthr ln1 cl1 lfs1 cl2
student10 : lisp prol kri1 kri2 es1 aied nnet es2
student11 : lisp kri1 aied mthr kri2 cl1 cl2 conc
student12 : lisp prol kri1 kri2 es1 es2 nlp nnet
student13 : lisp prol kri1 kri2 es1 aied es2 conc
student14 : lisp prol kri1 kri2 iss es1 aied es2
student15 : lisp prol kri1 kri2 es1 isc nnet nlp
student16 : lisp prol kri1 kri2 es1 es2 iss nnet
student17 : lisp prol kri1 kri2 es1 es2 aied nnet
student18 : lisp kri1 kri2 ln1 cl1 ln2 cl2 lfs1
student19 : lisp kri1 kri2 ln1 cl1 cl2 conc lfs1
```

```
student20 : lisp kri1 kri2 mvis ias isc mthr conc
student21 : prol kri1 mvis ias isc mthr cvis conc
student22 : lisp prol kri1 kri2 mvis es1 nlp conc
student23 : lisp prol kri1 kri2 es1 aied es2 nlp
student24 : lisp prol kri1 kri2 nlp nnet ias aied
student25 : lisp prol kri1 kri2 es1 es2 aied gr
student26 : lisp kri1 mvis ias isc nlp cvis conc
student27 : lisp prol kri1 kri2 es1 mthr conc ci
student28 : lisp prol kri1 kri2 es1 aied mthr nlp
student29 : lisp prol kri1 mvis ias isc cvis conc
student30 : lisp kri1 kri2 la1 la2 cl1 cl2 mthr
student31 : lisp prol kri1 kri2 es1 es2 conc ci
student32 : lisp kri1 kri2 la1 la2 cl1 cl2 lfe1
student33 : lisp prol kri1 kri2 es1 aied es2 nlp
student34 : lisp prol kri1 kri2 es1 es2 nlp aied
student35 : lisp prol kri1 kri2 es1 es2 nlp aied
student36 : lisp prol kri1 mvis ias isc nlp conc
student37 : lisp prol kri1 kri2 es1 es2 nlp nnet
student38 : prol kri1 mvis ias isc es1 cvis conc
student39 : lisp prol kri1 kri2 mvis es1 isc conc
student40 : lisp prol kri1 kri2 es1 nnet conc ppc
student41 : lisp prol kri1 kri2 es1 es2 aied nlp
student42 : lisp kri1 mvis ias isc nnet cvis cad
student43 : lisp prol kri1 kri2 ias aied mthr conc
student44 : cca swp com ppc os gr cad asc
student45 : aaic si cad bav cmc com cca swp
student46 : com cmc swp ppc nnet aps cc cafr
student47 : cca com swp ds os gr asc db
student48 : aps fpls cmc cc swp ppc asc gr
student49 : cca com swp ds os gr asc db
student50 : cca cmc swp ds si aaic cad db
student51 : aps fpls ci cafr kri1 bav ppc prol
student52 : cca com ds swp ppc ci os gr
student53 : aaic si bav cad com swp asc gr
student54 : cca com swp ds os asc si cad
student55 : swp ds cca com os si asc db
student56 : aps fpls cmc cafr os gr com bav
student57 : cc
student58 : cca com swp ds aaic aps db prol
student59 : bav si aaic cad asc gr db cca
student60 : ppc cmc gr db
student61 : aps cc cafr fpls bav cca ds ppc
student62 : cca com ds swp os asc db gr
student63 : cca swp ds com asc os gr nnet
student64 : cca ppc swp ds os asc aps db
student65 : aaic si bav cca aps db gr ppc
student66 : cca com swp ds ppc os asc db
student67 : ds cca swp com gr os asc cad
student68 : com ds swp ppc aps asc gr os
student69 : cca com ppc swp aps os gr nnet
student70 : cca ds swp ppc si bav aps fpls
student71 : cca com swp ds asc os gr nnet
student72 : cca com swp ds asc aaic si gr
student73 : com cca swp ds kri1 os asc ppc
student74 : cca com swp ds os asc si db
student75 : cmc ci cc fpls ppc swp cafr bav
student76 : rs1 rs2 ip1 ip2 mvis ias nnet cvis
student77 : rs1 rs2 ip1 ip2 mvis cvis nnet swp
student78 : rs1 rs2 ip1 ip2 mvis gis gr swp
student79 : rs1 rs2 ip1 ip2 mvis gis gr cvis
student80 : rs1 rs2 ip1 ip2 mvis gis db swp
student81 : rs1 rs2 ip1 ip2 mvis gis swp nnet
student82 : rs1 rs2 ip1 ip2 mvis gis cvis swp
student83 : rs1 rs2 ip1 ip2 mvis es1 nnet swp
student84 : rs1 rs2 ip1 ip2 mvis gis db swp
```

The preset file (actual timetable) for the 1992/1993 AI/CS MSc exam:

```
aied : 2
conc : 26
cvis : 16
es1  : 4
es2  : 17
ias  : 22
isc  : 13
kri1 : 8
kri2 : 24
lisp : 0
mvis : 15
mthr : 19
nlp  : 10
prol : 7
aaic : 25
aps  : 16
bav  : 11
cad  : 5
cafr : 5
cc   : 1
cca  : 27
ci   : 3
com  : 31
cmc  : 33
```

```
db  : 19
ds  : 21
fpls : 35
gr  : 23
os  : 1
ppc : 15
sl  : 28
ssc : 3
swp : 12
cl1 : 20
cl2 : 12
la1 : 23
la2 : 7
lfa1 : 10
mast : 19
ra1 : 27
ra2 : 11
ip1 : 20
ip2 : 0
gis : 2
```

Appendix B

Sample Data for Lecture/Tutorial Timetable

```
Data for 1992/1993 1st term AI/CS MSc lecture/tutorial:

[MODULES SECTION]
aied`1 ## AI and Education
       room SB-F10
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
aied`2 ## AI and Education
       duration 120
       room SB-F10
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
es1`1  ## Expert Systems 1
       room AT-LT2 AT-LT3
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
es1`2  room AT-LT2 AT-LT3
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
ias`1 ## Intelligent Assembly Systems
      room SB-F10
      exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
ias`2 room SB-F10
      exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
kri1`1 ## Knowledge Representation and Inference I
       room AT-LT2 AT-LT3
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
kri1`2 room AT-LT2 AT-LT3
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
lisp`1 ## Programming in Lisp
       room AT-LT2 AT-LT3
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
lisp`2 room AT-LT2 AT-LT3
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
mvis`1 ## Machine Vision
       room SB-F10
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
mvis`2 room SB-F10
       exclusion 0 1 2 3 8 9 10 11 16 17 18 19 24 25 26 27 32 33 34 35
scsai`1 ## Student Computing Support
        room SB-F10
        exclusion 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
slcs`1 ## Systems Intergration
       room TH-A
       exclusion 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
asic`1 ## ASIC design
       room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
asic`2 room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
aps`1  ## applicative programming and specification
       room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
aps`2  room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
cc`1   ## computational complexity
       room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
cc`2   room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
cca`1  ## concurrent computer architecture
       room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
```

```
cca`2  room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
ci`1  ## computability and intractability
      room TH-A TH-B TH-C
      exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
ci`2  room TH-A TH-B TH-C
      exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
com`1  ## computer communications
       room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
com`2  room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
cmc`1  ## communications and concurrency
       room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
cmc`2  room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
fpla`1  ## formal programming language semantics
        room 3218 3315 3317 4310 5327 6206 6224 6324
        exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
fpla`2  room 3218 3315 3317 4310 5327 6206 6224 6324
        exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
os`1  ## introduction to operating systems
      room TH-A TH-B TH-C
      exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
os`2  room TH-A TH-B TH-C
      exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
si`1  ## systems integration
      room 3218 3315 3317 4310 5327 6206 6224 6324
      exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
si`2  room 3218 3315 3317 4310 5327 6206 6224 6324
      exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
ssc`1  ## software systems concepts
       room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
ssc`2  room 3218 3315 3317 4310 5327 6206 6224 6324
       exclusion 4 5 6 7 12 13 14 15 19 20 21 22 23 28 29 30 31 36 37 38 39
cl1`1 ## Computational Linguistics 1
      duration 120
      room DHT7.01
      exclusion 19
cl1`2 duration 120
      room DHT7.01
      exclusion 19
ln1`1 ## Linguistics 1
      duration 120
      room CSSR
      exclusion 19
ln1`2 duration 120
      room CSSR
      exclusion 19
lfs1`1 ## Logic and Formal Semantics 1
       duration 120
       room CSSR
       exclusion 19
lfs1`2 duration 120
       room CSSR
       exclusion 19
cp1`1 ## Cognitive Psychology 1
      duration 120
      room CSSR
      exclusion 19
cp1`2 duration 120
      room CSSR
      exclusion 19
rs1`1 ## Remote Sensing 1
      room 8216
      exclusion 9 10 11 19
rs1`2 room 8216
      exclusion 9 10 11 19
ip1`1 ## Image Processing 1
      room 8216
      exclusion 9 10 11 19
ip1`2 room 8216
      exclusion 9 10 11 19
@aied-sem ## AI and Education-Seminars
          size 23
          unit 10
          duration 120
          room SB-A10
          exclusion 19
@aied-lab ## AI and Education-laboratories
          size 23
          unit 13
          duration 60
          room SB-C1
          exclusion 19
@es1-tut  ## Expert Systems 1-tutorials
          size 30
          room SB-A10 SB-C10 SB-C11 SB-C5
          exclusion 19
@ias-lab  ## Intelligent Assembly Systems-laboratories
          size 15
```

```
        unit 8
        room IAS-LAB
        exclusion 9 10 11 19
Okril-tut ## Knowledge Representation and Inference I-tutorials
        size 65
        duration 60
        room SB-A10 SB-C10 SB-C11 SB-C5
        exclusion 19
Olisp-tut ## Programming in Lisp-tutorials
        size 50
        room SB-A10 SB-C10 SB-C11 SB-C5 AT-2C
        exclusion 19
Omvis-tut ## Machine Vision-tutorials
        size 30
        room FH-B9
        exclusion 9 10 11 19
Oaps-tut ## applicative programming and specification-tutorials
        size 20
        room 3315
        exclusion 19
Oci-tut ## computability and intractability-tutorial
        size 10
        room 3315
        exclusion 19
Oos-tut ## introduction to operating systems-tutorial
        size 10
        duration 60
        room 3315
        exclusion 19

[ROOMS SECTION]
SB-A10  : 1 ## south bridge room A10
SB-C10  : 1 ## south bridge room C10
SB-C11  : 1 ## south bridge room C11
SB-C1   : 1 ## south bridge room C1(undergraduate terminal room)
SB-C5   : 1 ## south bridge room C5
SB-F10  : 1 ## south bridge room F10
FH-B9   : 1 ## forrest hill room B9
IAS-LAB : 1 ## forrest hill ias labatory
AT-LT2  : 1 ## appleton tower lecture theatre 2
AT-LT3  : 1 ## appleton tower lecture theatre 2
AT-2C   : 1 ## appleton tower room 2C
DHT7.01 : 1 ## cognitive science?
CSSR    : 1  ## Cog. Sci. Seminar Room at 1 Buccleuch Place
3218    : 2 ## JCML room 3218
3315    : 2 ## JCML room 3315
3317    : 2 ## JCML room 3317
4310    : 2 ## JCML room 4310
5327    : 2 ## JCML room 5327
6206    : 2 ## JCML room 6206
6224    : 2 ## JCML room 6224
6324    : 2 ## JCML room 6324
8216    : 2 ## JCML room 8216
TH-A    : 2 ## JCML room theatre A
TH-B    : 2 ## JCML room theatre B
TH-C    : 2 ## JCML room theatre C

[LECTURE-EXCLUSIONS SECTION]
lisp`1 lisp`2 kril`1 kril`2 es1`1 es1`2 aied`1 aied`2
lisp`1 lisp`2 kril`1 kril`2
lisp`1 lisp`2 kril`1 kril`2 ias`1 ias`2 mvis`1 mvis`2
lisp`1 lisp`2 kril`1 kril`2 cl1`1 cl1`2 ln1`1 ln1`2 lfo1`1 lfo1`2 cp1`1 cp1`2 aied`1 aied`2

[TUTORIAL-EXCLUSIONS SECTION]
Oaied-sem Oaied-lab : aied`1 lisp`1 lisp`2 kril`1 kril`2
Oes1-tut : es1`1 es1`2 lisp`1 lisp`2 kril`1 kril`2
Oias-lab : ias`1 ias`2 mvis`1 mvis`2 lisp`1 lisp`2 kril`1 kril`2
Okril-tut Olisp-tut : lisp`1 lisp`2 kril`1 kril`2
Omvis-tut : mvis`1 mvis`2 ias`1 ias`2 lisp`1 lisp`2 kril`1 kril`2
Oaps-tut : aps`1 aps`2 lisp`1 lisp`2 kril`1 kril`2
Oci-tut  : ci`1 ci`2 lisp`1 lisp`2 kril`1 kril`2
Oos-tut  : os`1 os`2 lisp`1 lisp`2 kril`1 kril`2

[PRECEDENCE SECTION]
aied`1 : Oaied-sem Oaied-lab
ias`1 : Oias-lab

[PREFERENCE SECTION]
es1`1 : Oes1-tut
kril`1 : Okril-tut
lisp`1 : Olisp-tut
mvis`1 : Omvis-tut
aps`1 : Oaps-tut
ci`1 : Oci-tut
os`1 : Oos-tut

[SIMULTANEOUS SECTION]
aied`2 aied-sem`1

[DIFFERENT-DAY SECTION]
aied`1 aied`2
es1`1 es1`2
ias`1 ias`2
```

```
kri1'1 kri1'2
lisp'1 lisp'2
mvis'1 mvis'2
asic'1 asic'2
aps'1 aps'2
cc'1 cc'2
cca'1 cca'2
ci'1 ci'2
com'1 com'2
cmc'1 cmc'2
fpls'1 fpls'2
os'1 os'2
ai'1 ai'2
asc'1 asc'2
cl1'1 cl1'2
ln1'1 ln1'2
lfs1'1 lfs1'2
cp1'1 cp1'2
rs1'1 rs1'2
ip1'1 ip1'2

[PRESET-TIME-ROOM SECTION]
os1'1 7
lisp'1 22 AT-LT2
kri1'1 15 AT-LT3
```

Appendix C

Publications

- *A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open-Shop Scheduling Problems*
  (Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA), 1993.)
- *Solving the Module Exam Scheduling Problem with Genetic Algorithms*
  (Proceedings of the Sixth International Conference in Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE), 1993.)
- *A Promising Hybrid GA/Heuristic Approach for Open-Shop Scheduling Problems*
  (Proceedings of the 11th European Conference on Artificial Intelligence (ECAI), 1994.)
- *Successful Lecture timetabling with Evolutionary Algorithms*
  (Proceedings of the 11th European Conference on Artificial Intelligence (ECAI) Workshop on Applied Genetic and Other Evolutionary Algorithms, 1994.)
- *Fast Practical Evolutionary Timetabling*
  (Proceedings of the Artificial Intelligence & Simulation of Behaviour (AISB) Workshop on Evolutionary Computation, 1994.)
- *Improving Evolutionary Timetabling with Delta Evaluation and Directed Mutation*
  (Parallel Problem Solving from Nature (PPSN) III, 1994.)

A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open-Shop Scheduling Problems

Hsiao-Lan Fang, Peter Ross, Dave Corne
Department of Artificial Intelligence
University of Edinburgh
Edinburgh, UK
Email: {hsiaolan,peter,dave}@aisb.edinburgh.ac.uk

Abstract

The general job-shop scheduling problem is known to be extremely hard. We describe a GA approach which produces reasonably good results very quickly on standard benchmark job-shop scheduling problems, better than previous efforts using genetic algorithms for this task, and comparable to existing conventional search-based methods. The representation used is a variant of one known to work moderately well for the traveling salesman problem. It has the considerable merit that crossover will always produce legal schedules. A novel method for performance enhancement is examined based on dynamic sampling of the convergence rates in different parts of the genome. Our approach also promises to effectively address the open-shop scheduling problem and the job-shop rescheduling problem.

1 INTRODUCTION

The job-shop scheduling problem (JSSP) is a very important practical problem. Efficient methods of solving it can have major effects on profitability and product quality, but with the JSSP being among the worst members of the class of NP-complete problems (Gary & Johnson 1979) there remains much room for improvement in current techniques. In general, the difficulty of the general JSSP makes it very hard for conventional search-based methods to find near–optima in reasonable time. This has led to recent interest in using genetic algorithms (GAs) to address these problems.

In the general JSSP, there are j jobs and m machines; each job comprises a set of tasks[1] which must each be done on a different machine for different specified processing times, in a given job-dependent order. Eg:, table 1 shows a standard 6×6 benchmark problem (*ie*, $j = 6, m = 6$), from (Muth & Thompson 1963). In this example, job 1 must go to machine 3 for 1 unit of time, then to machine 1 for 3 units of time, and so on. A legal schedule is a schedule of job sequences on each machine such that each job's task order is preserved, a machine is not processing two different jobs at once, and different tasks of the same job are not simultaneously being processed on different machines. The problem is to minimise the total elapsed time between the beginning of the first task and the completion of the last task (the *makespan*). Other measures of schedule quality exist, but shortest makespan is the simplest and most widely used criterion. For the above problem the minimum makespan is known to be 55, as in, for example, the schedule shown in figure 1 .

	(m,t)	(m,t)	(m,t)	(m,t)	(m,t)	(m,t)
Job 1:	3,1	1,3	2,6	4,7	6,3	5,6
Job 2:	2,8	3,5	5,10	6,10	1,10	4,4
Job 3:	3,5	4,4	6,8	1,9	2,1	5,7
Job 4:	2,5	1,5	3,5	4,3	5,8	6,9
Job 5:	3,9	2,3	5,5	6,4	1,3	4,1
Job 6:	2,3	4,3	6,9	1,10	5,4	3,1

Table 1: The 6x6 benchmark problem

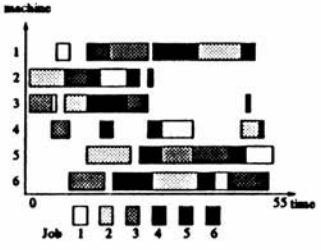

Figure 1: An optimal schedule for 6×6 JSSP benchmark

There are two similar benchmarks, of sizes 10×10 and 20×5. The best results on these benchmarks for traditional (B & B – branch & bound search) and GA methods published

[1]Note: what we call a "task" is often called an "operation" in the JSSP literature.

so far are shown in table 2[2]

Paper	Method	6x6	10x10	20x5
McMahon 75	B & B	55	972	1165
Baker 85	B & B	55	960	1303
Carlier 89	B & B	55	930	1165
Nakano 91	GA	55	965	1215

Table 2: Some published benchmark results

The branch & bound method (*eg*: see (Carlier & Pinson 1989)) produces good results but takes considerable computer time even for the 10×10 problem because of the significant amount of schedule generation implicit in the method. (Davis 1985) was the first to suggest and demonstrate the feasibility of using a GA on a simple JSSP, employing an essentially *ad-hoc* set of genetic operators and a memory-intensive chromosome representation, paving the way for future improvements. Meanwhile, the general success of GAs on other kinds of hard scheduling problems, such as the traveling salesman problem (TSP), started to lead to clues for more effective representations and operators for GA approaches. *Eg*: (Whitley *et al* 1989) defined a new edge recombination operator for the TSP, although noted that performance degraded when applied to more typical scheduling problems; (Bagchi *et al* 1991) used problem-specific information in the representation and genetic operators, addressing a limited form of JSSP in which certain batches of tasks must be scheduled continuously. More recently, (Nakano 1991) used a conventional (binary) GA for the JSSP, supplemented with algorithms for interpreting and repairing genomes, and was successful in improving on the performance of some previously reported branch & bound search methods on benchmark problems, though did not improve on the best results found with these methods.

Our approach uses a variant of the ordinal representation introduced in (Grefenstette *et al* 1985) and used for the TSP. This representation has the considerable merit of producing only legal schedules under crossover and mutation. When applied to the JSSP, it produces better results than those of (Nakano 1991) with pleasingly small computational effort, and thus provides a convenient way to handle the rescheduling problem too. The rescheduling problem involves modifying a schedule in process of execution in order to take account of changed, canceled or new jobs. Because this sort of thing happens frequently in the kind of organisation that has to deal with JSSPs, it is as important to find efficient rescheduling algorithms (which hopefully don't involve rebuilding the schedule from scratch) as it is to find effective algorithms for the full JSSP.

2 OVERVIEW

In section 3 we describe our encoding technique, and outline the basic activities of the schedule builder which performs the interpretation of a genome for the JSSP. In sectio 4 we go on to discuss the application of this approach *Open-Shop* scheduling, and outline the more sophisticat schedule builder we employ in this latter case. In section we briefly describe the job-shop rescheduling problem, ar how it can be addressed via our approach. In section 6 we g on to discuss the qualitative GA dynamics which arise fro the representation we use, making points in particular abo the redundancy of the representation, and the variation convergence rates for different genes (or 'chunks' of th genome). This leads us towards introducing a method f combating premature convergence in general GA applic tions that involve significant variation in gene convergenc rates, which is discussed further in section 7. Section presents some basic results: concerning the performanc of our basic approach on two benchmark JSSPs, show ing how this approach outperforms previously reported G attempts at this task which we know of; concerning th performance of our basic approach, enhanced by 'gen variance-based operator targeting', showing improvemer on the initial unenhanced results; and concerning perfo mance on a selection of benchmark open-shop schedulir problems, showing how our approach comes within a fe percent (sometimes 0%) of the optimal or best-known s lutions for the problems tried. We know of no GA-base efforts on the OSSP with which to compare, so we prese these latter results in order to show the potential for a G approach to open-shop scheduling, and invite fellow G researchers to experiment with the same problems. At th end of this section, we describe how to obtain the proble definitions for the benchmarks used in this paper. Finall section 9 summarises our results and discusses the gener approach and further work.

3 THE REPRESENTATION

The genotype for a $j \times m$ problem is a string containir $j \times m$ chunks, each chunk being large enough to hold th largest job number (j). A chunk is atomic as far as the G is concerned. It provides instructions for building a leg schedule as follows: the string of chunks $abc \cdots$ mean put the first untackled task of the a-th uncompleted job int the earliest place where it will fit in the developing sche ule, then put the first untackled task of the b-th uncomplete job into the earliest place where it will fit in the developir schedule, and so on. The representation can be seen encode all active schedules, and also lends itself to obviou extensions which would enable the encoding of necessary unnecessary delays on machines. The task of constructir an actual schedule is handled by a schedule builder whic maintains a circular list of uncompleted jobs and a list untackled tasks for each such job. Thus the notion of "a-t uncompleted job" is taken modulo the length of the circul list to find the actual uncompleted job. Note: instead of en ploying a circular list, (Grefenstette *et al* 1985) constrai alleles of the i-th chunk to range from 1 to $N - i + 1$ i value; it is unclear how to directly extend this technique t a JSSP (with more than one machine), hence our use of

[2]Adapted from (Nakano 1991).

circular list.

The schedule builder is straightforward and computationally cheap. It must consider four cases when slotting a task into a developing schedule. For instance, suppose it is asked to slot job 1 into machine 2, with processing time 2. If there is a suitable gap in the schedule for machine 2, it may be possible to fit the task in there with or without compulsory idle time. If no suitable gap exists, that task has to be added to the end of the machine's schedule with or without compulsory idle time. Figure 2 shows the choices. The symbol "##" shows where the schedule builder would

```
 With suitable gap,
 idle time needed:     ... not needed:
mc1:.. 11333          mc1:.. 114444
mc2:.. 2 ## 3333      mc2:.. 222## 44

  No suitable gap,
 idle time needed:     ... not needed:
mc1:.. 555111         mc1:.. 66611
mc2:.. 22 5  ##       mc2:.. 22 666##
```

Figure 2: Scheduler builder choices of task placement

place the task in each case.

4 OPEN-SHOP SCHEDULING

The Open-Shop Scheduling Problem (OSSP) is similar to the JSSP, with the exception that there is no *a priori* ordering on the tasks within a job. The OSSP has a considerably larger search space than the JSSP, and seems to be less heavily addressed in the literature, although it is an important and ubiquitous problem, occurring in any job-shop situation in which tasks for a particular job may be carried out in (almost) any order, such as automotive repairs (tasks) for cars (jobs), or upgrades/repairs (tasks) for PCs (jobs).

Table 3 shows a standard 5×5 benchmark OSSP (that is, $j = 5, m = 5$) taken from (Beasley 1990). In the above

	(m,t)	(m,t)	(m,t)	(m,t)	(m,t)
Job 1:	4,85	1,64	3,31	5,44	2,66
Job 2:	1,7	4,14	2,69	5,18	3,68
Job 3:	4,1	1,74	2,70	5,90	3,60
Job 4:	2,45	4,76	5,13	3,98	1,54
Job 5:	1,80	4,15	2,45	5,91	3,10

Table 3: A 5x5 benchmark OSSP

example, task 1 of job 1 must go to machine 4 for 85 units of time, task 2 of job 1 must go to machine 1 for 64 units of processing time, and so on, with no restrictions on the order in which the tasks for any job are to be processed. The problem is to generate a valid schedule with minimal makespan. Figure 3 shows a minimum-makespan (300) schedule for the benchmark in table 3.

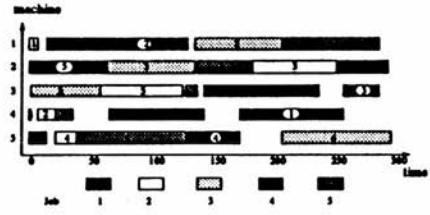

Figure 3: Minimal-makespan schedule for a 5x5 OSSP benchmark

The basic extension of the representation described in section 3 to the OSSP involves a genome *abcd*... meaning: put the *a*th untackled task of the *b*th uncompleted job into the earliest place it will fit in the developing schedule, put the *c*th untackled task of the *d*th uncompleted job into the earliest place it will fit in the developing schedule, and so on. Whereas previously, for the JSSP, the 'first untackled task' for any particular job was always predetermined owing to the *a priori* ordering on tasks, in this case we need to incorporate an extra gene for each job to encode which of the remaining tasks for a job to choose (since with no predetermined ordering, any may be chosen).

An alternative is to use precisely the same representation as for the JSSP, but change the interpretation of *abc*... to: heuristically choose an untackled task from the *a*th uncompleted job and place it in the earliest place it will fit in the developing schedule, heuristically choose an untackled task from the *b*th uncompleted job and and place it in the earliest place it will fit in the developing schedule, and so on. In this case, at each step the schedule builder looks ahead to find the earliest available slot(s) in the developing schedule into which a non-empty set of tasks from the current job can be placed. If lookahead determines that more than one equally early slots are available, then a simple heuristic is used to choose which task to actually place in which slot. Two simple heuristics we have used are: (a) choose randomly from the available tasks; (b) choose the task with the largest processing time. The random method seems to work best on small problems, but best results are found on larger problems with the "largest-first" heuristic. In general, this lookahead/heuristic method for the OSSP works better than the basic extension to the JSSP approach described in the above paragraph.

5 JOB-SHOP RESCHEDULING

Job-shops are beset by the continual need to alter previously worked out schedules in the light of problems which arise. This typically means revising the expected processing time

for some job in the schedule, or revising (typically delaying) the start time for a particular task. There is thus a need for efficient methods of rescheduling. If work has not yet begun on the current schedule, then an obvious and simple approach to rescheduling would be to rerun the schedule-finding program (eg: in this case, a GA) from scratch on the changed data. Strict rescheduling, however, means *not* scheduling the entire problem from scratch; rescheduling is thus strictly necessary when either there is not enough time to be able to schedule from scratch, or when part of the current schedule is already in progress. A proper rescheduling method would be to re-use some of the work already done in finding the previous schedule. This might involve augmenting the previous schedule with the new change, and iteratively modifying it until it is acceptable. Another method would be to recover a new, smaller scheduling problem made up from all and only those parts of the previous schedule that are affected by the change.

Rescheduling from scratch is obviously to be avoided in the light of the large processing time required for large problems and the frequency of the need to reschedule. Also, sophisticated use of previous work is very difficult to achieve with a typical GA (although see (Louis *et al* 1993) for a recent attempt at storing schema information in a case base). Nominal use of previous work done could involve seeding; we have not yet tried this. Our representation and schedule builder, however, lend themselves naturally to a method in which we make a smaller scheduling problem, via a simple dependency analysis which finds out which tasks are affected by the changes.

Two kinds of situation are dealt with: a change in the *processing time* of some task (which includes the case of removing a task entirely), and a change in the *start time* of some task (if, for example, a task must be delayed because of problems with a machine or delays in obtaining resources). Input to the rescheduler is simply the genome representing the schedule which must be altered. The user then enters the required modification (to the processing time and/or start time of one or more tasks). With reference to figure 1, suppose we need to increase the processing time of the machine2 task of job1. A simple dependency analysis discovers that the affected tasks are those that occur later in the schedule on machine 2 (as well as the changed task itself), as well as the machine 4, 5 and 6 tasks of job 1. Recursively, other affected tasks are found for each of the initially affected tasks until the complete set of affected tasks is found. Along with values from the previous schedule which contain new available start times for each machine, this set of affected tasks constitutes a reduced JSSP which can be solved by the GA much more quickly than fully rescheduling from scratch. A similar dependency analysis and reduced JSSP formulation is done for the case in which a task's earliest possible start time is shifted.

This method does not guarantee an optimal new schedule; the GA, of course, never guarantees optimality anyway, but the point is that the retention of a fixed (unaffected) portion of the previous (near) optimal schedule might preclude the discovery of an optimal schedule which might otherwise be possible to find by rescheduling from scratch. The strength of this rescheduling method, however, lies in its speed. There is thus a tradeoff between the speed in which a good new schedule can be found via retaining parts of the previous schedule, and the potential advantage of rescheduling from scratch with the (probably low) possibility of evolving a significantly better schedule. Experiments are underway to quantitatively analyse this tradeoff.

6 PERFORMANCE ENHANCEMENTS

On hard problems like the JSSP, GA researchers routinely need to use either problem-specific or problem-type specific performance enhancements to improve performance. These enhancements are interesting because of the light they shed on the dynamics of the GA approach and the aspects of problems which make it hard or easy for GAs to solve them. For example, Nakano's representation is highly redundant (with $2^{mj(j-1)/2}$ genomes representing approximately $j!^m$ distinct schedules) and so leads to the possibility of false competition among genotypes, in which different representations of the same schedule compete against one another, possibly to yield inferior descendants which combine aspects of their parents' representations which do not translate into good building blocks. There is, in fact, very little chance (but see below) of two representations of the same schedule competing in early generations — although there may be a huge number of possible representations of the same schedule, this number is entirely swamped by the number of distinct schedules. However, false competition will still be manifest with different representations of the same *building block* or, to be more correct, the same *forma*. A forma (Radcliffe 1990) can be viewed as any dimension along which two genomes are equivalent. False competition will then be relevant if the schemata in the representation do not directly coincide with the formae which (intuitively) represent the important building blocks; this is typically the case in sophisticated GA applications. Eg; in our case, the forma: "schedules in which the machine2 task of job 1 is scheduled before the machine2 task of job 2" may well be a good building block (ie: have high average fitness), but, since it does not correspond to a particular schema, two schedules which are instances of this forma may well recombine to produce children which are not.

Nakano partially combats false competition with *forcing*, in which he replaces illegal genotypes in the pool with their 'nearest' legal matches. This forces a one-to-one genotype/schedule mapping in a gene pool, eliminating false competition in the selection step (although still typically resulting in illegal schedules after crossover). Nakano hence uses a highly redundant representation with a complex evaluation technique for the basic GA, and then significantly improves performance by using forcing to reduce false competition. Our approach does not require forcing, since the representation always encodes legal schedules, but there is high redundancy (though less high than Nakano's), and we

similarly need a way of countering false competition.

Our choice of representation is highly context sensitive, and leads to front parts of the genotype converge more quickly than later parts. This seems to happen because schemata defined early in the genome correspond more precisely to good formae; that is: a *schema* such as 1,2,□,□,..., *always* corresponds to the forma "first schedule the first task of job1, and then the first task of job2". If it so happens that this forma has high fitness, then this schema will have high fitness. However, schemata defined later in the genotype, such as □,□,□,□,□,□,2,□,3, are likely to represent radically different formae in different genomes (contexts) — the sampled mean fitness of such a schema will thus tend towards the mean fitness of the population as a whole. Hence, high-fitness schemata will only be found early in the genome, and these will converge first (providing a 'context' which then leads to high fitness schemata being found a little later in the genome, and so on).

False competition thus leads to differing convergence rates for schemata across the genotype. This effect actually rises quite sharply towards the tail of the genotype owing to the fact that as the context becomes set by convergence in the rest of the genome, the j alleles of any tail end gene are 'competing' for, and thus multiply representing, fewer and fewer unscheduled tasks.

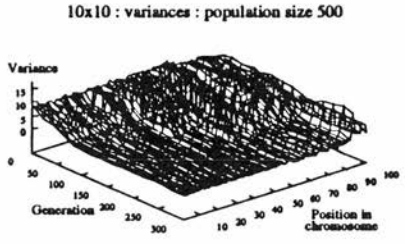

Figure 4: Plot of variance of chunks of the genome with time, and with genome position, on the 10 × 10 JSSP.

We can visualise the overall effect of this in figure 4, in which we can clearly see gradually decreasing convergence speed as we traverse the chromosome from left to right. This figure shows a plot of the variance of each chunk of the genotype within the pool (size 500) with its position in the genotype, and with generation as the GA operates, for 300 generations of a run on the 10 × 10 benchmark JSSP. As figure 4 shows, gene convergence rates fall fairly smoothly as a function of position in the genotype. This kind of behaviour should be typical of GA problems where the representation, for whatever reason, is such that there is a variation in 'significance' across the genotype. In the JSSP case, in which large scale problems not only cost significant computational time, but in which the solutions produced might significantly affect profits and/or product quality, we should be able to exploit this effect by using it to inform ways of increasing overall convergence speed and/or solution quality. In section 7, we describe a gene-variance based operator targeting strategy, which is a principled first attempt at doing just this, by making sure that genetic operators are concentrated where and when they seem to be most 'needed'. This initial attempt has led to significant improvement in solution quality.

7 GENE-VARIANCE BASED OPERATOR TARGETING

The situation in figure 4 suggests a strategy to improve solution quality. First, the faster stabilisation of early parts of the genome suggest premature convergence. This is because the fast converging early schemata may not have been adequately tested in the context of good formae that may be (partly) encoded later in the genome. Increasing mutation rates at fast converging sites may thus improve performance; also, this measure should obviate 'wasted' mutation in later, slow-converging parts of the schedule which are still in relatively early stages of exploration. Second, we can expect crossover at early, more stable positions to have minimal effect on sampling adequacy, since this leads only to re-examining schemata over and over again in similar contexts. So, encouraging crossover more at later, less stable positions should lead to more effective exploitation. On the whole, it would seem a good idea to increase the extent to which schemata are effectively sampled in new contexts, in proportion to the degree to which the GA seems 'unsure' about them. Conversely, it would seem a good idea to increase the extent to which new schemata are explored (via mutation), in proportion to the extent to which schemata defined at the same positions have already been (perhaps prematurely) converged to.

A way of implementing these effects is what we term *gene-variance based operator targeting* (GVOT). This works by measuring the diversity of genes at each position of the genotype in a pool (in our experiments, we do this by sampling statistical variance after every ten generations), and choosing the actual point of crossover or mutation via roulette-wheel selection based on these variances. Sites for N-point uniform crossover are selected probabilistically but according to the square of chunk variance, while order-based mutation positions are selected according to the inverse of chunk variance. Hence, high variance sections are more likely to be chosen for crossover; low variance sections for mutation.

This can be seen as a specific instance of an idea which should be of more general use in GA performance enhancement on hard problems, particularly where there is a significant variance in convergence rates at different sites in the genotype. In many other kinds of problem we can't expect smooth changes in variance across the genotype; this would not occur in the JSSP, for instance if (unusually) task processing times were to grow as a function of advancing position in the job sequence. However, whenever signif-

icant variation in convergence rate does occur (smooth or not), the GVOT strategy, targeting operators solely on the basis of dynamically sampled variance, should work just as well.

This performance enhancement method complements those discussed in, for example, (Booker 1987) and (Eshelman & Schaffer 1991), which present ways of improving performance by, eg, encouraging recombination between adequately 'different' genomes (incest prevention), and avoiding wasted crossover operations by only recombining the 'reduced surrogate' of two parents (the smaller genome made up of those sites at which the parents are different). There are complex interactions between such methods and GVOT. Roughly speaking, GVOT slows down convergence of otherwise fast-converging schemata in order to wait for other schemata to catch up, while encouraging vigorous recombination to more effectively test the latter; incest prevention in conjunction with reduced-surrogate recombination, on the other hand, will partially reproduce this effect to the extent that less converged schemata will be more likely to be present in the reduced surrogates of parents which are far enough apart to sanction recombination. The latter method, however, does not 'slow down' fast-converging schemata (which GVOT does via targeting mutation at fast-converging sites). We intend extensive experimentation to tease out the relative effectiveness of these methods in conjunction with, and other than, GVOT on problems with highly context sensitive genome representations. Our feeling is that GVOT, owing to the direct selective targeting of operators according to convergence rates, will be more and more effective the more varied the schemata convergence rates are in the application.

GVOT is less effective (though still produces better results), for example, with the representations we discuss above for the OSSP. This is because the plot analogous to figure 4 for the OSSP is rather more flat; because of much higher epistasis in the OSSP case (low-variance highly fit schemata only begin to occur at relatively long defining lengths), schemata sampled in earlier generations have a less significant advantage over others than in the JSSP case, and hence there is reduced variation in convergence rates.

8 RESULTS

The JSSP results below all involve population sizes of 500, using rank-based selection with elitism and a fixed crossover rate, running for 300 generations (unless otherwise specified), hence involving 150,000 evaluations. The comparative figures for Nakano involve the same number of evaluations, though based on 1,000 generations with populations of size 150. The raw fitness of a chromosome was taken to be the makespan of the schedule it represents. The OSSP results similarly involve rank-based selection with elitism but use adaptive crossover and runs of 1,000 generations. The two smaller OSSPs were tackled with populations of size 100, while the rest were tackled with populations of size 200. We found that results did not vary significantly across changes in crossover rate and adaptation regime. The reported JSSP experiments used a crossover rate of 0.6 and adaptive mutation (starting at 0, rising by 0.001 per generation), while the OSSP experiments use adaptive crossover (starting with p_C at 0.6, falling by 0.002 per generation, with a limit of 0.2) and adaptive mutation at $1 - p_C$. Typically, order-based mutation (swap alleles between two randomly chosen genes) was used. For the OSSP, the mutation rate was the probability of mutating a genome; so, for example, where p_M (ie: $1 - p_C$) was 0.6, this roughly translates to a bit-mutation rate of, for example, 0.012 for the 10×10 OSSP (divide by half the genome length).

GVOT involves calculating a measure of the diversity of alleles of a gene (or chunk of genes) within a population. We are still experimenting to find the most suitable measure of this diversity. Both JSSP-with-GVOT and OSSP-with-GVOT results use statistical variance of the numerical value of the alleles as a simple approximation to this measure; we are also investigating the use of allele entropy as a more well-founded information-theoretic measure of the diversity of alleles. In addition, we are experimenting with different ways of using the diversity measure to target operators. For the JSSP with GVOT, the method we used was roulette-wheel selection of crossover points based on variance (mutation sites based on inverse variance). For the OSSP with GVOT, we employed what we term *multiform* crossover, in which the probability of swapping genes between parents at a particular site is adjusted (from the normal 0.5, for uniform crossover) in accordance with the relative variance at that site.

Our main results are that we have been able to find better solutions on benchmark JSSPs than previous GA-based methods and have thus closed the gap somewhat between GA-based approaches and the best solutions so far found with branch & bound search.

In the two following tables, 'average' figures refer to the mean result over 10 trials; these are not available for Nakano's technique. Also, Nakano's 'without-forcing' result on the 10×10 benchmark is read from a graph in (Nakano 1991), hence our estimated error margin.

Table 4 summarises our results *without* gene-variance based operator targeting (GVOT), compared with Nakano's results (where available) *without* forcing, showing how, the representation we describe leads to better results when false competition is highly evident in both approaches[3].

With performance-enhancements in place, our results using GVOT are compared with Nakano's results using forcing in table 5. It can also be noted that our best solutions without GVOT are marginally better than Nakano's *with* forcing.

Although improvement in solution quality is modest as a percentage (though significant considering that these so-

[3]It is difficult for us to quantitatively compare our with approaches other than Nakano's since we have not yet found other reported GA approaches which use the benchmarks.

	10 × 10	20 × 5
Average sol'n without GVOT (Fang *et al*)	985	1225
Best sol'n without GVOT (Fang *et al*)	960	1213
Best sol'n without forcing (Nakano 91)	1160(±10)	—

Table 4: Our approach vs. Nakano's, without GVOT

	10 × 10	20 × 5
Average sol'n with GVOT (Fang *et al*)	977	1215
Best sol'n with GVOT (Fang *et al*)	949	1189
Best sol'n with forcing (Nakano 91)	965	1215

Table 5: Our approach vs. Nakano's, with GVOT

lutions may be very close to optimal anyway), the real advantage of our technique over previous GA methods is the combination of its apparent promise and the straightforwardness of applying it (arising from the absence of any need to repair invalid genomes). We also feel that it significantly improves on other techniques in terms of computational complexity, though unfortunately we cannot yet provide more quantitative results with regard to comparative speed because of a lack of available figures for comparison; however we can report that experiments on the 10 × 10 JSSP take less than 25 minutes of CPU time and the 20 × 5 JSSP less than 30 minutes, with our experiments implemented in C and run on a Sun-4 (without using GVOT, CPU time drops by about 30%).

We also experimented with one-point *vs* uniform crossover, and adaptive *vs* fixed order-based mutation rates. The graphs in 5 show our results on the 10 × 10 and 20 × 5 JSSP benchmarks, comparing different GA variants. *Fixed-1P* employs a fixed mutation rate per chromosome of 0.05 and one-point crossover; *one-point* employs a mutation rate per chromosome which begins at 0 and increases by 0.001 in each generation (stopping at 0.1); *uniform* employs the same adaptive mutation strategy as *one-point* and uses uniform crossover; finally, *GVOT* is as *uniform*, except for the use of N-point uniform crossover (where N is half the genome length) with GVOT.

Initial experiments with a (pseudo-)parallel GA with migration every 20 generations show improved average solution quality, as do experiments with larger population sizes (though obviously at the expense of time); but more work is needed properly to investigate and quantify these aspects.

Our initial results for a set of benchmark open shop schedul-

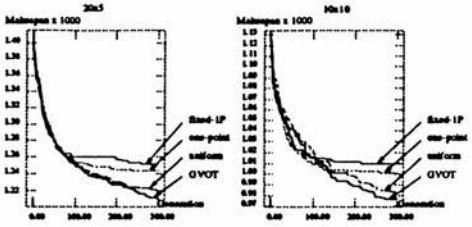

Figure 5: Relative performance of different variants on the 10 × 10 and 20 × 5 JSSP benchmarks

ing problems are shown in table 6. Results for the two smaller problems were obtained with the 'break-ties-randomly' heuristic, while the results for the larger problems were obtained with the 'largest-first' heuristic. For the two smaller problems and two larger problems, 'Best Known' is the optimal solution; for the rest, it is the best known solution. All OSSP experiments involved use of GVOT, which produced reliably better results than without GVOT, though less markedly so than with the JSSP.

OSS Problem	Best Known	Results: mean/best
4 × 4	193	193 / 193
5 × 5	300	302.2 / 300
7 × 7	438	447.1 / 439
10 × 10	645	679.5 / 669
15 × 15	937	980.0 / 969
20 × 20	1155	1235.1 / 1213

Table 6: Results on benchmark OSSPs

The JSSP benchmark problems used in this paper can be obtained from (Muth & Thompson 1963). The OSSP problems referred to in table 6 can be obtained via (Beasley 1990). The OR library referred to in the latter article is an electronic library from which may be obtained benchmarks for a wide range of OR problems. These are distributed in the form of Pascal code which generates the problems. Researchers wishing to compare with our results will need to know that the problems referred to in table 6 are each the *problem No. 1* of their specified size. Alternatively, problem data may be obtained directly from us.

9 CONCLUSIONS AND FURTHER WORK

We present a promising new representation for GA approaches to the JSSP, and described novel techniques for analysing the GA dynamics in terms of the variation in gene variance across the genotype, and targeting operator positions according to dynamically sampled measures of gene convergence rates. Our approach improves on the results obtained from other GA methods we know of, and brings

us closer to closing the gap in solution quality between the best solutions found by branch & bound search and those found by GA approaches so far.

The approach also conveniently handles rescheduling in the job-shop problem, and seems promising for application to the open shop scheduling problem. More tests are needed, however, before we can report a thorough comparison of our method against other techniques, and before we can determine the efficacy of our method when applied to real-world problems (benchmark problems are unrepresentative of the true difficulty of the general JSSP; the same might also be true of most real-life JSSPs!). In this vein, further work is under way to more thoroughly test the performance of our technique on the benchmarks, and on a set of real world JSSPs which we are planning to collate.

Finally, we hope to have shown further promise for GA-based approaches to job-shop problems, and hope and expect that further improvements will be reported (by us and others) via the use of various problem-specific heuristic improvements, as well as via approaches based on different genome representations. For example (Grefenstette 1987) discusses the general idea of incorporating problem-specific knowledge into various parts of the GA, while (Beasley *et al* 1993), describes a GA approach to combinatorial problems based on an epistasis reducing representation, which may be of use for the JSSP.

Acknowledgments

We gratefully acknowledge the constructive criticism and comment received from three anonymous referees, from which this paper has benefited, and we hope that we have satisfactorily accommodated their concerns and suggestions. Many thanks also to the China Steel Corporation, Taiwan, R.O.C., for financial support of Hsiao-Lan Fang.

References

S. Bagchi, S. Uckun, Y. Miyabe, & K. Kawamura (1991). Exploring problem-specific recombination operators for job shop. In R.K. Belew & L.B. Booker (eds) *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 10–17. San Mateo: Morgan Kaufmann, 1991.

D. Beasley, D. R. Bull, & R. R. Martin (1993). Reducing Epistasis in Combinatorial Problems by Expansive Coding. In S. Forrest (ed), *Genetic Algorithms; Proceedings of the Fifth International Conference (GA93)*, Morgan Kaufmann, San Mateo, CA.

J. E. Beasley (1990). OR-Library: Distributing test problems by electronic mail. In *Journal of the Operational Research Society*, Vol 41, pp. 1069–1072.

L. Booker (1987). Improving Search in Genetic Algorithms. In L. Davis (ed) *Genetic Algorithms and Simulated Annealing*, San Mateo: Morgan Kaufmann, 1987, pages 61–73.

J. Carlier & E. Pinson (1989). An algorithm for solving the job-shop problem. In *Management Science*, 35(2):164–176, February 1989.

L. Davis (1985). Job shop scheduling with genetic algorithms. In J. J. Grefenstette, (ed) *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 136–140. San Mateo: Morgan Kaufmann, 1985.

L. J. Eshelman & J. D. Schaffer (1991). Preventing Premature Convergence in Genetic Algorithms by Preventing Incest. In R. K. Belew & L. B. Booker (eds) *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo: Morgan Kaufmann, 1991, pages 115–120.

M. R. Gary & D. S. Johnson (1979) *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman.

J. J. Grefenstette, R. Gopal, B. Rosmaita, & D. Van Gucht (1985). Genetic algorithms for the traveling salesman problem. In J. J. Grefenstette (ed) *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 160–168. San Mateo: Morgan Kaufmann, 1985.

J. J. Grefenstette (1987). Incorporating Problem-Specific Knowledge into Genetic Algorithms. In L. Davis (ed) *Genetic Algorithms and Simulated Annealing*, San Mateo: Morgan Kaufmann, 1987, pages 43–60.

S. Louis, G. McGraw, & R. O. Wyckoff (1993). Case-based reasoning assisted explanation of genetic algorithm results. *J. Expt. Theor. Artif. Intell.*, Vol. 5, 1993, pp. 21–37.

J. F. Muth & G. L. Thompson (1963). *Industrial Scheduling*. Prentice Hall, Englewood Cliffs, New Jersey, 1963.

R. Nakano (1991). Conventional Genetic Algorithms for Job-Shop Problems. In R. K. Belew & L. B. Booker (eds) *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo: Morgan Kaufmann, 1991, pages 474–479.

N. J. Radcliffe (1990). Equivalence Class Analysis of Genetic Algorithms. In *Complex Systems* Volume 5, No. 2, pp.183–205, 1990.

D. Whitley, T. Starkweather, & D'Ann Fuquay (1989). Scheduling problems and traveling salesmen: The genetic edge recombination operator. In J. D. Schaffer (ed) *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 133–140. San Mateo: Morgan Kaufmann, 1989.

SOLVING THE MODULAR EXAM SCHEDULING PROBLEM WITH GENETIC ALGORITHMS

Dave Corne, Hsiao-Lan Fang, Chris Mellish,
Department of Artificial Intelligence,
University of Edinburgh,
Edinburgh, UK
Email: {hsiaolan, dave, chrism}@aisb.ed.ac.uk

ABSTRACT

Scheduling exam timetables for large modular courses is a complex problem which often has to be solved in university departments. This is usually done 'by hand', taking several days or weeks of iterative repair after feedback from students complaining that the timetable is unfair to them in some way. We describe an effective solution to this problem involving the use of an appropriately configured genetic algorithm (GA). Using real student data from a large multi-departmental modular degree scheme, the method we describe never failed to find a significantly better timetable than those that were actually employed (produced by hand), always taking less than half an hour.

INTRODUCTION

In a general scheduling problem, events must be arranged around a set of timeslots, so as to satisfy a number of hard constraints and optimise a set of objectives. Types of scheduling problem differ in terms of the kinds of constraints and objectives involved. In this paper we examine what we call the MESP (Modular Exam Scheduling Problem). This typically arises in universities running large modular degree schemes, in which each student takes an individual selection of exams from a wide inter-departmental pool of modules, many outwith their own department. The events are exams, the timeslots are possible start-times for those exams, the hard constraints are that no student should take more than one exam at a time, and the objectives are to generally minimise pressure on students, so that as few as possible have multiple exams in a day, consecutive exams, and so on.

Typical MESPs are NP-hard, and strewn with local minima which make it particularly difficult to address by way of heuristic search or hill-climbing techniques. MESP complexity is also illustrated by the size of the solution space. Eg: if there are t possible start times, and e exams, then there are t^e candidate schedules. In the particular MESP which occurs within the EDAI[1], this was 28^{44} in 1992, or c. 5×10^{63}.

When an MESP is tackled, typically in a university or college department, but very similar problems often occur in industry it is usually addressed by hand (eg: by a course organiser). This involves producing an initial draft timetable, followed by perhaps weeks of redrafting after student feedback complaining about the latest draft. The initial draft is often based on merging different departments' teaching/course timetables; but in large modular degree schemes, the fact that many students from one department typically take courses in others, and the fact that there are different lecture timetables for different terms, makes this a recipe for finding local minima, which then typically need extensive repair if better solutions are to be found (and hence better solutions are often not found).

(GAs) provide a way of addressing hard search and optimisation problems. GAs are particularly good at finding global optima in very hilly spaces; for these reasons, we investigated the use of GAs on the MESP. Following a very basic description of GAs for the uninitiated (for an excellent introduction, see [1]), we outline our GA approach to the MESP. Finally, we outline experiments in which different GA variants were used on a typical, real MESP, describe the promising results that ensued, and discuss future work and implications.

GENETIC ALGORITHMS

If we want to maximise a function $f(x_1, x_2, ..., x_n)$, where each x_i can take on any of its own range of values from a set v_i, then we can do this with a GA as follows. First, randomly generate a population of

[1]University of Edinburgh Department of Artificial Intelligence.

P candidate solutions. Eg: if each x_i has range $v_i = \{0,1\}$, then this simply corresponds to generating P random n-bit binary strings. Each candidate solution is called a chromosome (or genome). Call this initial population the current generation, then, until the number of generations g has reached a specified figure, or until all the chromosomes in the current generation have converged (have the same fitness), do:

1. Evaluate (ie: apply f to) each chromosome in the current generation. Let the sum of all the resulting fitness scores be S.

2. Stochastically select $P/2$ pairs of chromosomes from the current generation to act as parents for the next generation, such that the probability of a particular chromosome c being selected is $f(c)/S$.

3. For each parent pair, apply a *recombination* operator, with probability p_R, which yields two child chromosomes from the parents. Also, to each child, apply a *mutation* operator with probability p_M. In this way a new population of P chromosomes will be produced. Call this the current generation, and return to step 1.

GAs vary considerably in the different choices for steps 2 and 3. Eg, the selection method ('roulette wheel' selection) we describe in step 2 above is common, but just one of a number of possible methods. In step 3, the general idea is that the children produced by recombination will tend to have higher fitness scores than the parents. This can easily be seen to happen if two highly fit parents are fit for different reasons, eg: parent pqr is fit mainly because of its third gene being an r, while parent xyz may be highly fit because of its first gene being an x. A recombination operator may then produce xqr, which combines good aspects of the parents to produce an even fitter child.

One typical recombination operator is one-point crossover. If we have two chromosomes, $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_n$, then a random number i is chosen between 1 and $n-1$; this serves as a crossover point. One child is then $x_1, x_2, .., x_i, y_{i+1}, y_{i+2}, ..., y_n$, while the other is $y_1, y_2, .., y_i, x_{i+1}, x_{i+2}, ..., x_n$. A more general crossover operation is *uniform* crossover, in which for i from 1 to n, the ith bit of child1 is randomly chosen from $\{x_i, y_i\}$, and the ith bit of child2 is x_i if y_i was chosen for child1, or y_i if x_i was chosen for child1. In *fixed-point* uniform crossover (1pu), a fixed number m of bit-positions are chosen from a parent; one child then has x_i in the ith position if i is one of the chosen positions, and y_i otherwise, and *vice versa* for the other child. In the experiments described later using 1pu, m was chosen to be half the chromosome length.

Many other choices of recombination operator are possible, with different operators working best for different problems. Recombination, however, is the essential aspect of a GA which seems to give it enormous power in searching the fitnes landscape. On the other hand, because of the nature of recombination, there may be parts of the search space which will be unavailable without the presence of a *mutation* operator. Mutation acts by randomly changing the values in bit positions, perhaps (re)introducing a possibly useful value. Recombination operators are normally applied to a pair of parents with a probability p_R, where typically $0.5 < p_R < 1$, while mutation is applied with a probability p_M, where typically $0 < p_M < 0.05$.

APPLYING GA TO THE MESP

In applying a GA to a problem, we must specify both a representation and an evaluation function for candidate solutions. Here we describe these aspects with regard to applying GAs to the MESP, while the next section outlines experiments involving the use of GA variants on a real MESP.

Representation: The representation that we have successfully used is simply a list of numbers of length e (the number of exams to be scheduled), each element of which is a number between 1 and t (the number of timeslots available). The interpretation of such a chromosome is that if the nth number in the list is t, then exam n is scheduled to occur at time t. For example, the chromosome [2,4,7,3,7] represents a candidate solution in which exam 1 takes place at time 2, exam 2 takes place at time 4, and so on. Other work on applying GAs to timetabling problems in schools (see [2]) has used an alternative representation where the *position* in a chromosome represents the timeslot, while what appears at that position is a set of exams. This however leads to the *label replacement* problem: crossover often may produce children which are not valid solutions in that some exams may not be scheduled at all, or scheduled more than once; this necessitates the use of a label replacement algorithm after crossover, to replace missing exams or remove duplicates. Our representation completely avoids this problem: crossover may certainly lead to missing or duplicated *timeslots* (so that at certain times no exams, or multiple exams, are scheduled), but these are perfectly valid, and possibly good timetables.

Evaluation: For a general MESP, the evaluation function must take a chromosome, along with the *student data*, and return a *punishment* value for that chromosome. The student data is simply a collection of lists, where each list is the set of exams to be taken by a particular student. Chromosome fitness can then be taken

as the inverse of punishment (or 1/(1+punishment) — to avoid division by zero). The evaluation function may comprise a weighted set of individual functions, each 'punishing' the chromosome in terms of a particular punishable 'offence'. In the MESP we experimented with, the components of the evaluation function are functions which respectively count the number of instances of the following 'offences':

- A student taking more than one exam at once (weight = 30).
- ... more than two exams in one day (weight = 10).
- ... two exams in consecutive timeslots on the same day (weight = 3).
- ... an exam just before and another just after lunch on the same day (weight = 1).

In other MESPs, different sets of component functions and weightings (the above were chosen intuitively) may be more appropriate, Eg: components which treat very early exams, or perhaps occurences of four exams in two days, as separate offences. In general, the fitness function for an MESP, where c is a chromosome, d is the student data, $\{m_1,...,m_n\}$ is a set of functions which each record the number of instances of a particular 'offence', and $\{w_1, ...,w_n\}$ are the weights attached to these offences, is: $f(c,d) = 1/1 + sum_i^n w_i m_i^{c,d}$.

Evaluation is generally the computational bottleneck of a GA. With the MESP, and 'offence-counting' modules of the type we have described, it is easy to see that the time to evaluate a chromosome increases linearly with the number of students, and can involve many computations depending on the kinds of punishment being looked for. It so happens (which is partly the message of this paper), that the GA technique is so powerful that a typical MESP can be quickly and effectively solved despite this bottleneck.

EXPERIMENTS AND RESULTS

Our experiments used real data from the MESPs of the EDAI postgraduate AI/CS exams for 1991 and 1992, involving 60 and 93 students respectively (from two departments), and 38 and 44 module exams respectively (including modules from four departments) to be scheduled , each student generally taking a selection of 8 exams from this pool. The different GA variants below were applied to both problems:

- **GA1**: basic GA with p_R at 0.7, p_M at 0.003.
- **GA2**: Inverse Square Pressure; as GA1, except that instead of using the inverse of punishment as the fitness function, we use the inverse square.
- **GA3**: GA2 + Elitism: here, one instance of the best chromosome in a generation was always copied into the next generation (note: because of the stochastic nature of selection in a GA without elitism, there is no guarantee that the best chromosome in one generation will appear in the next).
- **GA4**: GA3 + Operator Rate Interpolation (ORI); ORI involves gradually decreasing p_R and increasing p_M with each new generation.

We report on four experiments, each using one of the above GA variants on both the 1991 and 1992 timetabling problems. In the GA1 and GA4 experiments, we also experimented with different crossover operators.

SUMMARY OF RESULTS

The results we describe represent averages over 10 trial runs of 300 generations, with a population size of 50 in each generation. GA1 always managed to produce timetables comparable in fitness to the actual timetables used within 300 generations when `fpu`[2] was used, and GAs 2, 3 and 4 performed at least as well as the actual timetable whatever crossover was used, but `fpu` crossover was always best. In terms of the fitness score of the best chromosome after 300 generations, GA2 was 25% better than GA1, GA3 was about 250% better than GA1, and GA4 was about 400% better. Using GA4 with `fpu` on the 1991 problem reliably resulted in a timetable better than the actual one, and the best (of ten trial runs) was a timetable with zero punishment – ie: no instances of consecutive exams, or more than two exams in a day (and certainly no clashes!) for any student. The actual timetable used for these exams had a punishment of 37, involving 11 consecutive exams offences, and 4 'before-&-after-lunch' offences. In the 1992 case, again ten out of ten trials of GA4 with `fpu` crossover produced better timetables than that produced by the course organisers. The best from ten trials was a timetable with a punishment of 3 (only one occurence of a student having consecutive exams). The actual timetable used for these exams had a punishment of 101, involving 33 consecutive exams offences, 16 before-&-after-lunch offences, and 2 three-exams-in-a-day offences.

CONCLUSIONS

We have described the MESP, a common and very hard problem which occurs frequently in schools and universities. We have found that a simple, traditional GA using one-point crossover can quickly produce results comparable to human schedulers on a typical MESP. Further, combining elitism, operator rate interpolation, inverse square pressure, and `fpu` crossover leads to reliable production of optimal or near-optimal

[2] The fixed number of positions swapped each time being half the chromosome length (ie: half the number of exams involved).

timetables, faring much better than an unaided human team. Our approach is straightforward to implement, and hence should be easy to adopt in any department or institution which continually needs to solve MESPs. The work already done is described fully in [3].

There is, however, a great deal more work to be done. So far, we have shown that GAs show great promise on timetabling problems of the kind and size reported in this paper. Also, at the time of writing the system has just been used for the 1992/93 version of this problem, involving yet more exams and more students, resulting in the production of a successful timetable. We have no comparative results for this particular latest application; this is to be expected however, because the course organisers who now use the system certainly do *not* want to go to the trouble of producing an exam timetable by hand, now that they don't need to.

Nevertheless, we intend to pursue systematic studies from which we can learn how our approach scales up to larger problems, how the approach compares with conventional timetabling methods, and how performance varies with parametric, representational and algorithmic variations in the GA configuration.

Conventional computer-based timetabling methods concern themselves more with simply finding the shortest timetable that satisfies all the constraints, usually using a graph-coluring algorithm (finding sets of exams that can be scheduled at the same time corresponds to finding sets of vertices in a graph which are not adjacent), and less with optimising over a collection of soft constraints. As this is an NP-hard problem, conventional methods use approximate graph-colouring algorithms which usually find a reasonable, if not optimal, solution, though may perform arbitrarily badly. No such system that we know of, however, attempts to optimise over constraints of varying importance such as exam-consecutivity, etc ... ; this is because it is hard to fit priority based optimisiation into the conventional method. The GA approach, on the other hand, can deal with hard and soft constraints in a uniform way; further, adding or changing the importance of a constraint simply corresponds to adding/altering a component of the fitness function, rather than extensively revising the algorithm itself. At the moment, then, our approach is incommensurable with the conventional approach in that it attempts to solve a harder, more constrained problem. It still remains to be seen, however, how well our approach scales up to larger problems, both on its own and in comparison with conventional methods. Another important avenue to explore is the more general problem, usually handled by conventional methods, in which exams (or lectures) may occur in a set of rooms with different capacities. Investigating this requires us to modify the chromosome representation and evaluation function, and will very shortly be tried via a project at the EDAI; the method used in [2] handles this more general problem, but suffers from a fairly poor choice of representation.

Examining performance sensitivity will come by testing a large number of GA variants on a standard corpus of problems, coupled with some theoretical work. The first few techniques of a long list that we intend to examine in this respect are: performance *vs* variation in population size; performance *vs* altered interpretation of representation (ie: we can rewrite the evaluation function to interpret a chromsome as an implicit representation of a timetable which satisfies all the hard constraints; this slows down evaluation but may potentially improve overall speed and solution quality); performace *vs* different penalty settings for constraint violations, and so on.

Finally, we hope to have described how it is possible to greatly ease the burden on course organisers and also ease the pressure on students at exam time by applying a GA to the modular exam scheduling problem. We are unsure about the precise limitations of the technique at present, but feel confident about its general wider applicability. Work continuing at the EDAI will hopefully soon produce more conclusive performance data. We also feel confident that the general GA approach to timetabling will apply well to some other important problems of a similar form. An appropriate example is the problem of timetabling paper presentations at conferences with parallel sessions; if each delegate was given the opportunity to provide the organisers with their individual preferences regarding the papers to be presented, a GA based timetabling system very similar to the one we describe could arrange for parallel sessions to be organised such that delegates are collectively satisfied as far as possible, in terms of minimising the degree to which two presentations a delegate wishes to see are scheduled to occur at the same time. We plan to do this, with the help of the organisers, for the next European Conference on Artificial Intelligence.

REFERENCES

1. Goldberg, D.E., Genetic Algorithms in Search Optimisation & Machine Learning, Addison Wesley Reading, 1989.
2. Abramson & Abela "A Parallel Genetic Algorithm for Solving the School Timetabling Problem", Technical Report, Division of I.T., C.S.I.R.O, April 1991.
3. Fang, H-L., "Investigating Genetic Algorithms for Scheduling", MSc Dissertation, Department of Artificial Intelligence, University of Edinburgh, Edinburgh UK, 1992.

A Promising Hybrid GA/Heuristic Approach for Open-Shop Scheduling Problems

Hsiao-Lan Fang[1] and Peter Ross[1] and Dave Corne[2]

Abstract. Many problems in industry are a form of open-shop scheduling problem (OSSP). We describe a hybrid approach to this problem which combines a Genetic Algorithm (GA) with simple heuristic schedule building rules. Excellent performance is found on some benchmark OSS problems, including improvements on previous best-known results. We describe how our approach can be simply amended to deal with the more complex style of open shop scheduling problems which occur in industry, and discuss issues relating to further improvement of performance and integration of the approach into industrial job shop environments.

INTRODUCTION

The Open-Shop Scheduling Problem (OSSP) is a complex and common industrial problem [6]. OSSPs arise in an environment where there is a collection of operations to perform on one or more machines. Efficient production and manufacturing demands effective methods to optimise various aspects of schedule, usually focussing on the total time taken to process all of the operations. We present a hybrid GA/heuristic approach which performs very successfully in comparison with previous results on some simple benchmark OSSPs [12]. In two cases (for which global optima had not already been found), our results improve on a previously best known result produced by tabu search [12]. Our approach is flexible and easy to use in terms of development time, and also exhibits several ideas for future improvement.

We concentrate on three chromosome representation strategies. One is a straightforward extension to the OSSP of a strategy used for the job-shop scheduling problem (JSSP) in earlier work [5], which does not involve any hybridisation. The other two strategies incorporate simple means of hybridising the GA with heuristic rules. One of these methods seems more powerful and robust than the other two.

We also note that the benchmark problems used can be freely obtained for comparative research, and describe how our approach can be extended to address more complex open shop scheduling problems. We know of no GA-based efforts except ours on the OSSP with which to compare, so we present results in order to show the potential for a GA approach to open-shop scheduling, and invite fellow researchers to experiment with the same problems.

[1] University of Edinburgh, Department of Artificial Intelligence, 80 South Bridge, Edinburgh, EH1 1FN, UK
[2] University of Edinburgh, Department of Artificial Intelligence, 5 Forrest Hill, Edinburgh, EH1 2QL, UK

ECAI 94. *11th European Conference on Artificial Intelligence* Edited by A. Cohn
Published in 1994 by John Wiley & Sons, Ltd.

1.1 Overview

Section 2 describes the OSSP in detail, and our GA approach is described in section 3. Experiments and results on benchmark problems are then presented in section 4. Section 5 discusses these results, advances various issues concerning performance improvement, and notes how the approach may be extended to cope with more complex OSSPs.

2 OPEN-SHOP SCHEDULING PROBLEMS

A commonly used simplification of the OSSP is to specify that each given operation can only be processed on a given specified machine. In reality, an operation can often be processed in a number of alternative ways, any of which may involve more than one machine. There may also be due dates and machine setup times to consider. In the following however we will concentrate on a simplified form of the general problem; this is done mainly because the benchmark problems on which we test the performance of our GA approach are thus simplified. We will later discuss simple amendments to our approach which promise to successfully cope with the more general problem.

An OSSP involves a collection of m machines and a collection of j jobs; each job comprises a collection of operations (sometimes called tasks). An operation is an ordered pair (a, b), in which a is the machine on which the operation must be performed, and b is the time it will take to process this operation on machine a. A feasible OSSP schedule assigns a start time to each operation, satisfying the constraint that a machine can only process one operation at a time, and that two or more operations from the same job cannot be processed at the same time. The main objective is usually to generate a schedule with a makespan as short as possible; the makespan is simply the total elapsed time in the schedule. More complex objectives often arise in practice, where due dates and machine set up times must also be taken into account, for example.

The common illustration of this kind of problem is that of an automotive repair shop [6]. In such a shop, a typical job might involve the operations 'spray-paint', and 'change-tyres' to be performed on the same vehicle. These operations cannot usually be performed concurrently (especially if the stations at which these operations are performed are in different places, for instance), but can be performed in any order. Also it is usually true that different stations (ie: 'machines'

) can concurrently process operations from different jobs (eg: involving different vehicles). If the operations in a job must be performed in some fixed order, then this becomes a 'Job-Shop Scheduling Problem' (JSSP).

Certain benchmark OSSPs have been used for comparative research. In these, each job comprises precisely one operation for each machine. These benchmarks are hence completely defined by an ordered collection of m processing times for each job. For example, table 1 shows a 5×5 (ie: 5 jobs and 5 machines) benchmark problem, taken from [9].

Table 1. A 5x5 benchmark OSSP

Machines:	1	2	3	4	5
Job 1:	64	66	31	85	44
Job 2:	7	69	68	14	18
Job 3:	74	70	60	1	90
Job 4:	54	45	98	76	13
Job 5:	80	45	10	15	91

In the above example, operation 1 of job 1 must go to machine 4 for 85 units of processing time, operation 2 of job 1 must go to machine 1 for 64 units of processing time, and so on, with no restrictions on the order in which the tasks for any job are to be processed. The problem is to generate a valid schedule with minimal makespan. Figure 1 shows a minimum-makespan (300) schedule for the benchmark in table 1.

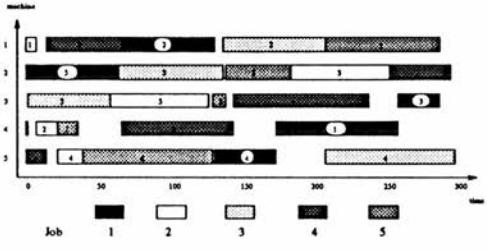

Figure 1. Minimal-makespan schedule for a 5x5 OSSP benchmark

3 A GA/HEURISTIC APPROACH TO THE OSSP

A common general technique for hybridising a GA with a heuristic search or heuristic rule based method is to use the GA to search a space of *abstractions* of solutions, and employ a heuristic or some other method to convert the points delivered by the GA into candidate solutions. Such hybridisation is one way of avoiding the often highly complicated problem of representing a complete solution as a chromosome in a way that facilitates effective GA-based search; it is usually more easy to represent abstract regions of the solution space, and have these abstractions converted into (ie: interpreted as) solutions by some other technique.

This paper presents some simple examples of such hyb GA/heuristic methods for the OSSP. Similar GA/heuri hybridisation occurs variously in the GA literature. Eg, a cent discussion of hybrid GA/heuristic hybrids for bin-pack and related problems appears in [11]. In the following, we scribe three simple strategies for using a GA to address OSSP.

3.1 Basic chromosome representation

Each of the representations we discuss is based on the foll ing basic technique. The genotype for a problem is a strin p genes, where p is the total number of operations involv summed over each job. Each gene can take alleles in the ra $\{1,2,\ldots,j\}$, where j is the largest job number. A chromoso provides instructions for building a schedule as follows: string of genes $abc\cdots$ means: "choose an untackled operat from the the a-th uncompleted job, and place it in the ea est place where it will fit in the developing schedule, cho an untackled operation from the b-th uncompleted job place it into the earliest place where it will fit in the de oping schedule, ...", and so on. Building a schedule is complished by a schedule builder, which maintains a circu list of uncompleted jobs and a list of untackled operations each such job. Thus the notion of "a-th uncompleted job taken modulo the length of the circular list to find the act uncompleted job being referred to.

Evidently, this description is incomplete because of word "choose" in the interpretation method. In this se each chromosome represents a region of the space of possi solutions. For example, the region of solutions which may represented by the chromosome "1,2,1,..." is that in wh the first operation scheduled comes from job 1, the sec from job 2, the third from job 1, and so on. The way tha chromosome is interpreted as a single solution somewher this region can vary. In this paper we look at three way doing this.

3.2 Directly encoding the operation

This is the most straightforward method; we simply dou the size of the chromosome by incorporating genes for choice of operation in addition to those for choice of Hence, $abcd\cdots$ will now mean: "choose the a-th untack operation from the the b-th uncompleted job, and place i the earliest place where it will fit in the developing sched choose the c-th untackled operation from the d-th unc pleted job and place it into the earliest place where it wil in the developing schedule, ...", and so on. We will refe this method with the abbreviation JOB+OP.

3.3 Fixed heuristic choice

In this method, a fixed heuristic is decided upon beforeha and used by the schedule builder to make the choice of eration at each step. Hence, if we use heuristic X to m this choice, then the interpretation of $abcd\cdots$ becomes : " heuristic X to choose an operation from the a-th uncomple job, and place it in the earliest place where it will fit in developing schedule, use heuristic X to choose an operat from the b-th uncompleted job, and place it in ...", and so

e use the abbreviation FH(X) in referring to this strategy, respect of some particular heuristic X. The simple heuris- s we will refer to in this paper are the following eight. In ch case, the set of possible operations to choose from are ose in the 'current job'. This 'current job' is that repre- ated (via the circular list of uncompleted jobs) by the allele the chromosome which is being interpreted at this step.

PT: choose the operation with largest processing time, breaking ties according to an *a priori* ordering over the operations.

PT: choose the operation with shortest processing time, breaking ties according to an *a priori* ordering over the operations.

F-LPT: Let t be the earliest time at which an operation can be scheduled, and let S be the set of operations that can be scheduled at t. Simply apply LPT to the operations in S.

F-SPT: As above, but using SPT instead of LPT.

F-BTR: As above, but simply choosing randomly from the set S.

G-LPT: Let G be the set of operations that can be placed in a gap in the schedule; that is, those operations which fit inbetween two already scheduled operations on the same machine. Apply LPT to the operations in G. If G is empty, proceed as with LPT.

RG: Choose the operation from G which leaves the longest amount of time in its gap, breaking ties randomly. If G is empty, then simply use LPT.

RG: Choose the operation from G which leaves the shortest amount of time in its gap, breaking ties randomly. If G is empty, then simply use LPT.

For example, in later experiments using FH(LPT), this ers to the fixed-heuristic hybrid method, with LPT being heuristic used in this case.

4 Evolving heuristic choice

ally, note that there is no good reason to rely on a fixed ristic for each choice of operation while building a sched- . Indeed, it is quite easy to see that varying the choice of ristic according to the particular job being processed, and o according to the particular stage in the schedule building ocess, may make more sense. It is hard to find some prin- led *a priori* method for making these varied choices, but can implement a simple adaptive strategy by extending basic chromosome representation as follows. A chromo- ne $abcd\cdots$ now means: "use the a-th heuristic to choose an eration from the b-th uncompleted job, and place it in the liest place where it will fit in the developing schedule, use c-th heuristic to choose an operation from the d-th uncom- ted job, and place it in ...", and so on. We dub this method HC', for 'Evolving Heuristic Choice'. Alleles of genes which interpreted as heuristic choices (eg: odd-numbered genes he above example) range through the number of available ristics; in the experiments described later, the set of pos- le choices are the eight described earlier. We have found eneficial for these alleles to be preferentially set to one in ticular of these choices (LPT for most problems) in the ial generation, thereafter being allowed to vary via muta- n and recombination. In the next section then, when we refer to the use of a *particular* heuristic in association with EHC, we simply mean that the *initial generation* of chromosomes have their heuristic choice alleles set to this heuristic, but are allowed to vary from then on.

4 EXPERIMENTS AND RESULTS

We tested each approach on six benchmark OSSPs of sizes 4×4, 5×5, 7×7, 10×10, 15×15, and 20×20. In each case, the GA used fitness-proportionate selection based on the objective function makespan - lower bound, lower bounds being provided in [12]. Elitist generational reproduction was used; uniform crossover was used, applied adaptively. That is, the crossover rate began at 0.8, and was reduced by 0.0005 after each generation down to a minimum of 0.3. Two children were produced from each crossover; these children, or the parents (when crossover was not applied) were each mutated with a fixed probability of 0.5; mutation involved swapping two randomly chosen genes. In each case, the results report the average of the best makespan found from each of ten trial runs of a maximum of 1000 generations, and the best makespan found overall. Convergence typically occured very quickly (sometimes in the initial generation) on the 4×4 problem; on larger problems, convergence ranged from an average of around 30 generations for the 5×5 problem to an average of 350 generations for the 20×20 problem. The population size was 200 in each case.

Table 2 shows results for the smaller three benchmarks. The 'Previous Best' row gives the best previously known solution. In the 4×4 and 5×5 cases, these are known to be global optima.

Table 2. Results on small benchmark OSSPs

	Benchmark OSSP (jobs x machines)		
	4 × 4	5 × 5	7 × 7
Previous Best	193	300	438
JOB+OP			
Mean	194.4	308.5	454.1
Best	193	302	441
Fixed Heuristic			
Mean (EF-SPT)	193.4	303.9	449.7
Best (EF-SPT)	193	301	445
Mean (EF-LPT)	211.0	312.2	449.8
Best (EF-LPT)	211	305	443
Mean (EF-BTR)	195.6	305.6	448.9
Best (EF-BTR)	195	301	436
Evolving Heuristic Choice			
Mean (EF-SPT)	193.0	305.0	449.7
Best (EF-SPT)	193	300	441
Mean (EF-LPT)	194.3	307.6	444.9
Best (EF-LPT)	193	305	435
Mean (EF-BTR)	194.1	305.3	449.8
Best (EF-BTR)	193	300	435

The most striking aspect of these results was that EHC yielded a better result, 435, than any previously found (that we know of) on Taillard's 7×7 benchmark ; the previous best for this was reached by tabu search [12]. Note also that FH(EF-BTR) also improved on this previous best. More generally, it appears that the methods incorporating SPT (as

fixed in FH, or *initially* fixed in EHC) are best on the two smallest problems while LPT shines through on the larger problem. We find that this reliably extends to problems larger than 7 × 7, and hence only incorporate LPT (as the fixed or initially fixed choice) in the experiments to follow. All of the FH and EHC methods improved on the JOB+OP trials, showing a clear benefit for some form of hybridisation.

In table 3, we compare JOB+OP, FH(LPT), and EHC(LPT) for the three larger benchmarks. Note that in this case the previous best results are known to be global optima for the 15 × 15 and 20 × 20 cases [12].

Table 3. Results on large benchmark OSSPs

	Benchmark OSSP (jobs × machines)		
	10 × 10	15 × 15	20 × 20
Previous Best	645	937	1155
JOB+OP			
Mean	690.7	968.9	1244.5
Best	668	951	1224
Fixed Heuristic (LPT)			
Mean	662.7	942.9	1163.7
Best	646	937	1155
Evolving Heuristic Choice			
Mean	660.1	940.1	1167.7
Best	641	937	1156

The most striking result in this case is the discovery of a new best result for the 10 × 10 benchmark, which again was obtained using the EHC method. Note also that these results further underline the quality of a hybrid approach as compared to that of the 'pure GA' JOB+OP method. Beyond these observations we cannot really discern any clear indications as to the relative quality of FH and EHC on the two larger benchmarks.

5 DISCUSSION

The approach we describe provides excellent results on difficult benchmark problems. Although this is no guarantee that the approach will generalise successfully to real problems, and/or perform just as well on different and larger benchmarks, it is clearly a promising enough basis for continued research along these lines.

It is particularly encouraging that even the best results were achieved with an essentially simple technique, improvements on which can be readily imagined. This augments a continuing theme in GA research literature, which shows that GAs begin to compete closely with or outperform other known methods on some problems when successfully hybridised [10, 11, 8]. Further work is under way to study more sophisticated heuristics and hybridisation strategies.

In the context of the interplay between the GA and the heuristics, these results appear counter to findings like those of [1], which suggest that the more search done by the GA at the expense of the heuristic, the better in terms of final solution quality, though probably at the expense of time. Our results, and those of other authors in other applications, tend to show the opposite: better quality results arrive through hybridisation with a heuristic, with little extra time cost. Reconciliation of such counter observations are readily found ho ever, by recognising that most generalisations we make fr necessarily small forays into the space of possible experime are at the mercy of being overturned by further such inv tigation. Better solution quality may well have arrived h through a 'pure' GA approach (such as JOB+OP) but only a rather extreme cost in time; eg: for JOB+OP to compete terms of solution quality with EHC may well be possible, b perhaps only if we use far larger population sizes and con quently wait far longer for convergence. Bagchi *et al*'s noti makes intuitive sense if we consider that it allows the GA f rein over the space of possible solutions, rather than searc ing a contracted space as is effectively done in most hyb methods. However, this 'expansion' of the space that we all the GA to survey carries with it the need for more extensi sampling and hence much larger population sizes. A hyb GA/heuristic method thus tends to seem the better practi choice, offering a better tradeoff in terms of speed *vs* qual on most problems.

5.1 Extension to more 'real' OSSPs

The more general statement of a jobshop problem is mo complex than that described here in two main ways. First, operation has a collection of alternative process plans (it c be done on different machines), rather than the single proc plan of being specified to be done on a particular machi Let each job j_i have p_i alternative process plans. Each su plan is a distinct set of machines and associated processi times, each representing an alternative way of discharging t job. We might extend our representation to incorporate su alternatives as follows: a schedule $abcd\cdots$ means: "choose untackled operation from the the a-th uncompleted job, ing the b-th valid process plan for this job, and place it in the earliest place where it will fit in the developing schedu ...", and so on. Here, when the schedule builder identifies t job currently referred to in the chromosome (via the heur tic choice), earlier choices in the schedule constrain the of valid alternative process plans that are still 'live' for t job. The valid set is treated as circular, and chosen from directed by the chromosome. There are of course several oth possibilities. For example, we could use essentially the sa representation as used for the simpler OSSP, but change t interpretation to: "heuristically choose a valid process pl from the a-th uncompleted job and then heuristically choo an operation from this plan, and place it into the earliest pla where it will fit in the developing schedule, ...", and so o This involves the addition of a heuristic to choose the p cess plan as well as choose an operation. Possibilities for t heuristic which chooses the process plan are easily imagine For example, we might choose the plan for which the to processing time remaining is smallest.

The second important difference is that jobs have due da which need to be considered, and also relative precedenc This can be dealt with in our approach simply by incorpora ing these considerations into the fitness function. That is, t fitness measure is a combination of the makespan of the sch ule and the extents to which job precedences are honoured a due dates are met. Alternatively, preference and due date formation may be readily incorporated into a heuristic. E example, instead of a heuristic which chooses the job wi

he largest processing time, we might choose the job which naximises some function of processing time and the extent to which its due date is met.

Hence, various possibilities are apparent for extending the approach to deal with more general problems. Such has been reported, for example, in the context of highly generalised manufacturing scheduling problems [7], although this did not report on the hybridisation of the GA with simple heuristics (chromosomes were much more direct representations of schedules). Our main point here is to show how our approach readily allows for extensions which will allow it to cope with problems of the fully general kind found in real machine shop environments, while still retaining its basic flavour, and hence retaining the presumed source of its success. It may not be immediately apparent that the success we demonstrate on simplified benchmark OSSPs will carry over to effective performance on more complex problems in an extended approach, but there is no apparent reason to be too skeptical of this possibility. Further work along these lines will be reported in due course.

5.2 Conclusion

We have presented an approach to the OSSP which performs very promisingly on benchmark OSSPs, twice outperforming previous reported attempts. We discussed how the approach may be extended to deal with more realistic problems; the simplicity of the approach, its apparent success, and the evident potential for much further improvement and extension, seem to render it a promising method warranting further research. Ultimately, of course, comparisons with other AI- or OR- based methods will be instructive. Also, the approach as presented fails to meet some possible needs which schedule managers may have in machine shop environments; eg: there is no clear way in which *rescheduling* can be addressed, other than by redefining the problem as necessary and running the GA from scratch; a more sophisticated technique however would be one which made use of information gained during formation of the previous schedule, which makes rescheduling a potentially very speedy process.

Finally, it should be noted that the GA configuration used in the experiments here is not optimal. Continuing GA research reveals variants and techniques that GA application researchers will find rewarding to heed. For example, [2, 3] both describe spatially-oriented selection strategies which seem to consistently outperform others, while [4] describes, among other things, reinitialisation strategies which can enhance overall robustness and reliability.

5.3 Notes

The benchmarks used here can be obtained via [9]. The OR library referred to in [9] is an electronic library of benchmarks for a wide range of OR problems. Researchers wishing to compare with our results will need to know that the problems referred to here are each the *problem No. 1* of their specified size and kind. Alternatively, problem data may be obtained directly from the authors, as can the details of the schedules found here which improve on previous best known results.

ACKNOWLEDGEMENTS

We are grateful to two anonymous referees for helpful and constructive comments on an earlier version of this paper. Thanks too to the UK Science and Engineering Research Council for support of Dave Corne via a grant with reference number GR/J44513.

REFERENCES

[1] Sugato Bagchi, Serdar Uckun, Yutaka Miyabi, and Kazuhiko Kawamura, 'Exploring problem-specific recombination operators for job-shop scheduling', in *Proceedings of the Fourth International Conference on Genetic Algorithms"*, eds., R.K. Belew and L.B. Booker, pp. 10–17. San Mateo: Morgan Kaufmann, (1991).

[2] Robert J. Collins and David R. Jefferson, 'Selection in massively parallel genetic algorithms', in *Proceedings of the Fourth International Conference on Genetic Algorithms*, eds., R.K. Belew and L.B. Booker, pp. 249–256. San Mateo: Morgan Kaufmann, (1991).

[3] Yural Davidor, 'A naturally occuring niche & species phenomenon: The model and first results', in *Proceedings of the Fourth International Conference on Genetic Algorithms*, eds., R.K. Belew and L.B. Booker, pp. 257–263. San Mateo: Morgan Kaufmann, (1991).

[4] Larry J. Eshelman, 'The chc adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination', in *Foundations of Genetic Algorithms*, ed., G. Rawlins, 265–283, Morgan Kaufmann, (1991).

[5] Hsiao-Lan Fang, Peter Ross, and Dave Corne, 'A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems', in *Proceedings of the Fifth International Conference on Genetic Algorithms*, ed., S. Forrest, 375–382, San Mateo: Morgan Kaufmann, (1993).

[6] Teofilo Gonzales and Sartaj Sahni, 'Open shop scheduling to minimize finish time', *Journal of the Association for Computing Machinery*, **23**(4), 665–679, (October 1976).

[7] P. Husbands and F. Mill, 'Simulated co-evolution as the mechanism for emergent planning and scheduling', in *Proceedings of the Fourth International Conference on Genetic Algorithms*, 264–270, San Mateo: Morgan Kaufmann, (1991).

[8] Jr. James D. Kelly and Lawrence Davis, 'Hybridising the genetic algorithm and the k nearest neighbours classification algorithm', in *Proceedings of the Fourth International Conference on Genetic Algorithms*, eds., R.K. Belew and L.B. Booker, pp. 377–383. San Mateo: Morgan Kaufmann, (1991).

[9] Beasley J.E., 'OR-library: Distributing test problems by electronic mail', *Journal of the Operational Research Society*, **41**, 1069–1072, (1990).

[10] Si-Eng Ling, 'Intergating genetic algorithms with a prolog assignment problem as a hybrid solution for a polytechnic timetable problem', in *Parallel Problem Solving from Nature, 2*, eds., R. Manner and B. Manderick, 321–329, Elsevier Science Publisher B.V., (1992).

[11] Colin Reeves, 'Hybrid genetic algorithms for bin-packing and related problems', Technical report, School of Mathematical and Information Sciences, Coventry University, (1994). submitted to *Annals of OR*.

[12] E. Taillard, 'Benchmarks for basic scheduling problems', *European Journal of operations research*, **64**, 278–285, (1993).

Successful Lecture Timetabling with Evolutionary Algorithms

Peter Ross, Dave Corne, Hsiao-Lan Fang
Department of Artificial Intelligence
University of Edinburgh
80 South Bridge, Edinburgh EH1 1FN
email: {peter|dave|hsiaolan}@aisb.ed.ac.uk

Abstract

Arranging a lecture/tutorial/lab timetable in a large university department or school is a hard problem faced continually in educational establishments. We describe how this problem has been solved in one institution via the use of evolutionary algorithms. The technique extends easily and straightforwardly to any lecture timetabling problem. Although there may be more effective ways to handle particular instances of the general lecture timetabling problem, we note that the combination of speedy, good results and ease of development for the particular application in hand make the EA based technique we present potentially widely useful in general.

1 Introduction

Lecture timetabling is the problem of assigning times and places to a many separate lectures, tutorials, etc ..., to satisfy several constraints concerning capacities and locations of available rooms, free-time needs and other such considerations for lecturers, and relationships between particular courses. The most prominent overall constraint (central to all timetabling problems) is that there should be no *clashes*; that is: any pair of lectures (or tutorials, etc ...) which are expected to share common students or teachers should not be scheduled simultaneously.

Typically, this is addressed by drawing up an initial draft timetable, followed by perhaps weeks of redrafting as complaints about the most recent draft flow in from various sources. The space of possible timetables can be nightmarish to traverse, and there have been several attempts to find useful AI/OR approaches to aid the process [2, 15, 4, 11, 3]. Success has been recently reported for using evolutionary algorithms (EAs) for timetabling [1, 6, 14, 7, 8, 17, 16]. Here we describe one such EA approach, and present illustrative results on some real lecture timetabling problems.

In presenting results on some real problems, we augment similar work which also reports results on real problems [6, 7, 8, 17, 16], but we also present a fuller and deeper discussion of the general approach, clarifying how it may be used on a much wider range of problems than that studied. Some comparison is also made between the use of different EA selection schemes. Also,

in reporting EA-based results in comparison with the independently and 'expertly' calculated timetables for four real lecture timetabling problems we show clearly how this approach can yield very beneficial improvements.

Overview

We first describe the kind of problem addressed in more detail in section 2. Description of our EA based approach follows in section 3, and notes on implementing the approach then appear in section 4. Illustrative experiments on real problems appear in section 5, followed by general discussion in section 6.

2 Lecture Timetabling Problems

The basic element of a lecture timetabling problem is a set of events $E = \{e_1, e_2, \ldots, e_v\}$. Each member of E is a unique event requiring assignment of a time and a place. That is, it may be a lecture, a tutorial, a lab session, or some other event which plays a part in the term timetable. We could alternatively formulate this, for example, in terms of a set of subjects S, each of which has associated numbers of lectures, tutorials, lab sessions, etc ... per week. However it is simpler to take as our starting point the set E as described, which is easily generated from the latter kind of data. Eg, two separate Lisp Programming lectures and five Lisp Programming tutorials will constitute seven members of E.

Each event e_i has an associated length l_i (how long the event is in, say, minutes), and an associated size s_i, which is either known or an estimate of the number of students expected to attend that event. There is also a set of 'agents' $A = \{a_1, a_2, \ldots, a_t\}$; these are lecturers, tutors, technicians, etc ... — people with some kind of distinguished role to play in an event. Finally, there is a set of places $P = \{p_1, p_2, \ldots, p_q\}$, and a set of times $T = \{t_1, t_2, \ldots, t_s\}$. An *assignment* is a four-tuple (a, b, c, d), in which $a \in E, b \in T, c \in P, d \in A$, with the interpretation "event a starts at time b in place c and is taught (lectured, tutored, ...) by agent d. A lecture timetable is simply a collection of n assignments, one for each event.

Such problems are beset by many kinds of constraint. A fuller presentation of these appears in [18], and various more simplified treatments have been presented elsewhere [2, 1, 3]. The following briefly notes the necessary aspects of the approach discussed in [18] which are needed for understanding the rest of this paper. Following this we describe the constraints which need to be met in a *specific* series of lecture timetabling problems; this serves as a background case-study against which we can illutrate some general aspects of the implementation of the approach to an arbitrary timetabling problem.

3 Evolutionary Timetabling

Assuming familiarity with the basic processes in EAs, it suffices to describe our approach by reference only to the chromosome representation and the fitness function. Those unfamiliar with the basics of EAs can consult good texts such as [12, 10].

The Chromosome Representation

A 'timetable' chromosome is a vector of symbols of total length $3v$ (recall: v is the number of events), divided into contiguous three-gene chunks. The three alleles in the ith chunk, where $1 <= i <= v$, represent the time, place, and agent assignments of ith event. Naturally, the sets of possible alleles at time, place, and agent genes are respectively identified with the sets T, P, and A. The simple example chromosome "abcdef" represents a timetable in which event e_1 starts at time a in place b, involving agent c, and event e_2 starts at time d in place e, involving agent f.

This constitutes a 'direct' representation, as opposed to the more indirect style common in EA-based job shop scheduling work, and recently implemented for timetabling in [16]. Relative advantages and disadvantages of these two styles are beyond the scope of this paper, but are a central point of interest. Suffice to say here that comparison of the two is beset by complications, but it so far seems that the direct approach *enhanced* with the introduction of intelligent mutation operators (which take great advantage of the directness of the representation, and hence cannot be feasibly constructed for use with the indirect style) [8], vies on equal terms with an ingenious version of the indirect style [16]; these observations are yet to be properly backed up empirically.

3.1 The Fitness Function

A maximally fit timetable is clearly one which satisfies all of the imposed constraints. Also, it seems reasonable to distinguish between timetables in terms of fitness based on the numbers and kinds of different constraints violated. A choice of fitness function which meets this behaviour is as follows, where C is the set of constraints in the problem, P_i is a penalty associated with constraint i, and $v_i(g) = 1$ if timetable g violates constraint i, and 0 otherwise:

$$f(g) = 1/(1 + \sum_{i \in C} P_i v_i(g)) \tag{1}$$

The relative penalty values may be chosen to reflect intuitive judgement of the relative importance of satisfying different kinds of constraint. Further discussion of this kind of objective function and other possibilities is beyond the scope of this paper, but appears elsewhere, eg: [8, 19, 13]. In general, however, a penalty function as above, using a rough choice of penalty settings derived from the course organisers' notion of relative importance of different constraints, appears adequately robust for many problems.

4 A Specific Lecture Timetabling Problem

An MSc course in the Department of Artificial Intelligence, University of Edinburgh (EDAI) involves eight taught course modules spread over two terms. The course is organised into 'themes', each involving a particular combination of 8 modules, of which some are compulsory and some optional. As well as choices from among the 30+ modules available in the AI Department,

students may also choose modules from the Computer Science (CS) Department and others. A complicating factor here is that the CS Dept is an inconvenient bus ride away from the AI Dept.

T comprises 80 start times, 16 per day on each day of a five day week. Each day's slots are at half-hourly intervals from 9am to 4:30pm. E comprises a large collection of lectures, tutorials, and lab sessions, mostly an hour long, but sometimes two hours long. A student enrolled on a course must attend all the lectures in E involving the course, but only one of the tutorials or labs (E will include several tutorials or labs for each course). Separate courses are pre-assigned to either term 1 or term 2. Hence, in one academic year there is a separate lecture timetabling problem for each term involving roughly half of the modules available on the course as a whole. The full set of constraints which need to be faced are as follow:

Options : Student's options should be kept open as far as possible. No pair of lectures in the same theme should overlap in the timetable. More generally, lectures x and y should not overlap if there is expectation that one or more students may wish to take both courses x and y.

Event Spread : The individual timetable for any student must be spread out fairly evenly. Eg: A student should not have to sit through four lectures in a single day. Rather, the events an individual student must attend should be evenly spaced out during the week. Also, different lectures on the same topic (eg: there may be 2 Prolog lectures per week) should occur on different days.

Travel Time : A student should have at least 30 minutes free for travel between events in the CS Dept and events in the AI Dept.

Slot Exclusions : CS lectures should occur in morning slots, and AI lectures in afternoon slots (this arises from an inter-departmental agreement). Also, lectures at lunchtime (starting at 1:00pm or 1:30pm) should be avoided if possible. In a similar vein, various constraints of the form "event e cannot start at time t" arise owing to other commitments of the staff involved.

Slot Specifications : Various constraints are given in the form "event e must start at time t", arising for various reasons.

Capacity : The size of a lecture or tutorial should not exceed the capacity of the room it occurs in. Also, a room can only cope with one event (lecture, tutorial, or lab) at a time (this is the main difference between lecture and examination timetabling).

Room Exclusions : Many constraints on room assignments for particular events can easily be derived from the Capacity constraints, along with information about the expected sizes of events. In addition however, there are other considerations which lead to several *a priori* constraints of the form "event e cannot occur in room r". For example, event e may demand disabled access, or certain audio-visual requirements unavailable in room r.

Room Specifications : Similarly, several constraints are apparent of the form "event e must occur in room r".

Juxtaposition : Preferably, all tutorials or laboratory sessions for any course should occur later in the week than the week's first lecture on that course. Sometimes this is particularly necessary, since a tutorial or lab session may be based on the lectures which were held (hopefully) earlier in the week. In other cases this is desirable but not vital.

Translating the Constraints into a Fitness Function

In this case, it so happens that every lecture's agent (ie: lecturer) is predetermined, and individual lecturers have already decided in advance (providing details in the style of exclusion constraints) which slots they are not available for. Tutors for tutorials and lab sessions are not pre-determined in this way, but for these there is no point in incorporating them into the timetable at this stage (usually well in advance of term), since we simply do not know for sure who will be available and when. For this problem we therefore need not consider the set A, and hence can use chromosomes of length $2v$. The above constraints can be dealt with as follows:

Keeping Options Open

The Options constraint is handled by interpreting it as a large collection of binary constraints, each involving a distinct pair of events expected to share students. For convenience, we derive a set of 'virtual' students, each of whom takes a distinct one of the set of possible four or five-module options for the term. This set of virtual students is then used to derive the collection of distinct binary constraints between events. Here, such a binary constraint occurs between every pair of distinct lectures taken by some virtual student. A similar constraint also holds between distinct lectures on the same module, and between lectures and tutorials on the same module. No such constraint is needed between different tutorials or labs for the same course, or even between tutorials and labs on different courses. Such may be scheduled simultaneously, and often are; in due course this gives rise to constraints which affect each students' choice from the set of tutorial and/or lab sessions available for each module.

This amounts to a collection of binary constraints of the form "e_1 must not overlap in time with e_2"; the fitness function must check for violation of each of these in turn. Similarly, it should be clear how the basic problem of avoiding clashes in any other lecture or exam timetabling problem can be dealt with. Notice too that we can incorporate the Travel Time constraint here. If e_1 and e_2 are timetabled with a break of, say, k minutes between them, but are assigned to rooms which take more than k minutes to travel between, then any Options constraint between them is effectively violated. Hence, the violation check for Options constraints can, simply via accessing the 'place' genes for e_1 and e_2 and a given travel-time matrix for the places, also account for Travel Time constraints.

Event Spread constraints

Event spread constraints can be handled in a number of ways. Eg, to even out the event spread for individual students we might explicitly calculate some measure or measures of event spread for each virtual student. Alternatively, we could reasonably approximate this by examining some

measure or measures of the spread of the timetable as a whole. Both would seem to offer the same overall effect; the latter will typically be computationally cheaper, but the former approach would seem to offer more potential for control and tradeoff of different aspects of the event spread for individual students.

The method used in the experiments detailed later is as follows: the fitness function notes, for each virtual student, the number of instances of the following two 'offenses': a) four events are scheduled in one day for this virtual student; b) five or more events are scheduled in one day for this virtual student. A different penalty term is associated with each, and the penalty weighted sum of instances of these offenses, summed over virtual students, makes up the contribution to (or, rather, detraction from ...) fitness of the overall event spread constraint.

Finally, 'different-day' constraints can clearly be handled in the same way as Options constraints. For any pair of lectures on the same module, we simply check directly from the chromosome whether or not their assigned slots are on the same day. If they are, then an appropriate penalty is added.

Exclusions and Specifications

It is easy to see how exclusion constraints can be directly translated into one or more simple violation-check functions, given the chromosome representation in use. Notice however that we can just as simply pre-arrange it so that chromosomes never violate these constraints in the first place. We can doctor the allele range of each gene so that it is always the specified allele (if any), or only ranges over the non-excluded alleles.

Choosing between such pre satisfaction of exclusion and specification constraints, and the option of penalising violations of them, is not always straightforward. If many such constraints exist, the 'pre-satisfied' space may well lack excellent timetables which violate a few exclusions, for example, but make up for this in other ways. On the other hand, pre-satisfaction speeds up evaluation and promises to speed up search via reducing the search space. The full ramifications of this choice are beyond the scope of this paper, but it suffices to point out here that either option should be available in a system which implements this technique. In the experiments discussed later, most exclusion and specification constraints were prespecified. The only 'penalised' such constraint was that for lunchtime lectures. According to the EDAI MSc course organisers, it is preferable to avoid these, but acceptable to trade these off against other constraint violations.

Capacity constraints

To check that a room's capacity isn't exceeded, we must first translate the overall room capacity constraint into constraints of the form "room r should contain no more than r_{cap} students in timeslot t", for each room r and timeslot t, where r_{cap} is the student capacity of room r. During evaluation, the system simply precomputes from the current candidate timetable the student load for each room in each slot, and then runs through this list of constraints checking each in turn and accumulating penalties for violations. Evidently, the same technique can be applied for a very wide range of similar problems, and also applies to constraints concerned with 'teaching loads' constraints in problems for which agents must be considered in the representation.

Juxtaposition Constraints

Finally, it is evident how the ordering constraints between given groups of events can be incorporated. We first derive a collection of binary constraints from those given. Eg, "all lisp tutorials should occur later than the first lisp lecture of the week" is translated into a collection of binary constraints of the form "lisp_1 must be before lisp_t3". Checking for violations of such constraints is then straightforward . Similarly, it should be clear how any juxtaposition constraint (eg: "there should be at least two days between event e_1 and event e_2") can be similarly handled.

5 Experiments

Experimental Setup

We address the EDAI MSc lecture/tutorial problems for both terms of the academic years 92/93 and 93/94, respectively involving 76, 73, 82, and 73 events. Other features of these problems are as described in section 4, and full details are available from the authors.

In all cases, the EA used a population size of 50, uniform crossover, gene-by-gene mutation, and elitist generational reproduction. The crossover and mutation rates p_C and p_M were dynamically altered as follows. p_C started at 0.8 and was decreased by 0.001 after each generation (ie: after every 50 evaluations), with a lower limit of 0.6, while p_M started at 0.003 and increased by 0.0003 each generation, with an upper limit of 0.02. Each trial was run for 200 generations (10000 evaluations). Separate experiments are recorded for each problem for each of three different selection strategies: fitness-proportionate (FIT), GENITOR-style rank-based with bias 2 (RANK) [20], and tournament selection with tournament size 10 (TOUR).

The result of a trial was a maximally fit timetable found during the trial. From this we record a vector of violations $V = \{c, j, p, l, e, d_{5+}, d_4, t\}$, which respectively denote violations of Options constraints (clashes), *important* juxtaposition constraints, *desirable* juxtaposition constraints, lunchtime constraints, slot-exclusion constraints, cases where a 'virtual student' faced more than four events in a a day, cases where a 'virtual student' faced four events in a single day, and, finally, travel time constraints. Violations of all other constraints mentioned above (eg: room exclusions, capacity constraints) are not recorded, since they were fully satisfied in all cases.

Ten trials were run for each experiment, and results for each problem record the best V found overall, and the mean of V over the ten trials, for each of three selection schemes. We also present V for the timetables produced by the course organisers for each problem, and which were the actual timetables used (or in use), since the EA system itself was not yet in regular use. 'Best' means relative to the penalty-weighted sum of violations. The fixed penalty values used in these trials for the various violations were, in the order in which they appear in the tables: 500, 300, 30, 30, 10, 5, 1, 1. Hence, violations are listed in order of decreasing importance, as judged by the course organisers.

5.1 Results

Problem	c	j	p	l	e	d_{5+}	d_4	t
92/93_term1								
Course Organisers	0	0	4	4	14	0	2	65
FIT / Best of 10	0	0	0	0	0	0	0	8
FIT / Mean of 10	0	0.1	0.2	1.2	0	0	0.9	16.6
RANK / Best of 10	0	0	0	0	0	0	0	17
RANK / Mean of 10	0	0	0	1.6	0	0	0.6	17.2
TOUR / Best of 10	0	0	0	0	0	0	0	0
TOUR / Mean of 10	0	0	0	1	0	0	0.4	0.7
92/93_term2								
Course Organisers	0	1	9	2	0	1	4	49
FIT / Best of 10	0	0	0	0	0	0	1	0
FIT / Mean of 10	0	0	0.1	0.5	0	0	2.1	9.2
RANK / Best of 10	0	0	0	0	0	0	3	9
RANK / Mean of 10	0	0	0.5	1	0	0	2.4	13.4
TOUR / Best of 10	0	0	0	0	0	0	0	0
TOUR / Mean of 10	0	0	0.3	0.6	0	0	0.7	0.1

Table 1: Comparative performance on the 92/93 problems

Tables 1 and 2 clearly show that the EA approach leads to much better timetables in each case. One course-organiser produced timetable failed to keep all reasonable options open (ie: had clashes), while many failed to fully keep lectures and tutorials away from various restricted slots, and all failed constraints at least as important as the need to avoid lunchtime events. Problems caused involve lecturers and tutors being forced to work during timeslots previously designated for other things (eg: regular weekly seminars), forced to give up free afternoons or mornings designated for research, students facing excessively demanding days, and so on.

On the other hand, for each problem, tournament selection found either a perfect timetable or one with a single travel-time violation in at least one of ten trial runs. For each selection scheme, mean and best results compared very favourably with the experts' efforts. Each EA trial was completed within 5 minutes on a sun SPARC. Occasionally, a EA solution, was worse in terms of some attribute (eg: violations of desirable juxtaposition constraints) than the course organisers' solution, but better overall in terms of the penalty-weighted sum of violations. This suggests that the EA was more successful at trading off the relative occurrences of violations of different importance.

Tournament selection appears to be the best choice, with rank-based selection of the style used in [20] and fitness proportionate selection, in that order, being next best. This relative performance of different selection schemes cannot strictly be taken as read from these results without further experiments using different tournament sizes, biases, and so on; however, much EA literature backs up this ordering of relative performance.

Problem	c	j	p	l	e	d_{5+}	d_4	t
93/94_term1								
Course Organisers	0	0	0	5	7	0	2	84
FIT / Best of 10	0	0	0	2	0	0	0	15
FIT / Mean of 10	0	0	2.6	2.3	0	0	0.7	28.6
RANK / Best of 10	0	0	1	2	0	0	0	23
RANK / Mean of 10	0	0	2.2	2.5	0	0.1	0.3	31.5
TOUR / Best of 10	0	0	0	0	0	0	0	1
TOUR / Mean of 10	0	0	0.6	1.4	0	0	0.2	0.6
93/94_term2								
Course Organisers	2	0	3	3	1	0	0	50
FIT / Best of 10	0	0	1	0	0	0	0	11
FIT / Mean of 10	0	0	0.2	1.4	0	0	1.4	14
RANK / Best of 10	0	0	0	1	0	0	1	13
RANK / Mean of 10	0	0	0.6	1.7	0	0	1.3	13.6
TOUR / Best of 10	0	0	0	0	0	0	0	0
TOUR / Mean of 10	0	0	0	0.7	0	0	0.2	0.2

Table 2: Comparative performance on the 93/94 problems

6 Discussion

The results clearly indicate the benefits of using a penalty-function based EA approach on this problem, and by implication suggest similar utility for the same approach on similar problems. In designing the penalty-weighted fitness function itself for these experiments, several of the design decisions were *ad hoc*. For example, penalty values were chosen according to a rough judgement of relative importance. Also, there were several other possibilities, as discussed earlier, for dealing with the event-spread constraints. Even the underlying EA itself was far from optimal in terms of parameter settings and general configuration. Better choices of selection scheme, for example, are spatially-oriented schemes as presented in [5] and [9], while better overall choices for the EA are certainly possible.

The 'rough-and-ready' aspect of the experimental configurations used in this paper, coupled with the good results reported and the ease of implementing the approach strongly suggests a promising future for both further research and also practical use of EAs on general timetabling problems. Naturally, there are a considerable number of theoretical and practical issues that need to be answered. An illustrative collection of these follow:

Scaling Up

How does this approach scale up to larger and more tightly constrained problems? The real problems addressed here, and similarly those addressed in [6, 14, 7], are similar in size or larger than a large proportion of the timetabling problems faced in many institutions. Hence the usefulness of this approach seems justified, inasmuch as we can expect the beneficial results displayed

here to carry over to different timetabling problems of similar or smaller size. How the approach scales with increasing size and/or complexity is a harder question, which is the subject of continuing research. Initial indications in unpublished work are that the basic approach scales well, but suffers from a problem common to EA-based optimisation: that is, solutions near optimal regions are rapidly found on large complex problems, but further evolution towards optima becomes considerably slow, and may stop altogether. Fortunately this difficulty is readily aided by the use of smart hillclimbing mutation operators. As detailed in [8], use of such operators helps to vastly increase the scope of the approach in terms of problem size. Similarly, an alternative chromosome representation used in [16] is also found to significantly improve on solution quality when compared with the basic approach as presented here.

Generalising Across

How does the approach perform on other timetabling problems? The general nature of the approach suggests that it would be just as well employed on many similar problems. A key aspect which matters here is speed of evaluation. As long as the numbers of constraints which need checking coupled with the computational ease of checking them make for a relatively speedy evaluation function, it seems safe to suggest that useful performance is promised. The kinds of individual constraints that usually occur in timetabling problems are computationally quick to check when using the direct chromosome representation. The general prospects for EA-based timetabling in this respect are in any case illustrated by the variety of real problems so far successfully addressed.

Different Approaches

Different representations, use of domain specific recombination operators, and hybridisation of the EA with other techniques are all candidates for refinement of this approach. Much further research in this vein is in order. It is also interesting and important to compare EA approaches with other methods such as branch & bound search, simulated annealing, and so on. This endeavour is complicated by the differences between the techniques themselves. Eg, the promise of the EA-based approach is most strongly manifest in its robustness across a very wide range of different timetabling problems. Comparison with rule-based approaches to test this claim on the same variety of problems would then necessitate the lengthy and difficult development process of building rule-based systems with similarly wide applicability. Comparison with simulated annealing is a more likely prospect, and such is planned in due course.

Practice

Some common needs are not met by the approach as discussed here. Eg: timetablers may wish to generate several distinct timetables to choose from. Such considerations require refinements and extensions, although the basic approach we have discussed remains a useful partial tool for such requirements. More relevantly, space prevents us from properly covering here the general process and the many possible choices involved in interpreting the constraints of a problem into a particular choice of fitness function. It is rarely apparent how best to do this, and further work is required to assess the various possibilities. Experience shows, however, that there is unlikely to be a major difference in performance between different such choices for problems of the size and type addressed here; hence, we feel that natural and/or arbitrary choices may be made with impunity for penalty settings, pre-specifications, event-spread constraint handling, and so on ..., at least for problems of the size and type found in small or medium sized university departments.

Acknowledgements

We would like to thank Bob Fisher and Alan Smaill for their help in describing the lecture timetabling problems used here and providing data. Thanks also to the UK Science and Engineering Research Council for support of Dave Corne via a grant with reference number GR/J44513.

References

[1] D. Abramson and J. Abela, 'A parallel genetic algorithm for solving the school timetabling problem', Technical report, Division of Information Technology, C.S.I.R.O., (April 1991).

[2] Alan M. Barham and John B. Westwood, 'A simple heuristic to facilitate course timetabling', *Journal of the Operational Research Society*, **29**, 1055–1060, (1978).

[3] Mirjana Cangalovic and Jan A.M. Schreuder, 'Exact colouring algorithm for weighted graph applied to timetabling problems with lectures of different lengths', *European Journal of operations research*, **51**, 248–258, (1991).

[4] Michael W. Carter, 'A survey of pratical applications of examination timetabling algorithms', *Operations Research*, **34**(2), 193–202, (March-April 1986).

[5] Robert J. Collins and David R. Jefferson, 'Selection in massively parallel genetic algorithms', in *Proceedings of the Fourth International Conference on Genetic Algorithms*, eds., R.K. Belew and L.B. Booker, pp. 249–256. San Mateo: Morgan Kaufmann, (1991).

[6] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo, 'Genetic algorithms and highly constrained problems: The time-table case', in *Parallel Problem Solving from Nature*, eds., G. Goos and J. Hartmanis, 55–59, Springer-Verlag, (1990).

[7] Dave Corne, Hsiao-Lan Fang, and Chris Mellish, 'Solving the module exam scheduling problem with genetic algorithms', in *Proceedings of the Sixth International Conference in Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, eds., Paul W.H. Chung, Gillian Lovegrove, and Moonis Ali, 370–373, Gordon and Breach Science Publishers, (1993).

[8] Dave Corne, Peter Ross, and Hsiao-Lan Fang, 'Fast practical evaolutionary timetabling', in *Proceedings of the AISB Workshop on Evolutionary Computation*, (1994).

[9] Yural Davidor, 'A naturally occuring niche & species phenomenon: The model and first results', in *Proceedings of the Fourth International Conference on Genetic Algorithms*, eds., R.K. Belew and L.B. Booker, pp. 257–263. San Mateo: Morgan Kaufmann, (1991).

[10] *Handbook of Genetic Algorithms*, ed., L. Davis, New York: Van Nostrand Reinhold, 1991.

[11] K. A. Dowsland, 'A timetabling problem in which clashes are inevitable', *Journal of operations research society*, **41**, 907–918, (1990).

[12] David E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*, Reading: Addison Wesley, 1989.

[13] Sami Khuri, Thomas Bäck, and Jörg Heitkötter, 'An evolutionary approach to combinatorial optimization problems', in *Proceedings of the 1994 Computer Science Conference (CSC94)*, Phoenix, Arizona, (March 1994). ACM Press. to appear.

[14] Si-Eng Ling, 'Intergating genetic algorithms with a prolog assignment problem as a hybrid solution for a polytechnic timetable problem', in *Parallel Problem Solving from Nature, 2*, eds., R. Manner and B. Manderick, 321–329, Elsevier Science Publisher B.V., (1992).

[15] N. K. Mehta, 'The application of a graph coloring method to an examination scheduling problem', *Interfaces*, **11**, 57–64, (1981).

[16] B. Paechter, H. Luchian, A. Cumming, and M. Petruic, 'Two solutions to the general timetable poblem using evolutionary methods', in *Proceedings of the IEEE Conference on Evolutionary Computation*, (1994).

[17] Ben Paechter, 'Optimising a presentation timetable using evolutionary algorithms', in *Proceedings of the AISB Workshop on Evolutionary Computation*, (1994).

[18] Peter Ross, Dave Corne, and Hsiao Lan Fang, 'Timetabling by genetic algorithms: Issues and approaches', Technical Report AIGA-006-94, Department of Artificial Intelligence, University of Edinburgh, (1994). revised version to appear in Applied Intelligence.

[19] Alice E. Smith and David M. Tate, 'Genetic optimisation using a penalty function', in *Proceedings of the Fifth International Conference on Genetic Algorithms*, ed., S. Forrest, pp. 499–503. San Mateo: Morgan Kaufmann, (1993).

[20] Darrell Whitley, 'The GENITOR algorithm and selection pressure', in *Proceedings of the Third International Conference on Genetic Algorithms*, ed., J. D. Schaffer, 116–121, San Mateo: Morgan Kaufmann, (1989).

Fast Practical Evolutionary Timetabling

Dave Corne, Peter Ross, Hsiao-Lan Fang

Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, U.K.

Abstract. We describe the General Examination/Lecture Timetabling Problem (GELTP), which covers a very broad range of real problems faced continually in educational institutions, and we describe how Evolutionary Algorithms (EAs) can be employed to effectively address arbitrary instances of the GELTP. Some benchmark GELTPs are described, including real and randomly generated problems. Results are presented for several of these benchmarks, and several research and implementation issues concerning EAs in timetabling are discussed.

1 Introduction

A number of researchers have applied evolutionary algorithms (EAs) to timetabling problems [2, 1, 7, 3, 8, 9]. Work so far has however tended to be isolated, applying a range of techniques to disconnected problems with little cross-comparison. The intent of this paper is to fully set out the nature of the problems addressed in EA timetabling research, and to present a series of results on some real and randomly generated problems which form part of a benchmark set we are collecting, using the (so far) most successful variant of the 'direct' approach. These benchmarks will hopefully spawn further, focussed work in the EA timetabling arena.

Section 2 describes the general form of the timetabling problem addressed by researchers using EAs and/or other techniques. Section 3 then notes how an EA may be set up to address an arbitrary instance of such a problem. In Sect. 4, we describe various test problems, both real and random, and gives tables of results on these problems. Section 5 then discusses a variety of aspects of the general approach which are worth mentioning, and some summary and conclusions appear in Sect. 6.

2 Evolutionary Timetabling

A large class of timetabling problems can be described as follows. There is a finite set of events $E = \{e_1, e_2, \ldots, e_v\}$ (for example, exams, seminars, project meetings), a finite set of potential start-times for these events $T = \{t_1, t_2, \ldots, t_s\}$, a finite set of places in which the events can occur $P = \{p_1, p_2, \ldots, p_n\}$, and a finite set of *agents* which have some distinguished role to play in particular events (eg: lecturers, tutors, invigilators, ...) $A = \{a_1, a_2, \ldots, a_m\}$. Each event e_i can be regarded as an ordered pair $e_i = (e_i^l, e_i^s)$, where e_i^l is the length of event e_i (eg:

in minutes), and e_i^s is its size (eg: if e_i is an examination, e_i^s might be the number of students attending that examination). Further, each place can be regarded as an ordered pair $p_i = (p_i^e, p_i^s)$, where p_i^e is the *event* capacity of the place (the number of different events that can occur concurrently in this place), and p_i^s is its size (eg: the total number of students it can hold). The event capacity matters: two exams can take place in the same room simultaneously, but two lectures cannot. There is also an $n \times n$ matrix D of travel-times between each pair of places.

An *assignment* is an ordered 4-tuple (a, b, c, d), where $a \in E$, $b \in T$, $c \in P$, and $d \in A$. An assignment has the straightforward general interpretation: "event a starts at time b in place c, and with agent d". If, for example, the problem was one of lecture timetabling then a more natural interpretation would be: "lecture a starts at time b in room c, and is taught by lecturer d".

Given E, T, P, A, and the matrix D, the GELTP involves producing a timetable which meets a large set of constraints C. A timetable is simply a collection of v assignments, one per event. How easy it is to produce a useful timetable in reasonable time depends crucially on the kind of constraints involved. In the rest of this section, we discuss what C may contain.

2.1 GELTP Constraints

Different instances of GELTPs are distinguished by the constraints and objectives involved, which typically make many (or even all) of the s^{vmn} possible timetables poor or unacceptable. Each constraint may be hard (must be satisfied) or soft (should be satisfied if possible). Many conventional timetabling algorithms address this distinction inadequately: if they cannot solve a given problem they relax one or more constraints and restart, thus trying to *solve a different problem*. The kinds of constraints that normally arise can be conveniently classified as follows.

Unary Constraints Unary constraints involve just one event. Examples include: "The science exam must take place on a Tuesday", or "The Plenary talk must be in the main function suite". They naturally fall into two classes:

Exclusions : An event must not take place in a given room, must not start at a given time, or cannot be assigned to a certain agent.

Specifications : An event must take place at a given time, in a given place, or must be assigned to a given agent.

Binary constraints A binary constraint involves restrictions on the assignments to a pair of events. These also fall conveniently into two classes:

Edge Constraints : These are the most common of all, arising because of the simple fact that people cannot be in two places (or doing two different things) at once. A general example is: "event x and event y must not overlap

in time". The term 'edge' arises from a commonly employed abstraction of simple timetabling problems as graph colouring problems [12].

Juxtaposition Constraints : This is a wide class of constraints in which the ordering and/or time gap between two events is restricted in some way. Examples include: "event x must finish at least 30 minutes before event y starts", and "event x and event y must start at the same time".

Edge constraints are of course subsumed by juxtaposition constraints, but we single out the former because of their importance and ubiquity. Edge constraints appear in virtually all timetabling problems, and in some problems they may be the *only* constraints involved.

Capacity Constraints Capacity constraints specify that some function of the given set of events occurring simultaneously at a certain place must not exceed a given maximum. Eg: in lecturing timetabling we must usually specify that a room can hold just one lecture at a time. In exam timetabling this capacity may be higher for many rooms, but in both cases we need also to consider the total student capacity of a room. We may allow up to six examinations at once in a given hall, but only as long as that hall's maximum capacity of 200 candidates is not exceeded.

Event-Spread Constraints Timetablers are usually concerned with the way that events are spread out in time. In exam timetabling, for example, there may be an overall constraint of the form "A candidate should not be expected to sit more than four exams in two days". In lecture timetabling, we may require that multiple lectures on the same topic should be spread out as evenly as possible (using some problem-specific definition of what that means) during the week. Event-spread constraints can turn a timetabling problem from one that can be solved easily by more familiar graph-colouring methods into one which requires general optimisation procedures and for which we can at best hope for a near optimal solution.

Agent Constraints Agent constraints can involve restrictions on the total time assigned for an agent in the timetable, and restrictions and specifications on the events that each individual agent can be involved in. In addition to those already discussed (exclusions and specifications) we typically also need to deal with constraints involving agnet's preferences (for teaching certain courses, for example), and constraints involving teaching loads.

3 Applying EAs to the GELTP

In applying an EA to a problem, central considerations are the choice of a chromosome representation and the design of the fitness function. In this section we describe the approach we have found most successful so far.

3.1 Representation

For the GELTP, a chromosome is a vector of symbols of total length $3v$ (where v is the number of events), divided into contiguous chunks each containing three genes. The three alleles in the ith chunk, where $1 <= i <= v$, represent the time, place, and agent assignments of event i. Naturally, the set of possible alleles at time, place, and agent genes are respectively identified with the sets T, P, and A. The simple example chromosome "abcdef" represents a timetable in which event e_1 starts at time a in place b, involving agent c, and event e_2 starts at time d in place e, involving agent f.

Very many timetables thus represented will involve edge-constraint violations ('clashes'). Eg, the chromosome "abcabcabc....." is well-formed, even though it puts every event in the same place at the same time and involving the same agent. The job of the EA is to gradually remove such violations of constraints during the artificial evolutionary process.

3.2 The Evaluation function

It is important to be able to differentiate the relative quality of different timetables. An apparently satisfactory solution is widely used in the EA literature: it is simply to have fitness inversely proportional to the number of constraints violated in a timetable with each instance of a violated constraint weighted according to how important or not it is to satisfy it.

Let C_j be the set of constraints of type j (for example, event-spread constraints). The specific type classification used can be tailored to suit the problem. Each violated member of C_j attracts a specific penalty w_j. For each $c \in C_j$, let $v(c,t) = 1$ if c is violated in timetable t, and $v(c,t) = 0$ if c is satisfied. A simple fitness function for a GELTP is thus:

$$f(t) = 1/(1 + \sum_{\text{types}j} w_j \sum_{c \in C_j} v(c,t)) \qquad (1)$$

Assuming that all the penalties are positive, this function is 1 if and only if all the constraints hold, otherwise it is less than 1. The general idea is that an appropriate choice of values for the penalty terms should lead both to reasonable tradeoffs between different kinds of constraint violations, and (by virtue of defining the shape of the fitness landscape) effective guidance of the EA towards highly fit feasible timetables.

The approach works best if the constraint set C is fine grained. For example, if C contained only the 'single' constraint: "the timetable has no clashes", then the fitness landscape would contain a few spikes in an otherwise flat space, which is quite intractable to any form of search. C is hence best composed of low order constraints, each of which involves only one or two events. For example, the important constraint "No lectures which share common students should clash" appears in C as a large collection of separate constraints each involving a distinct pair of lectures which (are expected to) share common students or teachers. The

set of such edge constraints are usually the largest single block of constraints in a timetabling problem. Commonly, applications will be faced with data in a form like: "student Jones sits exams Maths, Physics, Chemistry, ..."; this is then easily transformed to collections of binary constraints between events. In this case, each distinct pair of exams taken by the same student constitutes an edge constraint; typically, we may also automatically create an event-spread constraint between the same pair.

Given such a collection, a question arises as to how to treat them: they may either be treated as a uniform collection of distinct edge constraints of identical importance, or each edge may be weighted according to how many students share these events. In this way, and typically throughout this 'problem transformation' process, the constraint satisfaction problem (CSP) of finding a timetable which satisfies all the constraints can be transformed into any of several different constrained optimisation problems (COPs)[4]; each such COP will share at least one global optimum with the CSP (and preferably all of them) but will be otherwise different. In general, it seems important to make the landscape of this COP as meaningful as possible. For example, in an exam timetabling problem in which we treat each distinct edge constraint the same and each distinct event-spread constraint the same (but typically with event-spread constraints being less important edge constraints), we may find that the best we can do is find a collection of answers which violate a single event-spread constraint. These may be markedly different however; although each answer has just one pair of edge-constrained events timetabled too closely, some may involve only one student, while others may involve several. By weighting edge-constrained pairs of events (in this case according to the number of students sharing the given exams), the COP becomes more meaningful in that it is able to distinguish between such cases. This is particularly important in cases where the CSP has no solution, or where its solution is difficult to find; one or other of which is quite common norm in exam and lecture timetabling problems. In such cases, the COPs addressed by the EA (or some other method) may have different global optima, and so it becomes particularly important to use as 'meaningful' a COP as possible. In an example similar to that just discussed, for example, the less meaningful COP may have an optimum which violates just one event spread constraint (which involves 50 students suffering consecutive exams, say), while a global optimum of the more meaningful version may involve just 2 students suffering consecutive exams, although having violations of two distinct event-spread constraints.

3.3 Speed

An important general consideration is that calculation of fitness be fast. Fortunately, this is usually true for most of the constraints we need to consider in the GELTP, which mainly comprise unary and binary constraints. More to the point, however, when using the direct representation it is particularly easy to set up 'delta evaluation', whereby to evaluate a timetable we need only consider the changes between it and an already-evaluated reasonably similar one such as one of its parents.

[9] discusses the use of delta evaluation further, noting that it is slightly more than just an obvious speedup measure. In particular, [9] notes that speed comparisons made in terms of 'number of evaluations', as commonly done, may often be overturned when delta evaluation is in use. That is, EA configuration X might regularly find results in fewer evaluations than EA configuration Y (and hence be faster when *full evaluation* is in use, but Y may be found to be faster than X when delta evaluation is employed. The reason for this is just that evaluations when using X in conjunction with delta evaluation generally take longer than with Y, because, for example, X employs a highly diversifying recombination operator which means that delta evaluation has to consider several changes each time (though will typically still be faster than full evaluation). It is important to note here that delta evaluation leads to the notion of *evaluation equivalents* EEs, which we employ later on. This simply records the time taken for a run in terms of the number of *full evaluations* that would have been done in the same time. We measure this, for example, by dividing the total number of constraint checks made during a delta evaluation run by the (constant) number that would be made during a full evaluation. This measure hence allows a machine independent measure of the time taken by an EA/timetabling run employing delta evaluation. An alternative is simply to record the total number of constraint checks made, but EE's provide a more accessible measure and give a more reasonable indication of total time taken.

3.4 Penalty Settings

Clearly, we can choose penalty terms for different constraints according to our particular idea of how we would trade off the advantages and disadvantages of different solutions to the problem in hand. Penalties must be set with care, however. If the ratio between two penalties (say, ordering constraints *vs* event-spread constraints) is too high, then search will quickly concentrate on a region of the space low in violations of the more penalised constraint, but perhaps missing a less dense region in which better tradeoffs could be found. If too low, then the capacity for the search to trade off between different objectives is lost. Evidently, optimal penalty settings depend on many things, primarily involving the density of regions of the fitness landscape in relation to each constraint, as well as the subjective relative disadvantages of different constraint violations in the problem at hand. Alternative possibilities include the approach in [10] (in the context of multiple objective facility layout problems), in which penalty settings are revised dynamically in accordance with the gradually discovered nature of the constrained fitness landscape. Also, a principled method for constructing scalar functions for multi-objective problems is discussed in the context of EA optimisation in [6]. In this method, called MAUA (Multi-Attribute Utility Analysis), extensive questioning of an expert decision maker (in our case, an experienced timetable constructor) on example cases (pairs of distinct timetables) would lead to the construction of a nonlinear function M of the vector of summed constraint violations, designed so as to best match the judgement of the expert in that the ordering on timetables imposed by M optimally matches

that imposed by the expert. The effort involved in performing MAUA, however, is unlikely to be rewarded with a function M which is significantly closer to the expert decision-maker's judgement than an essentially *ad-hoc* but intuitive linear weighted penalty function and it is not clear that this is desirable. MAUA involves no attempt to structure M such that the fitness landscape is more helpful to the search process.

It suffices to say here that extensive experience so far suggests that for a wide range of problems we can settle for a simple linear penalty-weighted sum of violations with an intuitive choice of penalty settings.

3.5 Violation Directed Mutation Operators

In [9], a family of Violation Directed Mutation VDM operators for timetabling problems are examined, and it is found that a certain subclass of variations on VDM are particularly powerful for use on a range of realistic problems. Similar operators are studied in [4] for graph colouring and other constraint satisfaction problems. Here we adopt the use of one particular VDM variant, called (`rand, tn10`), as standard. The action of this operator is roughly as follows (fuller description appears in [9]): an application of (`rand, tn10`) to a timetable amounts to randomly choosing an event (gene), and then selecting a new allele (timeslot) for it via tournament selection with a tournament size of 10; an allele's 'fitness' for this purpose is a measure of the degree to which it will reduce the degree of constraint violation involving the chosen event. This 'allele choice' operation involves some computational expense, but since the substantial part of it involves numbers of constraint checks, the time it takes can be meaningfully absorbed into our EE measure.

4 Some Benchmark Timetabling Problems

We first look at five real examination timetabling problems, and later consider 32 randomly generated highly over-constrained test problems. These are not fully general, in the sense of having a full repertoire of place and agent constraints as well as several kinds of timeslot constraint, but only consider edge, event-spread, exclusion, and, in the case of two of the real problems, timeslot capacity constraints (arising from a limit on the number of seats available in examination halls at any one time). Realistic fully general benchmarks will appear anon, but for now it seems reasonable to provide more simply defined problems (and hence more accessible for comparative performance research) which are nevertheless realistically difficult and/or common GELTP variants..

4.1 Some Real Examination Timetabling Problems

Three of these arise from MSc examinations at the EDAI[1], and two from Kingston University, London. Each of the EDAI problems, named in turn: `edai-ett-91`,

[1] University of Edinburgh Department of Artificial Intelligence

edai-ett-92, and edai-ett-93, involve a four slots per day timetable structure, and involve a number of edge constraints and exclusions. In each case, there is an event-spread constraint as follows: if any pair of edge-constrained events are timetabled to appear on the same day, then they must have at least one full slot between them. That is, they must occupy the first and third, first and fourth, or second and fourth slots. edai-ett-91 has 314 edge constraints, and no exclusions, and must be timetabled over six days (hence 24 slots). edai-ett-92 has 431 edges, no exclusions, and must occupy seven days, while edai-ett-93 has 414 edges, 480 exclusions, and must occupy nine days.

The Kingston University problems respectively represent the first and second semester exams faced by Kingston University students in 1994. In the first semester problem, ku-ett-94-1, 97 exams have to be arranged over 5 days with with 3 slots per day. There are 399 edge constraints, and hence 399 individual event spread constraints. The event-spread constraint in this case is that if an edge constrained pair of exams both occur on the same day, they must occupy the first and third slot. ku-ett-94-2 is the second semester problem; 128 exams must be arranged over an 8 day period, with 3 slots per day. The event spread objective this time is to avoid a student facing more than one exam per day. Hence, edge-constrained pairs of events should not occupy slots on the same day. There are 536 edge constraints, and hence 536 event spread constraints too. An additional constraint faced by both of the Kingston University problems is that a maximum of just 470 candidates can be seated in any timeslot. Hence, associated with each event is a weight representing the number of students taking the appropriate exam. These weights are then used by the fitness function (as well as by the VDM operator) to penalise (avoid) violations of this capacity constraint.

4.2 Default EA Configuration

The EA configuration used in all experiments was as follows. A reproduction cycle consisted of a breeding step (in which one new chromosome was produced) followed by an insertion step, in which this new chromosome replaced the currently least fit individual (but only if the new individual was fitter). With probability 0.2, a breeding step involved the selection of one parent and the simple gene-wise mutation of it with a probability of 0.02 of randomly reassigning the allele of each gene in turn. With probability 0.8, a breeding step involved the selection of one parent, and the application of the VDM operator (rand, tn10. Tournament selection was used with a tournament size of 6, and the population size was always 1,000.

Using the default configuration, we examine the reliability of this EA on five real timetabling problems. The default configuration was applied in 100 separate trials to each of the five problems detailed above. We record, in each case, the number of such trials which found an optimum (the 'number of perfect trials' column; on each of these problems, optima violating no constraints exist), the least, average, and most evaluations taken to reach an optimum for those trials which did, and the the least, average, and most evaluation equivalents taken to

reach an optimum for those trials which did. These results appear in Table 1. Each trial on an **edai** problem was run for 25,000 evaluations, while trials on the **ku** problems ran for 40,000 evaluations each.

Table 1. Performance on five real lecture timetabling problems

	No. Perfect	Evaluations			Eval. Equiv's		
Problem	Trials	Lowest	Mean	Highest	Lowest	Mean	Highest
edai-ett-91	84	4595	7693	24933	4793	8084	26241
edai-ett-92	50	7701	14087	21703	5000	13771	21179
edai-ett-93	98	6611	9592	14818	4717	8501	13166
ku-ett-94-1	93	10273	14890	20243	4227	6181	8420
ku-ett-94-2	57	12594	19241	28808	3731	6109	9103

It may first be pointed out that these results show great general potential for EAs on timetabling problems. All trials are relatively fast; for example. Speed on the problems above ranged 250 to 400 evaluations per second[2] Even in the two cases where the EA found an optimum only 50% or so of the time, this means that a small number of trials would be more or less guaranteed to stumble on a perfect timetable soon enough. In the case of the **edai-ett-91** and **edai-ett-92**, the timetables actually used for these problems were independently produced by the relevant course administrators. As detailed in [3], these were very poor in relation to average EA-produced timetables on the same problems, and certainly far from the perfect timetables that the EA used here can typically find. From the **edai-ett-93** case on, the course administrators have been (thankfully) using an EA to do their timetabling work, and hence we do not have independently produced efforts for comparison. In the case of the **ku** problems, the course administrators at Kingston University tried to manually produce timetables for these problems but were having great difficulty, owing in particular to the recent modularisation of the underlying course; they also note that satisfying the capacity constraint in each case was particularly troublesome. Kingston University's timetablers notified us of their problems halfway through their troubled attempts, and sent us the relevant data, whereupon the EA managed to find perfect results regularly in each case.

The main intent in presenting these results is to provide benchmarks for future comparisons with other techniques. As detailed at the end of the paper, all problems we use are freely available. Comparative performance results on the above problems will focus on the *reliability* figure. That is, in looking for improvements on our timetabling EA, we are looking for a single method which will improve reliability in finding the optimum over all of the above set (and others). without any major speed sacrifice. Alternative valid targets include achieving

[2] On a Sun Sparcstation 2, using a not-necessarily optimised C program.

similar reliability faster, or perhaps less (but >50%, say) reliability but *much* faster.

4.3 Some Random Over-Constrained Timetabling Problems

A further kind of target, as is commonly the case with benchmark job shop scheduling problems, for example [11], is find increasingly lower bounds on the total penalty values for a suite of problems. Here we describe some timetabling problems involving edge and event-spread constraints which are constrained enough for this purpose, and present our best results so far using the EA described, and running to a limit of 200,000 evaluations in each of 10 trials for each problem. Each of these problems involves the same temporal structure and event-spread constraint as the `edai-ett` problems; that is, there are four timeslots per day, and for each edge constrained pair of events there is also an edge constraint which specifies that there should be at least one entire slot between them these events if they appear on the same day.

Each problem is named `bench-ett(X,Y,Z,W)`, where:

- X is simply an identifier of the set of random edge constraints involved.
- Y is the number of events to be timetabled.
- Z is the number of days allowed; hence there will be $4 \times Z$ slots altogether.
- W is the number of edge constraints. Each edge constraint is a pair of events, (e_1, e_2), constrained not to appear in the same timeslot. Associated with each constraint is the `edai-ett` style event-spread constraint, Further, associated with each edge constraint c in the set of edge constraints C is a weight c_w. This is an integer between 1 and 100 inclusive.

In each case, the EA applied a penalty P to a candidate timetable t as follows, in which $v(c,t)$ for $c \in C$ is 1 (0) if edge constraint c is violated (satisfied) in timetable t, while $e(c,t)$ for is 1 only if the event-spread constraint is violated but the corresponding edge constraint is *not* violated.

$$P(t) = \Sigma_{c \in C} 2v(c,t)c_w + e(c,t)c_w \tag{2}$$

Figures in the second column of Table 2 are hence the minimal value for P we have so far found on these problems. The third and fourth columns respectively give the weighted penalty for edge constraints violated by the best timetable found, and the weighted penalty for event-spread constraints violated in the same timetable.

The problems in Table 2 are extremely highly constrained; almost certainly more so than any real timetabling problem. Nevertheless, comparative performance of techniques (EA-based or not) on this set (and similar sets) of problems will be useful because these are indeed timetabling-style problems, similar in the nature of the constraints involved than a very common class of real exam timetabling problems. Hence, better performance on these benchmarks will almost certainly reflect potential for better performance on real timetabling problems. Also, since these are very highly constrained COPs, it seems likely that

Table 2. Performance on Very Highly Constrained Random Timetabling Problems

Problem	Smallest Penalty	Edge Penalty	Event-Spread Penalty
bench-ett(1,50,5,1000)	1116	212	692
bench-ett(1,50,6,1000)	613	81	451
bench-ett(1,50,7,1000)	248	17	214
bench-ett(1,50,8,1000)	79	2	75
bench-ett(2,50,5,1000)	1011	174	663
bench-ett(2,50,6,1000)	477	57	363
bench-ett(2,50,7,1000)	151	15	121
bench-ett(2,50,8,1000)	84	7	70
bench-ett(3,50,5,1000)	1129	180	769
bench-ett(3,50,6,1000)	550	65	420
bench-ett(3,50,7,1000)	205	24	157
bench-ett(3,50,8,1000)	61	3	55
bench-ett(1,100,10,4000)	1359	159	1041
bench-ett(1,100,11,4000)	825	125	575
bench-ett(1,100,12,4000)	479	49	381
bench-ett(1,100,13,4000)	312	29	254
bench-ett(2,100,10,4000)	1492	265	962
bench-ett(2,100,11,4000)	1025	158	709
bench-ett(2,100,12,4000)	620	60	500
bench-ett(2,100,13,4000)	346	44	258
bench-ett(3,100,10,4000)	1493	331	831
bench-ett(3,100,11,4000)	874	170	534
bench-ett(3,100,12,4000)	400	46	308
bench-ett(3,100,13,4000)	232	35	162
bench-ett(1,500,1,5000)	122502	30619	61264
bench-ett(1,500,2,5000)	23576	3948	15680
bench-ett(1,500,3,5000)	4580	411	3758
bench-ett(1,500,4,5000)	453	48	357

there is great scope for *continual* improvement on the 'best so far' figures given above. That is, techniques better than the EA we have used so far may be found which achieve 100% reliability on the five real timetabling problems discussed earlier, and hence be indistinguishable on those problems. However, such techniques are likely to be separated in terms of their comparative performance on these highly constrained benchmarks, while still leaving room for further improvement. Lastly, being COPs with (almost certainly) no perfect solutions possible in each case, these problems reflect the difficulties involved in many real modular lecture timetabling problems; in such a problem, for example, a typical approach may be to attempt to allow *any* combination of courses as feasible in the timetable, to give students greater flexibility in their module choices. Since this goal is almost always infeasible, such a problem becomes a COP, with weightings on pairs of module choices reflecting the desirability of allowing these

as viable combinations for a student.

5 Prospects for EAs in Timetabling

Indications so far point to the approach we have described as highly promising for general timetabling needs, at least of the kind usually found in educational institutions. However, several matters need pointing out with respect to its more general application. Firstly, as may be discerned, and as we have found, it is not always immediately apparent how best to describe the problem itself in terms of the kinds of constraints processed. Eg, to incorporate the overall event-spread constraint "no student should sit more than four exams in two successive days", one choice would be to have the objective function (partially) calculate each student's individual timetable, and directly penalise every instance of a student suffering the appropriate constraint violation. A potentially far less computationally expensive choice, however, might be simply to penalise all consecutive edge-constrained events which are less than, say, 2 hours apart in the overall timetable; though not addressing the constraint directly, this certainly applies the artificial evolutionary pressure in the right direction. These are just two of several possibilities for addressing this constraint. Generally, choices occur at many stages in the process of translating the constraints of a real GELTP into a penalty function, and much work is needed to discern the best way to do this. The difficulty may not be too great, however. At least *some* way of handling any given constraint will be naturally apparent, and, in our experience, it is unlikely that different choices will bring significantly different results, unless the problem instance is very large and hence solution speed is a major factor.

Another important aspect is comparative performance with other techniques. There is reason to suggest that the EA approach described will succeed more as a *general timetabling tool* than some other methods. In many cases, however, some other technique may be significantly preferable. Eg, a large, continually faced problem which changes little (in terms of the kinds of constraints involved) from instance to instance may be better off treated with some specifically designed algorithm. This may itself be based on an EA, involve an alternative EA-based approach, or be based on best-first search with a specifically designed heuristic, for example. In general, much work is required to discern what promises to be the best method for different kinds of GELTP.

Also, several aspects of the approach itself warrant much further study. Among these are the design of the objective function itself, as briefly discussed above and also in Sect. 3.4, the use of alternative operators, and various aspects of the underlying EA configuration.

Finally, timetablers (ie; human course organisers, for example) often have needs which are not directly met within the described approach. There may be some desire to generate several different possible timetables, for example, or there may be need for extensive preproccssing and altering of the constraints (which may initially contain several inconsistencies). Use of the EA described (via iterative applications, for example) may be useful as a tool in each case,

but such considerations evidently require more useful refinements or extensions, if not other methods entirely. A planned extension to a future version of the EA described here, for example, is to have an ATMS-based constraint-checking front end. Also, experiments are under way which involve the generation of multiple distinct solutions in a single run, making use of EA variants specifically designed for multimodal optimisation.

6 Summary and Conclusions

We have described and noted various matters in the application of EAs to general educational-institution based timetabling problems. EAs seem to have great potential in this arena, and we illustrate this via presentation of various results. First, we show that a particular EA described here (but more fully in [9]), finds, quickly and reliably, perfect timetables for each of five real examination timetabling problems. The particular reliability results are offered as benchmarks against which to examine alternative techniques. Second, a collection of very highly constrained random timetabling-style problems are described, and our best results on these are given. Various research issues and considerations are then noted.

Finally, the test problems addressed may be freely obtained (and explained) from the authors, and/or via the FTP site ftp.dai.ed.ac.uk .

Acknowledgements

Thanks to Bob Fisher and Peter Sutcliffe for providing problem data. Thanks also to the UK Science and Engineering Research Council for support of Dave Corne via a grant with reference GR/J44513, and to the China Steel Corporation, Taiwan, R.O.C., for support of Hsiao-Lan Fang.

References

1. Abramson D., Abela, J. : A Parallel Genetic Algorithm for Solving the School Timetabling Problem. IJCAI workshop on Parallel Processing in AI, Sydney, August 1991
2. Colorni, A., Dorigo, M., Maniezzo, V.: Genetic Algorithms and Highly Constrained Problems: The Time-Table Case. Parallel Problem Solving from Nature I, Goos and Hartmanis (eds.) Springer-Verlag, 1990, pages 55–59
3. Corne, D., Fang H-L., Mellish, C.: Solving the Module Exam Scheduling Problem with Genetic Algorithms. Proceedings of the Sixth International Conference in Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Chung, Lovegrove and Ali (eds.), 1993, pages 370–373.
4. Eiben, A.E., Raue, P.E., Ruttkay, Z Heuristic Genetic Algorithms for Constrained Problems. Working papers of the Dutch AI Conference, 1993, Twente, pages 341–353.

5. Corne, D., Ross, P., and Fang, H-L.: Fast Practical Evolutionary Timetabling. Proceedings of the AISB Workshop on Evolutionary Computation, Springer Verlag, 1994, to appear.
6. Horn, J., and Nafpliotis, N.: Multiobjective Optimisation Using The Niched Pareto Genetic Algorithm. Illinois Genetic Algorithms Laboratory (IlliGAL) Technical Report No. 93005, July 1993.
7. Ling, S-E.: Intergating Genetic Algorithms with a Prolog Assignment Problem as a Hybrid Solution for a Polytechnic Timetable Problem. Parallel Problem Solving from Nature 2, Elsevier Science Publisher B.V., Manner and Manderick (eds.), 1992, pages 321–329.
8. Paechter, B. Optimising a Presentation Timetable Using Evolutionary Algorithms. Proceedings of the AISB Workshop on Evolutionary Computation, Springer Verlag, 1994, to appear.
9. Ross, P., Corne, D., and Fang, H-L.: Improving Evolutionary Timetabling with Delta Evaluation and Directed Mutation.: Proceedings of PPSN III, Jerusalem, October 1994, Springer Verlag, to appear.
10. Smith, A.E., and Tate, D. M.: Genetic Optimisation Using a Penalty Function. Proceedings of the Fifth International Conference on Genetic Algorithms, San Mateo: Morgan Kauffman, S. Forrest (ed), 1993, pages 499–503.
11. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of operations research, Volume 64, 1993, pages 278–285.
12. Wilson, R. J.: Introduction to Graph Theory. Longman, London, 1979.

This article was processed using the LaTeX macro package with LLNCS style

Improving Evolutionary Timetabling with Delta Evaluation and Directed Mutation

Peter Ross, Dave Corne, Hsiao-Lan Fang

Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, U.K.

Abstract. Researchers are turning more and more to evolutionary algorithms (EAs) as a flexible and effective technique for addressing timetabling problems in their institutions. We present a class of specialised mutation operators for use in conjunction with the commonly employed penalty-function based EA approach to timetabling which shows significant improvement in performance over a range of real and realistic problems. We also discuss the use of *delta evaluation*, an obvious and recommended technique which speeds up the implementation of the approach, and leads to a more pertinent measure of speed than the commonly used 'number of evaluations'. A suite of realistically difficult benchmark timetabling problems is described and made available for use in comparative research.

1 Introduction

Recent research suggests that Evolutionary Algorithms (EAs) are a viable and effective method for addressing timetabling problems [2, 1, 7, 3, 5, 8]. Following the general success shown in these endeavours, the general importance and ubiquity of timetabling problems now warrants systematic attempts to assess the general success of applying EAs to them. That is, there is a need for some standard problem definitions, benchmark problems, benchmark results, and standardised techniques for reference. Towards this end, we discuss a common penalty-function based approach to timetabling, outline a working definition for general timetabling problems, and presents several benchmark problems. In particular some domain-specific operators are described, generally applicable to all timetabling problems, and compared empirically with standard operators on a range of realistic problems.

Our timetabling experiments employ the commonly used (but often *not* used) implementation technique we call ***delta evaluation***, whereby the evaluation of a new individual is speeded up by making use of previously evaluated similar individuals. We discuss this for two main reasons: (a) to emphasise and encourage its use as a way of significantly speeding up EA/timetabling applications (especially those involving the direct representation discussed later), and (b) given the use of delta evaluation as standard in this endeavour, it makes sense to record 'machine-independent' indicators of speed in terms other than simply 'number of evaluations', since the speed of an evaluation will vary greatly throughout the process. We hence describe the idea of 'evaluation equivalents' as a useful comparative measure.

We begin in Sect. 2 with a brief description of a common kind of timetabling problem. Section 3 then reviews the basic penalty-function approach, and introduces several new candidate genetic operators, and discusses the use of delta evaluation. Section 4 describes a family of benchmark timetabling problems we are developing, attending mainly to those we use later in experiments. Section 5 describes and records a collection of experiments comparing the performance of the operators described earlier on some of these benchmarks. Finally, Sect. 6 provides some discussion and summary of the paper.

2 Basic Timetabling problems

Timetabling problems involve a set of events $E = \{e_1, e_2, \ldots e_v\}$, and a set of times $T = \{t_1, t_2, \ldots, t_s\}$. Timetablers often also need to take into account a set of places $P = \{p_1, p_2, \ldots, p_m\}$, and/or a set of agents $A = \{a_1, a_2, \ldots, a_n\}$. An *assignment* is a 4-tuple (e, t, p, a) such that $e \in E$, $t \in T$, $p \in P$, and $a \in A$, with the simple interpretation: "event e occurs at time t in place p involving agent a". In a real case, this may for example mean: "The AI lecture starts at 9:00 in room LT-5, given by Minsky".

A timetable is simply a collection of assignments, one per event. The problem is to find a timetable that satisfies, or minimally violates, a (usually large) collection of constraints. The most common such constraint is simply that there should be no clashes; that is, a person should not have to be in two places at once. Considering the relationship with graph colouring problems, we call this an 'edge constraint'. A related and important constraint is what we call an 'event-spread' constraint, which expresses that a person should normally have at least a certain amount of time free between certain of the events in which he or she is involved. There are several other kinds of constraints. The most prominent among them are illustrated in Table 1. This table defines common kinds of constraints in terms of inequalities over assignments (or sets of them). In the table, $t(e)$ refers to the time assignment of event e; an ordering is assumed over the set of times T; e and f are events where, without loss of generality, we assume that $t(e) < t(f)$; and $l(e)$ stands for the duration of event e. Note that from now on in this paper, and for space reasons in Table 1, we will look at problems involving only time assignments, ignoring for now any constraints involving places or agents.

This is a convenience for present purposes, rather than any gross simplification. Place and agent constraints are often easy to handle in exam timetabling problems, for example, and such assignments can be made manually after the harder job of producing the events/times timetable. Also, constraints involving rooms and agents can often be dealt with implicitly via the constraints detailed in Table 1 alone. In many cases, of course, room and agent constraints do make the problem significantly harder; we begin to address this type of problem in [6], for example.

Table 1. Kinds of Timetabling Constraint

Constraint Type	Name	Assignment version
No clash between e and f	Edge	$t(e) + l(e) \leq t(f)$
Spare time s between e and f	Event-Spread	$t(e) + l(e) + s \leq t(f)$
Excluded times S for e	Exclusion	$NOT(t(e) \in S)$
Specified time u for e	Specification	$t(e) = u$
e must be before/after/same-time f	Juxtaposition	$t(e)(<\|>\|=)t(f)$

3 Timetabling with EAs

The most commonly employed EA approach is the use of a direct representation coupled with a penalty-allocating fitness function. A timetable is represented by a chromosome in which the allele of the ith gene directly represents the time assigned to the ith event; extensions dealing with room and/or agent assignments may be readily imagined. Fitness is simply a weighted sum of constraint violations; ie: high penalties accrue from violations of important constraints, while low penalties are given for soft or unimportant constraint violations. The EA then attempts to mimimise this sum.

It is wise to use *delta evaluation*, in which the computation of a chromosome's fitness is simplified as follows. Consider two timetables, g and h, which differ only in the assignments made to some subset D of the events E. Let $P(t) = \sum_{c \in C} w_c v(c,t)$ be the total penalty accruing to timetable t, where w_c is the weight associated with constraint c and $v(c,t)$ is 1 (0) if constraint c is (not) violated in t. C is the set of constraints. Now, if C_D is the subset of C containing only those constraints which involve one or more events from the subset D of E, then we can easily deduce:

$$P(h) = P(g) - \sum_{c \in C_D} w_c v(c,g) + \sum_{c \in C_D} w_c v(c,h) \qquad (1)$$

which expresses P(h) solely in terms of constraints involving the events in D. If the size of C_D is small in relation to C, then this promises to save much time. In general, if h differs from g in k places, then evaluating g needs $2k$ 'constraint-checks'.

We can use of the concept of a 'constraint-check' to form a convenient measure of the speed of an EA/timetabling run. Number of evaluations is no longer a useful measure of this, because the time of an evaluation will vary quite significantly. By summing the total number of constraint-checks, we can thus form more useful cross-comparisons. Further, by noting the constant number of constraint checks that would be needed in a *full* evaluation, we can convert a constraint-checks sum into *evaluation equivalents* (EEs). This gives both a pertinent measure for comparing speed of different configurations, and also enables us to easily see the speedup effect of using delta evaluation.

Applying delta evaluation of timetable h in an EA needs a decision as to which already-evaluated timetable to take as g. A natural choice, which we use, choose g arbitrarily from the parents of h (or just use the single parent, if h was produced via a single parent operator).

3.1 Violation-Directed Operators

Consider the process involved in working out the penalty score for a timetable t; the major part of the process involves checking each constraint in turn. While doing this, extra steps may easily be incorporated which help keep track of the events whose assignments seem to be causing the most difficulty. This might be done as follows. Keep a separate 'violation score' v_i for each event e_i; initially (before evaluation) set $v_i = 0$ for each i. Then, for each constraint c, if $v(c,t) = 1$ (and hence the constraint is violated), add w_c to the violation scores of each event involved in c. At the end of this process, the set of violation scores can be used to infer which events are most problematic in t, and hence inform as to where in the chromosome it would probably be best to direct mutation. We can put this information to use in a variety of ways. the way we examine in this paper is what we call *violation directed mutation* (VDM).

There are two key aspects to VDM; the choice of a gene to mutate, and the choice of allele to mutate it to. Three possibilities for the former are: (a) simply choose the 'best' candidate for mutation, ie: randomly choose one of a set of genes with maximal violation score, or (b) stochastically select one, biased towards genes with higher violation scores, or (c) simply choose one at random (in which case the 'directedness' of the mutation will arise through the later choice of allele).

For allele choice, it is simple to conceive of a directly similar set of possibilities; this needs, though, some analogue of 'violation score' for alleles. This can be done as follows: if the gene chosen for mutation is g, then a violation score for the candidate new allele a is calculated by simply amassing the penalty-weighted sum of violations of constraints involving g which would occur if it was given allele a.

In both cases, the 'stochastic selection' aspect can be done in various ways; in fact we can use almost any standard EA selection scheme. We choose to use straightforward tournament selection for this. A good reason for this choice follows from the added computational complexity associated with the allele-choice operation; rank-based or fitness-based selection, for example, would require us to calculate, every time, a violation score for each possible new allele, which means checking s times through the list of all constraints involved with g (where s is the number of alleles). Using tournament selection with a tournament size of k, however, this only need be done k times at each application of the VDM operator. Notice that we can absorb the extra time complexity of VDM, which only arises significantly when the allele choice component is other than `rand`, into our 'evaluation equivalents' notion by simply recording the number of constraint checks made during the allele choice operation.

We shall refer to variants of the VDM operator as ordered pairs (**g**,**a**), where **g** stands for the gene choice operation, and **a** stands for the allele choice operation. Instances we look at later will involve **rand** (random choice), **tnK** (tournament selection with a given tournament size of K), and **best** (choose a gene (allele) with a maximal (minimal) violation score).

Finally, we note that very similar operators are described by Eiben *et al*, among others, in [4] for use on graph-colouring and other benchmark constraint satisfaction problems. The differences are that Eiben also considers variants of the operator in which either one, two, or a random number of genes are mutated each time, whereas we only look at single gene mutations here, and Eiben *et al* do not look at variations of bias in the stochastic choice component, and do not look at a **best** component.

4 Benchmark Timetabling problems

A fully general timetabling problem involves, as we have suggested time, place, and agent assignments along with many and varied kinds of constraints involving, for example, room capacities and teaching loads, as well as those already mentioned. Benchmark problems of this sort are in preparation, but for convenience it makes sense to also investigate a simpler variant of the general problem which is, nevertheless, both realistic and common. Here we will describe one template for such a problem, based on the recurring EDAI[1] multi-departmental modular MSc examination timetabling problem (**edai-ett**).

The **edai-ett** involves v events, all of the same duration, and s timeslots spread out evenly over d days. There are four timeslots per day. The problem involves edge constraints, event-spread constraints, and exclusions. The overall event-spread objective is that, whenever a pair of events is edge constrained, these events should also (as well as being in different slots) either be on different days, or have at least one full slot between them. Our **edai-ett** benchmark set contains the four years' versions of the real problem, coupled with an arbitrarily large collection of randomly generated solvable problems of the same type produced via a problem generator.

Randomly generated problems of the **edai-ett** type are based on the construction of a random complete timetable $\mathcal{T}$ of 50 events within a four-slots-per-day, nine-day timetable structure. A set of edge constraints (each with an accompanying event-spread constraint as above) is then generated which render $\mathcal{T}$'s particular partition of events into days, coupled with certain details of its partition of a day's events into slots, as unique in satisfying these edge constraints. A set of exclusion constraints is also generated, which by themselves make $\mathcal{T}$ the only solution. A problem itself is then constructed by filtering these edges and exclusions. For example **edai-ett-rand(3,40,10)** is a problem involving version 3 of $\mathcal{T}$, and containing 40% of the generated edges and 10% of the exclusions.

[1] University of Edinburgh Department of Artificial Intelligence

The benchmark set we are developing is much along the lines discussed; for each real timetabling problem it contains, there is also a generator for random variants of problems of the same type. Here we will experiment with a small but useful sample of these problems.

5 Experiments

Two general questions guided the following set of experiments; a) What is the most successful variant of violation-directed mutation on timetabling problems?, and b) How can we generally expect performance to vary as we alter the difficulty of the problem? These are both extremely difficult questions to investigate, owing to the multiplicity of different variations on timetabling problems that can be imagined, to the many distinct (and interacting) ways in which we could alter the so-called 'difficulty' of such a problem, to the many variations possible on violation directed mutation, and also to inevitable restrictions on computational resources. The approach we take is to simplify the questions, and hence the investigation, without, we believe, sacrificing the usefulness of our results. Our investigation thus concentrates on these two issues: (a) How do a reasonable set of variants on violation-directed mutation compare on a typical exam timetabling problem?, and (b) How does performance of the best such variant behave as we vary the number of constraints on a similarly structured problem? To address (a), we investigated several variants of violation directed mutation on `edai-ett-93`, and to address (b) we looked at `edai-ett-rand(1,N,15)` for each of N from 20 to 100 in steps of 20.

5.1 EA Configuration

All experiments shared the following common EA configuration: A reproduction cycle consisted of a breeding step (in which one new chromosome was produced) followed by an insertion step, in which this new chromosome replaced the currently least fit (if strictly fitter itself). With probability 0.2, a breeding step involved the selection of one parent and the simple gene-wise mutation of it with a probability of 0.02 of randomly reassigning the allele of each gene in turn. With probability 0.8, a breeding step involved the selection of one parent (or two, when the operator was uniform crossover)and the application of the given operator. Tournament selection was used with a tournament size of 6, and the population size was always 1,000.

5.2 Variations on Directed Mutation

Several variations on violation directed mutation were explored using the real `edai-ett-93` problem as a benchmark. For each variant of the VDM operator. A trial consists of running the EA for 20,000 reproduction cycles, or until a perfect timetable was found. If convergence occurred to a non-perfect timetable during a trial (the entire population represented the same non-perfect timetable), then a

complete reinitialisation of the population occurred at the next reproduction step — this counted as 1 reproduction cycle, but 1,000 evaluations (translated further into the appropriate number of evaluation equivalents). For each variation of the VDM operator, 100 trials were performed with a different random seed each time. Eight results are given for each set of trials: the number of trials in which an optimum was found (which, as a percentage, can be taken as a measure of the reliability of the EA variant on this problem), the mean number of constraints violated by the best timetable over all trials, the lowest, mean, and highest number of evaluations recorded to find an optimum over those trials in which an optimum was found, and the lowest, mean, and highest number of EEs recorded to find an optimum over those trials in which an optimum was found.

Table 2. Performance of VDM Variants on edai-ett-93

	No. Perfect	Mean	Evaluations			Eval. Equiv's		
VDM Variant	Trials	Violations	Lowest	Mean	Highest	Lowest	Mean	Highest
uniform	0	3.8	n/a					
(rand, rand)	3	4.2	18704	19733	19950	1610	1701	1940
(rand, tn3)	80	0.4	10613	13706	17417	3168	4109	5247
(rand, tn6)	92	0.1	7591	10598	15458	3881	5427	7906
(rand, tn9)	98	0.02	6146	8398	11259	4450	6086	8187
(rand, best)	0	29.0	n/a					
(tn3, rand)	5	2.1	17063	17660	18257	1532	1573	1613
(tn3, tn3)	65	1.4	11713	16344	19655	4717	10194	12248
(tn3, tn6)	83	0.3	6752	9160	12090	3603	4796	6190
(tn3, tn9)	93	0.9	5326	8530	19119	4029	6403	15190
(tn3, best)	0	22.4	n/a					
(tn6, rand)	8	2.1	17874	18811	19746	1646	1725	1779
(tn6, tn3)	88	0.20	10463	13930	18644	3175	4335	5687
(tn6, tn6)	82	0.03	6741	9458	13242	3537	4969	7061
(tn6, tn9)	97	0.03	5831	9572	20583	4304	7266	16309
(tn6, best)	0	22.4	n/a					
(tn9, rand)	3	3.1	12120	15756	16044	1237	1419	1554
(tn9, tn3)	77	0.3	10228	14880	18909	3253	4667	6242
(tn9, tn6)	92	0.08	7484	10656	13802	4141	5623	7532
(tn9, tn9)	85	0.19	5359	10684	20771	3875	8186	17121
(tn9, best)	0	23.9	n/a					
(best, rand)	0	20.7	n/a					
(best, tn3)	0	21.5	n/a					
(best, tn6)	0	19.1	n/a					
(best, tn9)	0	13.9	n/a					
(best, best)	0	30.1	n/a					

Clearly enough, it seems that stochastic variations of VDM, in which both the choice of gene to mutate and choice of new allele are biased, stochastic

choices, are superior to basic uniform crossover on this problem. Points to note are how reliability varies with the selection pressure for these choices, and also how evaluation equivalents, rather than evaluations, are clearly a more useful indicator of speed than number of evaluations. To see the latter, note how mean EEs rises as we increase selection pressure (because we are performing more computations in, especially, the 'choice of allele' side of the operator), although mean evaluations falls. For example, (rand, tn6) and (tn3, tn9) are closely comparable in reliability on this problem, and a glance at the mean number of evaluations would seem to indicate that (tn3, tn9) is the better of the two owing to its speed. This would be true if we *had to* use full evaluation; using delta evaluation though, it is clear from the mean EEs figures that (rand, tn6) uses significantly fewer constraint checks to achieve similar reliability, and is hence overall the better of the two on this problem.

One clear point is the different degrees to which gene choice and allele choice matter. Looking solely at variations in performance with allele choice fixed, we find there is little difference between, say, (rand, tn3) and (tn9, tn3). Allele choice is rather more significant. For example, (tn3, tn9) is much more reliable than (tn3, rand). Whenever best appears for either allele choice or gene choice however, performance plummets. Another interesting point is that although biased gene choice is better than rand when the allele choice component is rand, it generally leads to slight degradation in reliability when the allele choice component is stochastic.

5.3 Increasingly Hard edai-ett-rand Problems

Some VDM variants were applied to each of 5 randomly generated solvable edai-ett-like problems of increasing constrainedness. The problems addressed were edai-ett-rand(1,N,15) for N from 20 to 100 in steps of 20. Using the same EA configuration as above (except of course for the operator choice), each entry in Table 3 records the number of optima found over 100 trials, and the mean number of EEs over those trials which found an optimum, for the configuration using the operator defined by the row, and on the problem defined by the column. Taking heed of the results discussed above, we look only at variation in bias in biased, stochastic versions of the allele choice component, keeping gene choice fixed at rand.

Table 3 reveals some clear trends. For the simpler problems, with N = 20, 40, and 60, all configurations are 100% reliable at finding an optimum, however the lower bias versions are better simply because they are faster. Again, this is directly attributable to taking advantage of delta evaluation. Number of evaluations actually reduces with increased bias, but the extra constraint checks done during allele choice more than counteract this. Hence, low-biased VDM seems best on simpler problems. As the problem gets harder, the same remains true in the sense that lower-bias versions are generally faster when they find an optimum, although they are rather less reliable. Overall, on the basis that the differences in speed are rather less significant than differences in reliability at finding an

Table 3. Performance of VDM on edai-ett-rand(1,N,15) for N from 20 to 100

	Reliability/Mean EEs				
Operator	N=20	N=40	N=60	N=80	N=100
(rand, tn3)	100/844	100/1715	100/3240	33/5007	0/—
(rand, tn6)	100/1212	100/2244	100/3780	100/6534	58/8436
(rand, tn9)	100/1642	100/2839	100/4535	100/7641	97/10154
(rand, tn12)	100/2178	100/3519	100/5349	100/8653	99/11464
(rand tn15)	100/2607	100/4160	100/6199	100/9660	100/12714

optimum, general indications are that higher pressure in the allele choice component is the preferred option for timetabling problems of this type; although not too high, since best rapidly decreases in reliability as N increases[2]. The critical region, at which higher bias starts to lose reliability on these problems, has yet to be found.

6 Concluding Discussion

We have shown that realistic timetabling problems can be effectively addressed by an EA whose main operator is VDM, particularly when incorporating a biased, stochastic method for the allele choice component, and either a random or biased stochastic method for the gene choice component. Such an EA is certainly far more effective than a straightforward EA using uniform crossover, achieving, for example 97% reliability on the edai-ett-93 problem in certain configurations, as opposed to the 0% reliability of uniform crossover. In particular, we have found EA configurations to be very useful and effective on a range of other real timetabling problems in addition to edai-ett-93 [5, 6].

Looking at the relative effect of different variants of VDM, it seems that the gene choice component may as well be random, while the real power and effect of the operator lies in the allele choice component. This is quite good news for implementation purposes; the combination of delta evaluation with a non-random gene-choice component at least doubles the memory requirement, since the chromosome must carry its gene violation scores around, and also makes the mechanics of delta evaluation more tricky. Results indicate, however, that it is not really necessary to have a non-random gene choice component in VDM, which alleviates these problems.

Many difficult and interesting questions remain in the EA/timetabling research arena. Here we have concentrated on one style of approach, and subsequent enhancements to its performance. General statements on the performance of EAs on timetabling is however complicated by several important factors: first,

[2] Space restrictions prevent a fuller presentation of results with other configurations etc ..., but such is available from the authors.

alternative styles of EA approach are also worth investigating; for example, those in which the chromosome is *indirect*, representing a timetable via a list of instructions, or a particular ordering of events, for later interpretation by a timetable building procedure. EA/timetabling work along these lines is pursued in [8]. Second, there are many more choices of operator possible, including further variants on mutation (single parent operators) and several alternative recombination operators worth investigating. Some of these are looked at in [5]. Third, there is great variation in the space of possible general timetabling problems. We are currently looking for useful landmarks in this space, but for the time being we take the satisficing route of examining EA performance on real timetabling problems we have encountered, and constructing further test problems based on these.

In the interests of further promoting comparative research on this important kind of problem, we encourage researchers to use the benchmarks we have looked at here, and others, which can all be obtained (and explained) via the authors, or directly from the FTP site ftp.dai.ed.ac.uk.

References

1. Abramson D., Abela, J. : A Parallel Genetic Algorithm for Solving the School Timetabling Problem. IJCAI workshop on Parallel Processing in AI, Sydney, August 1991
2. Colorni, A., Dorigo, M., Maniezzo, V.: Genetic Algorithms and Highly Constrained Problems: The Time-Table Case. Parallel Problem Solving from Nature I, Goos and Hartmanis (eds.) Springer-Verlag, 1990, pages 55–59
3. Corne, D., Fang H-L., Mellish, C.: Solving the Module Exam Scheduling Problem with Genetic Algorithms. Proceedings of the Sixth International Conference in Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Chung, Lovegrove and Ali (eds.), 1993, pages 370–373.
4. Eiben, A.E., Raue, P.E., Ruttkay, Z Heuristic Genetic Algorithms for Constrained Problems. Working papers of the Dutch AI Conference, 1993, Twente, pages 341–353.
5. Corne, D., Ross, P., and Fang, H-L.: Fast Practical Evolutionary Timetabling. Proceedings of the AISB Workshop on Evolutionary Computation, Springer Verlag, 1994, to appear.
6. Ross, P., Corne, D., and Fang, H-L.: Successful Lecture Timetabling with Evolutionary Algorithms. Proceedings of the ECAI 94 Workshop on Applications of Evolutionary Algorithms, 1994, to appear.
7. Ling, S-E.: Intergating Genetic Algorithms with a Prolog Assignment Problem as a Hybrid Solution for a Polytechnic Timetable Problem. Parallel Problem Solving from Nature 2, Elsevier Science Publisher B.V., Manner and Manderick (eds.), 1992, pages 321–329.
8. Paechter, B. Optimising a Presentation Timetable Using Evolutionary Algorithms. Proceedings of the AISB Workshop on Evolutionary Computation, Springer Verlag, 1994, to appear.

This article was processed using the LaTeX macro package with LLNCS style