Metaheuristics for University Course Timetabling

Rhydian Marc Rhys Lewis

A thesis submitted in partial fulfilment of the requirements of
Napier University for the degree of Doctor of Philosophy

August, 2006

Abstract

The work presented in this thesis concerns the problem of timetabling at universities – particularly course-timetabling, and examines the various ways in which metaheuristic techniques might be applied to these sorts of problems. Using a popular benchmark version of a university course timetabling problem, we examine the implications of using a "two-staged" algorithmic approach, whereby in stage-one only the mandatory constraints are considered for satisfaction, with stage-two then being concerned with satisfying the remaining constraints *but without re-breaking any of the mandatory constraints in the process*. Consequently, algorithms for each stage of this approach are proposed and analysed in detail.

For the first stage we examine the applicability of the so-called Grouping Genetic Algorithm (GGA). In our analysis of this algorithm we discover a number of scaling-up issues surrounding the general GGA approach and discuss various reasons as to why this is so. Two separate ways of enhancing general performance are also explored. Secondly, an Iterated Heuristic Search algorithm is also proposed for the same problem, and in experiments it is shown to outperform the GGA in almost all cases. Similar observations to these are also witnessed in a second set of experiments, where the analogous problem of colouring equipartite graphs is also considered.

Two new metaheuristic algorithms are also proposed for the second stage of the two-staged approach: an evolutionary algorithm (with a number of new specialised evolutionary operators), and a simulated annealing-based approach. Detailed analyses of both algorithms are presented and reasons for their relative benefits and drawbacks are discussed.

Finally, suggestions are also made as to how our best performing algorithms might be modified in order to deal with further "real-world" constraints. In our analyses of these modified algorithms, as well as witnessing promising behaviour in some cases, we are also able to highlight some of the limitations of the two-stage approach in certain cases.

Table of Contents

List of Figures

List of Tables

1: Introduction

Timetables are ubiquitous in many areas of daily life such as work, education, transport, and entertainment. Indeed, it is quite difficult to imagine an organized and modern society coping without them. Yet in many real-world cases, particularly where resources (such as people, space, or time) are not in abundance, the problem of constructing workable and attractive timetables can be a very challenging one, even for the experienced timetable designer. However, given that timetables will often have a large effect on the day-to-day lives of the people who use them, timetable construction is certainly a problem that we should try to solve as best we can. Additionally, given that timetables will often need to be updated or completely remade (e.g. school timetables will often be redesigned at the beginning of each academic year, bus timetables will need to be modified to cope with new road layouts and bus stops, etc.), constructing timetables is also a problem that people will have to face on a fairly regular basis.

1.1 Timetabling and Metaheuristics

In this thesis we will be concerning ourselves with the problem of timetabling at universities, and in particular: course timetabling problems. As we will see, the problem of automating timetable construction at universities is often a very challenging one. First, from a computer-science perspective most timetabling problem-formulations belong to the class of computationally NP-complete problems (see the work of Garey and Johnson [62] and also Section 2.1 of this thesis); therefore implying that there is no known deterministic polynomially-bounded algorithm for solving them in general. Second, individual timetabling problems are also often complicated by the idiosyncratic nature of the institution and users concerned. For example, different universities will tend to have their own interpretation of what is a “feasible” and/or “good” timetable, and will therefore also tend to have their own particular set of timetable constraints that they wish to impose on their particular problem. Unfortunately however, it might often be the case that an algorithmic approach that is successful for one particular problem-version may not turn out to be suitable for others.

In computing terms, timetabling problems are often modelled as Combinatorial Optimisation Problems (COPs). The overall objective in a COP is to find an assignment of discrete values to variables (e.g. timeslots for each of the events that needs to be timetabled) so that that the solution is *optimal* according to some criteria. In other words, the problem is to find the *best possible solution* from all possible solutions. The techniques available for solving COPs fall into two main classes: *exact* algorithms and *approximation* algorithms. Exact algorithms are able to prove the optimality of a solution, whereas an approximation algorithm cannot. In the case of timetabling, however, exact algorithms will typically constitute a brute-force style approach, and due to the exponential growth rates of the search spaces for these problems, their application will often only be suitable for very small problem instances. On the other hand, while approximation algorithms do not always produce optimal solutions, they do operate in polynomial time and might be able to produce solutions that are "good enough" for practical purposes. (Obviously, *how often* and *how quickly* an approximation algorithm is able to produce solutions that are "good enough" will usually be some of the criteria used to judge how effective it actually is.)

In the next chapter of this thesis we will conduct a review of the various different works that have proposed using approximation algorithms for tackling timetabling problems, paying close attention to those that have applied metaheuristic-based techniques. Metaheuristic[1] algorithms, which include techniques such as evolutionary algorithms, simulated annealing, tabu search, and ant colony optimisation, are an important class of approximation algorithm that, over the past decade-or-so, have been applied to a variety of different COPs such as timetabling. In essence, they might be regarded as a general-purpose algorithmic framework, applicable to various COPs, with relatively few modifications generally needing to be made for each type of problem. Given this wide-ranging applicability it is arguable that they are therefore quite fitting in many areas of automated timetabling where, as we have noted, the types of constraints that are imposed will often vary from place to place.

In this thesis, we will also present arguments as to why the employment of a two-stage timetabling approach might sometimes be appropriate – particularly with regards to a specific university course timetabling problem-version that we will be studying in detail

[1] The term "metaheuristic" is derived from the Greek prefix "*meta*", meaning in this sense "higher level", and "heuristic", from the Greek "*heuriskein*", meaning "to find".

here. In essence, an algorithm that uses this two-stage approach operates by first attempting to satisfy just the mandatory constraints of the problem and, assuming this task is completed successfully, will then go on to try and satisfy the remaining non-mandatory constraints, *but without re-breaking the mandatory constraints in the process*. Consequently, metaheuristic-based algorithms will be presented for each of these two sub-problems, and detailed analyses will be carried out in all cases.

1.2 Summary of Contributions

As mentioned above, the majority of the scientific investigations in this thesis will be conducted using a well-known benchmark version of the University Course Timetabling Problem (UCTP). A detailed analysis of this particular problem-version will be presented in Chapter 3. The majority of this thesis will then be spent examining algorithms that constitute part of the two-stage approach for this problem. From our studies of these algorithms, the following scientific contributions are made:

- A detailed analysis concerning the suitability of a Grouping Genetic Algorithm (GGA) for the task of satisfying the mandatory constraints of our chosen UCTP is conducted. In this analysis we note that there are, in fact, scaling up issues surrounding the general GGA paradigm and, in particular, we show that it can behave in quite different ways with different sized problem instances.

- In our examinations of the general GGA approach, we introduce a new method for measuring population-diversity and distances between individuals with the GGA representation. We also demonstrate the sometimes negative effects that population-diversity can have on the behaviour of the GGA recombination operator and offer some arguments as to why.

- Two suggestions are made as to how we might improve the performance of the GGA for the UCTP. The first of these involves the use of various specialised, fine-grained fitness functions with the algorithm; the second involves supplementing the algorithm with an additional stochastic search operator. In both cases, the positive and negative effects of these modifications are commented upon.

- A new "Iterated Heuristic Search" (IHS) algorithm is also proposed for satisfying the mandatory constraints of our chosen UCTP version. This algorithm – which, unlike our GGA, does not make use of a population, selection pressure, or the standard GGA recombination operator – is shown to outperform the GGA in almost all cases. These observations serve to highlight many of the arguments that are presented concerning the potential limitations of the GGA in general. Additionally, in order to further back-up these arguments, a second set of experiments is also conducted on a similar combinatorial optimisation problem – graph colouring – where comparable behaviour is observed.

- A new specialised Evolutionary Algorithm (EA) intended for satisfying the non-mandatory constraints of our chosen UCTP version is proposed. We suggest a number of new recombination operators that might be used with this sort of algorithm and investigate whether any of these are able to capture the underlying building-blocks of the problem. In experiments with a set of benchmark problem instances, we show that these recombination operators do not actually seem to improve the search in any great deal, and we offer some arguments as to why this is so.
- A new specialised simulated annealing-based algorithm, also for non-mandatory constraint satisfaction, is proposed for our chosen UCTP version. In experiments we show that by using appropriate neighbourhood operators and sensible parameter settings, this algorithm is able to return competitive results for a number of different benchmark problem instances. We also show how the performance of this algorithm can be improved by treating some of the non-mandatory constraints of this problem differently to others.
- Finally, we propose methods by which the best performing algorithms from both stages of the two-stage timetabling approach can be extended in order to cope with additional mandatory constraints that we choose to impose on our chosen UCTP-version. First, we show how our IHS algorithm might be modified in order to cope with these constraints and investigate the effectiveness of this new algorithm empirically. Second, we also demonstrate some of the potential weaknesses of our techniques by showing how the inclusion of large numbers of extra mandatory constraints can ultimately lead to unsatisfactory levels of performance with our chosen two-stage timetabling approach.

1.3 Thesis Guide

In order to keep this document at a reasonable length, we have written this text based upon the assumption that the reader already has some basic knowledge of timetabling, scheduling, and metaheuristics. A grasp of the fundamentals in complexity theory is also preferable, particularly for earlier chapters. Readers who do not possess these prerequisites are invited to first consult some good texts surrounding these matters. For example:

- The PATAT series (The Practice and Theory of Automated Timetabling) contains many good studies on various different timetabling problems. Details of these publications can be found on the web at http://www.asap.cs.nott.ac.uk/patat/patat-index.shtml. Many individual PATAT papers relevant to this work are also listed in the bibliography at the end of this thesis.
- Good overviews on the basic principles of metaheuristics can be found at the following site: http://en.wikipedia.org/wiki/Metaheuristic, and also on the official website of the Metaheuristics Network: http://www.metaheuristics.org/.
- Finally, readers are invited to consult the excellent work of Garey and Johnson [62] for information regarding algorithm complexity and NP-completeness theory.

This remainder of this thesis is structured as follows. In Chapter 2 we will begin by providing an introduction to the problem of timetabling at universities. We will describe what a timetabling problem actually is, will take a look at the types of constraints that can be imposed upon them, and will discuss why these problems might often be hard to solve. We will also provide a review of the various different algorithmic approaches that have been proposed for these timetabling problems, concentrating our efforts mainly on metaheuristic approaches.

In Chapter 3 we will then conduct a detailed analysis of a particular version of a university course timetabling problem that has recently been used as a benchmark case for a number of relevant works in the literature. These works will also be reviewed here. In this chapter we will also make arguments as to why the two-stage timetabling approach might be an effective way of tackling this particular problem.

Next, in Chapter 4 we will present two algorithms intended for satisfying the mandatory constraints of our chosen university course timetabling problem-version, namely a Grouping Genetic Algorithm (GGA), and an Iterated Heuristic Search (IHS) algorithm. We will perform a deep analysis of these two algorithms and will also make some general observations about the possible limitations of GGAs in general.

In Chapter 5 we will take a brief excursion from the central theme of timetabling and, pursuing our arguments regarding the relative advantages and disadvantages of the GGAs and IHS algorithms for timetabling, we will perform a second analysis and comparison of these two algorithmic techniques on another combinatorial optimisation problem: graph colouring with equipartite graphs.

Moving away from GGAs and IHS algorithms, in Chapter 6 we will then move our attention on to the task of satisfying the non-mandatory constraints of our chosen university course timetabling problem-version. In this chapter we will present two distinct algorithms intended for this purpose and will conduct a thorough analysis of both. Results of these algorithms using a collection of publicly available problem instance-sets will also be reported, and reasons for their various advantages and disadvantages will be discussed.

The penultimate chapter of this thesis, Chapter 7, is then intended to move this work from the benchmark problem-version mainly considered in this thesis, and on towards more "real-world" problems. Consequently, we will describe methods by which our heuristic search-based algorithm might be modified for dealing with additional mandatory constraints – particularly those that specify that a particular resource is not available at a certain time (what we call "unavailability constraints"). We will also show how our simulated annealing algorithm (Chapter 6) might be extended to cope with these new constraints, and both of these new algorithms will then be empirically investigated. In the final section of this chapter we will also conduct a brief discussion as to how the algorithmic models developed during this thesis might also be extended for dealing with other real-world timetabling features not covered in this work.

Finally, Chapter 8 closes this thesis by providing a summary of the main conclusions that can be drawn from this work, together with some ideas about possible directions for further research.

1.4 Scientific Publications in Connection with Thesis

Some of the work described in this thesis is also the subject of a number of scientific papers that the author has prepared. These papers have either already been published or are currently in press. We will now list these in the chronological order in which they were published and will provide details of each.

Preliminary work on the evolutionary algorithm presented in the first half of Chapter 6 can be found in the following paper (which was also awarded "Best Student Paper" at the conference):

- R. Lewis and B. Paechter, "New Crossover Operators for Timetabling with Evolutionary Algorithms," presented at The Fifth International Conference on Recent Advances in Soft Computing RASC2004, Nottingham, England, 2004.

Initial studies on the applicability of the grouping genetic algorithm for satisfying the mandatory constraints of our chosen UCTP version can be found in:

- R. Lewis and B. Paechter, "Application of the Grouping Genetic Algorithm to University Course Timetabling," in *Evolutionary Computation in Combinatorial Optimization* (EvoCop), vol. 3448, Lecture Notes in Computer Science, G. Raidl and J. Gottlieb, Eds. Berlin: Springer-Verlag, 2005, pp. 144-153.

Additionally, a second preliminary paper examining the suitability of this GGA has also been produced. In this paper the various fitness functions and heuristic search operator that are given in Chapter 4 of this thesis are introduced.

- R. Lewis and B. Paechter, "An Empirical Analysis of the Grouping Genetic Algorithm: The Timetabling Case," presented at the IEEE Congress on Evolutionary Computation (IEEE CEC) 2005, Edinburgh, Scotland, 2005.

Next, in the following paper, work combining the latter two preliminary papers is presented. This paper also contains a much more detailed examination of the GGA and its general properties, as well as a more thorough experimental analysis. (This paper contains much of the work that is presented in Chapter 4.)

- R. Lewis and B. Paechter, "Finding Feasible Timetables using Group Based Operators," (Forthcoming) Accepted for publication in *IEEE Transactions of Evolutionary Computation*, 2006.

The following book chapter also contains ideas that are documented in this thesis. The first half of this publication proposes our method of classifying timetabling metaheuristics which we will discuss in Chapter 2 of this thesis. It also contains a literature review, together with an abridged analysis of the UCTP version, which is given in Chapter 3. Meanwhile, the second half of this book chapter contains much of the work concerning the simulated annealing-based algorithm for the UCTP that, in this thesis, is presented in the second half of Chapter 6.

- R. Lewis, B. Paechter, and O. Rossi-Doria, "Metaheuristics for University Course Timetabling," (Currently in Press) In *Evolutionary Scheduling*, Lecture Notes in Computer Science, P. Cowling and K. Dahal, Eds. Berlin: Springer-Verlag, 2006.

Finally, although not directly related to any of the work presented here, many of the techniques regarding algorithm design and analysis used in this thesis (particularly our simulated annealing algorithm of Chapter 5) have also been used in a recent paper that proposes a new approach for stochastically solving Sudoku-style puzzles. The details of this particular work are as follows:

- R. Lewis, "Metaheuristics can Solve Sudoku Puzzles," (Forthcoming) *Journal of Heuristics*, vol. 13, 2007.

2: Solving University Timetabling Problems

In this chapter we will be giving a general overview of timetabling problems with regards to what they actually are, the reasons why they are often troublesome, and the ways in which they might be solved. In the next section we will start this introduction by stating the general definition of a timetabling problem, and will then go on to look at the different types of problem that occur at universities, as well as the different types of rules (or constraints) that might be imposed upon them. Next, in Section 2.2, for various purposes that will become clear, we will provide a comparison between timetabling problems (in their simple form) and another well known combinatorial optimisation problem: graph colouring. After this, in Section 2.3, we will then conduct a detailed review of the current state of this field by taking a look at some of the many different algorithms – particularly metaheuristics – that have been proposed for the various timetabling problems put forward in the literature. We will then conclude the chapter in Section 2.4.

2.1 University Timetabling and Constraints

The generic definition of a university timetabling problem can be considered the task of assigning a number of *events*, such as lectures, exams, meetings, and so on, to a limited set of *timeslots* (and perhaps rooms), in accordance with a set of *constraints*. Generally speaking, it is usually accepted that within this definition, university timetabling problems can be arranged into two main categories: exam timetabling problems and course timetabling problems. In reality, and depending on the university involved, both types of problem might often exhibit very similar characteristics, but a common and generally acknowledged difference is that in exam timetabling, multiple events can be scheduled in the same room at the same time (providing seating-capacity constraints are not exceeded), whilst in course timetabling, we are generally only allowed one event per room, per timeslot. A second common difference between the two can also sometimes concern issues with the timeslots: course timetabling problems will generally involve assigning events to a fixed set of timeslots (e.g. those occurring in exactly one week), whilst exam timetabling problems might sometimes allow some flexibility in the number of timeslots being used. (However, as we will see later, this is not always the case.) Good surveys on exam timetabling have been conducted by Carter and Laporte [28, 30] and Burke *et al.* [18, 22, 24]. Other timetabling surveys can also be found at [32, 95, 100].

With regards to the various constraints that might be imposed on a particular timetabling problem, it is general practice to group these into two categories: the *hard* constraints, and the *soft* constraints. Hard constraints have a higher priority than soft, and will usually be mandatory in their satisfaction. Indeed, timetables will usually only be considered *feasible* if and only if all of the hard constraints of the problem have been satisfied. Soft constraints, meanwhile, are those that we want to obey *if possible*, and more often than not they will describe what it is for a timetable to be *good* with regards to the timetabling policies of the university concerned, as well as the experiences of the people who will have to use it.

Perhaps the most common hard constraint in timetabling is the so-called "event-clash" constraint. This constraint specifies that a person (or some other resource of which there is only one) is required to be present in a pair of events, then these events *conflict*, and must not, therefore, be assigned to the same timeslot (as obviously such an assignment will result in this person(s)/resource(s) having to be in two places at once). This particular constraint can be found in almost all university timetabling problems and its presence will often cause people to draw parallels between this problem and the well-known graph colouring problem (which we will look at in more detail in the next section). Beyond this example constraint, however, a great many other sorts of constraints – hard and soft – can ultimately be considered in timetabling, and in the real world it is usually the case that most universities will have their own specific idiosyncratic set of constraints that makes their particular timetabling problem different to most others.

Despite the wide variety of different constraints (and thus timetabling problems) that *can* be encountered, it has been suggested by Corne, Ross, and Fang [39] that the majority of these can be categorised into five main classes. We will now list these and will provide some example constraints belonging to each:

(1) **Unary Constraints.** These are constraints that involve just one event, such as the constraint "event *a* must not take place on a Tuesday", or the constraint "event *a* must occur in timeslot *b*". (Both of these examples can occur in exam and course timetabling and can be hard or soft.)

(2) **Binary Constraints**. These sorts of constraints concern pairs of events, such as the event-clash constraint mentioned earlier, or those that involve the *ordering* of events such as the constraint "event *a* must take place before event *b*" (this latter example can be common to both exam and course timetabling, and can be hard or soft).

(3) **Capacity Constraints**. These are constraints that are governed by room capacities, etc. For example "All events should be assigned to a room which has a sufficient capacity" (This particular example is common in both exam and course timetabling, and is usually a hard constraint.)

(4) **Event Spread Constraints**. These are the constraints that concern requirements such as the "spreading-out" or "clumping-together" of events within the timetable in order to ease student/teacher workload, and/or to agree with a university's timetabling policy (Common in both exam and course timetabling; usually a soft constraint.)

(5) **Agent Constraints**: These are the constraints that are imposed in order to promote the preferences of the people who will use the timetables, such as the constraint "lecturer *x* likes to teach event *a*", or "lecturer *y* likes to have *n* free mornings per week". (This is probably more common in course timetabling, and can be hard of soft.)

From this brief analysis, it should be appreciable that as well as different universities usually specifying their own particular set of constraints that will need to be satisfied, the *types* of constraint that can be encountered in timetabling can also vary over a wide range[2]. Of course, from a practical standpoint, this is entirely understandable as different universities are likely to have their own individual needs and timetabling policies (and

[2] It should also be noted that in some cases a problem definition for one institution may *directly oppose* another's in the criteria that defines a desirable timetable. For example, some institutions might specify that it is preferable for students' events to be bunched together in order to aid part-time students etc. (a policy of Napier University in Edinburgh, for example); other universities, however, may prefer the events of each student to be spaced out within the week as much as possible.

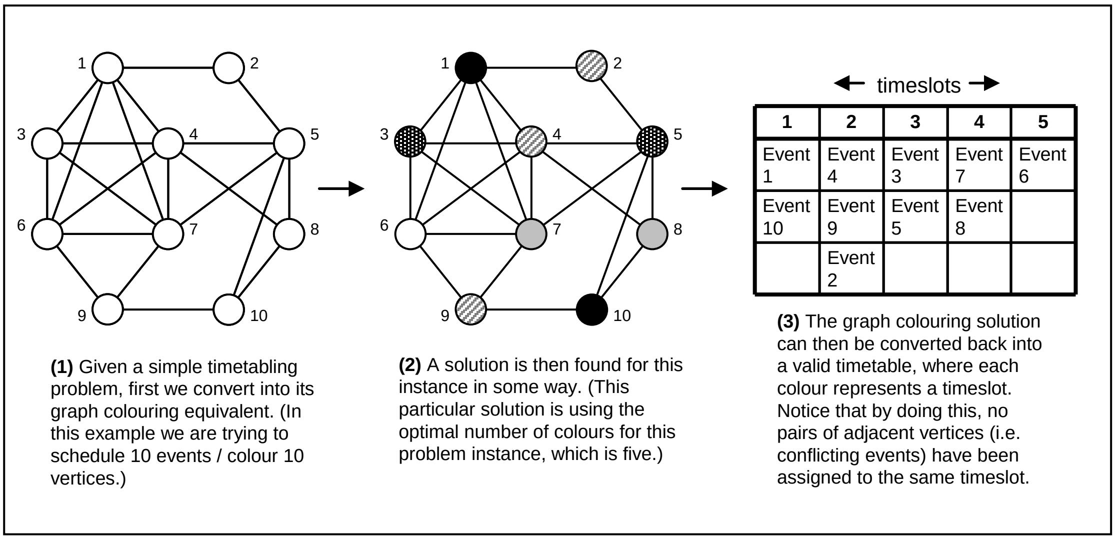
therefore set of constraints) that they need satisfied. However, from a research point-of-view, this idiosyncratic nature of timetabling can also make it very difficult to formulate meaningful and universal generalisations about the problem in general.

However, one important generalisation that we *can* make about timetabling problems at universities is that they are NP-complete in almost all variants [100]. Cooper and Kingston [38], for example, have shown a number of proofs to demonstrate that NP-completeness exists for a number of different problem interpretations that can often arise in practice. This, they achieve, by providing polynomially bounded transformations from various well-known NP-complete problems such as graph colouring (see Section 2.2), bin packing, and three-dimensional matching) to a number of different timetabling problem variants. Even, Itai, and Shamir [53] have also shown a transformation of the NP-complete 3-SAT into a timetabling problem. Of course, this general NP-completeness (or NP-hardness, if we are considering timetabling from an optimisation perspective) tells us that if we wish to obtain anything that might be considered a workable timetable in any sort of reasonable time, then this will depend very much on the nature of the problem instance being tackled. Some universities, for example, may have timetabling requirements that are fairly *loose*: perhaps, for example, there is an abundance of rooms, or only a very small number of events that need to be scheduled. In these cases, maybe there are lots of good (or at least acceptable) timetables within the total search space, of which one or more can be found quite easily. On the other hand, some university's requirements might be much more demanding, and perhaps only a very small proportion of the search space (if any) will be occupied by workable timetables. (It should also be noted that in practice, the combination of constraints that are imposed by timetabling administrators could often result in problems that are *impossible* to solve unless some of the constraints are relaxed). Thus, in cases where "harder" timetabling problems are encountered, there seems an implicit need for powerful and robust methods for tackling these sorts of problems.

2.2 Comparison to Graph Colouring

Before conducting a review of many of the various different heuristic timetabling algorithms that are available in the literature, it is first useful for us (as it will certainly assist us with various explanations within this thesis) to provide a comparison between the timetabling problem in its simplest form, and another well known computational problem: Graph Colouring. Given a simple and undirected graph G comprising a set of n vertices $V = \{v_1,...,v_n\}$ and a set of edges E which join various pairs of different vertices; the NP-hard graph colouring problem involves finding an assignment of "colours" for each vertex in V such that (a) no pair of vertices with a common edge are assigned the same colour, and (2) the number of colours being used is minimal[3].

It is straightforward to convert the most simple of timetabling problems into a graph colouring problems (and vice-versa) by considering each event as a vertex, and then simply adding edges between any pair of vertices that correspond to pairs of events that we do not want assigned to the same timeslot. Each timeslot that is available in the timetabling problem then corresponds to a colour, and the task (from a graph colouring point-of-view) it to simply try and find a solution that uses no more colours than there are available timeslots. (See figure 2.1, for an example.)



1	2	3	4	5
Event 1	Event 4	Event 3	Event 7	Event 6
Event 10	Event 9	Event 5	Event 8	
	Event 2			

(1) Given a simple timetabling problem, first we convert into its graph colouring equivalent. (In this example we are trying to schedule 10 events / colour 10 vertices.)

(2) A solution is then found for this instance in some way. (This particular solution is using the optimal number of colours for this problem instance, which is five.)

(3) The graph colouring solution can then be converted back into a valid timetable, where each colour represents a timeslot. Notice that by doing this, no pairs of adjacent vertices (i.e. conflicting events) have been assigned to the same timeslot.

Fig. 2.1: Demonstrating the relationship between the graph colouring problem and a simple timetabling problem where only the event-clash constraint is considered.

In graph colouring, the term "chromatic number" (commonly denoted χ) is used to refer to the minimum number of colours that are needed to feasibly colour a particular problem instance. Obviously, in simple timetabling problems this also represents the minimum number of timeslots that are needed for a clash-free timetable to be possible. Identifying χ is also an NP-hard problem, however.

A second parallel that we can draw between these problems involves the identification of features known as *cliques*. It is often noted that graph colouring problems that reflect

[3] See the work of Garey and Johnson [62] (page 133) for further details of this problem's NP-hardness. Also note that the NP-*complete* version of this problem defines a similar task, but as usual in the form of a decision problem: given $G = (V, E)$ and a positive integer $k < n$; is it possible to assign a colour to each vertex in V such that no pair of adjacent vertices has the same colour, and by only using k colours? (A proof of this decision problem's NP-completeness can be found in the work of Garey, Johnson, and Stockmeyer [61].)

real-world timetabling instances will often contain fairly large cliques. (A clique is simply a collection of vertices that are mutually adjacent, such as vertices 1, 3, 4, 6, and 7 in fig. 2.1, which is a clique of size 5.) This is because in many timetabling problems it is typical to encounter large collections of events that must not be scheduled together (e.g. a first year computer science degree might feature a number of compulsory events that all first-year computer scientists have to attend). In the equivalent graph colouring problem, the vertices that represent these events will form a clique, and it is easy to appreciate that no two vertices in this clique may be assigned the same colour (or equivalently, all of the corresponding events will need to be assigned to different timeslots). It is thus easy to deduce that if we are given a graph colouring instance (or equivalent timetabling problem instance) that has a maximum clique size of *C*, then *at least C* colours will be needed to colour the graph legally (that is $\chi \geq C$). The task of identifying the maximally-sized clique is also an NP-hard problem, however [62].

Note that this conversion to *pure* graph colouring problems only exists when we are considering constraints regarding *conflicting* events such as the event-clash constraint mentioned earlier. Indeed, when other sorts of constraints are also being considered, such as the *ordering* of the events within a timetable, then this will add extra complications. However, regardless of this, it is still the case that nearly all timetabling problems will still feature this underlying graph colouring problem in some form or another in their definitions, and it is certainly the case (as we will see) that many current timetabling algorithms use various bits of heuristic information extracted from this underlying problem as a driving force in their searches for a timetabling solution.

2.3 Algorithms for University Timetabling

Given the close relationship between graph colouring and university timetabling problems, it is perhaps unsurprising that many early techniques used in timetabling algorithms were derived directly from graph colouring-based heuristics (see the survey of Carter [28] from 1986 for a good review on some of these). One early example of this sort of algorithm was provided by White and Chan [112] in 1979. This particular approach was actually used for several years at the University of Ottawa in the 1970's, and in this study it is shown to be capable of scheduling 390 events involving 16,000 students into 25 timeslots. Basically, this method operates by first using a "largest degree first"-type heuristic to order the events. These events are then taken one-by-one according to this ordering and are assigned to the earliest timeslot that does not cause an event-clash, with new timeslots being opened whenever necessary. Next, if the resultant solution is seen to be using more timeslots than the desired amount, the algorithm then tries to move the events in these extra timeslots into the remaining ones. If this cannot be done, then these events are simply removed from the problem. Next, further heuristics are then used to try and find a suitable ordering for the timeslots that minimises the soft constraints of the problem, and finally, the events themselves are then shuffled in order to make further improvements.

Another early and commonly-cited example of this sort of algorithm is the *EXAMINE* timetabling system documented by Carter, Laporte, and Lee in [31]. In this paper, the system – which is a backtracking sequential-assignment algorithm – is applied to a set of real-world exam timetabling problems taken from a number of different universities. (These problem instances, which were first used in this study and which are often referred to as the Carter Instances, have been publicly available for a number of years now. They have also been used in a large number of exam timetabling papers, many of which we will look at later on in this chapter.) A number of algorithm variants are then tested and it is reported that the best performance is usually gained when two procedures are followed: first, when the events are inserted into the timetable following an ordering dictated by *saturation degree* heuristics (originally proposed by Brelaz [16], and which involves always selecting the node which has the highest number of colours adjacent to it); and second, when an additional algorithm is also used for trying to identify large cliques in the problem, so that the events within these cliques can then be given priority. The backtracking feature of this algorithm also enables the algorithm to undo previous assignments of events to timeslots when situations are encountered when an existing unplaced event has no feasible timeslots to which it can be assigned. In this case, the backtracking algorithm that is used is deterministic, meaning that for a given problem instance and ordering heuristic, the same timetable will always be produced.

Other approaches to timetabling problems have involved using constraint-based techniques (see the work of Deris *et al.* [44], Lajos [70], and Boizumault *et al.* [14], each of whom have applied these techniques to their own particular timetabling problems) and also integer programming (such as the algorithms of Carter [29], and Tripathy [108] in the 1980s and, more recently, Daskalaki *et al.* [43] in 2004). In the past decade-or-so there has also been a large growth of interest in the application of *metaheuristic*-based techniques to timetabling problems. In essence, the term "metaheuristics" is used to denote a variety of stochastic-based search techniques such as simulated annealing, tabu search, iterated local search, evolutionary algorithms, and ant colony optimization. According to the website of the *Metaheuristics Network* – an EU sponsored research project that was run from 2000 until 2004 [6] – a metaheuristic "… can be seen as a general algorithmic framework which can be applied to different optimization problems with relatively few modifications [being needed] to make them adapted to a specific problem." Given this latter characteristic, and also considering the idiosyncratic nature of timetabling problems that we have noted, it is perhaps unsurprising, therefore, that metaheuristics have become increasingly popular for addressing a number of timetabling problems in recent years.

However, when applying metaheuristic-based algorithms to timetabling problems, it is worth noting that an important aspect separating these sorts of problems from many other types of combinatorial optimisation problems is the presence of both hard and soft constraints in their formulations. For any metaheuristic approach to be worthwhile in this case there must therefore be some sort of system that is able to deal with timetabling constraints of both types in a satisfactory way. Our own survey of the timetabling literature with regards to this matter indicates that most metaheuristic algorithms for timetabling fall into one of three categories:

(1) **One-Stage Optimisation Algorithms:** where a satisfaction of both the hard and soft constraints is attempted simultaneously.

(2) **Two-Stage Optimisation Algorithms:** where a satisfaction of the soft constraints is only attempted once a feasible timetable has been found.

(3) **Algorithms that allow Relaxations:** Violations of the hard constraints are disallowed from the outset by relaxing some other feature of the problem. Attempts are then made to try and satisfy soft constraints, whilst also giving consideration to the task of eliminating these relaxations.

In the remainder of this section we will conduct a survey of the field of metaheuristics and timetabling using these three categories in order to classify the various algorithms. In each subsection we will first provide a simple and general overview of the method, and will then provide a description of various works that have used this basic approach. Note, however, that although this method of classification might be instructive for our reviewing purposes, it should not be interpreted as a definitive taxonomy, and it is arguable that some of the algorithms that will be mentioned here could belong to more than one of the categories.

2.3.1 One Stage Optimisation Algorithms

In general, timetabling algorithms of this type will allow the violation of both hard and soft constraints within the timetable, and the aim will be to search for a solution that has an adequate satisfaction of both. Ordinarily, this will be achieved through the use of some sort of weighted sum function intended for giving violations of the hard constraints much higher penalties than violations of the soft constraints. The following function is typical. Given a problem with *k* types of constraint, where the penalty weighting associated with each constraint *i* is w_i, and where $v_i(tt)$ represents the number of constraint violations of type *i* in a timetable *tt*, the quality of *tt* can be calculated using the formula:

$$f(tt) = \sum_{i=1}^{k} v_i . w_i (tt) , \tag{2.1}$$

There are possibly two main advantages to this sort of approach. First, because the aim is to simply search for a candidate solution that minimises[4] a single objective function, it can, of course, be used with any reasonable optimisation technique. Second, this approach is, in general, very flexible and easy to implement, because any sensible constraint can effectively be incorporated into the problem provided that an appropriate penalty weighting (which indicates its relative importance compared to others) is stipulated. This second factor, in particular, is highly convenient for timetabling problems where, as we have seen, we are likely to encounter an abundance of different combinations of constraints in practice.

It is perhaps for these reasons that we have seen a large number of timetabling algorithms of this type. Possibly, one of the earliest examples was provided by Colorni *et al.* in [35-37] where an evolutionary algorithm (EA) is proposed for a timetabling problem based on an Italian high school. In this particular approach, a number of hard and soft constraints are considered, and a weighted sum function is used to distinguish between the two types. Additionally, the bias introduced by the weightings is also complemented by a genetic repair procedure (that the authors call the filtering procedure), which is used to "fix" a particular timetable if the number of hard constraint violations is seen to rise above a certain level (this is defined by a system constant). This particular approach also encodes each timetable as a matrix where columns represent timeslots and each row represents a teacher involved with the timetable, and using this representation a novel crossover operator is then defined: taking two parent timetables p_1 and p_2, first, a local fitness function $f\,'$ is used to calculate the personal cost of each row (i.e. teacher) in both parent timetables. The crossover operator then takes the best *j* rows from p_1 and the remaining rows from p_2 to form the first offspring. The second offspring is then constructed by reversing the roles of the parents. (The value *j* is determined on the basis of the local fitness of both p_1 and p_2). The authors also report that improvements are gained when this operator is used in conjunction with a local search procedure that is used in order to try

[4] Depending on the particular evaluation function being used, the aim could be to maximise instead.

and move an offspring to its local optimum when it is first constructed. In [37] the authors compare this algorithm against a simulated annealing and tabu search approach and report that their EA produces better results (with respect to the minimal objective value found) than simulated annealing but slightly worse results than tabu search. However, they also note the obvious advantage that the EA possesses in that it is able to produce a large number of different, good quality timetables in a single run.

Another approach using EAs has been reported by Corne, Ross, and Fang in [39]. In this book chapter, weights are again used in the fitness function, but also three possible representations for the algorithm are discussed: namely the so-called clash-rich, clash-free and clash-sparse representations. In the first of these, a simple object-based encoding is used whereby each event is assigned an integer that represents a particular timeslot in the timetable. Thus a chromosome such as 9273453, for example, is interpreted as "event 1 is to be scheduled in timeslot 9", "event 2 is to be scheduled into timeslot 2", "event 3 into timeslot 7", and so on. As the authors note, it is likely that the vast majority of chromosomes in the search space defined by this encoding will probably have some hard constraint violations, especially due to event-clashes (hence the name clash-rich). However, one benefit of this approach is that because of its simplicity, the authors are able to define various *smart-mutation* operators, such as their *violation-directed* and *event-freeing* mutation operators, in which problem areas of the chromosome are identified and mutated in such a way so that constraint violations are hopefully rectified. Such operators are reported to improve the performance of the algorithm.

Corne *et al.*'s clash-free approach, meanwhile, uses a representation whereby each chromosome is represented by a *permutation* of the events. The algorithm then employs a scheme whereby timetables are formed by taking events one-by-one and assigning them to the first timeslot that causes no conflicts with any other events already in the timeslot. If no place exists, then new timeslots are opened accordingly. The main aim of this algorithm therefore becomes one of reducing the number of timeslots down to an acceptable level whilst never allowing conflicting events to be placed in the same timeslot (hence the name clash-free). This scheme is, of course, very similar to the greedy (or first-fit) algorithm used in graph colouring (see for example [42]). However, as is noted by the authors, this representation's main pitfall is that when constraints other than event-clashes are considered in the problem (such as event spreading constraints), we cannot be sure that the optimal timetable is actually representable. A potential solution to this latter problem is thus suggested with the authors' third method of encoding, that they call clash-sparse representation. Here, a single fixed ordering of events is used and instead of putting each into the first available slot, the event is put into the *k*th available slot (and where *k* is a value stored for each event in the chromosome). The approach is termed as clash-sparse because if there is no available slot for an event, then it is simply placed into the *k*th unavailable slot. Thus clashes are allowed, but are less common in the search space than the clash-rich approach. In [39] the authors show the results of experiments that compare these three approaches using a number of randomly generated test instances, as well as some real-world problem instances taken from Edinburgh University. In general the clash-rich approach shows the best performance.

Yet another one-stage EA approach to timetabling has been proposed Carrasco and Pato in [27]. In this work the authors consider a particular course timetabling problem originating from the Department of Hotel Management and Tourism at the University of the Algarve, Portugal. They also decide that two distinct measures should be used to evaluate a particular solution: the number of soft constraint violations from the students' perspective (the *class-oriented* objective), and the number of soft constraint violations from the point-of-view of the teaching staff (the *teacher-oriented* objective). As the authors note, in their problem these two measures of quality actually conflict, and they therefore decide to apply a multiobjective evolutionary algorithm (MOEA) to the problem, so that upon completion of a run, a user might be provided with a range of trade-off solutions with regards to these two competing objectives. The proposed algorithm operates as follows. First, making use of a (room × timeslots) matrix representation, an initial population is constructed using a constructive heuristic procedure. The resultant candidate solutions (which may or may not also contain violations of the hard constraints) are then evaluated according to both objective functions. Heavy penalties are then applied to both of these measures in order to penalise any occurrences of hard constraint violations. Next, using their own specialised evolutionary operators together with a selection pressure determined by the Non-Dominated Sorting approach of Srinivas and Deb [104], the population is then evolved, and all non-dominated solutions that are discovered during this process (and which are also deemed sufficiently distinct from one another) are copied into an archive population. Upon completion of a run, the final archive will contain a collection of timetables that can be both feasible and infeasible. The authors report that this algorithm, which in this case is only tested on one problem instance, is able to produce feasible solutions that are better than the previous manually produced timetables with regards to both of the competing objectives.

Moving away from evolutionary algorithms, various other authors including Abramson [9], Melicio *et al.* [79], and Elmohamed *et al.* [51] have also reported one-stage optimisation algorithms that make use of the simulated annealing (SA) metaheuristic. In the approach of Elmohamed *et al.*, for example, the authors consider the timetabling problem of Syracuse University in the USA and use a weighted-sum scoring function that heavily penalises violations of the hard constraints. As well as using general simulated annealing practices to optimise these timetables, the authors also point out that good results can be gained if three modifications are made to their system: (1) the introduction of *adaptive cooling*, (2) the use of a rule-based pre-processor to intelligently build good starting solutions for the annealing phase, and (3) the use of specialised methods for choosing neighbourhood moves. We will now take a brief look at each of these.

Firstly, adaptive cooling is basically a system by which a new temperature is calculated based on the *specific heat* of the current temperature. The idea is to keep the system close to equilibrium, by cooling more slowly at the temperatures near to which a *phase transition* occurs. In other words, the goal is to make sure that the global structure is arranged into an optimal a state as possible before moving on to resolve the finer details of the problem. The author's rule-based system, meanwhile, introduces a number of heuristics and additional data structures to the algorithm (such as distance matrices and room-event feasibility matrices) so that sensible choices are made when producing an initial solution. These heuristics and data structures are also then used for the third method of improvement. Here, the authors note that if the neighbourhood move applied during the optimisation process is selected at random (as is typical in SA), then when the search starts closing in on good regions of the search space, it will usually be the case that the vast majority of neighbourhood moves will cause an increase in the objective function and will thus be rejected. The authors therefore propose a system by which some bias is placed upon the selection of moves so that moves that have a higher chance of producing positive movements in the search space are favoured.

Another notable feature of this SA application is that the authors choose to use a method of reheating called Reheating as a Function of Cost, which was first introduced by Abramson *et al.* in [10]. The process of reheating in general SA practice can be useful, because a rise in the temperature parameter will increase the probability of negative movements (i.e. those that worsen a candidate solution with regards to the objective function) in the search space being accepted, and can therefore be used to help escape local optima if a search becomes trapped. Abramson *et al.* were the first authors to examine the effect that reheating could have with regards to SA and timetabling in [10]. In this study six separate cooling schedules are considered, of which four – namely Geometric Reheating, Enhanced Geometric Reheating, Non-Monotonic Cooling and Reheating as a Function of Cost – make use of reheating schemes. The latter scheme in particular deserves a special mention here, because as well as being the method that produced the best results in the paper, it is also, a technique that we will be making use of in Chapter 6. We will thus spend the next few paragraphs going over the theory of the technique here. (Note that this description will refer to those problems in which the aim is to *minimise* an objective function. However, the approach could, of course, also be easily modified to cope with maximisation tasks as well).

It is generally acknowledged that in the physical process of annealing (in metallurgy) the particular substance being cooled may experience a number of *phase transitions* before a frozen state is reached. This essentially means that in each transition the substance will undergo some sort of change in its molecular/atomic makeup that will alter the way in which it behaves. When Kirkpatrick *et al.* [68] first proposed SA as a general optimisation technique in 1983 a similar phenomenon was also proposed to occur. For instance, it is suggested that when the temperature of the system is high, then generally it will be the *gross* structure (or super-structure) of the problem that is resolved. During low temperatures, meanwhile, it will then be the more minute, *finer* details of the problem that are dealt with. We can use the term "phase transition" to thus denote the point at which the algorithm is observed to move from the former into the latter, and such transitions have been observed with various applications of SA [10, 68, 76]. In order to apply these concepts to a method of reheating, in [10] Abramson *et al.* propose the following scheme. First, during the run the temperature at which a phase transition occurs (T_{pt}) is calculated. In order to do this, it is necessary at each temperature T to determine the *specific heat* of the substance, which can be calculated using the following formula:

$$\frac{\sigma^2 C(T)}{T^2} \tag{2.2}$$

Here, $\sigma^2 C(T)$ represents the variance of the cost function at a particular temperature T. A phase transition is then deemed to occur at the temperature where the specific heat is maximal. (An example of these calculations can be seen in fig. 2.2.)

Having determined T_{pt}, Abramson suggests that a reheat temperature T_{reheat} can then be calculated using the formula:

$$T_{\text{reheat}} = \lambda C_{\text{best}} + T_{\text{pt}} \tag{2.3}$$

where C_{best} represents the cost of the best timetable found so far in the search, and λ represents a parameter that needs to be set to some value greater than zero[5]. The idea

[5] Generally, an appropriate value for λ will depend on the range of the objective function being considered. If the range is quite small (say, 0 to 100) then a value of between around 0.1 to 1.0 might turn out to be appropriate. For larger ranges, however, higher settings might often be needed.

behind Reheating as a Function of Cost is thus very simple: if the best solution found so far during the run is quite poor (and, thus, C_{best} is still quite high), then it is likely that the overall global structure of the solution will need to undergo some fundamental rearrangements, and so it is probably appropriate for T_{reheat} to take a value that is some way above T_{pt}. On the other hand, if the best solution found so far is very close to the optimal, then the global structure of the solution probably only needs to undergo very minor rearrangements. Consequently, T_{reheat} will take a value that is only a very small way above T_{pt}. In both [10] and [51] this method of reheating is reported to be very effective at producing good timetables; indeed, as we will see in our own SA algorithm documented in Chapter 6 later, it is also appropriate for our needs as well.

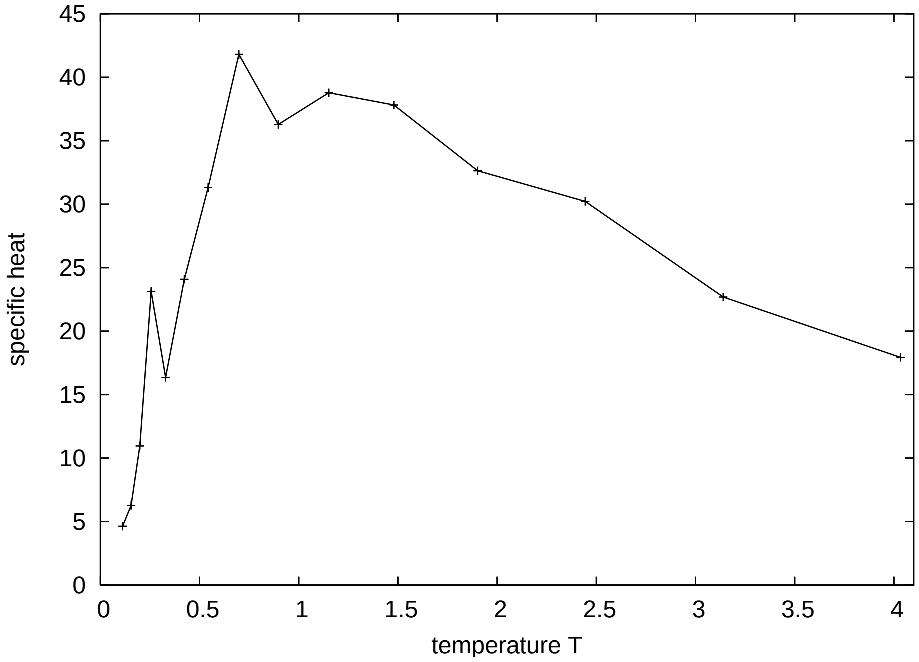


Fig. 2.2: Demonstrating how the *specific heat* can vary at different temperatures during a run of SA, and showing how we can use this information to calculate T_{pt}. This particular plot was constructed using data taken from an example run of our own SA algorithm for timetabling that we will see later in Chapter 6; it can be seen that the T_{pt} in this case is approximately 0.7

Returning now to our review of one-stage optimisation algorithms, it is worth noting that other metaheuristics apart from EAs and SA have also been proposed for various timetabling problems in the literature. For example, Costa [40], Hertz [66], and Schaerf [99] have all made use of tabu search for their timetabling problems. In the approach of Schaerf, for example, the author suggests inserting periods of local search in-between phases of tabu search, with the best results of each stage then being passed into the next stage, until no further progress in the search space can be made. In this approach the author makes use of a matrix representation in a similar manner to Colorni *et al.*'s encoding mentioned earlier, and also defines two neighbourhood operators. The first of these simply swaps the values contained in two cells of the matrix on a given row, one of which can be blank. However as Schaerf notes, this operator will often cause violations of the hard constraints, and so he also makes use of a second neighbourhood operator which uses the concept of *double moves*. Essentially, this works by making two moves simultaneously, with the second one being specifically chosen so that it has a good chance of cancelling out any infeasibility caused by the first one. In order to deal with other hard constraint violations that might occur, however, weights are also used in the evaluation function. The results from this paper (with some experiments being performed on some real data taken from some Italian schools) show that in their cases, the algorithm is able to always produce feasible solutions that are better than the hand-made ones originally used in the schools, and in reasonable amounts of time. In later work Di Gaspero and Schaerf [45] have also made use of similar methods (i.e. using tabu search, together with a weighted evaluation function) for the examination timetabling problem. A comparison with other algorithms proposed for the same problem, however, reveals that this approach is not always able to perform as well in some instances; although the authors do make note of a number of possible improvements that could be made in the future.

Di Gaspero and Schaerf [46] have also taken a look at the applicability of multi-neighbourhood search to a course timetabling problem. In this work, the authors investigate how various combinations of different neighbourhood operators can be combined in order to produce effective searches through the search space (in this case only two operators are considered). The combinations suggested are neighbourhood-*union*, neighbourhood-*composition* and *token-ring* search. A so-called *kick* operator is also defined which is used to provide perturbations to the search and to help it avoid getting stuck in local optima. In this work the authors once again make use of weights in the objective function in order to penalise the occurrence of hard constraint violations. The various proposed algorithms are then tested on four real-world based problems from an Italian engineering department. It is noted that timetable feasibility is found in all trials, and that in three of the four problem instances, the best results are achieved when the search space is explored using the neighbourhood-union operator, which basically involves randomly choosing one of the two neighbourhoods and then simply performing a random movement within this chosen neighbourhood to produce a new solution in each step of the algorithm.

At this point, we have reviewed a number of different algorithms that, in one way or another, have made use of weightings in their objective functions in order to penalise more heavily the occurrence of hard constraint violations within a candidate solution. However, it is worth noting here that although this technique has some good points (that we noted at the beginning of this subsection); it also has some inherent disadvantages. Various authors (e.g. Richardson *et al.* [91]) have argued that this sort of evaluation scheme does not work well in problems that are sparse (i.e. where only a few solutions exist in the search space). Also, even though the choice of weights for discriminating between different sort of violations will often critically influence the algorithm's navigation of the search space (and therefore its timing implications and solution quality), there does not actually seem to be any obvious method for choosing them. (Note that different instances of the same problem will usually require different weightings as well.) This means that a rather unsatisfactory amount of *ad hoc* reasoning is therefore usually required. Some authors (e.g. Salwach [98]) have also proposed that a weighted sum function can often introduce much more discontinuity into the fitness landscape of a problem, meaning that we will be presented with a situation in which small changes to a candidate solution may actually result in overly large changes to the fitness function.

With regards to timetabling, however, it is worth noting that some of the algorithms that we have reviewed above have had automated methods built into them in order to try and alleviate some of these problems. For example, in both [45] and [99] Schaerf and Di Gaspero have used a method by which the penalties awarded to hard constraint violations can actually be altered dynamically *during* the search. For example, in [99], Schaerf uses a weighting value *W* that is initially set to 20. However, at certain points during the search, the algorithm is able to increase *W* when it is felt that the search is drifting into search-space regions that are deemed too infeasible. Similarly, *W* can also be reduced when the search consistently finds itself in feasible regions. It is proposed that such a scheme allows a better quality search than simply keeping *W* fixed throughout the run. The *Tatties* timetabling system of Paechter *et al.* [86] (which we will also look at in Section 2.3.3) also allows weights to be altered during a run. However, this time the adjustment is performed manually by the users themselves rather than being done automatically by the algorithm. The user is thus presented with a graphical interface that allows him-or-her to specify a target and a weight for the various different problem measures that are considered. The user is then able to change these values as they see fit during a run of the algorithm.

2.3.2 Two-stage Optimisation Algorithms

The operational characteristics of two-stage optimisation algorithms for timetabling may be summarised as follows: in stage-one, the soft constraints are generally disregarded and only the hard constraints are considered for optimisation (i.e. only a *feasible* timetable is sought). Next, assuming feasibility has been found, attempts are then made to try and minimise the number of the soft constraint violations, using techniques that only allow feasible areas of the search space to be navigated.

Obviously, one immediate benefit of this technique is that it is no longer necessary to define weightings in order to distinguish between hard and soft constraints, because we no longer need to directly compare feasible and infeasible timetables. In practical situations, such a technique might also be more fitting when achieving timetable feasibility is the primary objective, and where we only wish to make allowances towards the soft constraints if this feasibility is definitely *not* compromised. Indeed, there are two main reasons why the use of a one-stage optimisation algorithm might turn out to be inappropriate in a situation such as this. First, whilst searching for a feasible timetable, a weighted-sum evaluation function will always take the soft constraints into account to some extent, and therefore it might provoke the adverse effect of leading the search *away* from attractive (i.e. in this case fully feasible) regions of the search space. Second, the operational characteristics of the one-stage optimisation approach also suggest that these sorts of algorithm will often allow a timetable to move away from feasibility in order to eliminate some soft constraint violations. However, in this particular case this strategy might be unwise, because there would, of course, be no guarantee that feasibility could then be re-established at a different point in the search space later on.

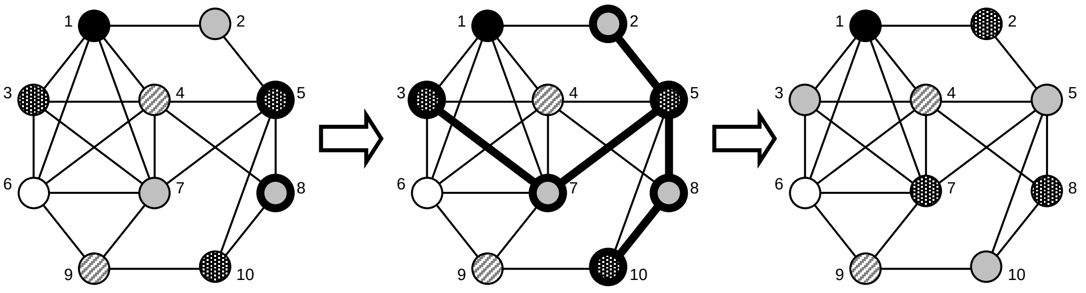
An early example of the two-stage approach was provided by Thompson and Dowsland in [107]. In this study, the authors consider a simple exam timetabling problem in which a feasible timetable is deemed one that contains no event-clashes and that obeys all of the capacity constraints. However, soft constraints are also considered in this problem, which concern the desirability of having each student's exams spread out *within* the pre-specified exam period in order to decrease student stress and to aid revision. In [107], Thompson and Dowsland choose to relate their exam timetabling problem directly to the underlying graph colouring problem and suggest a two-phase approach whereby a feasible colouring/timetable is first obtained, and then various neighbourhood operators that always preserve feasibility (but, at the same time, are still able to alter the other characteristics of solution) are explored. Although, as we noted earlier, the graph colouring problem is NP-hard, the authors state that the instances they use (data taken from the exam requirements at the University of Wales, Swansea, as well as some other universities) are loose enough such that feasible colourings can always be found fairly easily using standard graph-colouring algorithms which have been modified slightly to take into account capacity constraints of each timeslot.

A feasible colouring (exam timetable) having been found, Thompson and Dowsland then attempt to satisfy the soft constraints of their problem by making use of simulated annealing. In their tests, three different neighbourhood operators are proposed, each which is specially designed to only allow movements from one feasible timetable to another:

(1) **The Standard Neighbourhood**: whereby the colour of a particular vertex is changed to a new colour (i.e. an event is moved to a new timeslot) in such a way so that feasibility is preserved.

(2) **Kempe Chain Neighbourhood**: whereby the colours of all the vertices contained in a randomly selected Kempe chain are switched. (See fig. 2.3.)

(3) ***S*-chain neighbourhood**: which is, in essence, very similar to the Kempe chain neighbourhood above, but which involves more colours. Thus $S > 2$ colours are identified, and using an ordered list of these S colours together with a recursive procedure to produce the S-chain, we are able to identify various sub-graphs whose colours, when swapped, will result in a different but still feasible solution. (In this case only 3-chain neighbourhoods are considered. For further reference see the work of Morgenstern [82].)



(1) Select two adjacent nodes at random. (In this case we have selected vertices 5 and 8). Assuming that the graph contains no colour conflicts then these two vertices will, of course, be different in colour.

(2) Next, determine the corresponding Kempe chain of these vertices. This is done by identifying all of the connected vertices that are reachable from vertices 5 and 8 and which are assigned either of the two selected colours.

(3) Swapping all of the colours within this (and indeed any) Kempe chain will result in a different feasible colouring that uses no more colours than the original.

Fig. 2.3: Example of a Kempe chain neighbourhood operator.

The conclusions of Thompson and Dowsland's work in [107] is that the Kempe chain neighbourhood, when used in conjunction with SA, is able to produce significantly better results than the standard neighbourhood, and this is probably due to its increased flexibility over the standard neighbourhood operator. Additionally, they also state that in their case the standard neighbourhood operator often displays an unfavourable sampling bias, as not all colours are usually available for each vertex. They also note that there is little difference in the results between Kempe chain neighbourhoods and 3-chain neighbourhoods.

In [114], Yu and Sung have also proposed a two-stage course-timetabling algorithm and choose to follow an evolutionary approach. This algorithm starts with an initial population of totally feasible timetables (the input used is easy enough to allow this), and then attempts to optimise these with respect to a set of soft constraints, whilst always maintaining feasibility. A direct representation similar to that used by Colorni *et al.* [35-37] is also used, and because this method only considers feasible timetables, the evaluation function is only concerned with the number of soft constraint violations. This paper also introduces an interesting recombination operator called Sector Based Partially Mapped Crossover, in which a sector of one timetable is injected into another timetable as a means of crossover. The term "partially mapped" is applied in this case because a sector is rarely transformed in its entirety because of infeasibilities/illegalities arising in the resultant offspring. Instead, genes that *do* cause a problem when being transferred are dealt with by an appropriate genetic repair routine that enforces feasibility of the timetable at all times. Unfortunately, however, the authors carry out only limited experimentation with this algorithm and provide few details on the problem instances that are used.

A two-stage style approach for exam timetabling has also been proposed by Casey and Thompson in [33]. This particular algorithm is based upon a Greedy Randomised Adaptive Search Procedure (GRASP) and operates as follows. In stage-one, a feasible, clash-free timetable is first gained using a probabilistic-based constructive algorithm that works by taking events one at a time (based, to some extent, on some ordering heuristics derived from graph colouring-based algorithms) and attempts to place these into feasible timeslots, using backtracking when necessary [31]. Once feasibility has been achieved, the algorithm then moves onto the second phase in which a satisfaction of the soft constraints is attempted, whilst always remaining in feasible areas of the search space (this is achieved using Kempe chain interchanges and SA and is similar to the work of Thompson and Dowsland [107] mentioned above). After a period, the algorithm then returns to stage one, using the information gained in the pervious iteration to aid the constructive algorithm. This then continues for a set number of iterations. This algorithm is tested on the Carter Instances [1], and the results reported are generally promising.

Having now looked at a number of two-stage timetabling algorithms for this problem[6] it should be clear that for this approach to work effectively, it is essential that two

[6] It is worth noting that a number of two-stage timetabling algorithms have also recently been proposed for the benchmark timetabling problem that we will be studying in later chapters here. However, we will not discuss these algorithms at this point in the thesis; instead we will leave their examination for the next chapter where we will be looking at this particular problem-version (and the algorithms proposed for it) in much more detail.

criteria are met. First, feasibility must be obtainable in a reasonable amount of time. Second, a practical amount of movement within the resultant feasible-only search space must be achievable. With regards to the latter requirement, if the search space of the problem is quite unconstrained and a reasonable part of this space is made up of feasible solutions, then this may be so. However, it is also possible that if the "constrainedness" of the search space is quite high, then searches of this kind could also turn out to be extremely inefficient and/or ineffective at times, because it might simply be too difficult for the algorithm to move about and explore the feasible parts search space in any sort of useful way (this matter will be examined in more detail in Chapter 7); indeed, in these cases, perhaps, a method that allows the search to take "shortcuts" across infeasible parts of the search space might allow a better search to be conducted in some cases.

Additionally, whether this technique will be appropriate in a practical sense also depends largely on the user requirements: if a completely feasible timetable is an absolute necessity and the user is only interested in satisfying the soft constraints if this feasibility is assured, then this approach may well be fitting. On the other hand, if, for example, we were to be presented with a problem instance where feasibility was very difficult (or impossible) to achieve, then it may be the case that users might actually prefer to be given a solution timetable in which a suitable compromise between the number of hard and soft constraint violations has been achieved (suggesting that, perhaps one of the other two types of algorithm might be more appropriate instead).

2.3.3 Algorithms that allow Relaxations

Our final category of metaheuristic timetabling algorithm contains those methods in which some aspect of the problem has been relaxed so that the algorithm is able to make attempts in satisfying the soft constraints, but not at the expense of violating any of the hard constraints. Typically these "relaxations" will be achieved in one of two ways:

(1) Events that cannot be feasibly assigned to any place in the current timetable will be left to one side unplaced. The algorithm will then attempt to satisfy the soft constraints and will hope to assign these unplaced events to somewhere in the timetable at a later stage.

(2) Extra timeslots will be opened in order to deal with events that have no existing feasible timeslot available. If necessary, efforts will then be made to try and reduce the number of timeslots down to the required amount, whilst also taking into consideration the satisfaction of the soft constraints.

An early example of the second type above was proposed by Burke, Elliman, and Weare [19] and followed an evolutionary-based approach applied to exam timetabling. In this method each individual in the initial population is first created by a random permutation of the exams. These permutations are fed into a greedy graph colouring-style algorithm which takes each event in turn and places it into the first timeslot where no clash occurs and where capacity constraints are not exceeded. Extra timeslots are then opened for exams that cannot be placed into existing timeslots. (Note that building an initial population in this way will produce a collection of timetables that will contain no hard constraint violations, but which may well be using a variable number of timeslots.) The algorithm is then concerned with meeting two requirements: lowering the number of timeslots being used to a suitable level; and *spreading* the exams of each student within the timetable in a similar manner to Thompson and Dowsland [107] mentioned earlier (the latter is referred to as the *secondary objective* in the paper). In their approach, the authors choose to combine these two objectives into a single numerical function using weights. Knowledge augmented evolutionary operators specially designed to produce offspring that also contain no hard constraint violations are also used to evolve the population. The authors present results of the algorithm using various different weightings in their fitness function, and their results indicate the following trade-off: if, in the fitness function, a large emphasis is placed upon keeping the timetable short, then this will usually be to the detriment of the secondary objective, because there will be less opportunity for events to be spread out. Conversely, if a larger emphasis is placed upon the secondary objective this will likely mean that a longer timetable will be produced.

In [20] Burke, Elliman, and Weare have continued this work, this time with their research being directed towards the design of more specialised recombination operators for the problem. The authors define a basic scheme for recombination, and then experiment with various heuristics used in conjunction with this scheme in order to identify the most effective one. Good results to a real-world timetabling problem taken from Carter's problem instance set [1] are claimed.

A similar strategy of using variable length exam timetables has also been presented by Erben in [52], who chooses to use a modified version of his Grouping Genetic Algorithm (GGA) for graph colouring which is presented in the same paper (we will be looking at this particular graph colouring algorithm in much more detail in Chapter 5). In this case, an initial population of feasible exam timetables is made in a very similar way to Burke *et al.* above; however, this time the population is evolved using GGA-based operators instead (see the work of Falkenauer [54, 58] and also Chapter 4 of this thesis). As with Burke *et al.*, Erben then concerns himself with satisfying two objectives: shortening the timetable (i.e.

decreasing the number of timeslots), and spreading the exams to ease student workloads. Once again, these two objectives are combined into one fitness function, with heavier weightings being placed upon the former objective.

It is noticeable that in the exam timetabling problems tackled by Burke *et al.* [19, 20] and Erben [52], a compromise will usually have to be struck between (a) keeping the timetable short and (b) spreading-out each of the students' events within the timetable. In other words, this problem features two conflicting objectives, because if we wish to increase the average time that students are given between exams, then this will generally require larger numbers of timeslots; and conversely, if we reduce the number of timeslots being used by the timetable, then the potential for events to be spread-out will also diminish. In this sense, this formulation of the exam timetabling problem might be considered a type of Multiobjective Problem (MOP). As we have noted, in the work of both Burke *et al.* [19, 20] and Erben [52], the authors have chosen to cope with these two conflicting objectives by using pre-specified weightings in order to combine both into a single evaluation function. However, as is often noted in MOP literature, there is actually an inherent problem in using a single evaluation function for these sorts of problems, because in many cases the criteria used for measuring the different objectives might be incommensurable. For example, in this case it might be appropriate to ask whether it is wholly appropriate to combine a figure that represents the number of timeslots being used (similarly the number of unplaced events) with a figure that reflects how spread-out each of the students' events are. Other criticisms of using a single objective function are also often noted due to the fact that, from a user's perspective, the task of manually specifying weightings for these sorts of problems before a run is usually highly complex, and that the effects that these weights will have on the search will often be hard to predict[7].

For these reasons Cote, Wong, and Sabourin [41] have chosen to make use of multiobjective optimisation techniques for this particular bi-objective version of the exam timetabling problem. In essence, their approach involves the use of a hybrid multiobjective EA, which operates by ranking each member of the population using the concepts of Pareto Strength used in the *SPEA-II* multiobjective EA of Zitzler *et al.* [115]. The algorithm operates by first randomly producing a population of timetables that use various numbers of timeslots between some fixed bounds, and which may or may not be feasible. During a run the population is then evolved using two local search procedures which are used to

[7] We have mentioned some potential problems with weights in Section 2.3.1. Note also that are various other issues that surround the use of single objective functions in MOPs as well. However, we will not go into these here; instead, further information can be found in some good texts such as the work of Eiben and Smith [49] and Landa Silva, Burke, and Petrovic [71].

eliminate timetable infeasibilities and to make improvements on the spreading of the events. No form of recombination is used. The algorithm then stores any non-dominated solutions that are found in an archive population and, upon completion of a run, the user is presented with a number of alternative timetables of different length and differing levels of event spread. In practical applications this latter feature might be considered an advantage, because the user will be able to choose from a variety of solutions along the Pareto front, without there being any need for the manual specification of weights beforehand. (Presumably the process of manually choosing one of these timetables is easier than manually defining the weights in the first place.) In their experiments, for each run of the algorithm the authors choose to allow five separate timetable lengths to be stored in the archive and very good results are reported when the algorithm is allowed to run for long periods.

Given the fact that so many different criteria can ultimately be used to evaluate a particular timetable (due to the large number of constraints, hard and soft, that might be imposed on a particular problem) it seems that there is a large potential for conflicting objectives to actually occur in practical timetabling. However, apart from the bi-objective examples of Cote *et al.* [41] and Carrasco and Pato [27] (the latter which we saw in Section 2.3.1), there does not seem to have been a great deal of applications of multiobjective metaheuristics to university timetabling problems in general. Indeed, to our knowledge the only other notable examples from this field are provided by Burke, Bykov, and Petrovic in [21]; Petrovic and Bykov in [88]; and in an extended abstract by Paquete and Fonseca in [87]. In the first two of these works, the authors consider exam timetabling problems, and, again, the problem instances used in experiments are taken from Nottingham University and Carter instance set. In both cases the authors choose to impose nine different soft constraints on these problems, which deal with factors concerning room capacities, the spreading-out of exams, and the ordering of exams. Additionally, neighbourhood search techniques then are used in order to try and optimise these constraints and, in the case of Petrovic and Bykov, weights that are adjusted automatically during the run are used in order to help influence the direction of the search. (See [21] and [88] for further details.)

Moving away from the field of multiobjective metaheuristics, another evolutionary-based approach to exam timetabling, which uses methods that are slightly different to those discussed thus far, has also been proposed by Burke, Newall, and Weare in [25]. In this case the number of timeslots that are to be used in the timetable is defined explicitly in advance, and events that cannot be inserted into a timeslot are then deliberately kept to one side unplaced. Consequently, an evaluation function is then used that reflects (a) the number of unplaced events (to which a large weighting is applied) and (b) how spread-out each student's exams are in the timetable. During a run of the algorithm the authors then make use of three operators in order to evolve this population: their so-called *light* and *heavy* mutation operators and a deterministic hill-climbing procedure. (Again no recombination is used.) In particular, the latter operator, as well as attempting to optimise the spreading of exams, also places a special preference on inserting the unplaced exams if this is possible. The authors then test this algorithm using an instance from Nottingham University and the Carter problem instances [1], over a range of timeslot limits. In all cases, the algorithm is reported to eventually schedule all of the unplaced exams, as well as making significant improvements in the spread of the exams in the timetable.

Further work has also been carried out using this algorithm by Burke and Newall [26]. In this paper, however, the authors choose to investigate the idea of problem decomposition. Given a particular problem instance, the algorithm first splits the event set into a number of subsets E_1 to E_n. The method then attempts to schedule each of these groups, in turn, using the evolutionary-based algorithm just mentioned [25]. In other words, at each stage, the algorithm is only applied to one subset E_i of events at a time, and once timeslots have been found for each of the events in this subset (and a suitable compromise with regards to the spreading of events has been reached) these events are then fixed in place, and the algorithm then moves on to the next subset. It is hoped then, that at each stage of this algorithm the evolutionary-based approach will be able to schedule all of the events of a given subset E_i into the timetable *on top* of those groups E_1 to E_{i-1} that have already been scheduled, before going on to schedule the remaining subsets of events E_{i+1} to E_n. As the authors note, however, the obvious pitfall of this method is that by fixing events into timeslots like this, it may be impossible to schedule events later on in the process. They therefore experiment with a number of different heuristics for partitioning the event set, and also incorporate a look-ahead feature which allows two subsets to be considered at the same time, but which only fixes the events of the first subset at the end of each stage. The results indicate that this method of decomposition allows good results to be gained in much less time than when using the same evolutionary-based algorithm on the entire event set in one go.

The same decomposition method for the same exam timetabling problem has also been used with Batenburg and Palenstijn in [12]. In their own implementation of Burke and Newall's algorithm [26], they note that, in actual fact, the evolutionary-based algorithm used for scheduling each subset of events tends to converge quite quickly. Consequently, they choose to use their own parallel tabu search-based approach for optimising the population of partial timetables at each stage instead. According to results on one problem instance (the only one tested), the authors claim this approach ultimately brings better results, but at the expense of extra run time.

Merlot *et al.* have also looked at exam timetabling in [80]. Here the authors suggest a three stage approach involving constraint programming, simulated annealing and then hill climbing. In the first stage the constraint programming part of the algorithm is used to try and construct a feasible timetable. However, if events cannot be placed, these are placed into a so-called *dummy slot* at the end of the timetable. In the next two stages attempts are then made to try and improve the timetable (with regards to satisfying the soft constraints and also trying to deal with the events in the dummy slot) through a use of SA and a hill-climber, using Kempe chain interchanges. An evaluation function that places a large emphasis on emptying the dummy slot (through weights) is also used. In experiments the algorithm is tested using real-world problem instances from the University of Melbourne (to which it finds substantially better results than the university's existing methods) and also various benchmark problem instances such as the Carter instance set, the results of which compare well to other methods.

Finally, another evolutionary approach – this time for university course timetabling – has been proposed by Paechter *et al.* in [86]. Here, the authors describe a memetic approach whereby an evolutionary algorithm is supplemented with an additional local search routine that aims to improve each timetable when it is being built, with the results then being written back into the chromosome (i.e. it is a Lamarckian approach). Additionally in this algorithm, rather than break any hard constraints, events that cannot be feasibly assigned to any of the available timeslots are left to one side unplaced. Soft constraint violations are then also penalised through the use of weightings that can be adjusted by the user during the search (as we mentioned in Section 2.3.1). This algorithm has been successfully used for module timetabling at Napier University – a problem that typically involves trying to schedule around 2000+ events into 45 timeslots and around 180 rooms.

One interesting aspect of this approach is the authors' use of sequential evaluation. When this algorithm is run, the user is given a choice as to whether he-or-she wants to give special priority to the task of inserting the unplaced events into the timetable. If this option is not taken, then unplaced events are simply treated as an additional soft constraint by the algorithm; however, if it *is* taken, then when two candidate timetables are compared, the algorithm always deems the one with the least number of unplaced events as the fitter. However, ties are then broken by then looking at the penalties caused by each of the timetable's soft constraint violations. This particular approach means that many of the problems encountered when judging a timetable's quality through a single numerical value alone (as is the case with one-stage optimisation algorithms for timetabling – see Section 2.3.1) can be avoided. Note, however, that this method of evaluation is only useful for algorithms where it is sufficient to know the ordering of a set of candidate solutions, rather than an explicit numerical quality-score for each (in this case, the authors use binary tournament selection with their evolutionary algorithm); it is thus perhaps less well suited to other types of optimisation algorithms.

2.4 Conclusions

Concluding this chapter, it should be clear to the reader that timetabling problems can be – and indeed have been – addressed using a variety of different computational approaches. In our review of the literature, we have devoted the majority of our discussion towards the application of metaheuristics to these problems, and have taken special note of the different ways in which these sorts of algorithm might be adapted for dealing with and distinguishing between the hard and soft constraints of a particular problem. Consequently, we have suggested that metaheuristic algorithms for timetabling can be separated into three main classes – one stage optimisation algorithms, two-stage optimisation algorithms, and algorithms that allow relaxations – although it is worth noting again that this classification method is not concrete, and it could well be the case that some algorithms might fit into more than one of the classes. It is also worth stressing that although each of these schemes will each have their relative advantages and disadvantages; it is probably not the case that any particular approach is universally superior to any other. Instead, it is more likely that certain approaches might be more suited to certain *types* of problem-situations and certain types of user requirements. It would thus seem reasonable to assume that when choosing a particular approach for one's own timetabling problem, these issues should therefore be given the most consideration, especially bearing in mind that the solutions to practical problems will inevitably have to be used by real people. It goes without saying, however, that it is also desirable for an algorithm to be fast, reliable and robust whenever possible.

Another noticeable trait from our review is the fact that in many cases, algorithms that have been proposed to solve a particular timetabling problem will have only been tested on a small number of benchmark instances and/or on the authors' own problem instances. This is particularly noticeable in the case of course-timetabling where, unlike exam timetabling, there has not been the adoption of a commonly-used set of benchmark problem instances. (That is, arguably, until recently – see the next chapter). From a practical perspective, this situation is perhaps understandable, because if a timetabling problem needs to be solved at the authors' own institution(s), they will, of course, be more motivated in solving this problem, rather than spending their time trying to solve other people's. However, from a research point of view, this characteristic will often make it very difficult to assess how well an algorithm is capable of performing in comparison to others, as no real benchmarking criteria will exist. Also, in many cases, other researchers will have no idea whether or not the instances used in a particular study are actually "hard" or not. These difficulties are in contrast to many other types of problems faced in operations research (such as the travelling salesperson problem and the bin packing problem, for example) where we will often have standardised problem definitions, together with an abundance of different problem instance libraries available for benchmarking algorithms (see for example [3, 5]).

As we have seen, it is also quite common – particularly in course timetabling applications – for authors to state that their algorithm was able to produce better timetables than the manually-produced solutions previously used by the institution. However, there are actually a number of potential pitfalls when using an argument such as this to justify an algorithm. First, this sort of claim involves making the explicit assumption that the university's criteria of measuring timetable "goodness" (whatever these might be) have all been effectively captured by the algorithms objective function. However, it could be the case that often, the people who construct timetables by hand may well make certain choices subconsciously and, consequently, may not explicitly described these processes and criteria to the algorithm designer. Secondly – and perhaps more obviously – the statement that an algorithm is better than a human-designed timetable is, of course, only actually meaningful when the reader is also given some indication of the skill-level of the human in question. This information is not normally provided, however.

It seems, therefore, that in this field there is a real need for some standardised course timetabling problem instances that can be used for the effective analysis and comparison of various different timetabling algorithms. In the next chapter we will look at a particular version of a timetabling problem that is intended for just this purpose.

3: Case Study: A Benchmark Timetabling Problem

For the majority of this thesis we will be focussing our attentions on a specific benchmark version of the university course timetabling problem. This problem-version was originally defined in 2001 so that it could be used for various research purposes by the Metaheuristics Network [6], and was intended to overcome some of the common ambiguities and inconsistencies that currently exist in the study of automated course timetabling. However, in 2002, it was also used for the International Timetabling Competition [2], of which further details will be given later. Formulated by Ben Paechter, the actual problem is closely based on many typical real-world timetabling problems, but is also slightly simplified. Although, from the outset, it was acknowledged that such simplifications were not wholly ideal, there were a number of reasons why this was done. Firstly, as we have just mentioned, the problem was intended for research purposes, particularly with regards to analysing what actually happens in algorithms designed to solve these sorts of problems. (In many cases, real problems are often too complicated and messy to allow researchers to study these processes in sufficient detail.) Second, the large number of hard and soft constraints that are often found in real-world problems will usually make the process of writing code (or updating existing programs to be suitable) a very long and arduous process for timetabling researchers. Third, as we saw in the last chapter, typically many of the constraints encountered in real-world problems are idiosyncratic and will only relate to one-or-two institutions. Thus, their inclusion in a particular problem set will not usually allow us to learn much about timetabling problems in general terms.

What is important to note, therefore, is that the timetabling problem-version that we are choosing to study is offering a compromise: a variety of common real-world aspects of timetabling are included (as we will see), yet for ease of scientific investigation, many of the messy fine-details found in practical problems have been removed.

In this chapter we will conduct a detailed case study of this particular timetabling problem-version and will look at various algorithms that have been proposed for it. In the following section we will first describe this particular problem and will also go over some of the various terms and notations that will be used throughout this thesis. Next, in Section 3.2 we will then provide a description and rationale of some of the data structures and encoding methods that we will be used in our own algorithms for this problem-version, to be described in later chapters. In sections 3.3 and 3.4 we will then review the various research papers that have already been proposed for this problem-version in the literature.

Finally, note that for convenience, throughout this chapter we will the acronym UCTP (University Course Timetabling Problem) to exclusively refer to this particular problem-version. Indeed, unless explicitly stated otherwise, this notation will also apply for the remainder of the thesis.

3.1 Problem Description and Analysis

A problem instance of the UCTP consists of a set E of n events that are to be scheduled into a set of timeslots T and a set of m rooms R, each with an associated seating capacity. We are also given a set of students S, and each student in S is required to *attend* some subset of E. (Pairs of events are said to *conflict* when a student is required to attend them both.) Finally, we are given a set of rooming features F, which are intended to represent real-world features such as wheel-chair access, computing facilities etc. Certain features are *required* by each event and are *satisfied* by certain rooms.

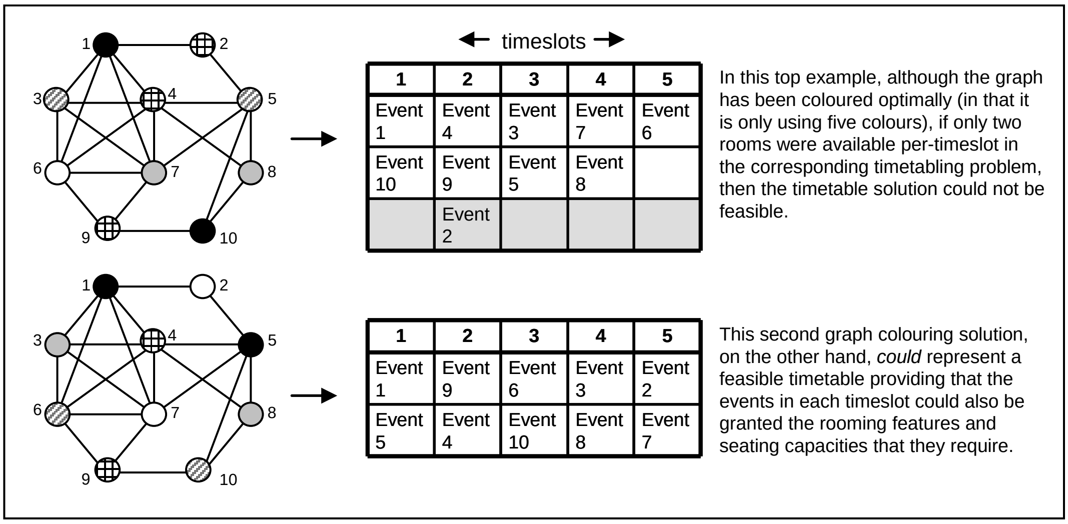
In this problem, in order for a timetable to be *feasible*, it is necessary that every event $e_1,\ldots,e_n$ is assigned to exactly one room $r_1,\ldots,r_m$ and exactly one of t timeslots (where in all cases $t \leq 45$, which is to be interpreted as five days of nine timeslots), such that the following three hard constraints are satisfied:

HC$_1$: No student is required to attend more than one event at any one time (or, in other words, conflicting events should not be assigned to the same timeslot);

HC$_2$: All events are to be assigned to *suitable* rooms. That is, all of the features required by an event are satisfied by its room, which must also have an adequate seating capacity;

HC$_3$: Only one event is assigned to any one room in any timeslot (i.e. no *double-booking* of rooms is allowed).

Note that the presence of the HC$_1$ – the usual event-clash constraint – makes the task of finding a feasible timetable similar to the graph colouring problem, as mentioned in the previous chapter. However, as we demonstrate in fig. 3.1, in this case the presence of HC$_2$ and HC$_3$ now add some further issues. First, we must now also ensure that no more than *m* events are assigned to any one timeslot (i.e. colour class); secondly, we need to make sure that every event in a given timeslot can also be given the rooming resources that it requires. From a pure graph colouring perspective, the addition of these extra constraints means that many feasible colourings that use *t* or fewer colours might still not actually correspond to feasible timetables.



1	2	3	4	5
Event 1	Event 4	Event 3	Event 7	Event 6
Event 10	Event 9	Event 5	Event 8	
	Event 2			

In this top example, although the graph has been coloured optimally (in that it is only using five colours), if only two rooms were available per-timeslot in the corresponding timetabling problem, then the timetable solution could not be feasible.

1	2	3	4	5
Event 1	Event 9	Event 6	Event 3	Event 2
Event 5	Event 4	Event 10	Event 8	Event 7

This second graph colouring solution, on the other hand, *could* represent a feasible timetable providing that the events in each timeslot could also be granted the rooming features and seating capacities that they require.

Fig. 3.1: Demonstrating the effects that hard constraints HC$_2$ and HC$_3$ have on the relationship between the UCTP and the underlying graph colouring problem.

It is worth noting that hard constraints HC$_2$ and HC$_3$ do, however, provide us with some additional clues about lower bounds that are not necessarily present in general graph colouring problems. First, it is easy to see that no feasible timetable using fewer than $\lceil n/m \rceil$ timeslots (colours) can possibly exist, because to use less than this figure would imply that either too many events have been assigned to one or more of the timeslots, or that some of the events have not been assigned to the timetable at all. Additionally, and in a similar vein, we can also deduce that if $n > (m \times t)$, then a problem instance will definitely not be solvable because this will obviously mean that there are not enough places[8] for all the events to be assigned. There are also various other pieces of information that we can infer from a UCTP instance: for example if we have some subset of events G that all require some combination of room features (or a room capacity) that only one room $r \in R$ actually satisfies, then it is easy to see that the instance can only be solvable if $|G| \leq t$.[9]

In addition to the hard constraints outlined above, in this problem there are also three soft constraints to be considered. These are as follows:

SC$_1$: No student should be required to attend an event in the last timeslot of a day;

SC$_2$: No student should sit more than two events in a row;

SC$_3$: No student should have a single event in a day.

Note that each of these soft constraints is slightly different (indeed, this was done deliberately by the problem formulator): violations of SC$_1$ can be checked with no knowledge of the rest of the timetable; violations of SC$_2$ can be checked when building the timetable; and violations of SC$_3$ can only be checked once all events have been assigned to the timetable.

Formally, we work out the number of soft constraint violations in the following way. For SC$_1$, if a student has a class in an end-of-day timeslot, we count this as one penalty point. (Naturally, if there are *x* students in this class, we consider this as *x* penalty points.) For SC$_2$ meanwhile, if one student has three events in a row we count this as one penalty point. If a student has four events in a row we count this as two, and so on. Note that adjacent events occurring over two separate days are not counted as a violation. Finally, each time we encounter a student with a single event on a day, we count this as one penalty point (two for two days with single events etc.). Our soft constraint evaluation function is simply the total of these three values.

For reference purposes, in Table 3.1 we outline the notation that will be used in our analysis of the UCTP throughout this thesis. When referring to a specific timetable we will use the term "feasible" to strictly denote a timetable in which all of the *n* events have been assigned to one of the *m* rooms and one of the *t* timeslots, so that none of the hard constraints has been violated. As we will see, however, in some of the later chapters we will

[8] For convenience, at various points in this thesis we will use the term "place" to denote a room/timeslot pair. (More formally, the set of all places $P = T \times R$)

[9] It is likely that various other lower bounds can be identified for any given UCTP instance with regards to both the hard and the soft constraints. Although we have provided some examples here, however, the identification of such bounds is not the main purpose of this thesis. Indeed as we will see, in almost all experiments conducted in this work we will make use of instances where we know feasibility to be obtainable.

also be using algorithms that allow some flexibility in the number of timeslots being used by a timetable. An extra variable *s* is thus also included in the table with an attached description. Finally, in this thesis we will use the term “perfect” to describe a timetable that is both feasible and which has no violations of the soft constraints.

Table 3.1: Description of the Main Notation used in This Thesis When Considering the UCTP

Name	Description
n	Number of events in the problem instance
m	Number of rooms in the problem instance
t	Maximum number of timeslots permitted in a feasible timetable (in all cases t is a constant 45, comprising five days of nine timeslots).
s	Variable used in some later chapters to denote the number of timeslots being used by a particular timetable
P	The set of *places* (i.e. room/timeslot pairs), where $P = R \times T$

3.2 Search Space Issues and Pre-processing

In simple terms, the total number of ways of assigning n events to p places (remembering that a *place* refers to each room/timeslot pair) is p^n. In anything but trivial problem instances, however, the vast majority of these assignments are likely to contain some level of infeasibility. In particular, it is worth noting that due to the presence of HC_3 above, only assignments in which all events have each been assigned to their own *unique* place even have the *possibility* of being feasible. For this reason, throughout this thesis, we therefore choose to encode all timetables using a matrix representation. In this scheme each timetable is represented by a two-dimensional matrix (i.e. grid) in which rows represent rooms and columns represent timeslots. Each cell in the matrix (i.e. place in the timetable) can then be blank or can contain *at most* one event. (See fig. 3.2 for an example). Note then, that this method of encoding a timetable allows us to disregard the third hard constraint of the UCTP altogether because, due to the latter characteristic above, it is now impossible to double-book a room. Additionally, beyond trivial instances this method of encoding will also drastically reduce the size of the search space that any algorithm will need to navigate.

In order to demonstrate this latter claim, it is necessary to take note of the fact that the number of ways of assigning n events to the p cells of a matrix in the manner we have described, is exactly:

$$\frac{p!}{(p-n)!} \tag{3.1}$$

Hence, assuming that *p* and *n* are positive integers such that $p \geq n$, it is sufficient to show that $p^n \geq$ (eq. (3.1)). This can be easily demonstrated by noting that:

$$\begin{aligned} \frac{p!}{(p-n)!} &= \frac{p(p-1)(p-2)\ldots(p-(n-1))(p-n)(p-(n+1))\ldots*2*1}{(p-n)(p-(n+1))\ldots*2*1} \\ &= p(p-1)(p-2)\ldots(p-(n-1)) \end{aligned} . \tag{3.2}$$

Hence, we are asking if:

$$p^n \geq p(p-1)(p-2)\ldots(p-(n-1)), \tag{3.3}$$

which it clearly is.

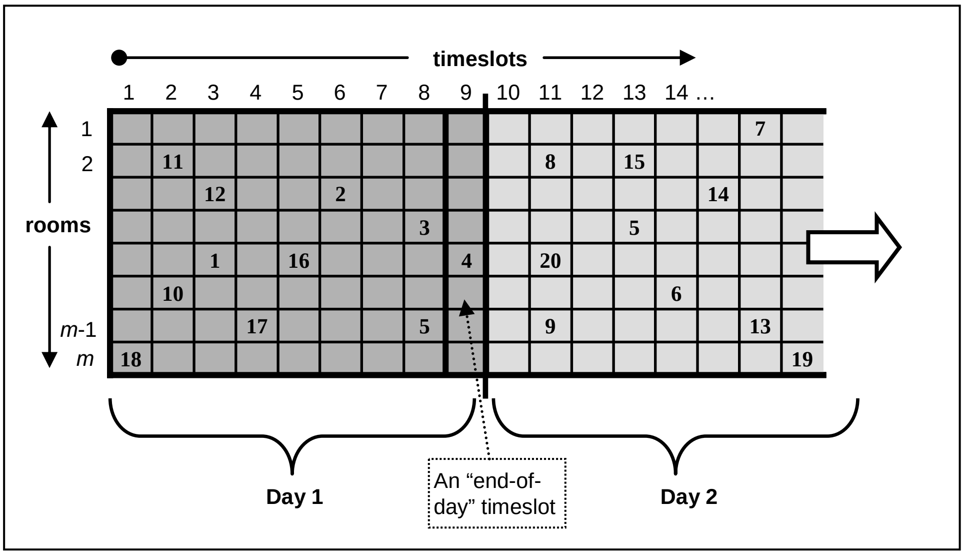


Fig. 3.2: A demonstration of the matrix representation for timetables used throughout this thesis. Here, event 11 has been assigned to room 2 and timeslot 2, event 8 has been assigned to room 2, timeslot 11, and so on. Also indicated in this diagram is the presence of the end-of-day timeslots (which will occur in timeslots 9, 18, 27, 36 and 45). These might be considered slightly different to the remaining forty timeslots, because events that are assigned to these will automatically cause soft constraint SC_1 to be violated.

As well as making use of a two-dimensional matrix representation in this thesis, in all of our algorithms (that we will see in later chapters) we also choose to carry out some useful pre-compilation steps that are intended to help speed up each of the algorithms' remaining procedures. These steps involve the construction of two additional matrices: the *event-room* matrix and the *conflicts* matrix. The Boolean ($n \times m$) *event-room* matrix is used to indicate which rooms are suitable for which events and can easily be calculated for an event *i* by simply identifying which rooms satisfy both the seating capacity and the features required by *i*. Thus if, room *j* is deemed suitable for *i*, then element (*i*, *j*) in the matrix is marked as true, otherwise it is marked as false. Meanwhile, the ($n \times n$) *conflicts* matrix can be considered very much like the standard adjacency matrix used for representing graphs and, for our purposes, indicates which pairs of events conflict (i.e. that cannot be scheduled into the same timeslot). Thus, if event *i* and event *j* have one of more common student, then elements (*i*, *j*) and (*j*, *i*) in the matrix are marked as true, otherwise they are marked as false. Additionally, as a final step in this procedure (and following the suggestions of Carter [28]), we are also able to add some further information to the conflicts matrix: note that if two events *i* and *j* do not conflict, but both can only be assigned to the same single room *r*, then it is easy to deduce that there can also exist no feasible timetable in which *i* and *j* are assigned to the same timeslot. Thus, we may also mark elements (*i*, *j*) and (*j*, *i*) as true in the conflicts matrix as well.

When encoding timetables using the matrix representation, the presence of the *event-room* and *conflicts* matrices makes it very easy and inexpensive to check for violations of the remaining two hard constraints of the UCTP. In order to check whether a timetable contains a violation of HC_1, for example, it is simply necessary to check each column (i.e. timeslot) of the timetable matrix in turn, and identify whether any pair of events within it correspond to a positive entry in the *conflicts* matrix. Checking for violations of HC_2 is also simple as we only need to ensure that every event *i* in the timetable matrix has been assigned to a row *r* such that the corresponding entry in the *event-room* matrix – i.e. element (*i*, *r*) – is true.

3.3 Initial Work and the International Timetabling Competition

It was Rossi-Doria *et al.* who conducted one of the first studies into the UCTP in 2002 [97]. In this research paper, the authors proposed five different metaheuristic-based algorithms (namely, evolutionary algorithms, ant colony optimisation, iterated local search, simulated annealing, and tabu search) for the UCTP and then attempted to provide an unbiased comparison between them. In some of these algorithms, the satisfaction of both hard and soft constraints was attempted simultaneously; for example, in the case of the evolutionary algorithm the following weighted sum function was used:

$$f(tt) = \alpha h(tt) + s(tt) . \tag{3.4}$$

Here $h(tt)$ indicates the number of hard constraint violations in a timetable tt, $s(tt)$ indicates the number of soft constraint violations, and α is a constant that, in the case of Rossi-Doria *et al.*, was always set to a figure larger than the maximum possible number of soft constraint violations. Meanwhile, other algorithms in the study, such as the iterated local search and simulated annealing approaches, made use of a two-stage timetabling approach (see Section 2.3.2), whereby once feasibility was obtained, no further violations of the hard constraints were allowed to take place.

Upon conducting a thorough comparison of these five metaheuristic algorithms, it was observed by Rossi Doria *et al.* that in the cases where feasibility was generally achieved, the two-stage algorithms tended to produce results that were significantly better than the remaining metaheuristic approaches. This feature, along with some other observations made during the comparison, caused the authors to offer two interesting conclusions about the particular problem-version being studied. These were as follows:

- "The performance of a metaheuristic [with the UCTP], with respect to satisfying hard constraints and soft constraints may be different";
- "Our results suggest that a hybrid algorithm consisting of at least two phases, one for taking care of feasibility, the other taking care of minimizing the number of soft constraint violations, is a promising direction."

Following this work, the International Timetabling Competition [2] was then organised and run in 2002-3. The idea of this competition was for participants to design algorithms for this timetabling problem, which could then be compared against each other using a common set of benchmark instances and a fixed time limit, measured in elapsed time. The time limit was determined separately for each of the participants' computers using a program that measured various characteristics of their machine during execution, and the effectiveness of this benchmarking program was then later verified by running the best competition entries on a single standard machine. (A number of points regarding the pros and cons of using a time limit for this purpose will be discussed later in Chapter 6.) Upon the close of the competition, the participant whose algorithm was deemed to perform best across these instances (and checked against a number of unseen instances that were only available to the organisers) was awarded a prize. The exact criteria for choosing the winning algorithm when judged across all of the problem instances can be found on the competition's web site [2].

The twenty problem instances used for the competition were made using an instance generator designed by Ben Paechter. This generator requires eight command line parameters to be defined (listed in Table 3.2) together with a random seed. When run, the generator then produces a problem instance-file together with a corresponding solution-file. Additionally, the generator also offers the user the choice of (a) producing a problem instance with a corresponding *perfect* solution, or (b) producing a problem instance only with a corresponding *feasible* solution. If the second option is chosen, then obviously it will not always be known if the produced problem instance will have an obtainable perfect solution. For the International Timetabling Competition the problem instances consisted of between 200 and 300 students, 350 to 440 events, and 10 or 11 rooms, and all were ensured to have at least one perfect solution[10]. As usual, the number of timeslots was fixed at 45. Additionally, in 13 of the 20 instances the number of events *n* was made equal to the number of rooms multiplied by 40, meaning that optimal solutions to these instances had to have 40 timeslots completely filled with events (as, obviously, perfect solutions would not have any events assigned to the five end-of-day timeslots).

Table 3.2: Description of the Parameters Used with the UCTP Instance Generator.

	Parameter Description
(1)	Number of events
(2)	Number of rooms
(3)	Number of room-features
(4)	Approximate average number of features per room
(5)	Approximate average percentage of features required by events
(6)	Number of students
(7)	Maximum number of events per student
(8)	Maximum number of students per event

Another important aspect of the competition was the way in which the organisers chose to evaluate the timetables. The official rules of the competition stated that timetable quality would only be measured by looking at the number of soft constraint violations. Thus, if a timetable (a) contained any hard constraint violations, (b) used any extra timeslots, or (c) left any events unplaced, then it would immediately be considered worthless (the same judging criterion was also used when comparing various metaheuristics in the earlier work of Rossi-Doria *et al.* [97]). Indeed, the organisers imposed a strict rule

[10] Note, in fact, that in this case if a UCTP instance is known to have at least one perfect solution, then we actually know that there are at least 5! = 120 perfect solutions, because the soft constraints defined for this problem do not actually carry across different days. Thus, we can permute the days of any perfect timetable and the result is also guaranteed to be perfect.

that stated that participants would only be allowed to submit an entry to the competition if their algorithms could find feasibility on all twenty instances. Given this criterion, and also taking into consideration the conclusions of Rossi-Doria *et al.* [97] quoted above, it is perhaps unsurprising then, that many of the entrants to this competition therefore elected to use the two-stage timetabling approach.

The competition, which ended in March 2003, eventually saw a total of 21 official entries, plus 3 unofficial entries (the latter three were not permitted to enter the competition because they were existing members of the Metaheuristics Network). The submitted algorithms used a variety of techniques including simulated annealing, tabu search, iterated local search, ant colony optimisation, hybrid approaches, and heuristic construction with backtracking. As it turned out, according to the competition criteria, many of the best algorithms, including the top three, made use of a two-stage metaheuristic approach. These operated by using various methods to first find feasibility, followed by assorted neighbourhood search algorithms (such as simulated annealing, tabu search, and so on) to then deal with soft constraints. The winning algorithm, for example, was a two-stage, simulated annealing-based algorithm by Philipp Kostuch of Oxford University and details of this algorithm, plus many of the others submitted can be found at the official competition web page [2].

3.4 Review of Relevant Research

Since the running of the International Timetabling Competition, a number of good papers have been published regarding this particular timetabling problem, some of which have described modifications to algorithms that were originally entered into the competition. In this subsection we will now review some of the most notable and relevant works on this problem.

In [11], Arntzen and Løkketangen have described a two-stage tabu search-based approach. In the first stage, their algorithm uses a constructive procedure to build an initial feasible timetable and operates by taking events one by one, and assigning them to feasible places in the timetable, according to some specialised heuristics that also take into account the potential number of soft constraint violations that such assignments might cause. The order that events are inserted is determined dynamically, and decisions are based upon the state of the current partial timetable. The authors report that these heuristics successfully build feasible timetables in over 90% of runs with the competition instances. Next, with feasibility having been found, Arntzen and Løkketangen opt to use tabu search in conjunction with simple neighbourhood operators in order to optimise the soft constraints. In the latter stage, feasibility is always maintained by making use of neighbourhood operators that are restricted so as never to break any of the hard constraints.

Cordeau, Jaumard, and Morales (available at [2]) have also used tabu search to try and satisfy the soft constraints of the UCTP. However, this method is slightly different to Arntzen and Løkketangen above, because, when dealing with the soft constraints, the algorithm also allows a small number of hard constraints to be broken from time to time. The authors achieve this by using the same weighted sum function given in eq. (3.4) above. In this case, however, and rather like the methods used by Schaerf [99], the parameter α (which, we remember, is used to penalise violations of the hard constraints) is allowed to vary so that when the number of hard constraint violations in the timetable rises above a fairly small amount, α is increased to a value so that further infeasibilities will generally not be allowed. It is hoped then, that such a mechanism will help to control the level of infeasibility in the timetable, and it is claimed by the authors that such a scheme allows freer movement about the search space.

Another approach similar to these has also been proposed by Burke, *et al.* in [23]. In this paper the authors make use of the Great Deluge algorithm of Dueck [48] in order to try and reduce the number of soft constraint violations. Again, this is done whilst always remaining in feasible areas of the search space. In contrast to other neighbourhood search algorithms such as simulated annealing or tabu search, however, one advantage of this approach is that only a single parameter needs to be defined before running the algorithm: the amount of available search time. Given this value, the algorithm is then able to adapt the intensity of the search so that it will only start to converge on local (or global) optima when this time limit is being approached. As can be expected, in general the algorithm is reported to return better quality results when the amount of available search time is increased, and the algorithm is reported to return good results on the competition benchmark instances.

Socha, Knowles, and Sampels have also suggested ways of applying the ant colony optimisation metaheuristic to this problem. In [102, 103], the authors have presented two ant-based algorithms – an Ant Colony System and a $\mathcal{MAX}$-$\mathcal{MIN}$ system – and have provided a qualitative comparison between them. At each step in both of the algorithms, every ant first constructs a complete assignment of events to timeslots using heuristics and pheromone information (the latter which is accumulated in previous iterations of the algorithm). Timetables are then further improved by way of a local search procedure, originally proposed in [96]. The only major differences between the two approaches are, in fact, the way that heuristic and pheromone information is interpreted, and the approaches used for updating the pheromone matrix. However, tests using a range of problem instances indicate that the $\mathcal{MAX}$-$\mathcal{MIN}$ system generally achieves better results. A description of the latter algorithm – which was actually entered unofficially to the timetabling competition – can also be found at [2], where good results are reported.

Another good study looking at this problem is offered by Chiarandini *et al.* in [34]. In this research paper, which also outlines an unofficial competition entry, the authors present a broad study and comparison of various different heuristics and metaheuristics for the UCTP. After experimenting with a number of different approaches and also parameter settings (the latter which was done automatically using a tuning method proposed by Birattari *et al.* [13]), their decided method is a two-stage, hybrid algorithm that uses a variety of different search methods. In the first stage, constructive heuristics are initially employed in order to try and find a feasible timetable, although, as the authors note, these were usually unable to find complete feasibility unaided. Consequently, local search and tabu search schemes are also included to try and help eliminate any remaining hard constraint violations. Feasibility having been achieved, the algorithm then moves onto satisfying the soft constraints and conducts its search in exclusively feasible areas of the search space. It does this first by first using variable neighbourhood search and then by using simulated annealing. The annealing phase is reported to use more than 90% of the available run time of the total algorithm, and a simple reheat function for this phase is also implemented (this operates by resetting the temperature to its initial starting value when it is felt that the run is starting to stagnate). Extensive use of delta evaluation [92] is also made in order to try and speed up the algorithm, and according to the authors the final algorithm achieves results that are significantly better than the official competition winner.

Kostuch has also used simulated annealing as the main construct in his timetabling algorithm, described in [69]. Based upon his winning entry to the timetabling competition, this method works by first gaining feasibility via simple graph colouring heuristics (plus a series of improvement steps if these heuristics prove inadequate) and then uses simulated annealing to try and satisfy the soft constraints, first by ordering the timeslots, and then by swapping events between timeslots. One of the interesting aspects of Kostuch's approach is that when a feasible timetable is first being constructed, efforts are made to try and schedule the events into just forty of the available forty-five timeslots. Indeed, as the author notes, five of the available timeslots will automatically have penalties attached to them (due to the soft constraint SC_1) and so it could be a good idea to try and eliminate these from the search altogether from the outset. The author then only allows the five end-of-day timeslots to be opened if feasibility using forty timeslots cannot be achieved in reasonable time. (In reported experiments using the twenty competition instances the events in nine of the instances were always scheduled into forty timeslots.) Another interesting aspect of this approach is the way in which the cooling schedule for the SA stage is determined: in this case, the algorithm's time limit is used in order to calculate a temperature decrement rule that will allow a cooling that is as slow as possible, but that will also still spend enough time at low temperatures for sufficient movement towards the optimal to be possible. (This way of determining the search's characteristics according to the amount of computation time that is available is similar to the Great Deluge approach of Burke *et al.* [23] that we mentioned earlier.) Finally, the author also uses a slight "trick" by making note of the fact that in some of the competition instances, there are a small number of events that are actually empty (i.e. with no students). Thus, because these do not ultimately have any bearing on the number of soft constraint violations in a particular candidate solution, the author chooses to remove these from the timetable altogether before starting the SA process, and only reinserts them again at the end of the run (usually into the five end-of-day slots). The author suggests that such a "trick" helps to free up the timetable's resources and thus allows a better search to be performed during the SA phase.

3.5 Conclusions

From the review presented in this chapter, and in particular, bearing in mind the suggestions of Rossi-Doria *et al.* mentioned in Section 3.3, it should be appreciable that the two-stage timetabling approach seems to offer some promise for the UCTP; indeed, as we have seen it is certainly the case that many of the best algorithms currently available in the literature have used it in some form or another. However, it is also noticeable that one of the critical requirements of this type of approach is that in the first stage of such an algorithm, we still obviously require a reliable method of achieving feasibility – if indeed feasibility *is* achievable. Looking at the results that have been reported for the twenty competition instances, it would seem that this requirement is fairly easy to meet in most cases. (Note again that these problems were deliberately constructed with soft constraints in mind and were not intended to be too difficult to solve with regards to finding feasibility.) However, as a natural consequence of this characteristic, this has also meant that the vast majority of work conducted so far on the UCTP has pertained to the analysis of algorithms specialised in satisfying the *soft* constraints of the problem. But, of course, this still leaves a major issue of concern. How we can ensure that we will also be able to find feasibility when other, "harder" problems are encountered? As we saw earlier in this chapter, even the problem of finding a feasible timetable is NP-hard in this case, and it should not therefore be treated too lightly. Thus we believe that there are justifications and needs for more powerful search algorithms that specialise in finding feasible timetables, which can also cope across a much wider range of problem instances. An algorithm looking to achieve this will be the main aim of the next chapter.

4: Finding Feasibility Using a Grouping Genetic Algorithm

In this chapter, we will concern ourselves with the task of producing an effective algorithm for the first stage of the two-stage timetabling strategy: that is, an algorithm that ignores the soft constraints of the UCTP and simply tries to find a *feasible* solution by satisfying all of the hard constraints of the problem. For this purpose, we will mainly be examining the applicability of the so-called “Grouping Genetic Algorithm” (GGA) – a fairly well-known type of evolutionary algorithm that has been successfully applied to a number of different computational problems. In our analysis of this algorithm, however, we will actually see that when applied to this timetabling problem, there are, in fact, a number of scaling-up issues; in particular, we will see that the GGA can actually produce unsatisfactory performance in some cases, and we will offer a detailed description of why this might be so.

As a by-product of these investigations, we will also introduce a method for measuring population diversities and distances between individuals with the “grouping representation” used with this algorithm. We also look at how such an algorithm might be improved: first, by investigating the effects that various different fitness functions have on the algorithm, and second by introducing an additional heuristic search operator (making in effect a type of grouping *memetic* algorithm). Although we will see that the addition of such mechanisms can improve the performance of the GGA in some cases; eventually we will see that, in many cases, better results can actually be achieved when we remove the grouping genetic operators from the algorithm altogether. This latter observation, in particular, will serve to highlight some of the issues raised in this chapter concerning possible limitations of the general GGA approach.

4.1 Grouping Problems and Grouping Genetic Algorithms

Grouping genetic algorithms (GGAs) may be thought of as a special type of evolutionary algorithm specialised for *grouping problems*. Such problems are those where the task is to partition a set of items U into a collection of mutually disjoint subsets (or groups) u_i of U, such that:

$$\cup u_i = U \quad \text{and} \quad u_i \cap u_j = \varnothing, \quad i \neq j\,. \tag{4.1}$$

As well as this definition, it is also usual for grouping problems to feature some problem-specific constraints that define valid and legal groupings. For example, the bin packing problem requires that a set U of items of varying sizes be packed (grouped) into a minimal number of bins of a fixed capacity, such that no bin is overfilled. Another example is the graph colouring problem – in this case the task is to partition the set U of nodes (of a graph) into a minimal number of groups (i.e. colours), ensuring that none of these groups contains a pair of nodes with a common edge. Further examples of grouping problems include:

- The Bin Balancing (Equal Piles) Problem [56];
- The Edge Colouring Problem [67];
- Various versions of the Frequency Assignment Problem [8];
- The *k*-way Graph Partitioning Problem [111];
- The Economy of Scales Problem [58];
- The Pick-up and Delivery Problem [90];
- The Cell Formation Problem [17].

From an optimisation point-of-view, many grouping problems, including all of the above examples, are NP-hard. Given this fact, there is therefore scope for the application of approximation algorithms – such as evolutionary algorithms and other metaheuristics – to these sorts of problems.

In [54] and [58], Emanuel Falkenauer – the creator of the GGA – argues convincingly that when considering grouping problems like those just mentioned, the so-called "traditional" genetic operators and representations often used in evolutionary computation can actually be highly redundant, not least due to the fact that the usual sorts of operators are *item*-oriented rather than *group*-oriented. The upshot is a general tendency for these operators to recklessly break up the building-blocks that we might otherwise want promoted.

As an example, consider the traditional item-based encoding scheme, where a chromosome such as 31223 represents a solution where the first item is to appear in group three, the second in group one, the third is in group two, and so on. (As a point of interest, this has been used with timetabling in [39], [93] and [97].) First of all, when used with a grouping problem, such a representation immediately brings disadvantages because it goes against one of the fundamental design principles for evolutionary algorithms: The Principle of Minimum Redundancy [89], which states that each member of the search space should be represented by as few distinct chromosomes as possible. To expand upon this point, note that the ordering or naming of groups is not actually important in a typical grouping problem (e.g. the ordering/naming of the colours in a candidate solution for the graph colouring problem does not have any influence on the actual quality of the solution being represented); instead what *is* important is *how* the items are grouped. However, using this particular encoding, given a candidate solution that uses s groups (in the example chromosome above, $s = 3$), there are, in fact, another ($s! - 1$) distinct chromosomes that represent exactly the same grouping of items, due to the various ways in which the groups can be permuted. Of course, this means that the degree of redundancy will grow *exponentially* with the number of groups being used; meaning that the search space will generally be a lot larger than it needs to be.

Next, if we were to make use of a traditional recombination operator with this encoding, we would generally see that context-dependant information would actually be passed out-of-context and, as a result, the resultant offspring would rarely resemble either of its two parents (with respect to the solutions that they represent). For example, let us apply a standard two-point crossover to two chromosomes: 3|12|22 crossed with, say, 1|23|12 would give, as one of the offspring, 32322. Firstly, this offspring no longer has a group 1 and, depending on the problem being dealt with, this may mean that it is invalid or illegal. Secondly, in this case it can be seen that the groupings that occur in the resultant offspring seem to have very little in common with either parent, and in fact this operation has resulted in more of random jump-style movement in the search space. Obviously, such features go against the general aim of a recombination operator (see fig. 4.1).

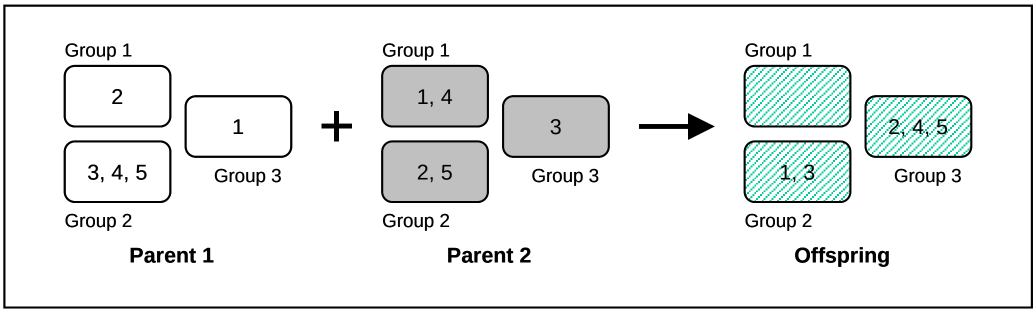


Fig. 4.1: Pictorial description of the "traditional" two-point crossover application described in the text.

Similar observations to these are also made by Falkenauer with regards to the standard sorts of mutation operators that are used with these sorts of encodings, and also with the typical genetic operators that work with permutation based encodings, such as the Partially Mapped Crossover [64], where similar problems are observed ([58], pages 85-96). On the whole, the arguments that Falkenauer presents leads him to the following conclusion: when considering grouping problems, it is essentially the groups themselves that are the underlying building-blocks of the problem and not, for example, the particular states of any of the items individually. Thus, if we are to use evolutionary computation to try to solve these problems then appropriate representations and genetic operators that allow these groups to be propagated effectively during evolution are advisable. In particular, Falkenauer argues that a successful recombination operator for these sorts of problems should allow *entire groups* to be transmitted from parent to offspring.

With this in mind, a standard GGA scheme has been proposed by Falkenauer in [54, 58]. In essence, he suggests using a specialised encoding in which the individual genes of a chromosome are represented by the *groups*, with the evolutionary operators then operating on these groups. (Note that the former point implies that a chromosome's length is determined solely by the number of groups that it is using. The implications of this fact will be examined later in the chapter.). For clarity, in fig. 4.2 we give a pictorial example of a typical GGA recombination operator. Notice that this sort of operation does indeed allow entire groups from both parents to be transmitted into the offspring. With regards to mutation, meanwhile, Falkenauer suggests that such an operator should also work with groups, and suggests three general strategies: the creation of a new group; the elimination of an existing group; or shuffling a small number of randomly selected objects among their respective groups.

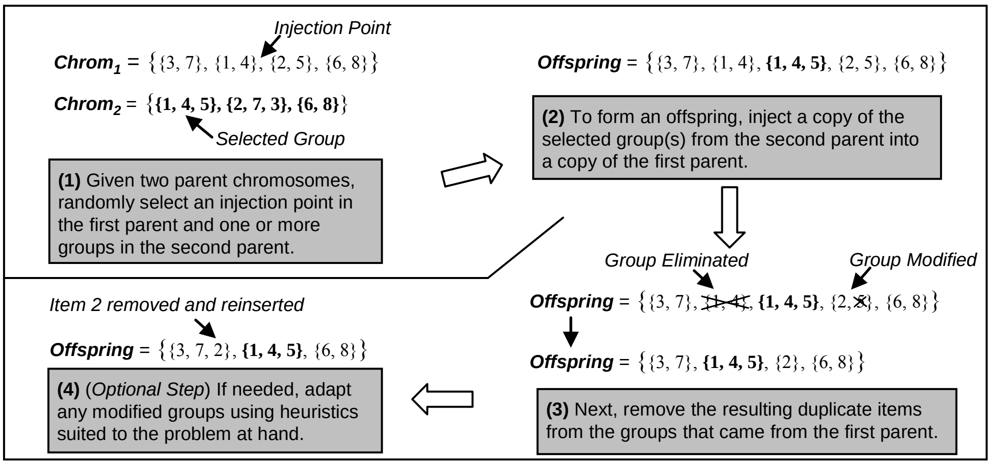

Injection Point
Chrom$_1$ = {{3, 7}, {1, 4}, {2, 5}, {6, 8}}
Chrom$_2$ = {{1, 4, 5}, {2, 7, 3}, {6, 8}}
Selected Group
(1) Given two parent chromosomes, randomly select an injection point in the first parent and one or more groups in the second parent.
Offspring = {{3, 7}, {1, 4}, {1, 4, 5}, {2, 5}, {6, 8}}
(2) To form an offspring, inject a copy of the selected group(s) from the second parent into a copy of the first parent.
Group Eliminated
Group Modified
Offspring = {{3, 7}, ~~{1, 4}~~, {1, 4, 5}, {2, ~~5~~}, {6, 8}}
Offspring = {{3, 7}, {1, 4, 5}, {2}, {6, 8}}
(3) Next, remove the resulting duplicate items from the groups that came from the first parent.
Item 2 removed and reinserted
Offspring = {{3, 7, 2}, {1, 4, 5}, {6, 8}}
(4) (Optional Step) If needed, adapt any modified groups using heuristics suited to the problem at hand.

Fig. 4.2: Falkenauer's GGA Recombination Operator.

Since the time when Falkenauer first made these suggestions in 1994 there have been a number of applications of these techniques to various grouping problems such as the bin packing and bin balancing problems by Falkenauer in [54, 56-58]; the graph colouring problem by Eiben, van der Hauw, and van Hermert in [50], and also Erben in [52]; the edge colouring problem by Khuri, Walters and Sugono in [67]; and also the exam-timetabling problem in [52]. Other example applications have also been suggested by Brown and Sumichrast [17] for the Cell Formation Problem, Rekiek *et al.* [90] for the Pickup and Delivery Problem, and Falkenauer again [58] for the Economies of Scale problem.

4.1.1 GGAs and the UCTP

It should be clear from the problem description given in Section 3.1 that the task of finding a feasible timetable for a given instance of this UCTP also constitutes a grouping problem: in this case, the "items" are the events themselves, and the "groups" are defined by the timeslots. Thus, in order for a timetable to be feasible, the events need to be grouped into at most *t* timeslots (remembering that, in this case, *t* represents the target number of timeslots = 45) such that all of the hard constraints are satisfied. (From a grouping perspective we may consider the same task to be one of partitioning the events into at most *t* groups such that (a) no group contains a pair of events that conflict, and (b) events in every group can each be assigned to their own feasible room.)

In the next section we will give a detailed description of a GGA specialised for the UCTP. The remainder of this chapter will then continue as follows: in Section 4.3 we will provide some general details of the experimental set-up used in this chapter and, using this framework, will go on to look at the effects that the various genetic operators seem to have on the quality of the search (positive and negative) in Section 4.4. Next, in Section 4.5, we will then go on to introduce a new way of measuring population diversities for this sort of representation, and will make some other general comments regarding this sort of algorithmic approach. In Section 4.6 we will then attempt to improve the algorithm: first, through the use of some new fitness functions and, second, via the introduction of a heuristic search operator. In particular, we will examine both the good and bad effects that this additional search operator might have. Next, in Section 4.7 we will investigate the consequences of actually removing the evolutionary parts of the algorithm altogether, by introducing a new algorithm that only makes use of this heuristic search operator. Finally, in Section 4.8 we outline the main conclusions of this chapter and, will present some further discussion on some of the issues raised.

4.2 Algorithm Description

4.2.1 Representation and Solution Building

For this algorithm, each timetable will be encoded using the two-dimensional matrix representation discussed in Section 3.2. Also, in addition to only allowing a maximum of one event to be assigned to a particular place (matrix-cell) as discussed, in this case we also choose to disallow any event from being inserted into a place where it causes a violation of any of the remaining hard constraints. Instead, and following a fairly typical GGA scheme [50, 52, 54, 58], we choose to open up extra timeslots (i.e. add columns to the matrix) in order to cope with events that cannot be feasibly assigned to any existing place in the current timetable. (Similar strategies have also been used in other timetabling algorithms such as Burke, Elliman and Weare's evolutionary approach in [19, 20]; and Socha and Sampels ant-based algorithm in [102, 103]). For this UCTP, it is thus easy to see that the overall aim of the GGA is to simply try and find a solution that uses no more that *t* timeslots.

As we will see, a scheme for building full solutions from empty or partial solutions is vital in the GGA algorithmic approach, not just for building members of the initial population, but also for use with the grouping genetic operators described below. A procedure, that we call Build, for achieving just this is outlined in fig. 4.3. As arguments the procedure BUILD takes an empty or partial timetable *tt* and a non-empty list of currently unplaced events *U*. Then, using the sub-procedure INSERT-EVENTS, the events in *U* are taken one-by-one (according to some heuristics, defined in Table 4.1) and are inserted into places in the timetable that are between timeslots *l* and *r* and that are feasible. Events for which there are no feasible places are ignored. Eventually then, *U* will be empty (in which case we have a complete timetable that may or may not be using the required number of timeslots), or *U* will only contain events that cannot be inserted anywhere in the current timetable *tt*. In the latter case, a number of new timeslots are therefore opened, and the process is repeated on these new timeslots. The number of timeslots that are opened is calculated in line 7 of INSERT-EVENTS in fig. 4.3. Note that the result of this calculation represents a lower bound, because we know that a maximum of *m* events can be assigned to a particular timeslot, and therefore *at least* $\lceil |U|/m \rceil$ extra timeslots will be needed to accommodate the remaining events in *U*.

```
BUILD (tt, U)
1. if (len(tt) < t)
2.     Open (t – len(tt)) new timeslots;
3. INSERT-EVENTS (tt, U, 1, len(tt));

INSERT-EVENTS (tt, U, l, r)
1. while (there are events in U with feasible places in tt
             between timeslots l and r)
2.     Choose an event e from U that has feasible places in tt;
3.     Pick a feasible place p for e;
4.     Move e to p;
5. if (U = ∅) end;
6. else
7.     Open ⌈|U|/m⌉ new timeslots;
8.     INSERT-EVENTS (tt, U, r, len(tt));
```

Fig. 4.3: Procedure for building initial solutions and also rebuilding partial solutions. In this pseudo-code *tt* represents the current timetable and *U* is a list of unplaced events of length $|U|$. The function len(*tt*) in the figure returns the number of timeslots currently being used by *tt*. Finally, and as usual, *m* indicates the number of rooms available per timeslot, and *t* represents the target number of timeslots = 45.

TABLE 4.1: THE VARIOUS EVENT AND PLACE SELECTION HEURISTICS USED WITH BUILD (FIG. 4.3)

Heuristic	Description
H_1	Choose the unplaced event with the smallest number of possible places to which it can be feasibly assigned in the current timetable.
H_2	Choose the unplaced event that conflicts with the most other events.
H_3	Choose an event randomly.
H_4	Choose the place that the least number of other unplaced events could be feasibly placed into in the current timetable.
H_5	Choose the place that defines the timeslot with the fewest events in.
H_6	Choose a place randomly.

In order to form an initial population for the GGA, the procedure Build is called for each individual. (Note that when an initial solution is being built, to start with *U* will contain the entire set of events.) At each step of Insert-Events, an event is chosen according to heuristic H_1 with ties being broken by H_3 (see Table 4.1). Next, a place is chosen for the event using heuristic H_4, with ties being broken by H_5 and further ties with H_6. Note that in our case the use of heuristics H_3 and H_6 (random choices) in the initial population generator provides us with enough randomisation to form a diverse initial population.

With regards to other heuristics described in Table 4.1, note that the rules that are used for determining the order in which events are to be inserted into the timetable are somewhat akin to the rules for determining the order in which nodes are to be coloured in Brelaz's classical Dsatur algorithm for graph colouring [16]. However, in this case we observe that H_1 also takes the issue of room allocation into account. Heuristic rule H_1 therefore selects events based on the state of the current partial timetable *tt*, and prioritises those with the least remaining feasible options. Meanwhile, rule H_2 (which we will use later), prioritises those events that have the highest conflicts degree, which – as a rule of thumb – are often the more problematic events to insert.

Our reasons for choosing the described place selection heuristics, on the other hand, are as follows: by using rule H_4 we are making the seemingly sensible choice of choosing the place that will have the least effect on the future place-options of the remaining unplaced events. Rule H_5, meanwhile, encourages events to be put into timeslots that already have large numbers of events assigned to them. This is done in the hope that the timeslots will be filled more effectively in general, thereby requiring fewer timeslots on the whole.

4.2.1.1 Aside: An Alternative Rebuild Strategy

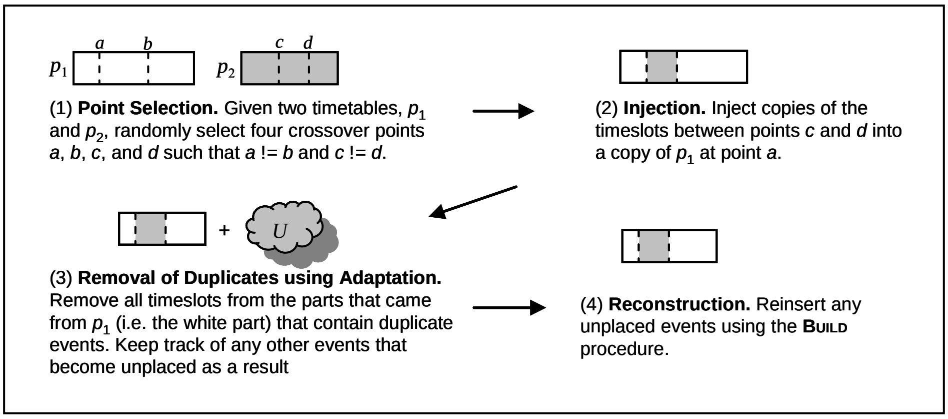
Finally, it is also worth mentioning at this point, that during our initial experiments with this algorithm we also implemented and tested a second building scheme that used the same heuristics as just discussed, but which operated in the slightly more "traditional" manner of opening timeslots in *tt* one-by-one, on the fly as soon as any event in *U* (due to preceding insertions) became unplaceable. However a detailed comparison of the two schemes revealed that the quality of the individual timetables produced by this second method was usually worse, and that the cost of this process was significantly higher. This second issue is particularly important in this case because, as we will see in the next section, a rebuilding procedure is also an integral part of the grouping genetic operators. We believe that this greater extra expense is due to the fact that, whilst looking for places for events in

U, the whole timetable (which would be continually growing) needs to be considered by the heuristics, whilst in our proposed building scheme, while *U* remains non-empty, the problem is actually being split into successively smaller sub-problems.

4.2.2 The GGA Genetic Operators

Because, as we have discussed, we have decided to consider the individual timeslots as the principal building-blocks of the problem in this case (Section 4.1), it follows that appropriate genetic operators should now be defined so that these "groups of events" can be propagated effectively during the evolutionary process. We choose to use the standard GGA recombination method (proposed in [54, 58]) modified to suit our particular needs and representation.

Fig. 4.4 depicts how we go about forming the first offspring timetable using parents p_1 and p_2 with four randomly selected crossover points *a*, *b*, *c*, and *d*. A second offspring is then formed by switching the roles of the parents and the crossover points. What is important to note about this operator is that it allows the offspring to inherit complete timeslots (the structures that we consider to be the underlying building-blocks of this problem) from *both* parent timetables.



(1) **Point Selection.** Given two timetables, p_1 and p_2, randomly select four crossover points *a*, *b*, *c*, and *d* such that *a* != *b* and *c* != *d*.

(2) **Injection.** Inject copies of the timeslots between points *c* and *d* into a copy of p_1 at point *a*.

(3) **Removal of Duplicates using Adaptation.** Remove all timeslots from the parts that came from p_1 (i.e. the white part) that contain duplicate events. Keep track of any other events that become unplaced as a result

(4) **Reconstruction.** Reinsert any unplaced events using the BUILD procedure.

Fig. 4.4: The four stages of GGA recombination – point selection, injection, removal of duplicates using adaptation, and rebuilding. Note that in order to form the second offspring, copies of the timeslots between points *a* and *b* in p_1 are injected into a copy of p_2 at point c.

During recombination, notice that as a result of stage-two (injection), there will be duplicate events in the offspring timetable, thus making it illegal. In order to correct this, we could, for example, simply go through the timetable and remove all duplicate events from the timeslots that came from p_1. However, although such an operation would result in a valid and complete offspring timetable, it is likely that the offspring would actually be of very poor quality because it would almost certainly be using more timeslots than either of its two parents, thus going against the main aim of the algorithm. Instead, we therefore choose to use the additional step of *adaptation* [59] in order to try to circumvent this issue. This process is described in stage-three of fig. 4.4 and, indeed, the same procedure has also been used in various other GGAs applications (see [50, 52, 54, 58], for example).

Finally, in the forth stage of recombination, events that have become unplaced as a result of this adaptation step are reinserted using the BUILD procedure (fig. 4.3) with heuristic H_1 being used to define the order in which events are reinserted (breaking ties with H_2 and any further ties with H_3). Places for events are then selected using the same heuristics as the initial population generator.

Our mutation operator also follows the typical GGA scheme: a small number of randomly selected timeslots are removed from the timetable and the events contained within these are reinserted using the rebuild procedure. (In our experiments the number of timeslots to remove was defined by the parameter *mr*, such that between one and *mr* distinct timeslots were always chosen.) Additionally, because we want this mutation operator to serve its normal purpose of adding diversity to the search, the order in which the events are reinserted is completely randomised (by only using heuristic H_3), with places being chosen using heuristic H_4, breaking ties with H_6.

Finally, in this GGA we also make use of an inversion operator. Similarly to other GGAs (e.g. [50, 54, 58]), this works by selecting two timeslots in a timetable at random, and then simply reverses the order of the timeslots contained between them. Note that inversion does not actually alter the number of timeslots being used, or indeed the packings of events into these timeslots. However, it may assist recombination if promising timeslots are moved closer together, as this would improve their chances of being propagated together later on[11].

4.2.3 A Preliminary Fitness Measure

Finally, for this algorithm, we need a way of measuring a timetable's quality. In our case, since we are only interested in finding feasibility, a suitable measurement need only reflect the timetable's distance-to-feasibility. As we have seen in Section 2.3, when applying metaheuristic techniques to timetabling problems, this can be measured by taking various factors into consideration such as the number of broken constraints, the number of

[11] It is probably worth noting that we could have actually implemented a uniform grouping crossover operator here as opposed to the two-point variant explained in this section. However, in this case we decided to keep all the genetic operators within the GGA design-guidelines specified by Falkenauer in [54, 58].

unplaced events, and so on. Of course, the criteria that *are* chosen will usually depend on the representation being used, and on user and/or algorithmic preference.

In the case of this GGA, because we explicitly prohibit the violation of hard constraints and, instead, open up extra timeslots as and when needed, we could therefore simply use the number of extra timeslots as a distance-to-feasibility measure. However, it is likely that such a method would hide useful information about a candidate solution, because it would not tell us anything about the number of events packed into these extra timeslots. We therefore use a more meaningful measure that we calculate by carrying out the following steps. As usual, let s represent the current number of timeslots being used in a particular timetable and let t represent the target number of timeslots (i.e. 45):

(1) Calculate the number of extra timeslots t' being used by the timetable (where $t' = s - t$);

(2) Identify the t' timeslots with the fewest events in them;

(3) Total up the number of events in these t' timeslots.

We may also think of this measure as the minimum number of events that would need to be removed from the timetable in order to reduce the number of timeslots to the required amount t. Obviously, a fully feasible timetable will have a distance-to-feasibility of zero.

4.3 Experimental Set-Up and Instance Generation

As it turned out, our initial tests with this algorithm showed that existing benchmark instances (on the web and otherwise) could easily be solved by this algorithm. For example, with the twenty competition benchmark instances, feasible solutions using a maximum of t timeslots were actually found in the initial populations of the GGA in most cases (in over 98% of our trials). Although this highlights the strength of our constructive heuristics, of course, it unfortunately tells us very little about the other operational characteristics of the algorithm. We therefore set about making some new problem instances, using the same instance generator that was used for the International Timetabling Competition. All in all, we made sixty instances, arranged into three sets of twenty: the *small* set, which contained instances of approximately 200 events and 5 rooms; the *medium* set, containing instances of approximately 400 events and 10 rooms; and the *large* set, containing instances using approximately 1000 events and 25 rooms. These instances were created with no reference to this algorithm, but were deliberately intended to be more troublesome for finding feasibility. This latter characteristic was achieved by conducting simple experiments whereby instances were created and run on two existing constructive algorithms, reported in [11] and [74]. Only instances that both of these algorithms struggled with were then considered for inclusion in the instance set. Indeed, given excess time, these two algorithms were generally unable to place around 20% to 40% of the events. Further details, including a description of how the instances were generated, plus the instances themselves, can be found online at [7]. However, it is worth mentioning here that all instances have at least one solution where the events can be feasibly packed into the target number of forty-five timeslots, and for some of them there is also a known perfect solution. However, for the remaining instances, some are known to definitely *not* have perfect solutions[12], whilst in other cases, whether or not a perfect solution exists still remains undetermined.

We also imposed certain time limits on our algorithm during testing that we considered fair for these sizes of problem. These were 30, 200 and 800 seconds of CPU time for the small, medium and large sets respectively. The hardware used for these experiments (and indeed for all experiments described in this thesis) was a Pentium IV 2.66Ghz processor with 1GB RAM under a Linux operating system.

For all experiments with the GGA, a steady-state population (of size ρ) using binary tournament-selection was used. The population was then evolved in the following manner: at each step of the algorithm, two offspring were produced by either recombination or replication, dictated by a recombination rate *rr*. (Note: an offspring made via replication is simply a copy of its first parent.) Next, the two offspring were mutated (according to the mutation rate *mr* – see Section 4.2.2), and evaluated. Finally the two offspring were reinserted into the population, in turn, over the individuals with the worst fitness. If at this point there was more than one least-fit individual, a choice between these was made at random. Finally, at each step a small number (*ir*) of randomly chosen individuals were also selected to undergo inversion.

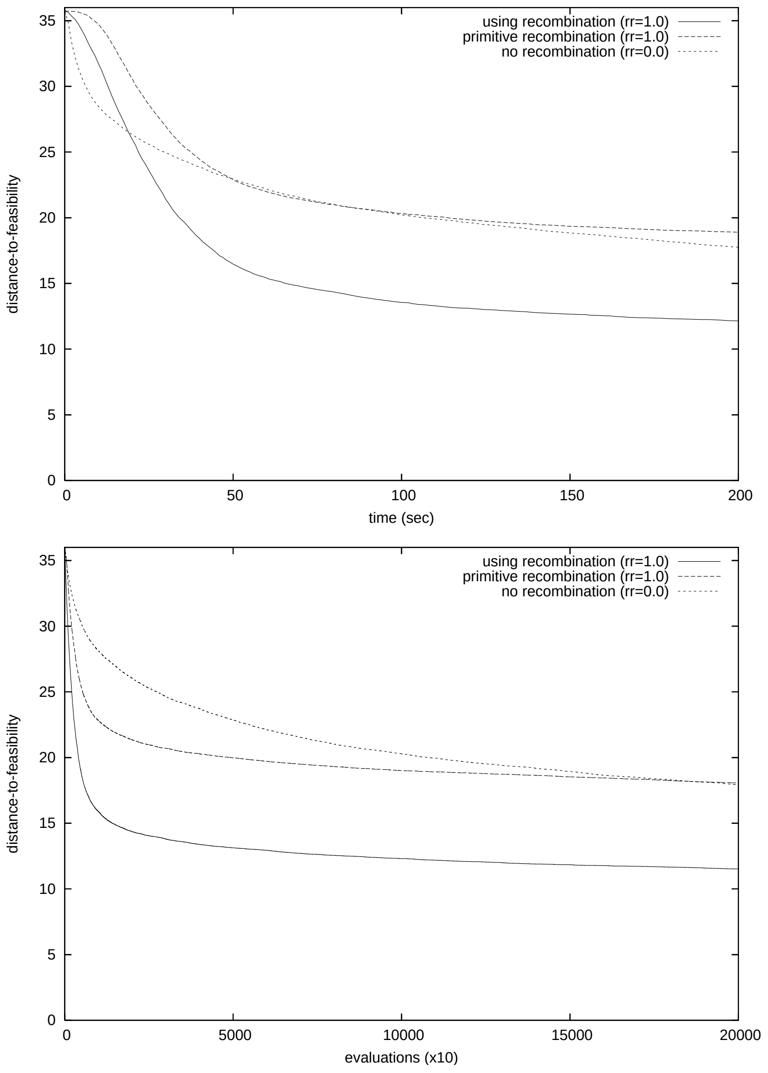
[12] We were able to determine that an instance had no perfect solution when $n > 40m$. In these instances we know that at least $(n - 40m)$ events will have to be assigned to the end-of-day slots, where they will cause a violation of soft constraint SC_1.

4.4 Investigating the Effects of the GGA Recombination Operator

For our first set of experiments with this algorithm, we will examine the general effects that the recombination operator has on the algorithm, by comparing runs that always use recombination to produce offspring (i.e. with a rate *rr* = 1.0) against runs that use none at all (*rr* = 0.0). The results of these experiments are depicted in figs. 4.5(a)-(d). If we consider first the results for the medium instances in fig. 4.5(a), we can see that after an initial lag period of around 20 seconds (where using no recombination seems to provide quicker movements through the search space on average) the use of this recombination operator benefits the search significantly[13].

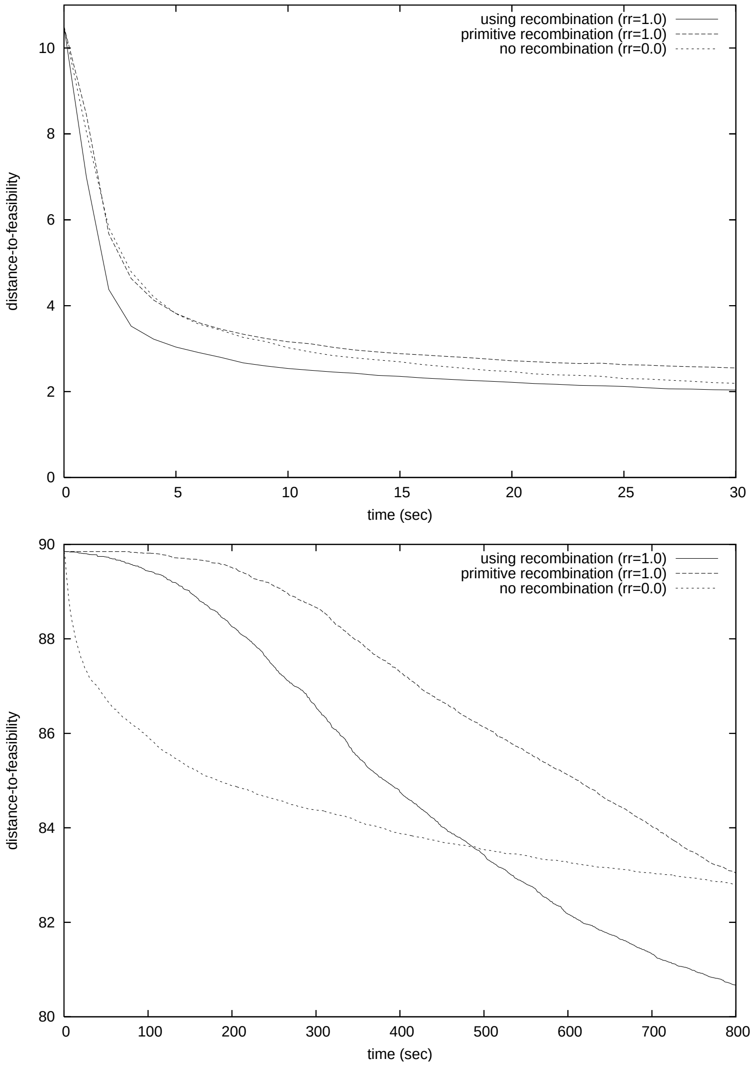
Note however, that such a simple comparison on its own is not completely fair because, as the reader may have noticed, the heuristics used for rebuilding with our recombination operator are different from those used with mutation, and therefore it might be the heuristics doing the work, and not the fact that the recombination operator is doing its job of successfully combining useful parts of different solutions. Thus, in figs. 4.5(a)-(d) we also include a third line that again uses recombination with a rate 1.0, but also uses the more primitive heuristics used by the mutation operator when rebuilding timetables after stage (3) of recombination. This line is labelled "primitive recombination" in the figures. In fig. 4.5(a) we can see that, in this case, the presence of this primitive recombination operator actually seems to *hinde*r the search slightly with regards to CPU time. However, it might also make sense to observe this behaviour from a second perspective: in timetabling, due to the large number of possible constraints that can be imposed on a particular problem, it can often be the case that the evaluation function becomes the most costly part of the algorithm, particularly when soft constraints are also being considered. If we now look at these same runs, but with regards to the number of evaluations (fig. 4.5(b)), we can see that according to this criterion, use of this more primitive recombination, up until around 150,000 evaluations, is clearly beneficial. We also see once more that the more advanced recombination operator provides the best search (this difference was also significant).

[13] In the experimental analyses appearing in this thesis we will use the word "significant" to indicate that a Wilcoxon signed-rank test performed on the results found at the time limit came from a different underlying distribution with a probability ≥ 95%.



Figs. 4.5(a) and (b): (top and bottom) Showing the behaviour of the algorithm with and without recombination with the medium instances. Each line represents, the distance to feasibility of the best solution in the population, averaged across 20 runs on each of the 20 instances (i.e. 400 runs) using mr = 2, ir = 4, and ρ = 50. Figure (a) shows the runs with regards to CPU time; figure (b) with regard to the number of evaluations performed.

Moving our attention to the behaviour of this algorithm with the small and large instance sets (figs. 4.5(c) and (d) respectively) in our experiments we actually noticed different behaviours in each case. With the small instances, the algorithm generally performed well across the set, and seemed quite insensitive to the various parameter settings (and whether recombination was being used or not). Indeed, although fig. 4.5(c) indicates a *slightly* better search when using recombination, this difference is small and in our experiments it was not seen to be significant. As a matter of fact, in most trials, optimal solutions were regularly found to over half of the instances within the time limit, making it difficult to draw any particularly interesting conclusions other than the fact that performance of the algorithm with these instances was generally quite good.



Figs. 4.5(c) and (d): (top and bottom) Showing the behaviour of the GGA with and without recombination for (a) the small instances, and (b) the large instances. The meaning of "primitive recombination" is explained in the text. Each line represents, at each CPU second, the distance to feasibility of the best solution in the population, averaged across 20 runs on each of the 20 instances (i.e. 400 runs) using *mr* = 2, *ir* = 4, and ρ = 50.

In the case of the large instances, meanwhile, the algorithm actually displayed yet another behavioural pattern. Looking at fig. 4.5(d), we can see that the use of recombination in these cases seems to drastically slow the search; indeed, no benefits of recombination can actually be seen until around 500 seconds. Clearly, in these cases if we were using shorter time limits, then a use of recombination might actually hinder rather than help. Secondly, if we consider the scale of the y-axis in fig. 4.5(d) which, we note, is heavily truncated, we can see that in all cases only very small improvements are actually being achieved during the entire run. For example, when using recombination we can see in fig. 4.5(d) that the distance-to-feasibility at the time limit is 80.7. This means that by the time the time limit is reached, approximately (80.7/1000) × 100 = 8.07% of the events cannot be inserted into the timetable on average. In comparison, with the small instances this figure is approximately $(2.0/200) \times 100 = 1.0\%$ of events, and with the medium instances, approximately $(12.2/400) \times 100 = 3.0\%$. Perhaps even more telling though is the fact that with the large instances, the GGA (using recombination) only reduces the distance-to-feasibility by an average of 10.2% during the entire run. This compares to an 80.6% reduction with the small instances and a 66.0% reduction with the medium instances. Considering all problem instances used in these tests are known to have at least one optimal solution (with respect to the hard constraints) the improvements achieved with the large instances are therefore quite disappointing.

The above observations immediately suggest that instance size is an important factor in the run characteristics of this GGA in terms of the effects of recombination, behaviour over time, and the general progress made through the search space. In the next section we will present some ideas as to why this might be so.

4.5 Scaling-up Issues with the GGA

4.5.1 A Diversity Measure for Grouping Representations

Before describing some of the possible scaling-up issues of this algorithm, it is first necessary for us to introduce a diversity measure for the grouping representation. Additionally, in this subsection many of the concepts that we will be describing will actually relate to grouping problems as a whole, and not just this UCTP. Therefore, in our descriptions we will revert to the more generic terms of "groups" and "items", as opposed to "timeslots" and "events", which are specific only to timetabling problems.

As we have discussed, the grouping representation used by the GGA admits two important properties: chromosomes are variable in length (in that they can contain varying numbers of groups), and the ordering of the groups within the chromosomes is irrelevant with regards to the overall solution being represented. Unfortunately, these characteristics mean that many of the usual ways of measuring population diversity, such as Hamming distances [83] or Leung-Gao-Xu diversity [72] are rendered inappropriate. Additionally, in the case of this timetabling problem, we believe that it would be misguided to use diversity measures based on population fitness information (such as the standard deviation etc.), because in our experiences it can often be the case that minor changes to a timetable might actually result in large changes to its fitness and, inversely, two very different timetables can often have a similar fitness.

We believe that a suitable diversity measure for this representation, however, can be obtained by using some ideas of the so-called ‘substring-count’ method of Mattiussi, Waibel, and Floreano, recently presented in [78]. In the grouping representation, it is worth considering that each group can only occur at most once in any particular candidate chromosome (otherwise the solution would be illegal because it would contain duplicates). Given a population *P*, a meaningful measurement of diversity might therefore be calculated via the formula:

$$div(P) = \rho(a/b) \tag{4.2}$$

where ρ is the population size, *a* represents the number of different groups in the population, and *b* is the total number of groups in the population. Using this measurement, a homogenous population will therefore have a diversity of 1.0, whilst a population of entirely distinct individuals will have a diversity of ρ .

Additionally, in agreement with [78], using these basic ideas we are also able to define a distance measurement for a pair of individuals, p_1 and p_2, via the formula:

$$dist(p_1, p_2) = 2(x/y) - 1 \tag{4.3}$$

where *x* represents the number of different groups in p_1 and p_2, and *y* is the total number of groups in p_1 and p_2. Thus, two homogenous timetables will have a distance of zero and two maximally distinct individuals will have a distance of one.

4.5.2 Diversity, Recombination, and Group Size

During our experiments with the GGA, we often noticed that evolution (with regards to the number of new individuals being produced over time) was quite slow at the beginning of a run, but then gradually quickened as the run progressed. These characteristics were particularly noticeable when dealing with the larger instances where we saw new individuals being produced at a somewhat unsatisfactory rate for quite a large proportion of the run (an effect of this characteristic can be seen in figs. 4.5(a) and (d), where we can see a slight s-shaped curve formed over time when using recombination with the GGA). Investigations into this matter revealed that this was due to the fact that the recombination operator usually tended to be more costly at the start of a run but then gradually became less expensive as evolution progressed. Further investigations revealed that this added expense seemed to be influenced by two factors: (a) population diversity, and (b) the sizes of the groups in the candidate solutions.

Fig. 4.6 shows three examples of steps (1)-(3) of the GGA recombination operator in order to illustrate these concepts. In fig. 4.6(a), the distance between candidate solutions p_1 and p_2 is comparatively small (i.e. $2\times(6/8) - 1 = 0.5$) and only one of the seven items becomes unplaced during recombination. In fig. 4.6(b) however, although the number of groups and items being injected is the same as fig. 4.6(a), the distance between p_1 and p_3 is larger (i.e. $2\times(8/8) - 1 = 1.0$); consequently, the duplicate items resulting from the injection are spread across more of the groups, meaning that a greater number of the groups coming from p_1 need to be eliminated. This also means that more items have to be dealt with by the rebuilding process, making the overall procedure more expensive.

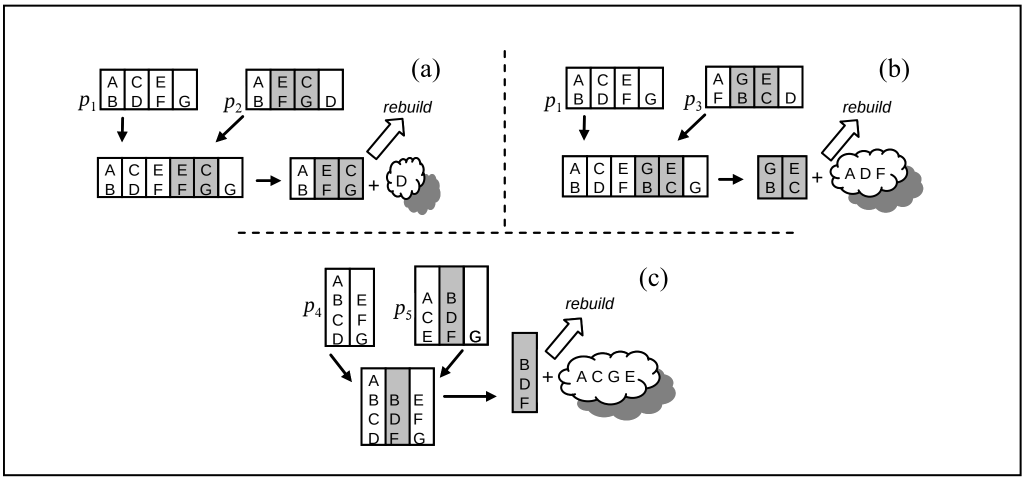


Fig. 4.6: Demonstrating how diversity and group size can influence: (a) the amount of reconstruction needed; and (b) the number of groups that are lost, using the standard GGA recombination operator.

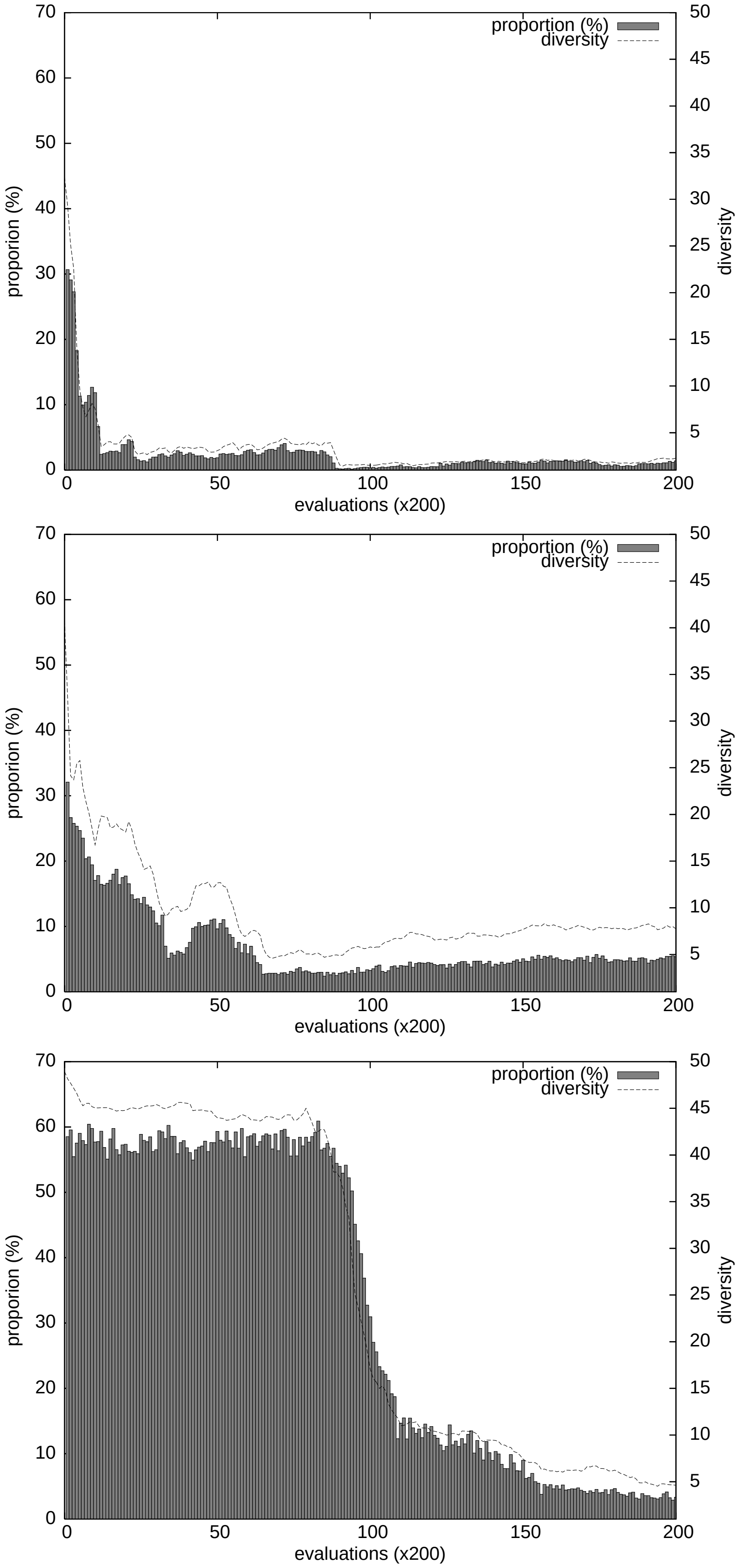
Next, in fig. 4.6(c) we illustrate the effects that larger groups can have on the recombination operator. In this case, the size of the problem may be considered the same as the previous two examples, because we are still only dealing with seven items. However, this time the *size of the groups* is larger and, as can be seen, the injection of a group from p_5 into p_4 causes a high proportion of the items to become unplaced during stage three of recombination.

In fact, figs. 4.6(b) and 4.6(c) depict cases of what we will term a *unilateral recombination*: after stage (2) of recombination (injection), all of the groups coming from the first parent have contained a duplicate and have therefore been eliminated. Thus, the resultant offspring does not actually end up containing building-blocks from both parents (as is usually desirable), but will instead be made up of some groups from the second parent, with the rest having to be formed, from scratch, by the rebuilding process. In this sense, it might be considered more of a macro-mutation operator than anything else.

In order to further illustrate these concepts, consider fig. 4.7, where we show details of example runs with a small, medium, and large problem instance respectively. In all three figures it can be seen that as evolution progresses, the level of diversity in the populations generally falls. This, of course, is typical of an evolutionary algorithm. We also see in these figures that the proportion of items (events) becoming unplaced during recombination mirrors this fall very closely, thus highlighting the strong relationship of the two measurements. However, the other noticeable characteristic in these figures is the way in which high levels of both of these measures are sustained for longer periods when the instance sizes (and therefore the number of events/items per timeslot/group) are larger. Looking at fig. 4.7(c), for example, we can see that no real drop in either measurement actually occurs until around 180,000 evaluations and, up until this point, over half of the items (events) are becoming unplaced, on average, with every application of the recombination operator.

We believe that this latter phenomenon is caused by the fact that, because in this case the groups *are* larger, the potential for losing the groups coming from the first parent is increased (as illustrated in fig. 4.6(c)). We may therefore view this as a more *destructive* recombinative process. Of course, not only does this make the operation more expensive (because a greater amount of rebuilding has to be performed), it also means that it is more difficult for the GGA to successfully combine and pass on complete groups from one generation to the next. Thus, it would seem that, in these cases, the recombination operator is indeed becoming more of a macro-mutation operator, and a useful component of the evolutionary algorithm might be being compromised.

It is also worth noting that in cases where the recombination operator *is* behaving in a more destructive manner, this will generally mean that more groups in an offspring will occur as a result of the rebuilding process, as opposed to being directly inherited from an ancestor. Unfortunately however, this may well add extra diversity to the population, therefore exacerbating the problem even further.



Figs. 4.7(a)-(c): (top, middle, and bottom) Example runs with a small, medium, and large instance respectively, demonstrating (1) the close relationship between diversity and the amount of reconstruction needing to be done with the recombination operator, and (2) the differing ways that the two measures vary during the runs as a result of the different sized timeslots/groups. (All runs using *rr* = 1.0, *ir* = 4, *mr* = 2, and ρ = 50.) In the figures, the amount of reconstruction being done with the recombination operator is labelled "Proportion (%)" – this refers to the percentage of events that are becoming unplaced, on average, with each application of the recombination operator.

4.5.3 Group Size and Chromosome Length

As a final point in this subsection, it is also worth mentioning some further characteristics of the GGA operators that relate to group size. If we refer back to fig. 4.4, which demonstrates the standard GGA recombination operator applied to this timetabling problem, we will see that the total number of possible values that can be chosen for crossover points *c* and *d* (that is, the total number of group-combinations that can be selected for injection from the second parent into the first parent) is always exactly:

$$\frac{s_2(s_2-1)}{2}, \tag{4.4}$$

where here, s_2 represents the length of the second parent p_2. Additionally, the total number of possible group-combinations that can be selected for removal by the mutation operator is always exactly:

$$\frac{s!}{j!(s-j)!}, \tag{4.5}$$

where *j*, which is in the range [1, *mr*], represents the number of groups that have been chosen to be removed from the chromosome, and *s* represents the length of the chromosome being mutated.

However, unlike most forms of evolutionary computation, where chromosome length will be defined by the size of the problem instance being tackled (see, for example, [39], [97], and [105])[14], with the GGA, a chromosome's length will actually be defined by the *number of groups* it is using. This, of course, will mean that when dealing with grouping problems of a certain size, problems that use larger groups will be represented by proportionately shorter chromosomes, and the values returned by equations (4.4) and (4.5) will, in turn, be lower.

The implications of these facts are particularly apparent with this timetabling problem. Here, chromosome length is defined by the number of timeslots being used, but it is the number of events that defines problem size. Thus, given that our aim is always to feasibly arrange the events into t = 45 timeslots, this means that the lengths of the chromosomes will remain more-or-less constant, regardless of problem size. Unfortunately, in practice this means that an increase in instance size will not only cause the timeslots (groups) to be larger (resulting in the unfavourable situations described in the previous sub-

[14] In grouping problems, for example, problem size will generally be defined by the number of items to be partitioned.

subsection), it also suggests that the potential for the genetic operators to provide sufficient exploration of the search space might also be more limited.

4.6 Improving the Algorithm

So far in this chapter we have presented a justification and description of a GGA for the UCTP, but have also noted that in some cases – particularly for larger instances – that the algorithm does not always seem to perform at a completely satisfactory level. In this section we will therefore investigate two separate ways in which we might go about improving algorithmic performance. First, we will introduce a number of new fitness functions that could also be used with this algorithm, and will investigate whether any of these are able to improve upon any of the results we have witnessed thus far. Second, we will introduce an additional heuristic search operator into the GGA (to make, in effect, a grouping *memetic* algorithm [84, 85]) and will examine the conditions that are needed in order to allow this operator to also improve the GGA's performance. These ventures will be reported in sections 4.6.1-2, and 4.6.3-4 respectively.

4.6.1 Using more Fine-Grained Fitness Functions

A central aspect of any evolutionary algorithm is the way in which candidate solutions in the population are evaluated against each other. Ideally, a good fitness function should convey meaningful information about the quality of a candidate solution and will also encourage the search into promising areas of the solution space. For many problems in operational research, a suitable measure is suggested naturally by the problem at hand (such as the travelling salesman problem [105]). In others, however, it is not so easy. For example, in [58] Falkenauer looks at an application of a GGA to the bin packing problem and suggests that while the most obvious way of measuring a candidate solution's fitness is to just calculate the number of bins being used (with the aim of minimisation), this is actually unsuitable because it will likely lead to a very inhospitable fitness landscape where "a very small number of optimal points in the search space are lost in the exponential number of points where this purported cost is just one above this optimum. Worse, these slightly sub-optimal points [all] yield the same cost [58]". In practical terms this could, for example, lead us to a situation where we might have a very diverse population, but all members appear to have the same fitness. In this situation, not only would selection pressure be lost, but also if all the scores were indeed one away from the optimum, any move from near-feasibility to full-feasibility would be more-or-less down to chance.

In Section 4.2.3, we mentioned two possible ways of measuring solution quality with this problem, and then used one of these (our so-called distance-to-feasibility measure) to perform the experiments in Section 4.4. However, there is no reason why we should necessarily use either of these during evolution. Indeed, both measurements are fairly coarse-grained and might well lead us to the undesirable situations described in the previous paragraph. We therefore introduce here four further fitness functions. These, plus the original two are defined as follows, and for simplicity's sake, all have been made maximisation functions:

$$f_1 = \frac{1}{1+s} \tag{4.6}$$

$$f_2 = \frac{1}{1+d} \tag{4.7}$$

$$f_3 = \frac{1}{1+d+(s-t)} \tag{4.8}$$

$$f_4 = \frac{\sum_{i=1}^{s} (E_i/m)^2}{s} \tag{4.9}$$

$$f_5 = \frac{\sum_{i=1}^{s} (C_i)^2}{s} \tag{4.10}$$

$$f_6 = \frac{\sum_{i=1}^{s} (S_i)^2}{s} \tag{4.11}$$

Here, as before *s* represents the number of timeslots being used by a particular timetable, *t* is the target number of timeslots (i.e. 45), and *m* is the number of available rooms per timeslot. In this case *d* represents the distance-to-feasibility measure already discussed. Additionally, we also define some new measures: E_i represents the number of events currently assigned to timeslot *i*; S_i tells us how many students are attending events in timeslot *i*; and, finally, C_i tells us the total conflicts-degree of timeslot *i* (that is, for each event in timeslot *i*, we determine its degree by calculating how many other events in the entire event set it conflicts with, and then C_i is simply the total of these values).

Essentially, fitness function f_3 is the same as Eiben, van der Hauw, and van Hermert's fitness function for graph colouring used in [50]. It uses the observation that if two timetables have the same value for *d*, then the one that uses the least number of extra timeslots is probably better and, similarly, if two timetables have the same number of extra timeslots, then the one with the smallest value for *d* is probably better.

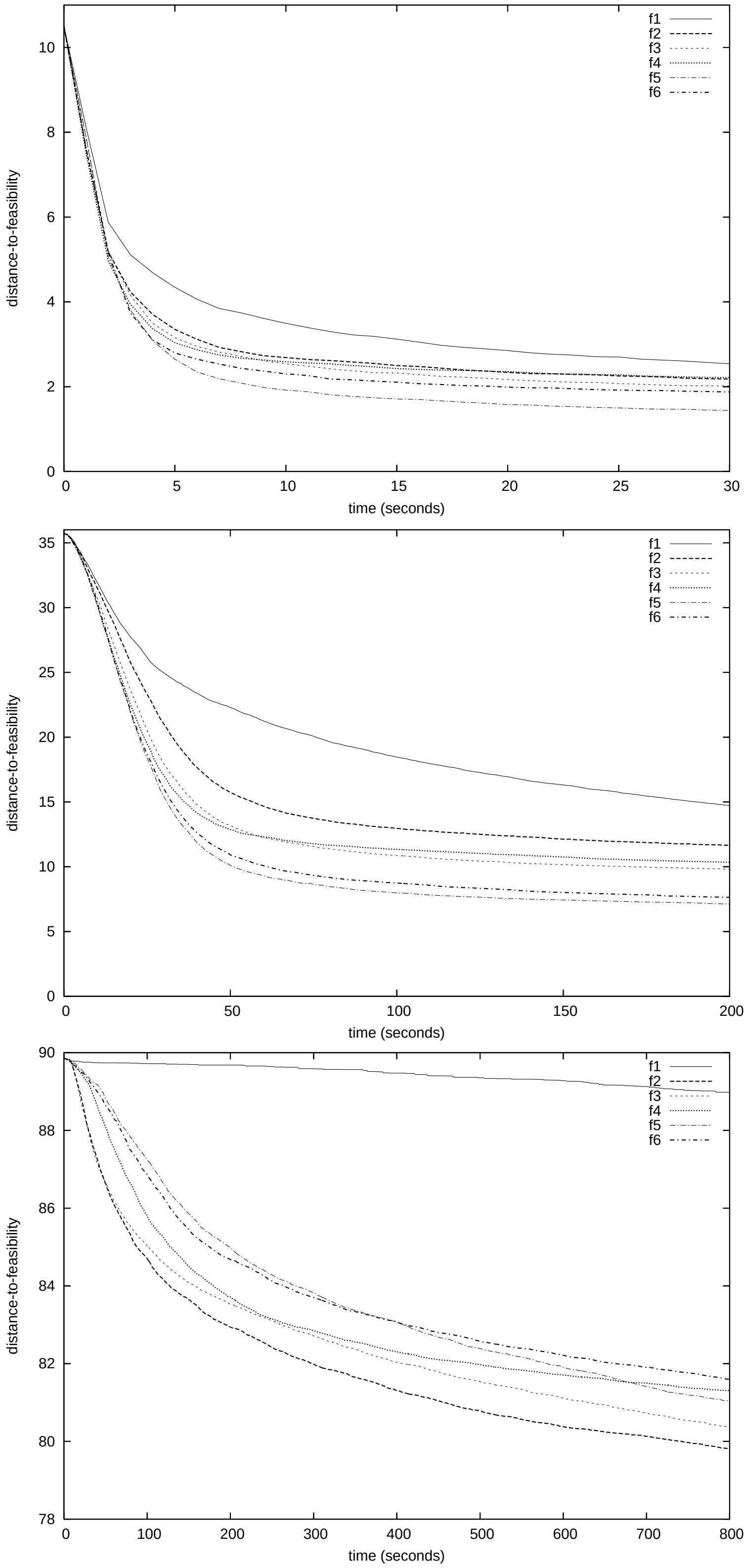
Functions f_4, f_5, and f_6, meanwhile, judge quality from a different viewpoint and attempt to place more emphasis on the individual timeslots. Thus, timetables that are made up of what are perceived to be promising timeslots (i.e. good *packings* of events) are usually favoured because their fitness will be accentuated by the squaring operations. The three functions differ, however, in their interpretations of what *defines* a good packing: function f_4 tries to encourage timetables that have timeslots with high numbers of events in them, and is similar to the fitness function suggested for bin packing [54, 58]; function f_5, meanwhile, uses the well-known heuristic from graph colouring [16] that recommends colouring as many nodes (events) of high degree as possible with the same colour [52]; finally, function f_6 attempts to promote timetables that contain timeslots with large total numbers of students attending some event in them – following the obvious heuristic that if many big events are packed into one timeslot, then other smaller (and presumably less troublesome) events will be left for easier packing into the remaining timeslots.

As a final point, it is worth noting that unlike the remaining four functions, functions f_2 and f_3 need to know in advance the target number of timeslots. If this is undefined, the task of calculating the minimum number of timeslots needed to accommodate the events of a given instance is equivalent to calculating the chromatic number in graph colouring. However, computing the chromatic number is itself an NP-hard problem [62]. In practical course timetabling, however, this detail is probably not so important because it is typical for the university to specify a target number of timeslots in advance.

4.6.2 Analysing the Effects of the Various Fitness Functions

To investigate the effects of these six fitness functions, we performed tests using the same steady-state population scheme as before (Section 4.3), and simply altered the fitness functions for each trial-set. Note then, that the only *actual* change to the algorithm's behaviour for each set of trials in this case is (a) the criterion used for choosing tournament winners during selection, and (b) the criterion used for picking which individuals to replace. Note also, that the computational costs of the fitness functions are roughly equivalent, as all require just one parse of the timetable.

Figures 4.8(a)-(c) show how the algorithm responds to the six fitness functions over time with the three different instance sets. If we first draw our attention to figs. 4.8(a) and (b), we can see that with regards to the small and medium instances, f_5, and then f_6, clearly give the best searches (on average), with respect to both the speed of the search and the best solutions that are found within the time limits. As expected, we can also see that functions f_1 and then f_2, also seem to provide the worst performance. We believe these differences in performance are due to the reasons mentioned above: when using f_5 and f_6 (and to a lesser extent, f_3 and f_4) the algorithm is able to distinguish between solutions that, according to f_1 or f_2, might be valued the same. Thus, it is possible to maintain selection pressure for a greater duration of the run. Furthermore, it would appear that the heuristic criteria that f_5 and f_6 use to make these distinctions (described above), is indeed conducive to the search. In both cases, the improvements that f_5 and f_6 provided were significant.



Figs. 4.8(a)-(c): (top, middle, and bottom) The effects of the six different fitness functions over time with the small, medium, and large instances respectively. Each line represents, at each second, the distance to feasibility of the best solution in the population, averaged across 20 runs on each of the 20 instances (i.e. 400 runs), using ρ = 50, *rr* = 1.0 (0.25 with 4.7(c)), *mr* = 2, and *ir* = 4.

Interestingly, we see that the algorithm responds differently to the fitness functions when considering the large instances (fig. 4.8(c)). As before, we see that f_1 is clearly the worst performer, but we also see that the characteristics of the remaining five fitness functions is now more-or-less reversed, with f_2 providing the best performance. This could be because the squaring functions used with f_4, f_5, and f_6 cannot accentuate the characteristics of a good timeslot as much as when used with the other instances (which have a smaller number of events per timeslot). However, most importantly one has to look again at the scale of the y-axis of fig. 4.8(c) to appreciate that the algorithm, again, actually performs fairly badly with all of the fitness functions on the large instances. Additionally, if we disregard f_1, the differences between the remaining five fitness functions were not actually seen to be significant.

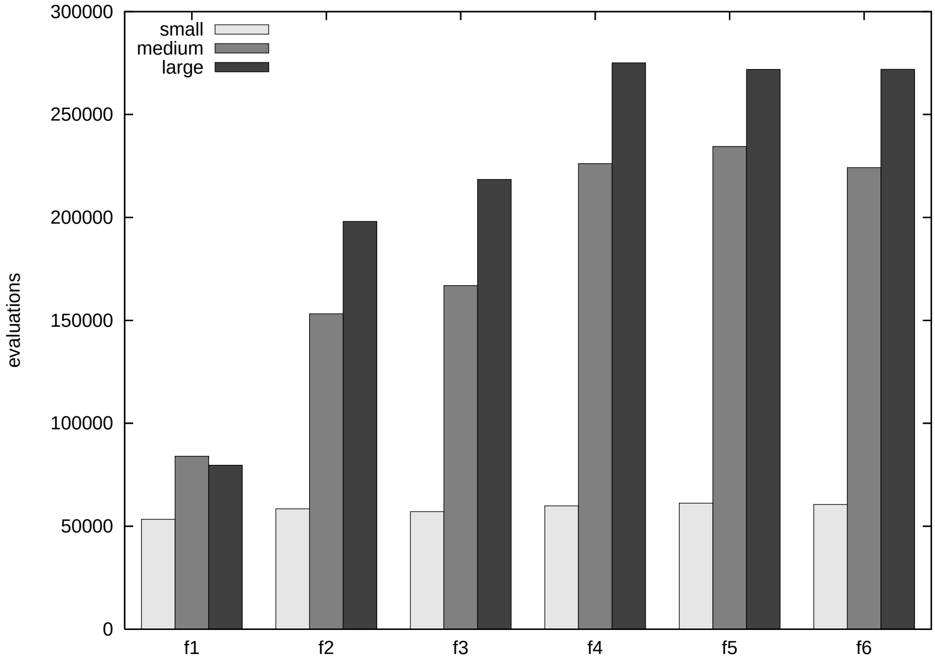
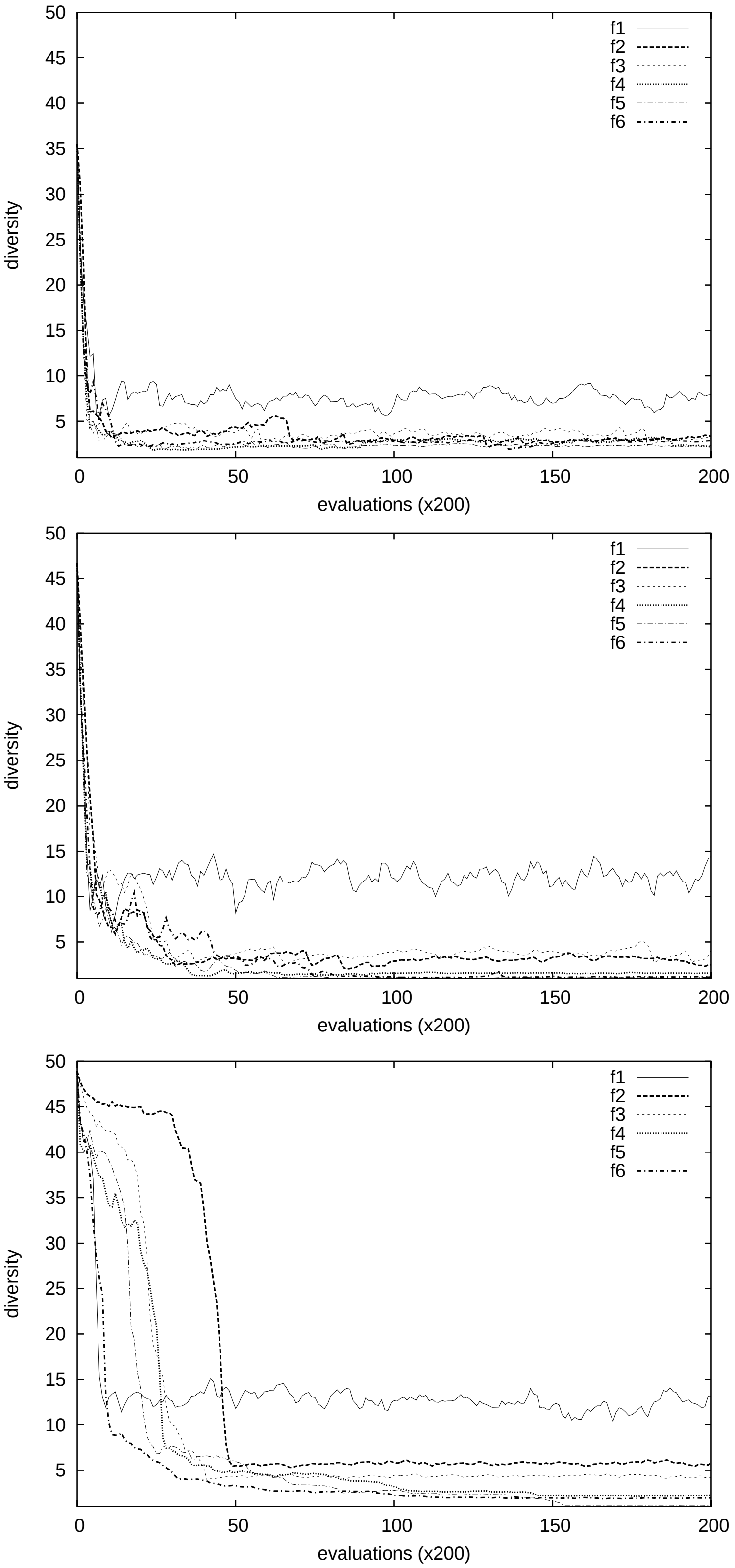


Fig. 4.9: Average number of evaluations performed during the runs shown in figure 4.8.

Figure 4.9 also shows some intriguing results of these experiments. As can be seen, when considering the medium and large instance sets, the number of evaluations performed (i.e. the number of new individuals produced) within the time limits alters drastically depending on which fitness function the algorithm is using. (This pattern also emerges with the small instances, but the distinction is more difficult to make in the figure.) The reasons why these characteristics occur start to become clear when we look at figs 4.10(a)-(c), where we contrast the influences that the six different fitness functions can have on a population's diversity as it is evolved.



Figs. 4.10(a)-(c): (top, middle, and bottom): Example runs with a small, medium, and large instance respectively, to demonstrate the effects that the various fitness functions have on the diversity of the population during evolution. All runs were performed using ρ = 50, *rr* = 1.0, *mr* = 2 and *ir* = 4.

As can be seen in all three cases, when using f_1 (which is the most coarse-grained of the six fitness functions), after an initial drop during the first few thousand evaluations, the diversity of the populations eventually persists at a higher level than when any of the other five fitness functions are used. Of course, this is due to the reasons that we described at the beginning of the previous sub-subsection: i.e. when using a coarse grained fitness function such as this, at some point all of the individuals in the population will start to appear very similar (according to the fitness function's criteria), and so much of the selection pressure will be lost. According to our reasoning in Section 4.5, this sustained diversity will naturally mean that the recombination operator will remain more expensive and destructive, eventually not allowing as many individuals to be produced within the time limit. Conversely, we can also see in figures 4.10(a)-(c) that when the more *fine*-grained fitness functions are used (i.e. f_4, f_5, and f_6), the diversity of the populations eventually drops to levels that are lower, because the fitness functions are able to distinguish between individuals that other fitness functions might see as identical for longer, meaning that selection pressure remains for a greater part of the run. In turn the lower levels of diversity will generally result in a less expensive recombination process, meaning that more individuals can be produced within the time limits.

Note that an overly rapid loss of diversity may sometimes be undesirable in an EA, because it might lead to a redundancy of the recombination operator and an under-sampling of the search space. However, in the case of this GGA there is clearly a trade-off because, as noted, a high amount of diversity can cause recombination to be both expensive and destructive. With regards to the quality of solutions that are found within the imposed time limits, in the cases of the small and medium instances, the trade-off seems to fall in favour of using f_5 and f_6 which, although exhibiting tendencies to lose diversity at a quicker rate, still both return superior results and in less time.

As a final point, it is also worth considering some further implications of these fitness functions: from a practical standpoint, in some real world timetabling problems there may be some events that every student is required to attend (such as a weekly seminar or assembly). Clearly, such an event must be given its own exclusive timeslot, because all other events will conflict with it (i.e. all other non-empty events will have common students with it). However, fitness function f_4 will, unfortunately, view this as an almost empty (or badly packed) timeslot and will penalise it, therefore possibly deceiving the algorithm. Fitness functions f_5 and f_6, however, will reward this assignment appropriately. On the other hand, although the work of Erben [52] has demonstrated that fitness function f_5, when used with a GGA, can perform very well with types of problem-instances that other sorts of EA might find very difficult (such as the "Pyramidal Chain"-style problems in graph colouring – see the work of Ross, Hart, and Corne [94]), it is worth bearing in mind that when we choose to judge a timeslot's quality by looking at the total degree of the events within it (with higher values being favoured) this criteria is ultimately a heuristic, and it is conceivable that counter examples could be constructed. Additionally, it is also worth remembering that in our experiments f_5 didn't seem to perform so well with the larger instances either.

4.6.3 Introduction of a Heuristic Search Operator

A second way in which we might go about improving this GGA is via the introduction of an additional search operator. In evolutionary computation, it is generally accepted that EAs are very good at coarse-grained global search, but are rather poor at fine-grained local-search [113]. It is therefore perfectly legitimate (and increasingly common) to try to enhance an EA by adding some sort of local search procedure. This combination of techniques is commonly referred to as a *memetic* algorithm (e.g.[84, 85]), and the underlying idea is that the two techniques will hopefully form a successful partnership where the genetic operators move the search into promising *regions* of the search space, with the search operator then being used to explore *within* these regions.

Looking at some other algorithms from this problem domain, both Dorne and Hao [47], and Galnier and Hao [60] have shown how good results can be found for many graph colouring instances through the combination of these techniques. In both cases, specialised recombination operators were used, with tabu search then being utilised to eliminate cases of adjacent nodes having the same colour. Rossi-Doria *et al.* have also used similar techniques for timetabling in [97], where their more global operators (such as uniform-crossover) are complemented by a stochastic, first-improvement local-search operator which, as one of its aims, attempts to rid the resultant timetables of any infeasibility.

As a matter of fact, it turns out that these methods are not actually suitable for our algorithm as they are intended for eliminating *violations* of hard constraints, and as we have already related (Section 4.2.1), in our representation we explicitly disallow these violations to occur as part of the encoding. Indeed, an appropriate searching procedure in this case should, instead, be able to take a timetable with no hard constraint violations, and somehow find a timetable that is hopefully better in some way, but still with no violations.

With regards to other grouping-based algorithms that have used of this sort of representation, but which have also made use of an additional search technique, Falkenauer's hybrid-GGA [55, 58], and Levine and Ducatelle's ant algorithm [73] (both for bin packing) have both been reported to return substantially better results when their global-search operators are coupled with an additional search method. These additional techniques are inspired by Martello and Toth's dominance criterion [77] and work by taking some unplaced items, and then attempting to swap some of these with items in existing bins so that (a) the bins become more full, but (b) the number of items in the bin stays the same (i.e. each item was only replaced by a bigger item).

However, even though such dominance criterion does not strictly apply in our timetabling problem, it would still be useful to define a similar operator that attempts to improve the packings of events into the timeslots somehow. An operator intended for doing just this is defined by the procedure that we will call Heuristic-Search in fig. 4.11. Taking a list of unplaced events and a partial timetable (*U* and *tt* respectively), this procedure basically operates by repeatedly taking events from *U* and trying to insert them into free (i.e. blank) and feasible places in *tt* (lines 4-7 of fig. 4.11). In order to complement these actions, however, at each iteration the procedure also attempts to move the events *within tt* (lines 9-15) in such a way that the free places in *tt* change position, thus offering the possibility of further events in *U* being moved into *tt* in the next iteration.

```
HEURISTIC-SEARCH (tt, U, itLimit)
1. Make a list V containing all the places in tt that have no events
   assigned to them;
2. $i := 0$;
3. while ($U \neq \emptyset$ and $V \neq \emptyset$ and $i < itLimit$)
4.    foreach($u \in U$ and $v \in V$)
5.       if (u can be feasibly assigned to v in tt)
6.          Put u into v in tt;
7.          Remove u from U and v from V;
8.    if ($U \neq \emptyset$ and $V \neq \emptyset$)
9.       repeat
10.         Choose a random event e in tt and $v \in V$;
11.         if(e can be feasibly moved to v in tt)
12.            Move e to v;
13.            Update V to reflect changes;
14.         $i := i + 1$;
15.      until ($i \geq itLimit$ or e has been moved to v)
```

Fig. 4.11: Pseudo-code description of the heuristic search procedure. Here, *tt* represents a partial timetable, *U* a list of unplaced events, and *itLimit* the iteration limit of the procedure.

Note that the addition of this operator to the GGA will have two important effects: first, whilst Heuristic-Search does not actually allow the number of events contained within *tt* to decrease, if its application *is* successful then events will be taken from *U* and added to *tt*, thereby improving the overall timeslot packings. Secondly, because this process causes events and free spaces within *tt* to be randomly shuffled amongst the timeslots (lines 9 to 15 of fig. 4.11) diversity will be added to the population.

In our experiments, which will be described in the next section, we chose to use this heuristic search operator in conjunction with our mutation operator. As before, each time a mutation occurs during a run, a small number of timeslots are randomly selected and removed from the timetable. The events in these timeslots now make up the list of unplaced events *U*, and HEURISTIC-SEARCH is applied. If *U* is non-empty when the iteration limit is reached, then the rebuilding scheme (fig. 4.3) is used to insert the remaining events.

4.6.4 Analysing the Effects of a Heuristic Search Operator

It can be noticed in fig. 4.11 that our heuristic search procedure needs to be supplied with an iteration limit that specifies the maximum number of steps that the procedure can run for each time it is called. In these experiments we chose to set this to be proportionate to the size of the problem instance being solved. This was achieved by using an additional parameter l such that the iteration limit $itLimit = (l \times n)$, remembering that in this problem, n represents the number of events in a particular problem instance.

With regards to algorithm performance, the introduction of this operator now presents an additional trade-off: too much heuristic search (i.e. a setting for l that is too high) may not allow enough new individuals to be produced within the time limit, and will probably result in too little global search; however, a setting for l that is too low could also be detrimental, because it may not allow the new operator enough opportunity to explore the regions of the search space that the global operators lead us to. To investigate these implications, we therefore empirically carried out a large number of trials on the three instance sets, using fitness function f_5 with various different recombination rates *rr*, settings for l, mutation rates *mr*, and population sizes ρ. (In all trials the same steady-state population scheme as described in 4.3 was also used.)

The first thing that we noticed from these experiments was the dramatic effect that the use of the heuristic search operator had on the number of new individuals that could be produced within the time limits. This is illustrated for the three instance sets in fig. 4.12. Here, we see that the introduction of this additional search operator, even in very small amounts, actually causes a dramatic decrease in the number of new individuals that are produced within the time limits. Although this is partly due to the obvious fact that the heuristic search procedure is adding extra expense to the mutation operator, we believe that the main reason is due to the fact that, because the heuristic search operator is continually adding diversity to the population (due to its various random elements), this causes the recombination operator to remain much more expensive and destructive for a greater part of the run. As fig. 4.12 shows, this is especially so for the medium and large instances, where the presence of larger groups further exacerbates this phenomenon.

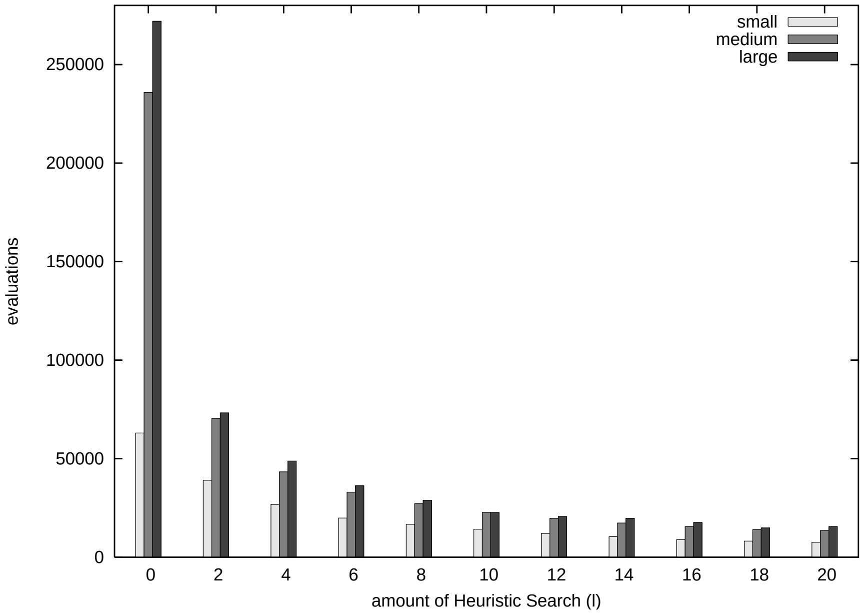
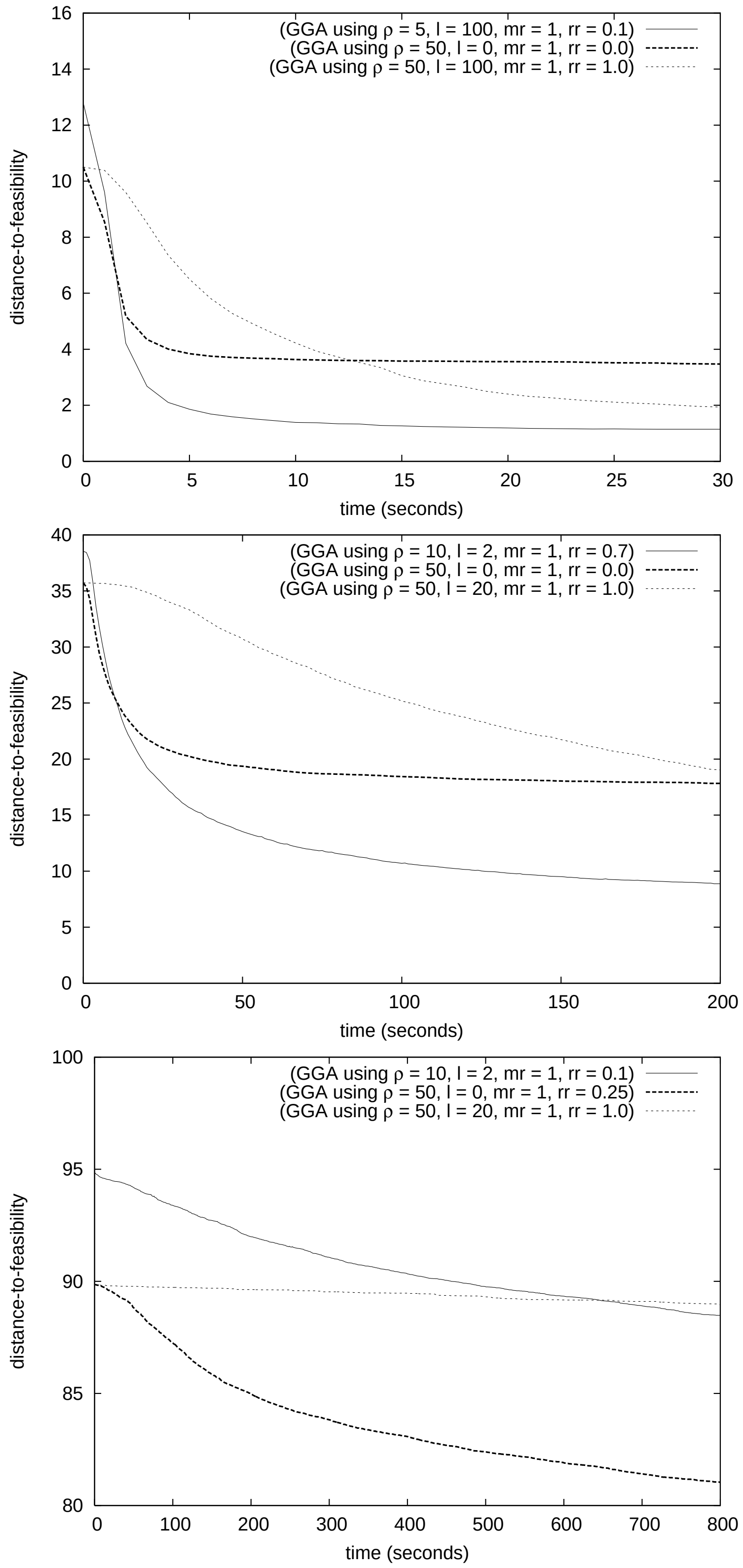


Fig. 4.12: Showing the influence that various amounts of heuristic search have on the number of evaluations performed within the time limits for the different instance sets (using ρ = 50, *rr* = 0.5, *mr* = 2, and *ir* = 4).

In these experiments we also saw that the GGA responded differently to the various parameter settings when dealing with the different instance sets. A short summary of these differences now follows, and example runs with some contrasting parameter settings can also be seen in figures 4.13(a)-(c):

- With the small instances, the best parameter settings (with regards to the distance-to-feasibility achieved within the imposed time limit, averaged across the twenty instances) generally involved using small populations with high amounts of heuristic search and very small (but still present) amounts of recombination. The best results were gained when using ρ = 5, *l* = 100, *mr* = 1 and *rr* = 0.1.

- With the medium instances, the best parameter settings for the GGA (again, with regards to the average distance-to-feasibility achieved within the time limit across the twenty medium instances) generally came when we used small populations, with small (but still present) amounts of heuristic search, and a fairly high rate of recombination. (The best results were given by the parameters ρ = 10, *l* = 2, *mr* = 1 and *rr* = 0.7.) We also found that an increase in any of these parameters usually caused the search to become much slower, particularly for increases in *l*, which would simply cause too much diversity in the population, thus making the recombination operator too destructive and expensive. Alternatively, decreases in any of these parameters usually tended to cause an earlier stagnation of the search.

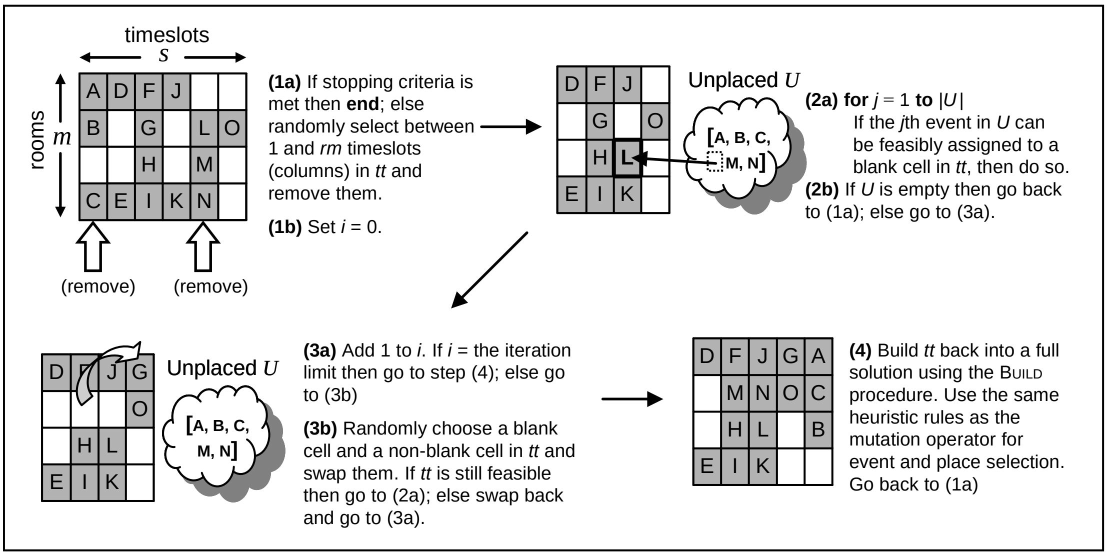


Figs. 4.13(a)-(c): (top, middle, bottom) The effects of various parameter settings with the small, medium, and large instance sets respectively. Each line represents, at each second, the distance to feasibility of the best solution in the population, averaged across 20 runs on each of the 20 instances (i.e. 400 runs). Note, because different population sizes are being used, the lines may not start at the same point on the y-axis.

- Finally, with the large instances, the best results of the GGA were generally gained when using big populations with small amounts of recombination, and no heuristic search whatsoever. (The best results were given using $\rho = 50$, $l = 0$, $mr = 1$, and $rr = 0.25$.) In particular, runs that used both heuristic search *and* recombination always provided disappointing searches. This, we believe, was because the sustained population-diversity offered by the heuristic search would generally cause the recombination operator to do more harm than good; thus, the best results were generally gained when we ensured that many regions of the search space were sampled (by using larger populations) with the majority of the downhill movements then being performed by the less destructive mutation operator.

4.7 Assessing the Overall Contribution of the GGA Operators

Given that the experiments in the previous subsection have indicated that the inclusion of the procedure Heuristic-Search, whilst being able to aid the search in some cases, can still cause the often unhelpful diversity that makes the GGA recombination operator expensive and destructive; the natural question to now ask is: What results can be gained if we abandon the GGA operators altogether, and simply use the heuristic search operator on its own?



(1a) If stopping criteria is met then **end**; else randomly select between 1 and *rm* timeslots (columns) in *tt* and remove them.

(1b) Set *i* = 0.

(2a) for *j* = 1 **to** |*U*|
    If the *j*th event in *U* can be feasibly assigned to a blank cell in *tt*, then do so.

(2b) If *U* is empty then go back to (1a); else go to (3a).

(3a) Add 1 to *i*. If *i* = the iteration limit then go to step (4); else go to (3b)

(3b) Randomly choose a blank cell and a non-blank cell in *tt* and swap them. If *tt* is still feasible then go to (2a); else swap back and go to (3a).

(4) Build *tt* back into a full solution using the Build procedure. Use the same heuristic rules as the mutation operator for event and place selection. Go back to (1a)

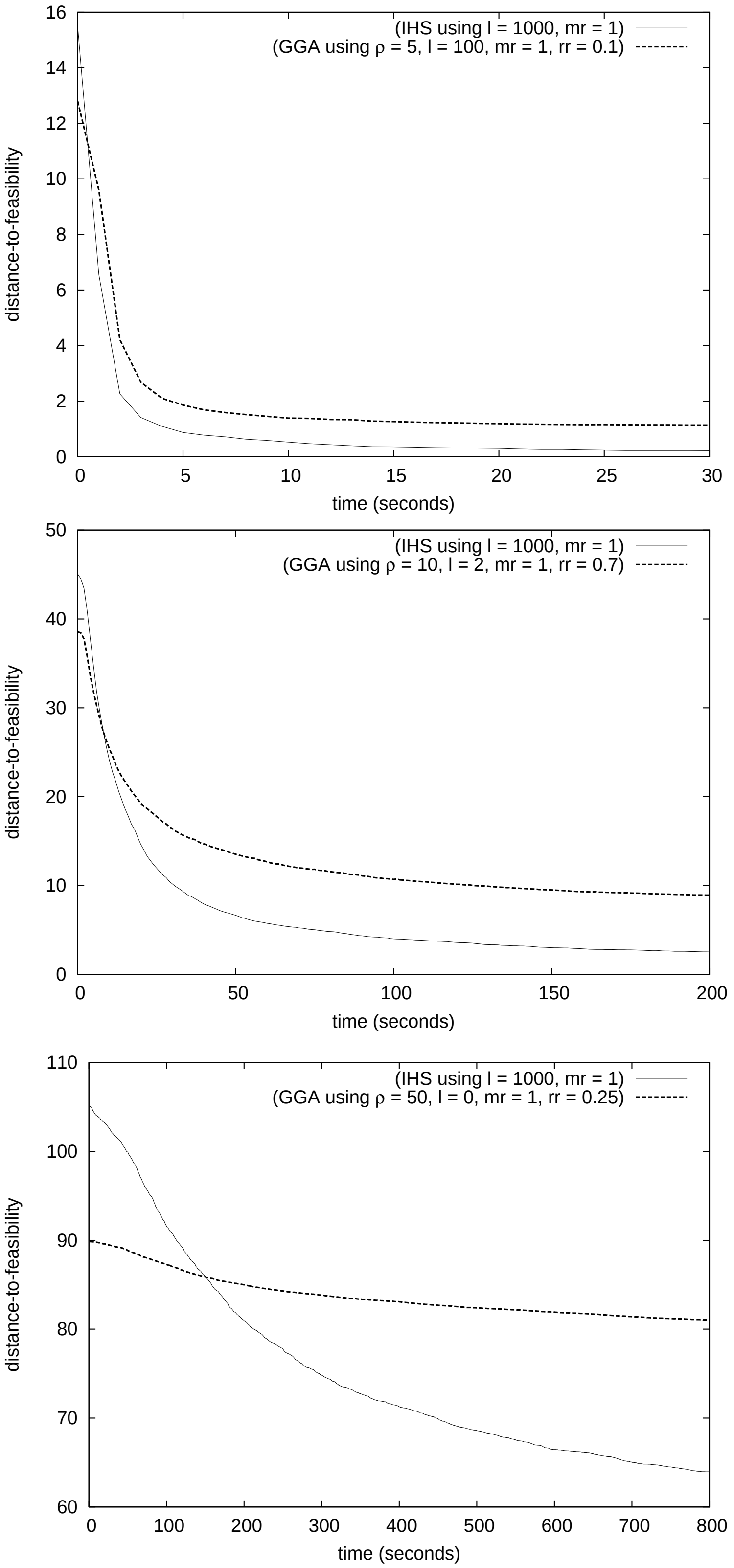
Fig. 4.14: Pictorial description of the Iterated Heuristic Search algorithm for the UCTP.

In order to try and answer this question, we implemented a new algorithm that operated by making just one initial timetable in the same way as described in Section 4.2.1, and then simply repeatedly applied the mutation operator incorporating heuristic search until the time limit was reached. In our descriptions, we will refer to this algorithm as the Iterated Heuristic Search (IHS) algorithm and, for convenience, a pictorial description of the complete IHS algorithm can also be found in fig. 4.14.

In fig. 4.15(a)-(c) we provide a comparison between the new IHS algorithm and the GGA (the latter which is using f_5 and the best performing parameter settings of the previous subsection). A breakdown of the results with regards to the two algorithms' performances is also provided in Table 4.2, where we show average performance, and Table 4.3, where we show the best performance (both taken from twenty runs on each instance).

Considering the large problem instances first (fig. 4.15(c)), because we have now plotted the IHS algorithm against the GGA, we are able to view, in context, some of the negative effects of the GGA operators. As can be seen, the IHS algorithm – which, we note, does not use a population, recombination, or any form of selection pressure – clearly provides the best results on average within the time limits. This, it is able to do, despite starting its run with a timetable that is usually worse than the best timetable present in the initial population of the GGA. These observations, we believe, clearly highlight the various issues raised in Section 4.5 – not least the observation that when the groups in candidate solutions are large, the genetic operators seem to be less beneficial to the overall search. (These differences in results found at the time limit were significant.)

Note that there are also some important differences between the GGA and the IHS algorithm. Firstly, because the GGA requires that a population of individual timetables is maintained, computation time generally has to be shared amongst the individual members. In the case of the IHS algorithm, this is not so. Additionally, the GGA operators of replication and recombination generally have to spend time copying chromosomes (or part of chromosomes) from parent to offspring, which in the space of an entire run, could amount to a consequential quantity of CPU time. Again, with the IHS algorithm, this is not necessary. Differences such as these might offer advantages to the IHS algorithm because, for example, more effort can be placed upon simply trying to improve just the one timetable. Indeed, if the chosen heuristic search operator is not particularly susceptible to getting caught in local optima (as would seem to be the case here) then this may well bring benefits.



Figs. 4.15(a)-(c): (top, middle, and bottom) Comparison of the GGA and the Iterated Heuristic Search algorithm with the small, medium, and large instance sets respectively. Each line represents, at each second, the distance to feasibility of the best solution found so far, averaged across 20 runs on each of the 20 instances (i.e. 400 runs). Note, because the IHS algorithm starts with just one initial solution, it is likely to have a higher distance-to-feasibility than the best candidate solution in the initial population of the GGA, and will thus generally start at a higher point on the y-axis.

Moving our attention next to figs. 4.15(a) and 4.15(b), we can see that the IHS algorithm also outperforms the GGA when dealing with the small and medium problem instances. Indeed, in our experiments the differences in both cases were also seen to be significant, thus demonstrating the superiority of the IHS algorithm in these cases as well. The more powerful search capabilities, presumably, are due to the same factors as those described in the previous paragraphs. However, it is worth noting that the differences in results do seem to be less stark than the results of the large instances, hinting that the GGA is perhaps able to be more competitive when the groups of events (defined by the timeslots) are smaller in size. This also agrees with the arguments regarding the effectiveness of the GGA operators given in Section 4.5.

Table 4.2: A Breakdown of the Average Results Found by the GGA and Iterated Heuristic Search Algorithm over the Sixty Problem Instances. (Results are averaged across twenty runs on Each Instance.) In the Three Columns Marked “P” , Some Supplementary Information About the Instances Is Provided: a “Y” Indicates that we Definitely Know a Perfect Solution to Exist, an “N” Indicates that we Definitely Know there Not to be a Perfect Solution, and a “?” indicates Otherwise.

	Distance to Feasibility (Average 20 runs)								
	Small			Medium			Large		
Instance #	**P**	**GGA**	**IHS**	**P**	**GGA**	**IHS**	**P**	**GGA**	**IHS**
1	Y	0	0	Y	0	0	Y	0	0
2	Y	0	0	Y	0	0	Y	0.7	0
3	?	0	0	?	0	0	Y	0	0
4	Y	0	0	N	0	0	N	32.2	20.5
5	?	1.05	0	N	3.95	0	N	29.15	38.15
6	Y	0	0	Y	6.2	0	N	88.9	92.3
7	?	0	0	?	41.65	18.05	N	157.35	168.5
8	N	6.45	1	Y	15.95	0	N	37.8	20.75
9	N	2.5	0.15	?	24.55	9.7	N	25	17.5
10	N	0.1	0	Y	0	0	N	38	39.95
11	Y	0	0	Y	3.2	0	N	42.35	26.05
12	N	0	0	?	0	0	Y	0.85	0
13	N	1.25	0.35	Y	13.35	0.5	Y	19.9	2.55
14	N	10.5	2.75	Y	0.25	0	Y	7.25	0
15	Y	0	0	N	4.85	0	Y	113.95	28.12
16	Y	0	0	?	43.15	6.4	Y	116.3	57.45
17	?	0.25	0	Y	3.55	0	?	266.55	174.9
18	N	0.7	0.2	?	8.2	3.1	?	194.75	179.25
19	N	0.15	0	N	9.25	3.15	?	266.65	247.35
20	N	0	0	N	2.1	11.45	?	183.15	164.15
Av ± σ		1.15± 2.6	0.22± 0.62		9.01± 12.78	2.6± 4.88		81.0± 86.33	63.9± 77.85

As a final point, it is also worth noting that whilst the results of the GGA that are presented in this section are the product of runs using tuned parameter settings for each instance set, the parameter settings used for the IHS algorithm were only decided upon following our own intuitions (i.e. little empirical parameter tuning was performed). Although, in fact, in our experiences we actually found that the IHS algorithm was quite robust regarding its run time parameter settings, there is, of course, a possibility that results could be further improved with different settings.

Table 4.3: A Breakdown of the Best Results Found by the GGA and Iterated Heuristic Search Algorithm with the Sixty Problem Instances (Taken From Twenty Runs on Each Instance).

	Distance to Feasibility (Average 20 runs)					
	Small		**Medium**		**Large**	
Instance #	**GGA**	**IHS**	**GGA**	**IHS**	**GGA**	**IHS**
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	30	8
5	0	0	0	0	24	30
6	0	0	0	0	71	77
7	0	0	34	14	145	150
8	4	0	9	0	30	5
9	0	0	17	2	18	3
10	0	0	0	0	32	24
11	0	0	0	0	37	22
12	0	0	0	0	0	0
13	0	0	3	0	10	0
14	3	0	0	0	0	0
15	0	0	0	0	98	0
16	0	0	30	1	100	19
17	0	0	0	0	243	163
18	0	0	0	0	173	164
19	0	0	0	0	253	232
20	0	0	0	3	165	149
Av ± σ	0.35± 1.1	0.0± 0.0	4.7± 10.0	1.0± 3.1	71.5± 80.3	52.3± 72.6

4.8 Conclusions and Discussion

In this chapter we have examined a number of different issues regarding GGAs and their applicability to the UCTP. We have noted that the task of satisfying the hard constraints of this problem fits the classical definition of a grouping problem, and consequently, using the guidelines suggested by Falkenauer [54, 58], we have designed and analysed a GGA specialised for this problem that combines the standard GGA operators with powerful constructive heuristics. In experiments we have observed that recombination *can* aid the evolutionary search in some cases, whilst in others, depending on the run time available, it might be more of a hindrance.

In this chapter we have also introduced a way of measuring population diversity and distances between pairs of individuals for the classical grouping representation. Consequently, we have seen that diversity, together with group size, can drastically influence (a) the overall expense of the recombination operator, and (b) the ease in which the underlying building-blocks are combined and passed from one generation to the next. We have also noted that there may be other issues with this type of representation, due to the fact that larger groups will mean that chromosomes become proportionally shorter in length. While we still believe that it is indeed the groups that encapsulate the underlying building-blocks of grouping problems such as the UCTP, in this chapter we have thus highlighted areas where the propagation of these can often be quite problematic.

In this chapter we have also examined two ways that the performance of this GGA might be improved: first, through the use of a number of different fitness functions and second, by introducing an additional heuristic search operator. In particular, we have seen that, in some cases the more fine-grained fitness functions (such as f_5 and f_6) *can* produce significantly better results, but in other instances this is not so. We have also seen that the introduction of a heuristic search operator to this GGA can improve results, but probably needs to be used with care, as in some cases its use can mean that (a) not enough new individuals are produced within the time limits, and (b) the added diversity that it brings can cause the recombination operator to be too destructive and expensive to benefit the search.

Note that an advantage of EAs in timetabling that is sometimes noted by authors is that they will usually result in a *population* of candidate solutions, as opposed to just one., meaning that the real-world user will usually have the option of choosing the timetable that best fits his-or-her needs (see for example the arguments of Colorni *et al.* [37]). However, as we have seen in our case, in order for this particular EA to perform at a reasonable level, population-diversity usually needs to be tightly controlled due to the effects that it can have on the behaviour of the GGA recombination operator. Additionally, as we can see in the example runs in figs. 4.10(a)-(c), by the time that reasonably good regions of the search space have usually been reached, the population-diversity has usually dropped to a fairly low level anyway, particularly for the more fine-grained fitness functions f_4, f_5, and f_6. It is debatable therefore, whether this particular advantage is really present in this case.

Note also that we have not yet considered the issue of soft constraint satisfaction with the algorithms of this chapter. This was never our aim, and indeed, we will be looking at this particular issue in more detail in Chapter 6. However, it is worth noting here that there might be additional complications if we *were* to attempt their inclusion. For instance, the soft constraints SC_2 and SC_3 (see Section 3.1) clearly depend on the *ordering* of the timeslots – a factor that is not considered by the GGA or the IHS algorithm in this chapter. One way in which we could incorporate these soft constraints might therefore be to add an additional step to these algorithms so that once a feasible timetable using a maximum of *t* timeslots has been achieved, the timeslots (groups) are then ordered in some way to try and satisfy as many of these soft constraints as possible. We will look at an algorithm that does something along these lines in Section 6.2 later. However, if we wanted the GGA or IHS algorithms to consider soft constraints *during* a run (i.e. when also looking for feasibility) then it is likely that we would have to follow the strategy of other authors such as Erben [52], and add penalty measures (possibly through the use of weights) to the fitness functions. However, because the chief aim of this algorithm is to find feasibility, such a modification might actually have the adverse effect of sometimes leading the search away from attractive (i.e. fully feasible) regions of the search space. Meanwhile, the incorporation of other sorts of soft constraints might present fewer difficulties. Consider, for example, soft constraint SC_1 (Section 3.1). If we were to try and reduce the target number of timeslots from forty-five down to forty, this would actually mean that candidate timetables would be deemed feasible when the number of timeslots being used fell below forty-five, but also the total number of violations of this soft constraint would also fall as the number of timeslots being used approached forty.

Finally, perhaps the most prominent observation made in this chapter is the fact that in the majority of instances, we have seen that the GGA – even in its improved states – has been outperformed by the more straightforward IHS algorithm that, notably, does not make use of a population, selection pressure, or the grouping recombination operator. In particular, we have seen that the superior performance of this algorithm is most evident in the large UCTP instances where, for reasons that we have discussed (particularly in Section 4.5), the GGA operators seem to display the least potential for aiding the search. We believe these observations to be important, because they seem to highlight difficulties that are inherent not just with this application, but with the GGA approach as a whole. However at this point, we must be careful about making such strong claims because we have little empirical evidence of this from other problem domains. In the next chapter we will thus temporarily divert our attentions from timetabling, and we will seek to reinforce these assertions. This we will do by performing a second comparison between a GGA and an IHS algorithm using a different type of grouping problem – Graph Colouring with Equipartite Graphs. If similar observations to the experiments in this chapter are observed then this will serve to strengthen our arguments about the GGA approach in general. In

Chapter 6 we will then return to our study of the UCTP, concentrating our efforts on the task of satisfying the soft constraints.

5: Further Investigation: Application to Equipartite Graph Colouring

In the previous chapter, one of the main observations made in our experimental analyses was that the performance of the grouping genetic algorithm (GGA) for the UCTP, generally worsened when instances with relatively high numbers of items per group were encountered. We hypothesised that this was due to two reasons: (a) because larger groups generally cause the GGA recombination operator to become more destructive, and (b) because larger groups cause the chromosome to become proportionally shorter in length, thus placing possible limitations on the search-capabilities of the GGA genetic operators.

Presently, however, the effects of these observations have only been witnessed when dealing with the UCTP, and we have little experimental evidence as to whether these characteristics might also occur when applying GGAs to other similar problems. In this chapter we will therefore take a slight excursion from our studies of university course timetabling and will take a look at a second type of grouping problem: Graph Colouring with Equipartite Graphs. By applying a GGA to this problem, we will investigate as to whether similar observations to those given in Chapter 4 can be made. As a means of comparison, we will once again compare and evaluate the GGA's performance against an

IHS algorithm which, on the whole, will be of a very similar style to our previous IHS algorithm, described in Section 4.7. Note that in this chapter, unlike other chapters in this thesis, the variable *n* will be used to denote the number of nodes in particular graph colouring problem instance. Additionally, the variable *m* will be used to denote the maximum number of nodes that can be assigned to any one particular colour class in an optimal solution.

The reason why we feel it is appropriate to study graph colouring with equipartite graphs here, as opposed to graph colouring in general, is that it is quite similar in structure to the timetabling problem that we have been considering thus far and therefore remains with the general theme of this thesis: in equipartite graphs, the *n* nodes of the graph are partitioned into χ almost equal-sized independent sets and edges are then only permitted between nodes from distinct sets. They are thus a special type of graph colouring problem where each colour class in an optimal solution cannot contain more than $m = \lceil n/\chi \rceil$ nodes. This latter feature therefore adds a characteristic to these problems that is similar to the UCTP where, as we have seen, we cannot assign more than *m* events to a particular timeslot without breaking a hard constraint (remembering that in the UCTP, *m* is used to represent the number of available rooms). Additionally, in this chapter we will also conduct our experiments mainly on instances where χ = 40, thus also giving these problems the same sort of "flavour" as problem instances of the UCTP in which perfect solutions are obtainable. Finally, the equipartite characteristic also allows us to easily control the sizes of the groups (colour classes) in the problem instances, which, as we will see, is also be useful for the purposes of our experiments.

In the next two subsections we will describe the two algorithms that we will use to perform this comparison. Next, in Section 5.3 we will describe the experiments and criteria used for comparing the two algorithms in the test, and in section 5.4 we will present our results. Finally Section 5.5 will conclude the chapter and also provide some further points of discussion.

5.1 A GGA for Graph Colouring

We noted in the previous chapter that two GGAs for graph colouring have already been proposed in the literature: first by Eiben, van der Hauw, and van Hemert in 1998 [50], and then later by Erben in 2000 [52]. In fact, the chief difference between these two algorithms is simply the fitness functions that are used: the former uses an evaluation method that is equivalent to f_3 described in Section 4.6.1, whilst the latter uses fitness function f_5. In [52], Erben claims that this function allows his algorithm to return substantially better results (than Eiben *et al.*'s) because it "provides … a search space landscape that is virtually free of plains, and thus offers more information to be exploited." (Remember that similar observations were also made in Section 4.6.2, where we saw better results being returned by f_5 with some of the UCTP instances).

Due to the claimed superior performance of Erben's GGA (in general), we therefore choose to use this GGA-version in our experiments. In the following paragraphs, we will now describe this algorithm. Note that although there is already a description of this algorithm in [52], it is necessary to include a description here, because some of the operational details that are necessary for repeating the experiments are not actually present in the original paper. These details are thus provided here, and were established through open contact with the author of the original paper. Interestingly, our initial tests with the algorithm indicated that if these supplementary details were not incorporated into the algorithm, then performance would sometimes drop quite considerably.

Using typical GGA terminology, Erben's GGA for graph colouring operates by considering the nodes of the graph as the *items*, with the objective then being to try and arrange these into *groups* (i.e. colours) of non-adjacent nodes, such that the number of colours is minimal. In order to form an initial population, each individual is created by first randomly permuting the n nodes, and by then applying a standard greedy (or first-fit) procedure [42] to colour them. (This greedy algorithm operates in the usual way of taking the nodes one-by-one one from left to right, and assigning them to the first colour that does not cause a colour conflict). Thus, members of the initial population do not feature any occurrences of a colour conflict, but, of course, the number of colours being used by each individual (i.e. chromosome length) can vary.

The GGA recombination operator then follows the standard GGA method [54, 58]: given two parent chromosomes, p_1 and p_2, some groups (colour classes) in p_1 are randomly selected, and copies of these are then injected into a copy of p_2. Next, duplicate nodes that occur as a result of this injection are removed using *adaptation* (see Section 4.2.2). Finally, a rebuild scheme is employed to re-colour any uncoloured nodes that occur as a result of this step. This scheme works by randomly permuting both the uncoloured nodes and the existing colour classes, and then simply applies the greedy algorithm as before. A second offspring is then produced by swapping the roles of the two parents. (Note that in the conclusions to his paper, Erben states that this crossover (recombination) operator was beneficial to the algorithm's search capabilities in all performed experiments.)

The mutation operator also functions in a typical GGA way: some colour classes in a chromosome are chosen at random and removed. The chromosome is then rebuilt in the same way as the recombination operator.

Finally, in his experiments Erben uses the *noisy sort* steady-state population strategy of Falkenauer [54, 58], and chooses to make use of three parameters: a population size ρ, a mutation rate *mr* in the range [0, 1] and a recombination rate *rr* in the range [0, 0.5]. The overall behaviour of this algorithm is as follows:

(1) Generate an initial population of ρ individuals and evaluate them.

(2) Perform a *noisy sort* of the population (i.e. starting with an empty list *L*, repeatedly select two individuals from the population at random, apply tournament selection, and transfer the loser to the head of *L*. When only one individual remains in the population, place it at the head of *L*. We thus have a partially sorted list with the weaker individuals tending to be towards the tail and with the best individual at the head).

(3) Take the top $\lfloor rr.\rho \rfloor$ individuals in *L* and recombine them with each other. Copy these offspring over the bottom $\lfloor rr.\rho \rfloor$ individuals in *L*.

(4) Apply mutation by going through every individual and removing each group with a probability *mr*.

(5) Re-evaluate the new and altered individuals.

(6) If any of the stopping criteria (see below) are met then end; else go back to step (2).

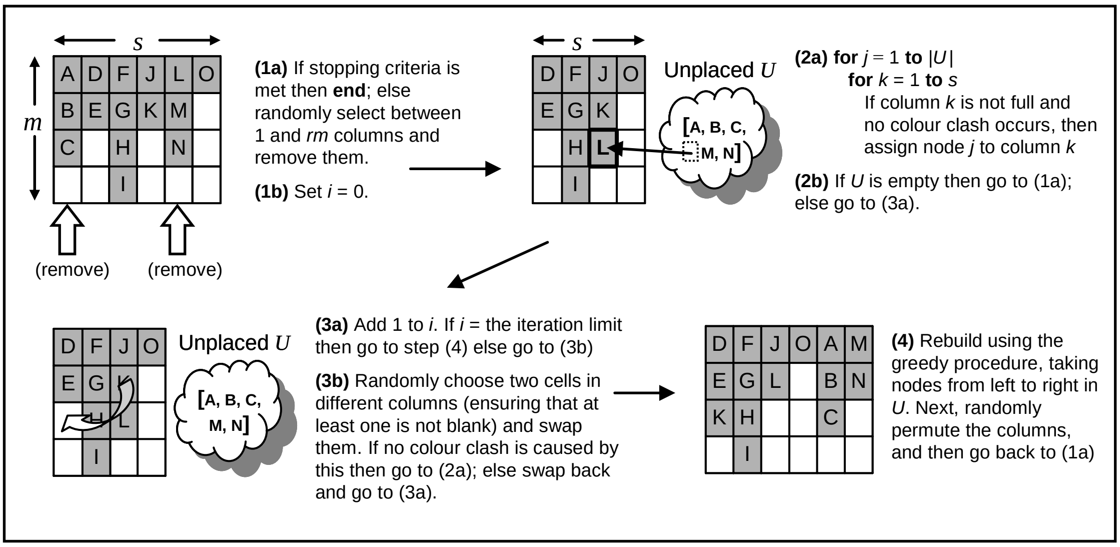
5.2 An Iterated Heuristic Search Algorithm for Equipartite Graph Colouring

The second algorithm in our comparison is very much like the Iterated Heuristic Search (IHS) algorithm for the UCTP that we described in Section 4.7: just one initial solution is created, and the algorithm then makes successive attempts to try and reduce the number of groups (colours) down to some target amount. There are, however, a few differences present due to the slight variations in these two problems, and so we will describe these differences here.

In order to construct an initial solution, we use the Dsatur algorithm of Brélaz [16] to determine the order in which the nodes are coloured, with a balancing heuristic [15] then being used to select a colour for each of these. This balancing heuristic simply assigns each node to the existing colour that contains the least number of nodes, creating a new colour when no exiting colour is suitable. This approach was chosen because in our initial tests with equipartite graph colouring problem instances, we saw that this heuristic nearly always produced solutions that used fewer colours that the standard Dsatur algorithm (which just chooses the first available colour for each node).

Similarly to our previous IHS algorithm in Section 4.7, throughout the run the candidate solution is stored in an ($m \times s$) matrix with each column, 1 to s representing a unique colour class. As before, the number of columns being used in the matrix is allowed to vary, with the aim of minimisation. Additionally, in this case the number of rows m in the matrix is a constant that represents the maximum number of nodes that can be contained in any one colour class, and because we are dealing with equipartite graphs, this is known to be at most $m = \lceil n / \chi \rceil$. Note, however, that unlike the matrix representation used with the UCTP, in this case we do not need to concern ourselves about which row each node is assigned to (any row is suitable) and so for consistency, we choose to always keep the blank cells of the matrix at the bottom of each column (see fig. 5.1 for an example).

An initial solution having been formed, the IHS algorithm then operates following the steps outlined in fig. 5.1. In these steps, two parameters are used: an iteration limit and a parameter *rm*, and the meanings of these are described in the figure.



(1a) If stopping criteria is met then **end**; else randomly select between 1 and *rm* columns and remove them.

(1b) Set $i = 0$.

```
(2a) for j = 1 to |U|
        for k = 1 to s
           If column k is not full and
           no colour clash occurs, then
           assign node j to column k
```

(2b) If U is empty then go to (1a); else go to (3a).

(3a) Add 1 to i. If i = the iteration limit then go to step (4) else go to (3b)

(3b) Randomly choose two cells in different columns (ensuring that at least one is not blank) and swap them. If no colour clash is caused by this then go to (2a); else swap back and go to (3a).

(4) Rebuild using the greedy procedure, taking nodes from left to right in U. Next, randomly permute the columns, and then go back to (1a)

Fig. 5.1: The Iterated Heuristic Search Algorithm for Equipartite Graph Colouring

With reference to fig. 5.1, as a final point it is worth mentioning a further detail about step (4), which is used to rebuild partial solutions when the iteration limit is reached. Our reasons for choosing the described method are due to some simple characteristics of the greedy (first-fit) algorithm that were first pointed out by Culberson and Luo in [42], and which are best illustrated by an example:

In step (1) of fig. 5.1 we notice that two columns are removed from the matrix. The nodes contained within these two columns are then placed into a list U = [A, B, C, L, M, N]. Next, in step (2a) we see that we have managed to insert one of these nodes, L, back into the matrix. Let us now assume that that for the remainder of this procedure, no more nodes are reinserted into the matrix before reaching the iteration limit, and so, as in fig. 5.1, we arrive at step (4) with U = [A, B, C, M, N]. Note that the original ordering of the nodes in U (minus K) has been preserved. Additionally, we know that nodes A, B, and C definitely cannot have a common edge between them, because otherwise they would never have been placed into the same column (colour class) in the first place (similarly for nodes M and N). This means that an application of the greedy algorithm using this ordering of nodes will *never* need to add more than two extra columns (colours) to the matrix. However, during the greedy process, there is still, of course, a possibility that some nodes in U could be assigned to *existing* colours, thus creating the possibility of needing to open *fewer* than two.

When the above feature is tied in with the rest of the algorithm this actually means that during an entire run, the number of colours being used in a solution can never actually increase. Thus the overall IHS algorithm might be considered a type of hill-climber, with a stochastic mechanism for crossing plateaus in the fitness landscape perhaps being supplied by steps (3) and (4).

5.3 Comparing the Algorithms

In our comparison, both algorithms were designed to halt when either a solution using χ colours was found, or when a predefined time limit (measured in CPU seconds) was reached. Note that in their experiments, neither Eiben *et al.* [50] nor Erben [52] actually consider CPU time in their analyses – instead choosing to measure algorithm performance according to the number of evaluations performed. However, the latter measure is not really appropriate in our case because the IHS algorithm does not actually need to "evaluate" a candidate solution as such, since its hill-climbing nature makes it guaranteed never to produce a worse colouring. Indeed, in this case, a candidate solution's "quality" (i.e. the number of colours it is using) only ever needs to be checked to see if the first halting condition above has been met. Secondly, it is also worth remembering that in the case of a GGA, quality measures according to the number of evaluations are also likely to hide the varying amounts of computational effort that are needed to perform recombination at different stages of the evolutionary process due to the extra amounts of rebuilding that usually need to take place when the population is still diverse and when the groups are large (as we discussed in the previous chapter).

However, it should also be noted that the comparison of algorithms according to CPU time also has its pitfalls, not least because it brings into question issues of implementation and hardware. In our case we have attempted to avoid potential discrepancies (as much as possible) by implementing both algorithms with the same language and compiler (C++ under Linux using g++ 3.2.2), and by using common data-structures and code-sharing whenever appropriate to do so. Naturally, all experiments were also conducted on the same hardware, the specification of which is the same as previous experiments.

In our tests, similarly to Eiben *et al.* and Erben, all problem instances were created using the instance generator of Culberson [4, 42]. For various values of χ and n (see below), we performed experiments for a range of edge connectivities p at and around the phase transitions in each case. More specifically, for each p-value, we produced 10 different problem instances and then performed 10 separate trials on each instance, giving a total of 100 trials at each setting. We also used four different criteria for comparing the algorithms which are as follows:

(1) **Robustness**, which is indicated by a *Success Rate* in the range [0, 1]. This tells us the proportion of runs in which the algorithms were able to find an optimal solution within the time limits. (Note that the instance generator used always produces problem instances where the optimum is achievable – thus the maximum possible success rate is always 1.0)

(2) **Computational Effort**, which is indicated by the *Solution Time*. This tells us how long the algorithms took to find an optimal solution in CPU time, if indeed one was found. (In cases where the success rate is less than 1.0, those runs where optimality could not found within the time limits are not considered for the calculation.)

(3) **Accuracy**, which is indicated by the *Number of Colours* used in the best solution found during the run, averaged across the 100 runs. This measurement is therefore meant to give us some indication of how *close* the algorithms got to optimality on average.

(4) **Run Characteristics**, Finally, this criterion is also included to help give us some indication as to how the algorithms actually progress through the search space over time when dealing with "hard" instances (i.e. those with an edge connectivity level p somewhere within the phase transition regions of both algorithms). This is

demonstrated by plotting over time the number of colours used in the best solution found so far (averaged across the 100 runs) at a selected p-value.

Lastly, the time limits for each set of experiments were determined by conducting all of our experiments with the GGA on each set of problem instances twice: in the first case we ran the algorithm for a fixed number of evaluations (this amount would be determined by looking at similar experiments of Eiben *et al.* [50] and Erben [52]). During these trials we would then record the largest amount of CPU time that was required in any of these runs, and a similar value would then be used as the official time limit in our comparison.

5.4 Analysis of Results

In our first comparison, shown in fig. 5.2, we compare the algorithms when considering equipartite graphs with a chromatic number $\chi = 8$ and number of nodes $n = 200$. As with the work of Erben [52], the parameters used for the GGA were $\rho = 20$, $mr = 0.05$, $rr = 0.2$, and for the IHS algorithm we used $rm = 1$ and an iteration limit of $1000n$ (our use of n in defining the latter allows the procedure to scale up with instance size). The CPU time limit used in these experiments was 90 seconds, which was the maximum amount of time that was required to perform 150,000 evaluations with the GGA (this was the number of evaluations used as the cut off point in the same experiments in Erben's algorithm analysis).

One reason for including a comparison with these particular instances is to demonstrate that the results of our implemented GGA are very similar to the results claimed in the same experiments in Erben's paper (which used the same parameters as above), thus justifying to a certain extent our method of CPU time limit calculation. However, we can also see that with regards to the robustness of the two algorithms, although the phase transition occurs around the same values of edge connectivities p, the curve of the IHS algorithm is definitely narrower, thus demonstrating a higher success rate on a greater range of instances. Note that the accuracy of the IHS algorithm is also superior across the phase transition region, and that, in cases of $p = 0.2$, it also seems to be able to make more positive movements through the search space for a greater period of the run.

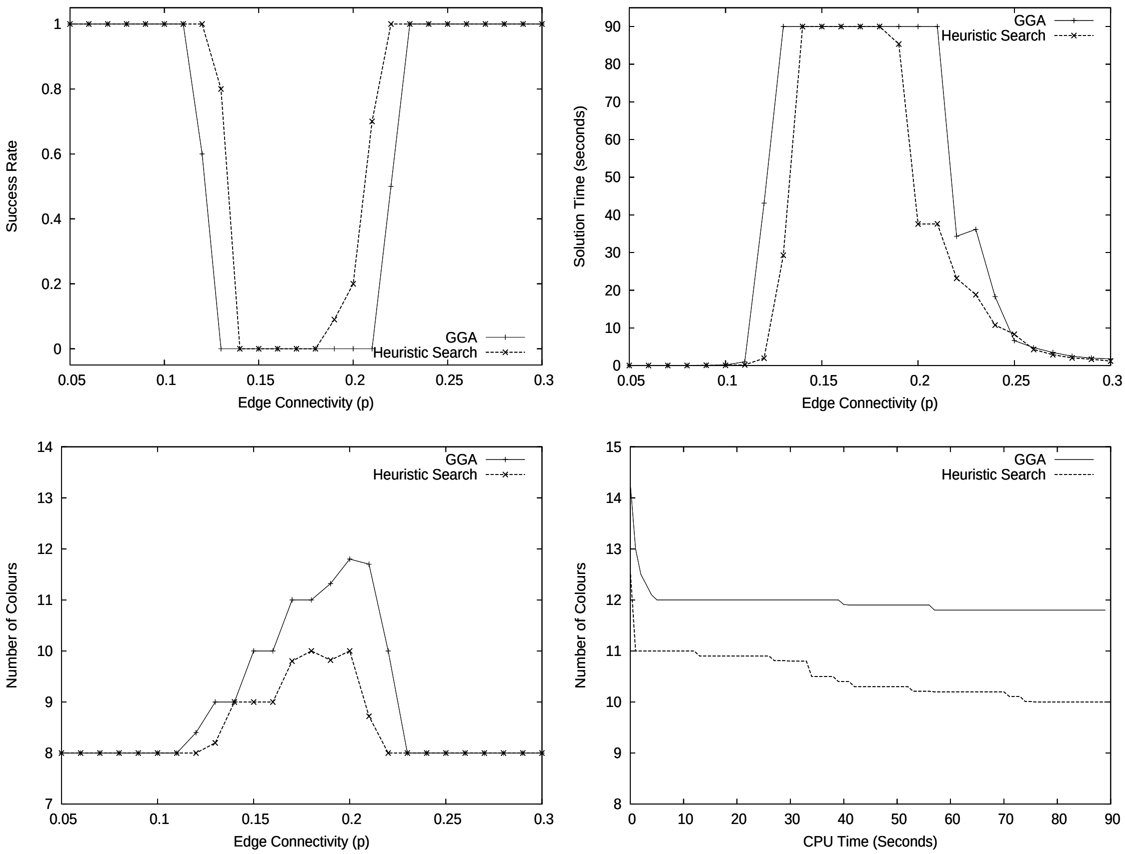


Fig. 5.2: Comparison for equipartite graphs, for n = 200 and χ = 8 (thus m = 25), using a time limit of 90 CPU-seconds. The Run Characteristics-graph (bottom right) was made using instances where p = 0.2.

For the remainder of our experiments and analysis we will now look at equipartite graphs in which the chromatic number χ = 40. These instances thus represent problems that are more similar in style to the UCTP. In all cases, time limits were determined by observing the maximum number of seconds needed for the GGA to perform 300,000 evaluations in total (this stopping criterion was used in similar experiments by Eiben *et al.*). Also, following the advice of Erben, for the GGA we used fairly small populations (ρ =20) and chose to set the recombination and mutation rates to 0.2 and 0.02 respectively. The parameters used for the IHS algorithm, meanwhile, were set the same as the previous experiment.

Figure 5.3 shows the behaviour on instances for n = 200 and χ = 40. Interestingly, in this case we can see that the phase transition regions of the two algorithms are slightly different, with the GGA's being slightly to the right of the IHS algorithm's. Indeed, between p-values 0.68 and 0.75 we can clearly see that the GGA exhibits a higher success rate, is more accurate and takes less computational effort on average. Moving beyond these values we see that the IHS algorithm then begins to show superior performance and, for p = 0.8, for example, we can see that the GGA actually needs to use six extra colours on average, whilst the IHS algorithm tends to need less than one on average. However, looking at the run characteristics on instances where both algorithms show a success rate of zero (in this case $p = 0.78$), we see that the algorithms seem to behave quite similarly over time and both tend to find solutions of similar quality of around 44 colours.

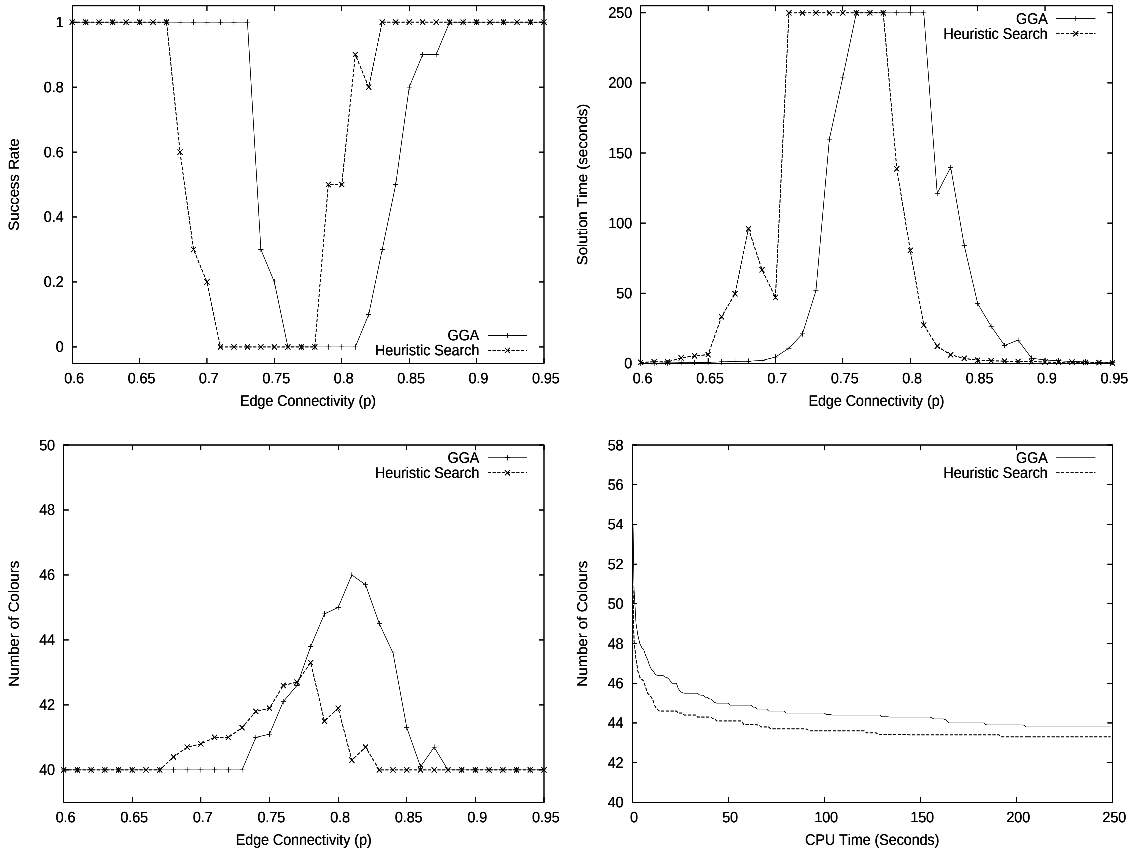


Fig. 5.3: Comparison for equipartite graphs, for $n = 200$ and $\chi = 40$ (thus $m = 5$), using a time limit of 250 CPU-seconds. The Run Characteristics-graph (bottom right) was made using instances where $p = 0.78$.

Looking next at the algorithms' behaviour with equipartite graphs for $n = 400$ and $\chi = 40$ (fig. 5.4), we can see that the phase transition region of the GGA is wider than the IHS algorithm's, thus indicating a larger range of instances that it cannot solve (or come close to solving). Perhaps even more telling though is the contrast between the algorithms' behaviour with regards to accuracy: here we can see that throughout the phase transition of both algorithms, even in cases where neither algorithm is able to find a solution, the IHS algorithm produces solutions using fewer colours than the GGA on average. Also note that between p-values 0.625-0.675, for example, whilst the IHS algorithm was able to find optimal solutions in relatively short amounts of time in every case, the GGA can only achieve solutions that are using 20-or-so extra colours within the time limit.

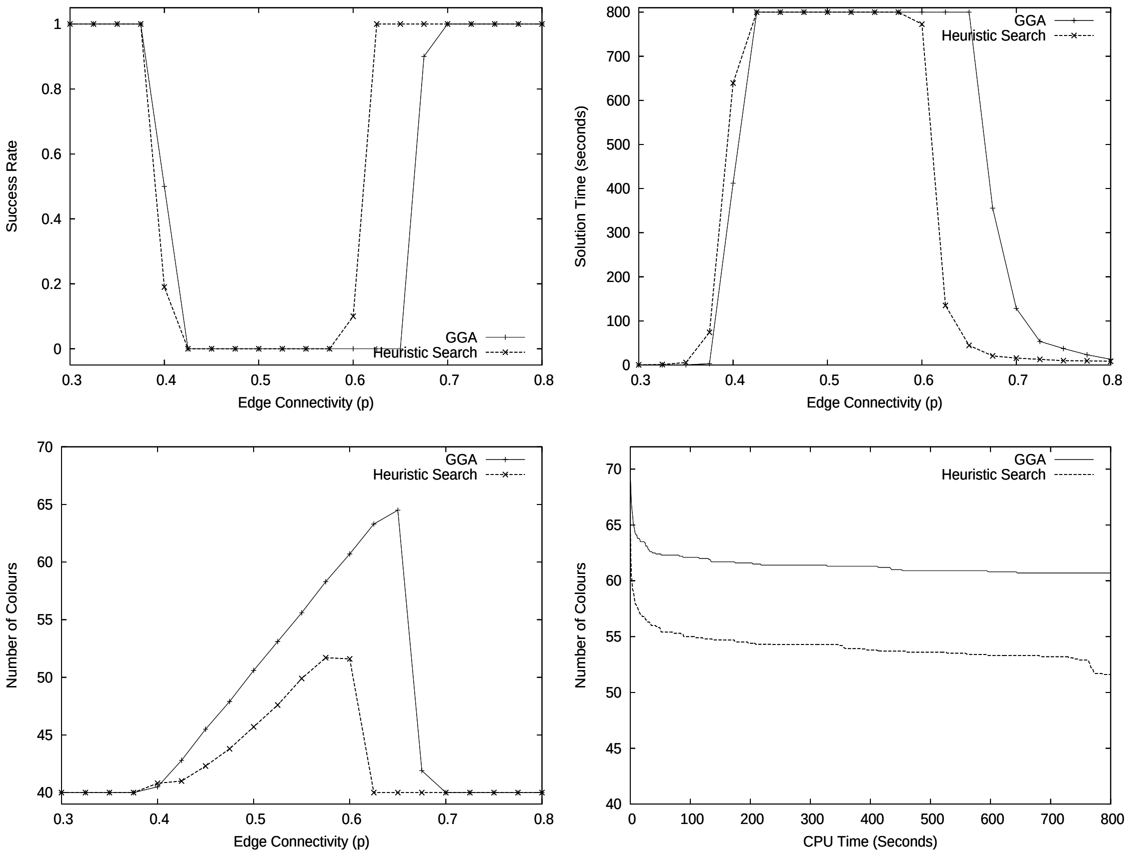


Fig. 5.4: Comparison for equipartite graphs, for n = 400 and χ = 40 (thus m = 10), using a time limit of 800 CPU-seconds. The Run Characteristics-graph (bottom right) was made using instances where p = 0.6.

Finally, in fig. 5.5, we compare the algorithms using equipartite graphs of n = 1000 and χ = 40. Here, we witness similar characteristics to the previous experiment. However, the difference in performance between the two algorithms seems to be even more apparent in this case, with the phase transition region of the GGA covering almost two times as many p-values as the IHS algorithm. Note also, for example, that for edge connectivities p = 0.45-0.6, whilst the IHS algorithm is able to find optimal solutions in relatively short amounts of time in every run, the GGA can only produce colourings of approximately 2χ colours within the time limit.

5.5 Conclusions and Further Discussion

From the results presented in this chapter, it should be clear to the reader that, similarly to our experimental observations made in the previous chapter, the IHS algorithm for equipartite graphs seems to give a better overall performance than the GGA. Also noticeable is that in cases where we witness low success rates from both algorithms (i.e. in the phase transition regions), the IHS algorithm also seems to show better accuracy than the GGA in the majority of cases.

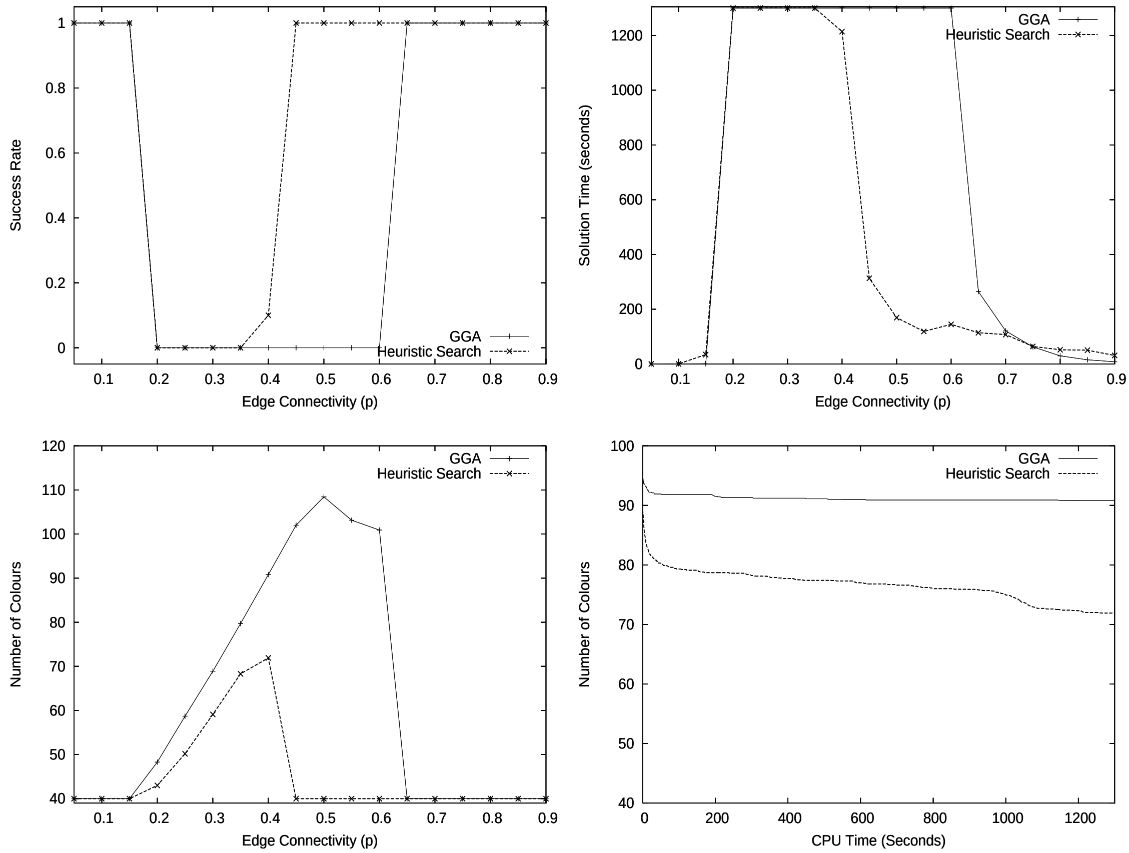


Fig. 5.5: Comparison for equipartite graphs, for n = 1000 and χ = 40 (thus m = 25), using a time limit of 1300 CPU-seconds. The Run Characteristics-graph (bottom right) was made using instances where p = 0.4.

We have also seen from these experiments that the inferior behaviour of the GGA seems once again to be at its most noticeable when the sizes of the groups are larger and thus the relative lengths of the chromosomes are shorter. Indeed, when the GGA *did* sometimes exhibit comparable or better performance than the IHS, this was usually when the problems being solved involved relatively small groups (in our case where χ = 40, n = 200, and thus m = 5).

Once again, clues as to why the GGA is able to compete in these cases are revealed when we examine how two related measures – (1) the population diversity and (2) the amount of rebuild being performed during recombination – seem to vary during the evolutionary process. Figure 5.6 shows these for a single run on instances with edge connectivities within the phase transition regions for each of the four instance sets. Here it can be seen quite clearly that only in fig. 5.6(b), where m = 5, do the two measures seem to actually settle at low levels during the run. As before, we hypothesise that the smaller groups in this case allow the underlying building-blocks of the problem (i.e. the groups) to be propagated more effectively during evolution, and therefore allow the population to converge around a relatively good point in the search space. In contrast, for the remaining three cases (shown in figs. 5.6(a), (c), and (d)), we see that these measures seem to remain at much higher levels throughout the run, implying that the recombination operator is once again acting as more of a macro-mutation operator in these cases, and less as an operator for combining and passing on good building-blocks from one generation to the next. This agrees with the observations that we made in the previous chapter.

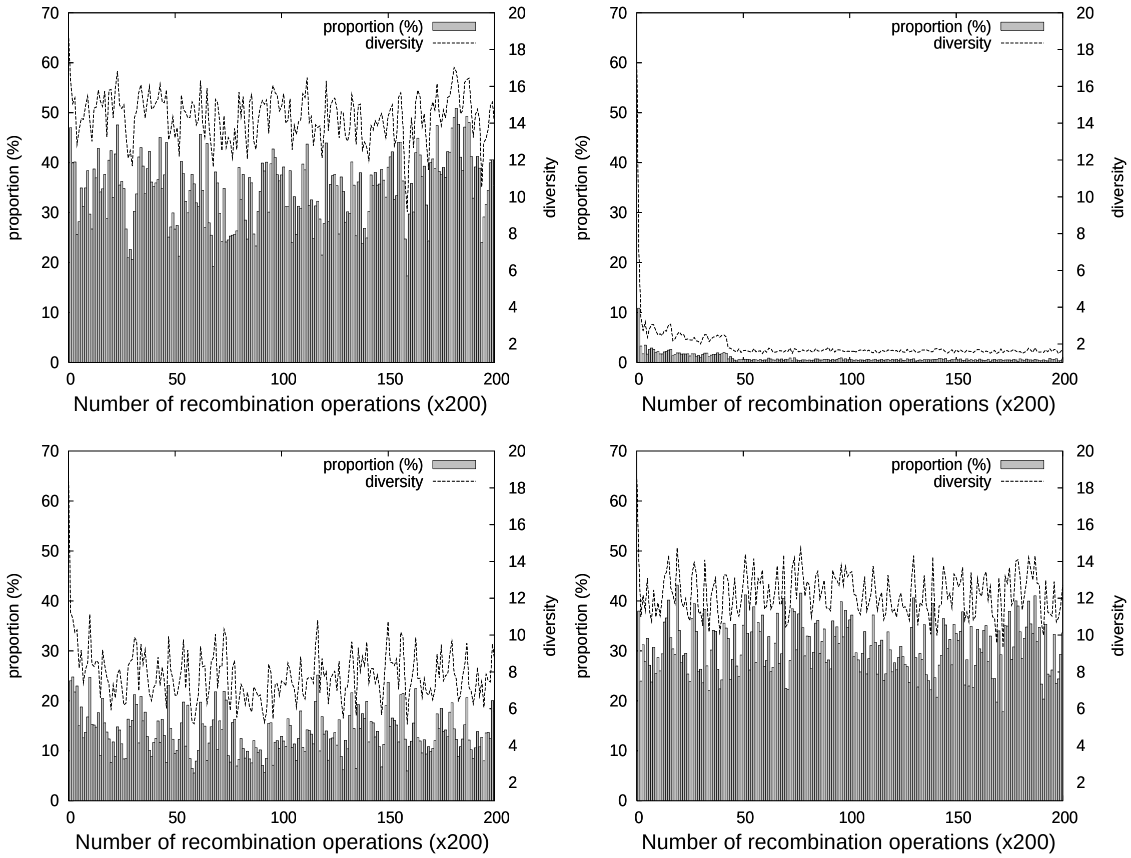


Fig. 5.6(a)-(d): (top-left, top-right, bottom-left, bottom-right respectively) Four example runs showing (1) the population diversity, and (2) the proportion of nodes becoming unplaced with each application of the recombination operator, during example runs with the GGA. Graphs (a)-(d) show runs with instances of n = 200, χ = 8, and p = 0.2; n = 200, χ = 40 and p = 0.78; n = 400, χ = 40 and p = 0.6; and n = 1000, χ = 40, and p = 0.4 respectively. Run-time parameters used in these runs were the same as the described experiments. In the figures, the amount of rebuilding needing to be done with the recombination operator is labelled *Proportion (%)* - this refers to the percentage of nodes that are becoming unplaced, on average, with each application of the recombination operator.

Finally, before ending this chapter, it is also probably worth mentioning some contrasting features of these two algorithms in order to further facilitate our understanding of this comparison so that we may view these results in their proper context.

Firstly, it should be noted that the GGA does not currently need to know the chromatic number χ of a problem instance in advance (although, in our case, we do actually make use of the information in order to specify when the algorithm can halt). On the other hand, the IHS algorithm uses this information for calculating *m* (i.e. the number of rows that will be present in the solution matrix)[15]. As we have already noted in Section 2.2, however, the task of calculating the chromatic number in graph colouring is NP-hard, and so this information might not always be easy to come by. If we consider these implications with regards to course timetabling, however, this particular feature might not be of such great consequence because, as we have seen in our analysis of the UCTP, it is usually the case that *m* (in this case the number of rooms available per timeslot) is known in advance, and/or the maximum number of timeslots is stipulated as part of the problem definition.

Secondly, it is also worth remembering that the IHS algorithm considered here is currently designed specifically for equipartite graphs, whilst the GGA, on the other hand, is intended for dealing with all types of graph colouring problem. In Erben's work [52], for instance, it has been shown to cope quite well with pyramidal chain-style graph colouring problems (see the work of Ross *et al.* [94]) – a type of problem that the IHS algorithm could not currently deal with without some modifications first being made.

With regards to the parameters of the algorithms, we found that the IHS algorithm generally seemed quite robust and, providing that *rm* was kept relatively small (e.g. 1-3) and the iteration limit was kept relatively high (more than 500*n*, say), we tended to witness fairly consistent behaviour across the instances. Conversely, we found that the GGA was usually very sensitive to changes in its three parameters, with alterations to any of them usually resulting in quite drastic changes to algorithmic performance. In our tests we thus kept their settings very close to those recommended by Erben [52], which were reported to perform well in his experiments. Additionally, in this case we also found that other population strategies, such as the steady-state strategy used for our GGA for the UCTP in (described in Section 4.3), generally seemed to give a much poorer overall performance than the "noisy sort" population strategy of Falkenauer [54, 58] and Erben [52] used here. These factors suggest therefore that in practical applications, some sort of tuning process would usually need to be applied if the GGA were to be used to its maximum capability.

Finally, looking back at the run characteristics of the two algorithms (figs. 5.2-5) we can see that our use of the more powerful Dsatur procedure for producing initial solutions for the IHS algorithm generally gives this algorithm a slight head-start, because the resultant solutions nearly always use fewer colours than any of the members of the GGA's

[15] Alternatively, if we were to know *m* in advance, then we could calculate χ because $m = \lceil n / \chi \rceil$, and therefore $\chi = \lceil n / m \rceil$.

initial population. Although it was not our intention to look into such matters here, in the future it would, of course, be interesting to see whether the performance of the GGA could improve were the initial population to have one or more individuals in it formed using a Dsatur-type algorithm.

In conclusion, at the beginning of this chapter our stated aims were to conduct an analysis of a second GGA to see if similar characteristics to those observed in Chapter 4 could be observed. From the results presented in this chapter we can see that this goal has been met, and we hope that the reader is now suitably convinced about the potential pitfalls of the GGA approach in some cases. Moving away from GGAs, in the next chapter we will now revert back to the overall theme of this thesis – university course timetabling. In particular, having now established that the IHS algorithm, in general, tends to be a better choice for finding timetable feasibility with the UCTP (at least for our problem instances), we will now move our attention on to the second phase of the two-stage timetabling strategy: satisfying the soft constraints.

6: Satisfying the Soft Constraints of the UCTP

Up until this point in the thesis, our proposed timetabling algorithms have been concerned with the problem of finding feasibility with the UCTP (or in other words, satisfying the hard constraints of this problem). In this chapter, continuing with our study of the two-stage approach to timetabling, we will now switch our efforts over to the task of satisfying the *soft* constraints of the problem. We will present two separate metaheuristic algorithms intended for this task, both of which will follow the two-stage timetabling strategy (Section 2.3.2) of first obtaining feasibility, and then proceeding to try and reduce a *cost function* (reflecting the number of soft constraint violations in a timetable, calculated as described in Section 3.1), whilst always remaining in feasible areas of the search space.

The first of these algorithms, as we will see, will operate using an Evolutionary Algorithm (EA) framework. In order to follow the desired two-stage approach, we will introduce a number of specialised, problem-specific genetic operators that are guaranteed to preserve timetable feasibility, and will conduct an analysis of their general effects. The second algorithm, meanwhile, will make use of the Simulated Annealing (SA) metaheuristic [68, 110] and will use neighbourhood operators that are also specifically designed to preserve feasibility.

In our experimental analyses of both algorithms, rather than using the sixty "hard" UCTP instances used in Chapter 4, we choose to make use the twenty problem instances used for the International Timetabling Competition [2]. These, we feel, are perhaps more appropriate in these circumstances for the following reasons. First, unlike the sixty "hard" instances, each of the competition instances is known in advance to have a perfect solution, thus we will always be able to tell exactly how close our proposed algorithms are able to get to the optimum. However, the vast majority of these instances have still not yet been solved to perfect optimality, so we can presume that they still present a challenge. Secondly, from experience, we know that our algorithms from Section 4 will nearly always be able to achieve feasibility in a non-restrictive amount of time with the competition instances (as we have seen with the sixty "hard" instances, this is not always so), thus we will be able to concentrate more fully on the task-in-hand: i.e. satisfying the soft constraints of the UCTP. Finally, unlike the sixty "hard" instances, various other algorithms in the literature have also used the competition instances and associated benchmark timing program in their experimental analyses. We will therefore also be able to compare our proposed algorithms against these in order to gain a fuller picture of how well they actually perform in general.

6.1 An Evolutionary Algorithm for Soft Constraint Satisfaction

The first algorithm of this chapter is a two-stage evolutionary algorithm (EA), which operates by first constructing a population of fully feasible timetables, and then evolves these whilst always remaining in feasible areas of the search space. Our initial motivations for designing an evolutionary algorithm of this type were as follows:

(1) Even though many different types of metaheuristic algorithm were submitted to the International Timetabling Competition (see Chapter 3), interestingly none of the entrants chose to make use of any sort of evolutionary technique;

(2) Our studies regarding the satisfaction of the hard constraints (Chapter 4) have shown that our constructive heuristics perform very well with the twenty competition problem instances, meaning that we have a mechanism by which we can produce populations of feasible timetables in relatively small amounts of time;

(3) As we have seen in Chapter 3, the two-stage approach for the UCTP has shown great promise when used in conjunction with other sorts of metaheuristics;

Given the latter point in particular, we therefore considered it interesting to see whether an EA that followed this two-stage approach could also perform competitively on the benchmark instances used for the competition; and if not, why not?

For our proposed EA we will again use the matrix representation (Section 3.2) for each individual timetable. However, in order to be consistent with EA terminology in the following descriptions, we will usually refer to each individual timetable matrix as a *chromosome*, and each cell (blank or otherwise) as a *gene*. Also, in order to denote a particular gene of a particular chromosome, we will also use the notation $tt[i, j]$, which will refer to the value that is contained in the gene appearing in the ith row and jth column of a chromosome tt.

In the following pages, a description and analysis of this algorithm will now be presented as follows: in Section 6.1.1 we will describe how populations of feasible timetables are first produced for this algorithm. In Section 6.1.2 and 6.1.3 we will then describe the recombination and mutation operators respectively. Next, in Section 6.1.4, we will outline the experiments that we conducted with this algorithm, and will present an analysis of the results. Finally Section 6.1.5 will conclude the section concerning this EA and will provide some further points of discussion.

6.1.1 Initial Population Generation

For this algorithm an initial population is formed by using the procedure Build (Section 4.2.1) for each individual. To determine the order in which the events are to be placed into the timetable, we use heuristic H_1; breaking ties with H_3 (refer to Table 4.1). Places are then selected for each event using H_4, breaking further ties with H_6. (We remember here that heuristic rules H_3 and H_6 are random choices.) Upon completion of the event assignments, if more than t timeslots are being used by the timetable, then our Iterated Heuristic Search (IHS) algorithm from Section 4.7 is called in order to try and reduce the number of timeslots down to the required amount. (As a matter of fact, in our trials we saw that this latter step was not actually necessary in over 98% of cases, because the heuristics that we used were actually capable of producing feasible timetables using t timeslots by themselves. Also, when this step *was* actually needed, it was usually able to achieve its goal very small amounts of time.)

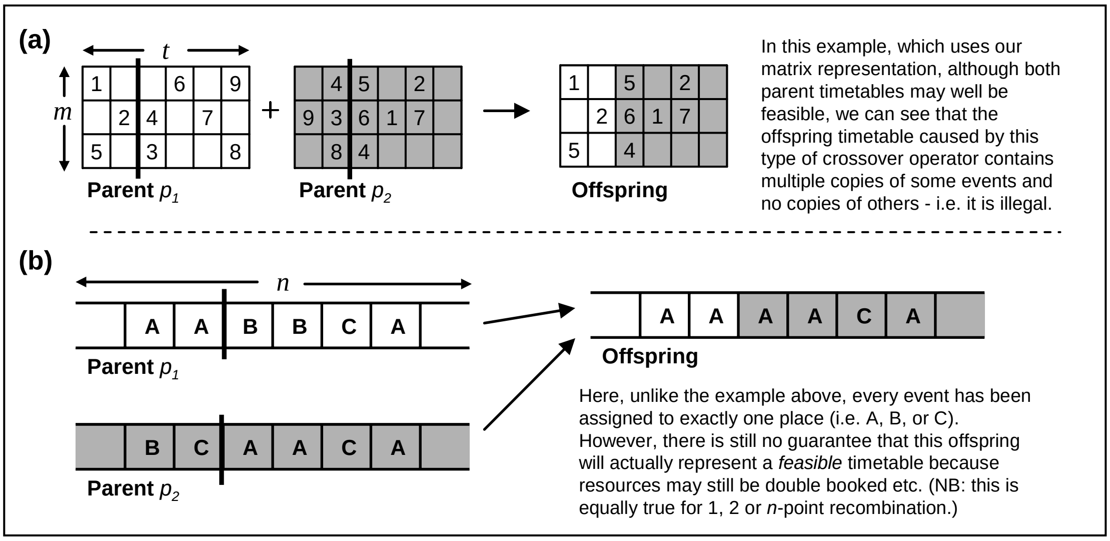
6.1.2 Defining "Useful" Recombination Operators

We have mentioned earlier in this thesis that in evolutionary computation, it is generally desirable for the recombination operator to facilitate the combining of useful and relevant parts (building-blocks) of various different candidate solutions, so that new offspring that are conducive to the search are produced. When designing such an operator for the problem of satisfying the soft constraints of the UCTP, however, we are faced with two important questions:

(1) What features actually constitute "useful and relevant" building-blocks in this case?

(2) How can we ensure that all offspring produced by the recombination operator are always feasible?

With regards to the first question above, we chose to investigate this by designing a number of different recombination operators. Although, as we shall see, all of these are actually quite similar in style, essentially each differs in their method of deciding *which* genes to copy from parent to offspring, and will base their decisions on some sort of perceived *relationship* that these various genes have with one another.



In this example, which uses our matrix representation, although both parent timetables may well be feasible, we can see that the offspring timetable caused by this type of crossover operator contains multiple copies of some events and no copies of others - i.e. it is illegal.

Here, unlike the example above, every event has been assigned to exactly one place (i.e. A, B, or C). However, there is still no guarantee that this offspring will actually represent a *feasible* timetable because resources may still be double booked etc. (NB: this is equally true for 1, 2 or *n*-point recombination.)

Fig. 6.1: Demonstration of the inappropriateness of some of the more "traditional" recombination operators with this two-stage EA for timetabling.

In order to address the second question, meanwhile, it is first useful for us to look at fig. 6.1, where we demonstrate the problems of using "traditional" sorts of recombination and representation with this sort of approach. In fig. 6.1(a), which demonstrates a simple cut-and-paste type recombination used with our matrix representation, we can see that simply combining different chunks of two parent chromosomes is actually highly inappropriate in this case, because it will almost inevitably produce offspring that are not only infeasible, but also *illegal*. Meanwhile, as we demonstrate in the second example fig. 6.1(b), where we make use of the "object-based" representation that we discussed earlier (and which has also been used in various EAs for timetabling such as [39, 93, 94, 97]), although we can see that offspring produced will not actually be illegal this time (because all events will still be assigned to some place in the offspring timetable), there is, of course, no guarantee that the resultant timetable will actually be feasible in most cases.

At this point it is also worth mentioning that the various *specialised* recombination operators used in some evolutionary timetabling applications (such as Burke, Elliman, and Weare's algorithm in [19, 20], and Paechter *et al.*'s algorithm in [86]) also don't seem to be wholly suitable in this case. This is because, as we have noted earlier, although these algorithms always manage to ensure that timetables are kept free of hard constraint violations, both are only actually able to do so by allowing relaxations of some other feature of the problem (i.e. by opening extra timeslots in the former, and by leaving certain events unplaced in the latter). Indeed, although such methods of relaxation may sometimes bring benefits during an algorithm's execution, as we have seen in the definition of this UCTP in Chapter 3, if such relaxations (i.e. temporary infeasibilities) have not been dealt with by the end of a run, then this will, of course, mean that the timetable provided by the algorithm is infeasible and, according to the judging criteria of the competition, worthless.

Given the above, perhaps the most suitable way of addressing this issue in our case is to add some sort of *genetic repair* function to the EA, that will allow genes from different chromosomes to be combined, but at the same time will somehow ensure that the resulting offspring are always feasible. However, like most timetabling problems, it is worth noting that this task of taking an entire chromosome timetable that is infeasible and then "fixing" it in some way so that it becomes feasible (i.e. repairing at the *chromosome level*), is actually equivalent to the NP-hard task of achieving feasibility in the first place. This seems to suggest that even if a process such as this *were* able to consistently succeed in a non-restrictive amount of time, it might still end up having to do a large amount of rearranging of the genes in order to achieve its goal. However, this would also mean that the offspring produced would probably have very little in common with either parent. In our case then, we choose to implement a repair mechanism that operates at the *gene-level* instead. That is, we will restrict our recombination operations so that each of the genes selected for copying from the parents to the offspring will be transferred one-by-one, with each one that causes an infeasibility then being dealt with immediately by a repair operator. Indeed, as we will see, our proposed operator does in fact guarantee that the offspring produced will always be feasible. However, whether it will still actually be beneficial to the search will be looked at in our experiments in Section 6.1.4.

In the following sections (6.1.2.1-6) we will now describe the five different recombination operators and also the associated repair operator used in our experiments. Note that for ease of reading, when describing these operations we will explain how we go about making the first offspring c_1 (which is initially an exact copy of its first parent) from parent timetables p_1 and p_2. In order to construct the second offspring, however, the roles of the two parents are simply to be reversed.

6.1.2.1 Sector-based Recombination

Our first recombination operator explores the idea that there will be "sectors" of a chromosome that will have a strong sub-fitness [114] – that is, it considers the prospect that some timetables will have *areas* of genes within them that are causing smaller numbers of soft constraint violations than others. The sector-based recombination operator thus attempts to provide a mechanism whereby such sectors can be propagated during evolution, and operates as follows: firstly, four values – *top*, *bottom*, *left*, and *right* – are selected. These are used to define a sector in the second parent p_2 (see fig. 6.2(a)). Next, each gene in p_2 within this sector is considered in turn, and an attempt is made to inject it into the same positions in c_1 (this is achieved using the gene transfer and genetic repair functions that will be defined in Section 6.1.2.6).

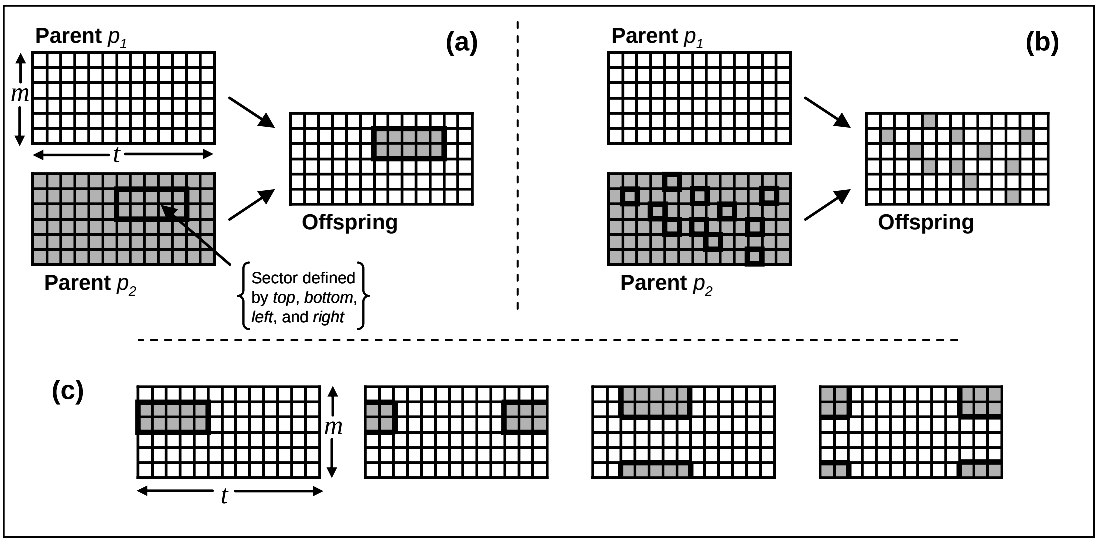


Fig. 6.2: Demonstration of (a) Sector-based recombination, (b) Student, Conflicts and/or Random-based recombination, and (c) the various ways that a sector can perform wraparound.

In our approach, when choosing the values that define the sector, we allow situations to occur where *top* < *bottom* and *right* < *left*. We name this feature "wraparound", and it is useful because it avoids showing unnecessary bias to genes in the centre of the chromosome. (See fig. 6.2(c).) Later, we will also see that it is useful when using this method to limit the size of the sector in some way. Our reasons for doing this are related to the nature of the genetic repair function, and will therefore be dealt with in the repair operator's description in Section 6.1.2.6. In our approach, however, we choose to limit the height of a sector so that cannot be greater than $\lfloor m/2 \rfloor$, and the width so that it cannot be greater than $\lfloor t/2 \rfloor$ (where t = 45).

6.1.2.2 Day-based Recombination

Our second recombination operator is very similar to the sector-based operator described above. However, whereas in the latter we allow large amounts of flexibility as to what dimensions the sectors can take; here, we choose to impose a restriction stipulating that a sector can only cover an entire day's assignments. The rationale for this operator is the observation that in this UCTP, none of the three soft constraints actually carry across different days; thus we might therefore consider the assignment of events to a particular day as a sub-timetable, with all of its soft constraint violations contained within it. This recombination operator will therefore attempt to exploit this fact by providing a mechanism whereby good days might be propagated and combined during a run.

6.1.2.3 Student-based Recombination

For student-based recombination we start by choosing a student at random, and then calculate all of the events that he or she is required to attend. We then go through parent p_2 and attempt to insert all of the genes that contain these events into the same position in c_1, applying the genetic repair function wherever necessary. (See fig 6.2(b) for an illustration.)

In order to justify such an operator, it is worth considering for a moment the notion of *personal-timetables*. As we have described in our definition of the UCTP (Chapter 3), each student in this problem is required to attend some subset of events. Thus, given a feasible timetable, we can easily determine the personal-timetable of a particular student by identifying the particular timeslots and rooms where he-or-she is attending some event. Now, given the fact that, in this case, all violations of the soft constraints are caused in some way by the whereabouts of the students during the working week (a student that has to attend three events in a row will result is a penalty cost of one, for example), it is reasonable to assume that some students will have personal-timetables that contain fewer soft constraint violations than others. This recombination operator therefore endeavours to use this fact by offering a mechanism whereby the good personal-timetables might be combined and propagated in order to help produce high quality offspring.

6.1.2.4 Conflicts-based Recombination

This operator follows a similar idea to student-based crossover. This time, however, the operator starts by choosing an event e at random and then identifies the subset $E' \subseteq E$ that represents all of the events that *conflict* with e. Next, just like our previous operator, attempts are then made to try and inject all of the genes in p_2 that contain these events into the same position in c_1.

With this operator, similarly to student-based crossover, we are thus identifying collections of genes that are related due to common students, and providing a mechanism whereby they might be propagated during evolution. However, it is worth noting that the UCTP, just like in real world timetabling situations, will usually feature pairs of events that conflict due to multiple students taking both events. In this case, therefore, we might be arming ourselves with the potential to alter the objective function to a greater degree.

6.1.2.5 Random-based Recombination

Our final recombination operator follows a similar pattern as the previous two examples. However, in this case the genes that are selected in p_2 for injection into c_1 are chosen entirely at random – i.e. no relationships between the various genes are considered. In our case it is useful to introduce this operator, because it can be used for conducting control experiments, thus allowing us to assess whether the relationships that are being used to define the previous four operators are actually aiding the search or not.

Note that with this recombination operator, we need to control in some way the number of genes that are to be selected for injection from p_2 into c_1. For our purposes (which we will outline in Section 6.1.4) we found that it was sufficient to control this through the introduction of a single parameter τ in the range $[0, t \times m]$. This parameter simply specifies the number of genes in p_2 that are to be randomly selected for transfer to c_1 with each application of the operator.

6.1.2.6 Gene Transfer and Genetic Repair

We have noted previously that at the start of the recombination process the child c_1 will be an exact copy of parent p_1, and the recombination operator of choice will define a set of genes in p_2 that are to be copied into the same positions in c_1. For each individual gene that is to be injected from p_2 into c_1, there are four things that can happen. We will now go through each of these in turn, and in each case, we will describe what action will occur, and how the feasibility of the chromosome will be maintained. For this description, let e and g represent events such that $e \neq g$, and let x and y define the coordinates of the particular gene in p_2 that we wish to inject:

(1) $p_2[x, y] = c_1[x, y]$. In this case we do nothing.

(2) $p_2[x, y]$ is blank and $c_1[x, y]$ contains e. In this case we make $c_1[x, y]$ blank, and attempt to find a new place for e using the genetic repair function (see below).

(3) $p_2[x, y]$ contains g, and $c_1[x, y]$ is blank. In this case we find and delete the original occurrence of g in c_1. Next, we insert g into $c_1[x, y]$. If this maintains the feasibility of timeslot y in the chromosome then we accept the move and end; otherwise, we reset the change and we consider this particular insertion as having failed.

(4) $p_2[x, y]$ contains g, and $c_1[x, y]$ contains e. In this case, we first delete the occurrence of g in c_1. Next, we insert g into $c_1[x, y]$. If this makes timeslot y infeasible, then we reset the change and end; otherwise we attempt to identify a new place for e using the genetic repair operator (see below).

Note that in cases (2) and (4) above, we might encounter a situation where we have an unplaced event e in c_1. Our genetic repair function is responsible for identifying a new blank gene elsewhere in the chromosome where e can be inserted without compromising c_1's feasibility. As before, let x and y indicate the original position of e in c_1. First, the repair operator searches horizontally along row x looking for a blank gene in which it can insert e without causing an infeasibility (note that we already know the room defined by row x to be suitable for e, so we only actually need to check for violation of hard constraint HC_1 here). Next, if it cannot find a suitable blank gene for e in row x, then the operator goes on to check any other rows (if there are any), which define other suitable rooms for e. This process halts when either (a) a suitable blank gene for e is found – in which case we can insert e here, or (b) until all blank genes in appropriate rows have been considered. If the latter occurs, then we consider the repair process as having failed in this case, and we therefore reset the timetable to its previous state and move on.

Note that in our approach, when performing genetic repair during an application of the sector-based or day-based recombination operators, only genes that are *outside* of the sector are considered by our repair operator. This would seem intuitive as the objective of these recombination procedures is to try and transfer as many of the genes *inside* the sector as possible from p_2 to c_1. This feature also explains why we feel it necessary to restrict the size of the sector when using sector-based recombination, because a sector that is too big would obviously allow very little room for genetic repair to operate, and would thus encourage a high amount of repair failure.

Finally, this method of repair does have one additional issue that we feel needs addressing. Generally, when a timetable has a low cost (in that it will contain a relatively low number of soft constraint violations), then according to soft constraint SC_1, it should not have many (if any) events scheduled in the five end-of-day timeslots. However, when performing genetic repair in this case, there will actually be a tendency for our repair operator to want to put events into these timeslots. The reasons for this are twofold: first, in a good timetable the end-of-day timeslots will usually contain a higher-than-usual number of blank genes; second, because there *are* lower numbers of events in these timeslots, then there will also be a smaller probability of there being an event already existing within the timeslot that conflicts with the event that we are trying to insert. These factors suggest that if we are to improve our chances of producing good quality offspring via recombination, we should try to show prejudice against the placing of events into these five timeslots.

In our approach, we dealt with this problem by introducing a parameter *eod* (end of day) of range [0, 1]. This parameter defines the probability of us allowing an event to be assigned into one of these penalised timeslots each time our repair function encounters one.

6.1.3 Mutation Operator

Finally, like the various recombination operators defined above, a mutation operator for this algorithm also needs to preserve a chromosome's feasibility. Given a timetable chromosome which, for consistency we will again call c_1, our method works by performing the following steps:

(1) Randomly select two distinct genes, $c_1[a, b]$ and $c_1[c, d]$, ensuring that $c_1[a, b] \neq c_1[c, d]$ (i.e. thereby ensuring that both genes are not blank);

(2) Swap the contents of genes $c_1[a, b]$ and $c_1[c, d]$. If feasibility is maintained then end; otherwise, reset the swap and go back to step (1).

Note that a procedure such as this has the potential of becoming an infinite loop, because we may encounter a situation where we cannot perform *any* swap that preserves feasibility. In our experiments, we limited the number of attempted swaps to 10,000. Using the competition problem instances, however, this limit was never reached in our trials.

6.1.4 Experiments and Analysis

In order to assess the relative effects of the five recombination operators (and associated repair operator), we performed experiments using the same steady-state population strategy as our grouping genetic algorithm, described in Section 4.3. In all cases we used a population size $\rho = 50$, set *rr* to 1.0, *eod* to 0.05, and used a mutation rate of $1/n$. (That is, each time a new individual was produced, a loop was performed n times. At each iteration, we would then apply the mutation operator with a probability $1/n$.) For each set of trials we performed twenty individual runs on each of the twenty competition instances, using the time limit specified by the competition benchmarking program as our stopping criteria, which equated to 270 seconds of CPU time on our computers. As a second control trial, we also performed experiments using no recombination at all (i.e. using *rr* = 0.0). In our descriptions below, these trials will be referred to as “Mutation only”.

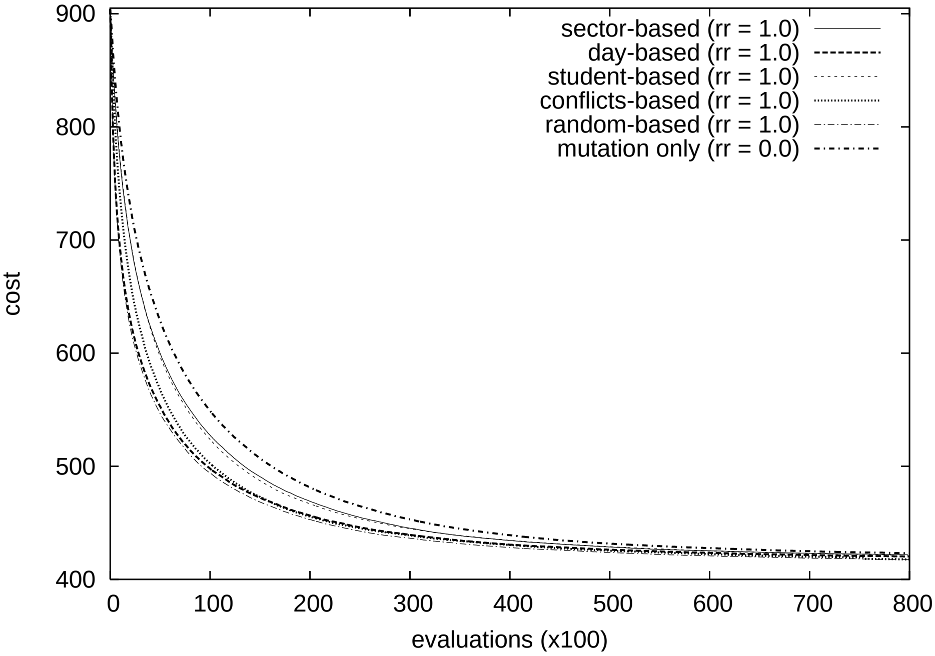


Fig. 6.3: Showing the behaviour of the algorithm with regards to the number of evaluations performed, with the various recombination operators, and also with mutation on its own. Each line represents the cost of the best solution found so far, averaged across 20 runs on each of the 20 instances (i.e. 400 runs).

If we look first at the performance of the algorithm with regards to the number of evaluations performed (depicted in fig. 6.3), we will notice that the operator that produces one of the fastest initial movements through the search space is day-based recombination. However, for these twenty problem instances, this also happened to be the operator that involved the largest number of genes being transferred (and thus repair being performed) per application on average. Thus, in order to assess whether this type of recombination was actually facilitating the propagation of useful building-blocks or, instead, was simply providing a mechanism by which the algorithm could make larger random-style jumps in the search space, we chose to set the parameter τ (which, we remember, is used for controlling the number of genes that are chosen to be transferred from parent to offspring with our random-based recombination operator) to a value that was equivalent to the number of genes contained in a day (i.e. $\tau = (m \times t)/5$).

As fig. 6.3 demonstrates, a random-based recombination operator using this setting for τ causes the algorithm to behave in a very similar manner to day-based recombination. This suggests that the “useful building-blocks” that we were attempting to identify by using entire day’s assignments are not actually being captured by the day-based operator at all, and instead, the quicker downhill movements (compared to the remaining trials) are simply being caused because the larger random-style jumps that are occurring are allowing a slightly broader search to occur during the initial parts of the run. Indeed, this latter hypothesis is also backed up by the fact that the sector and student-based operators, which, in these instances, involves slightly smaller numbers of genes being transferred on average, also produced slightly slower downhill movements in initial parts of the run. Equally, the conflicts-based recombination operator, which involves an intermediate number of genes being transferred, also shows an intermediate behaviour between the four other operators in the figure, whilst the mutation only trials, which involve no random-style jumps, shows the slowest.

Note also that with the day-based recombination operator, because there are only five days in any one chromosome, the actual number of gene combinations that can be chosen for injection from the second parent p_2 into the offspring c_1 is quite small. On the other hand, the number of gene combinations that can be selected during random-based recombination is much larger. The extra restrictions in the former might therefore lead to a more rapid drop in the diversity of a population during a run, thereby leading to a quicker redundancy of the recombination operator, and might explain why the random-based operator seems to achieve the slightly quicker downhill movement prevalent in the figure.

However, regardless of all these observations, perhaps the most striking feature in fig. 6.3 is that as the runs continue, the differences between the six trials become steadily smaller and, perhaps most crucially, once the searches stagnate at around 60,000-or-so evaluations, the trials where the recombination operators *are* being used don't actually appear to be producing results that are noticeably better than the mutation-only trials.

Moving our attention towards fig 6.4, where we show the behaviour of the same set of runs, but in this case with regards to CPU time, we see very similar patterns emerging: although we can see that the various recombination operators do offer a slightly faster movement through the search space for the first parts of the run (as we demonstrate more clearly in the projection), we can also see that when the time limit is reached, no real difference in performance exits between mutation only and the remaining trials. Indeed, none of the algorithm variants was seen to be significantly different to any other.

It is also worth noting that, in both figs. 6.3 and 6.4, in order to display these results more clearly we have truncated the y-axes. However, if we were to allow the scale of these axes to start at zero, which, we remember, is a cost that is achievable with all of the problem instances, the differences between these six trials would be even less clear.

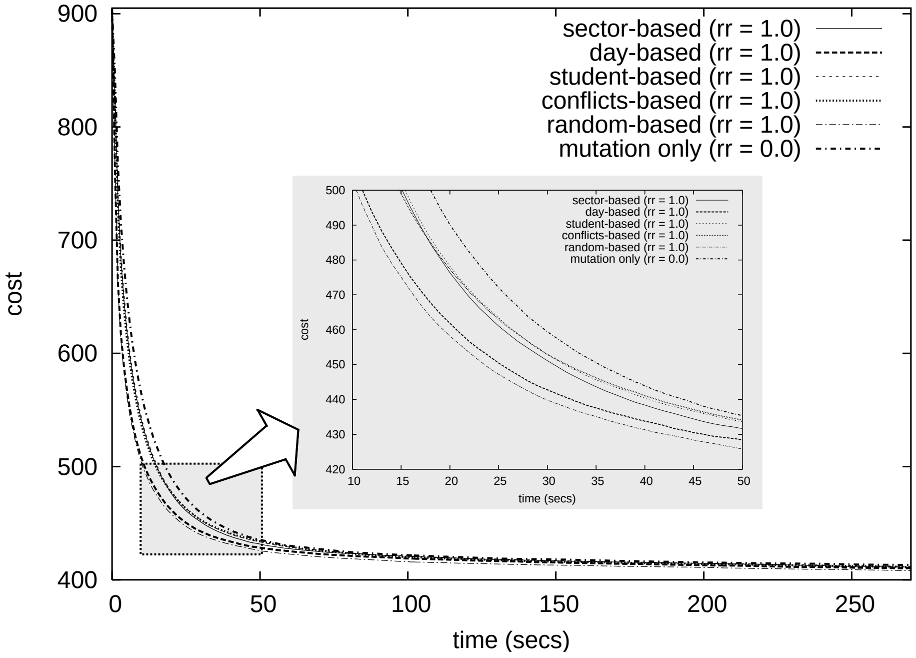


Fig. 6.4: Showing the behaviour of the algorithm over time, with the various recombination operators, and also with mutation on its own. Each line represents the cost of the best solution found so far, averaged across 20 runs on each of the 20 instances (i.e. 400 runs). The projection (inset) shows the effects between 10 and 50 seconds).

Finally, in Tables 6.1 and 6.2 we show a breakdown of the results gained at the time limit with each of the twenty problem instances in each of the six trial-sets. It can be seen from the averages presented in these tables that the results gained in each set of trials with regards to both the average performance *and* best performance, are very similar in general, with no clear winner presenting its self.

6.1.5 Conclusions and Discussion

We are now at approximately the half way stage of this chapter. So far we have concerned ourselves with the task of designing and analysing a two-stage EA for the UCTP. Although in our tests we have seen that this EA can successfully evolve a population of feasible timetables (in that it is able to improve the quality of the candidate solutions over time), crucially, we have witnessed that this evolution is done just as effectively within the time limit when mutation is used on its own. In other words, our proposed recombination operators do not seem to offer any extra search powers in these cases.

Table 6.1: A Breakdown of the Results Found in the Various Trials with the Twenty Problem Instances. (Results for all instances are averaged across twenty runs.) The Label: "Initial Pop. %" Indicates the Percentage of the Time Limit that was required to produce the Initial Population across the Trials.

#	Initial Pop. %	Lowest Cost Found During Run (Average ± std. dev. for 20 runs) Sector-based	Day-based	Student-based	Conflicts-based	Random-based	Mutation only
1	0.02	308.2 ± 24.1	317.1 ± 23.6	308.8 ± 15.7	308.8 ± 25.0	305.7 ± 18.5	312.7 ± 18.5
2	0.02	278.8 ± 17.8	283.6 ± 21.4	287.5 ± 23.4	286.6 ± 21.5	279.7 ± 25.9	278.7 ± 25.9
3	0.03	323.8 ± 29.7	323.2 ± 22.3	340.9 ± 20.2	344.5 ± 16.6	333.2 ± 25.5	324.9 ± 25.5
4	0.04	650.4 ± 23.5	643.8 ± 34.5	643.3 ± 39.2	645.8 ± 43.7	638.6 ± 48.6	665.0 ± 48.6
5	0.03	612.8 ± 46.7	624.4 ± 46.2	626.0 ± 38.3	627.9 ± 48.1	617.2 ± 42.1	618.3 ± 42.1
6	0.03	482.5 ± 43.8	478.1 ± 43.8	498.4 ± 48.5	484.2 ± 47.8	488.4 ± 46.4	487.2 ± 46.4
7	0.03	454.6 ± 43.3	446.6 ± 35.2	433.1 ± 35.7	444.2 ± 42.2	434.7 ± 49.8	444.9 ± 49.8
8	0.03	338.1 ± 29.8	345.3 ± 27.4	337.2 ± 29.3	327.7 ± 22.0	340.2 ± 22.2	341.2 ± 22.2
9	0.03	310.0 ± 20.5	307.6 ± 28.1	295.7 ± 21.6	301.8 ± 21.9	294.5 ± 24.2	301.6 ± 24.2
10	0.02	316.3 ± 18.4	323.5 ± 17.8	318.9 ± 26.8	319.5 ± 21.5	311.3 ± 15.5	326.0 ± 15.5
11	0.03	328.7 ± 30.4	340.1 ± 21.3	342.4 ± 22.6	342.7 ± 30.3	326.2 ± 17.6	331.1 ± 17.6
12	0.02	387.5 ± 38.0	384.3 ± 33.2	379.7 ± 33.5	373.7 ± 35.9	373.6 ± 33.2	380.8 ± 33.2
13	0.03	450.8 ± 29.7	438.6 ± 36.6	432.1 ± 21.9	438.6 ± 24.5	454.6 ± 25.1	442.5 ± 25.1
14	0.03	492.7 ± 54.3	477.6 ± 55.8	489.2 ± 49.9	501.2 ± 41.7	502.4 ± 51.3	505.0 ± 51.3
15	0.03	431.3 ± 27.1	429.8 ± 41.0	419.1 ± 30.8	425.6 ± 32.9	428.4 ± 25.8	433.6 ± 25.8
16	0.03	295.8 ± 14.2	299.4 ± 18.0	289.2 ± 20.5	300.0 ± 20.7	293.2 ± 22.4	290.4 ± 22.4
17	0.03	538.4 ± 56.2	533.2 ± 36.8	530.5 ± 43.8	524.3 ± 52.8	531.1 ± 50.2	539.0 ± 50.2
18	0.02	269.0 ± 20.7	266.2 ± 17.7	267.2 ± 19.3	265.1 ± 23.9	263.5 ± 17.3	263.7 ± 17.3
19	0.04	524.2 ± 39.1	524.9 ± 28.3	538.9 ± 30.7	532.3 ± 31.1	519.0 ± 36.0	521.8 ± 36.0
20	0.03	400.2 ± 38.6	387.0 ± 23.7	413.2 ± 28.5	398.3 ± 27.3	397.6 ± 39.5	408.2 ± 39.5
Average		409.7	408.7	409.5	409.6	406.6	410.8

In reality, the ineffectiveness of these recombination operators is not particularly surprising because, as we have seen, the nature of this particular timetabling problem coupled with our chosen representation seemingly makes it difficult to define genetic operators that can consistently combine meaningful parts of various chromosomes whilst always ensuring the offspring's feasibility. In our case, we believe that the following two factors are probably responsible for this lack of search capability:

(1) During recombination, when injecting a gene from the second parent into the offspring, if this process fails (as it often can – see Section 6.1.2.6) this will, of course mean that we are not injecting an entire building block into the offspring at all (if indeed anything we can arguably call a "building block" has been captured by the recombinative process in the first place).

(2) The nature of our repair mechanism means that foreign genes (that is, genes that do not occur in either parent but which *do* appear in the subsequent offspring) will often be introduced into the offspring, which may well disrupt various other parts of the

chromosome, as well as possibly undoing some of the work done in previous iterations of the algorithm.

Table 6.2: The Best Results Found from Twenty Runs on Each of the Twenty Problem Instances in Each Trial.

	Lowest Cost Found During Run (Best of 20 runs)					
#	**Sector-based**	**Day-based**	**Student -based**	**Conflict -based**	**Rand-based**	**Mutation only**
1	269	276	285	262	277	287
2	258	247	234	245	240	245
3	258	287	298	317	287	291
4	611	546	556	561	540	584
5	511	536	558	565	539	534
6	389	410	410	388	408	393
7	399	385	379	361	339	378
8	285	307	296	284	294	298
9	282	238	262	259	254	267
10	280	297	272	282	276	287
11	280	307	306	264	281	283
12	294	321	335	305	317	295
13	398	364	386	385	393	355
14	372	378	391	400	410	411
15	392	364	357	368	395	363
16	271	265	247	258	256	252
17	438	457	462	443	432	456
18	234	232	234	214	231	212
19	442	481	488	465	439	457
20	311	335	355	356	327	347
Average	348.7	351.7	355.6	349.1	346.8	349.8

In practice, of course, these factors will probably mean that these five "recombination" operators do not recombine different chromosomes at all; instead, they are more akin to some sort of macro-mutation operator which simply results in large and random-style leaps within the search space being made. Essentially then, they are not really fulfilling the desired task of a good recombination operator, meaning that a significant and potentially useful component of the general EA paradigm has been lost.

It is also worth noting that another potential pitfall of this algorithm is its reliance on being able to produce populations of different, feasible timetables in reasonable amounts of time. Whilst our proposed methods have been able to achieve this successfully with the twenty competition benchmark instances in relatively small amounts of time (although we can see in Table 6.1 that some instances require slightly longer than others), it is, of course, worth remembering that even this task might turn out to be unachievable when "harder" instances (such as the ones used in Chapter 4) are encountered.

Considering all of the above, it is also probably unsurprising to learn that neither the average nor the best results gained by this algorithm compare very favourably with the results achieved by other two-stage algorithms for this problem. Looking at the official results of the International Timetabling Competition [2], for example, we can see that the average costs of the solutions found by this EA are, in most cases, over twice as high as many of the official entrants, and even worse compared with some of the better performing algorithms. Additionally, although it is very likely that the performance of this EA could be improved by the addition of some extra search routines for soft constraint satisfaction (such as those described in [96, 97]), and/or some sort of smart-mutation operator[16] [39], it seems that if we are to persist in our quest for an effective two-stage EA for this problem, we will still always have to compensate for the fact that we do not currently seem to have a natural way in which we can successfully pass and propagate meaningful information between different chromosomes with this sort of approach and this sort of representation.

Indeed, it should be noted that in all of our experiments up until this point in the chapter, we have seen that the genetic operator that seems ultimately to provide this EA with its (perhaps rather limited) search capabilities is mutation: an operator that works by making very small – but essentially blind – stochastic perturbations to a candidate solution timetable, but which does not facilitate the sharing of genetic information between different members of a population. This factor might suggest, therefore, that perhaps a more promising direction to take at this point would be to tie such a perturbation operator in with some sort of *neighbourhood search* algorithm instead. We will present an algorithm doing just this in the next section, and a design and analysis of this algorithm will constitute the second half of this chapter

6.2 An SA Algorithm for Soft Constraint Satisfaction

Neighbourhood search algorithms (also sometimes known as *local search* algorithms) might be considered a family of optimisation techniques that include such popular algorithms as simulated annealing [68, 110], tabu search [63], iterated local search [65], variable neighbourhood search [81], and various sorts of hill-climbers. Generally, in order to operate, these algorithms require three things to be defined: (1) some way of encoding

[16] Note that we have not added these sorts of operators here because our objective was to examine the effects and shortcomings of the various recombination operators.

candidate solutions; (2) a *cost* function (used for measuring the quality of a given candidate solution); and (3) a *move* (or neighbourhood, or perturbation) operator that allows small changes to be made to a candidate solution in some way. Once these three ingredients have been defined, these algorithms all operate by taking some initial candidate solution (possibly produced at random) and then attempt to navigate through the search space in some way via a repeated application of the move operator, attempting to find the candidate solution with the best possible cost. (In general the different types of neighbourhood search algorithm will differ in their ways of deciding which neighbours to look at; their criteria for deciding which moves to accept and reject; and their proposed methods of escaping or avoiding local optima.)

In this section, we shall propose another two stage approach that utilises one of these types of neighbourhood search variants – namely simulated annealing (SA) – for the task of satisfying the soft constraints of the UCTP. As in our EA above, this algorithm will operate by first finding feasibility, and will then attempt to eliminate as many soft constraint violations as possible, whilst always remaining in feasible areas of the search space.

At this point, it should be clear to the reader that we already have at our disposal all of the essential ingredients for a neighbourhood algorithm such as this, and so an application of SA should be relatively straightforward:

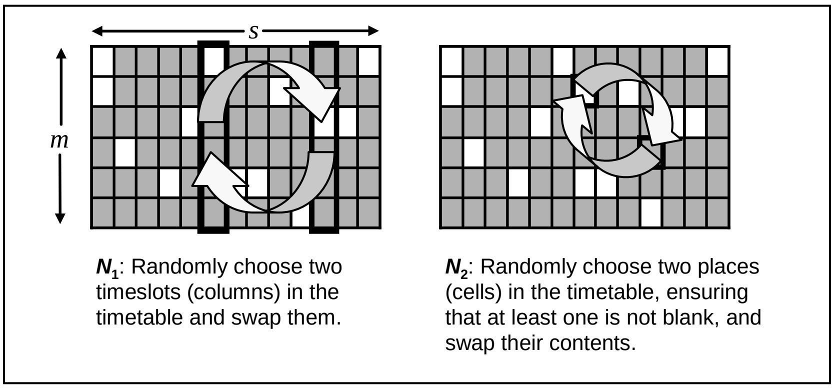
(1) Our matrix representation provides us with a suitable encoding for this task, and our IHS algorithm (given in Section 4.7) seems to give us a suitably robust method of randomly generating an initial feasible timetable;

(2) The method of calculating the number of soft constraint violations (given in Section 3.1) offers us a suitable cost function for this task;

(3) A move operator that preserves feasibility has already been used as a mutation operator for the previous EA and should also be applicable here. (Note that other feasibility-preserving operators are possible as well. We will consider one of these here also.)

The SA algorithm that we will describe in this section operates as follows. First, a feasible timetable is constructed using our IHS algorithm. Once this has been achieved, the SA metaheuristic is then applied in two successive phases. In the first phase (SA phase-1), attempts are made to try and order the timeslots of the timetable in an optimal way through the use of a simple neighbourhood operator that we will call N_1 – see fig. 6.5. Next, in SA phase-2 the algorithm then attempts to make further improvements by shuffling the events around the timetable, using a neighbourhood operator that we will call N_2 (which is essentially the same as our mutation operator from the previous section). Note that this two-phased approach for attempting to satisfy the soft constraints has also been used in the algorithms of White and Chan [112] and also Kostuch [69], both of which we reviewed earlier. However, there are a number of features that distinguish our methods from these, including our (more effective) methods for producing initial feasible solutions; our use of various automated functions to determine some of the algorithm's run-time parameters; our cooling schedules, which include a sophisticated reheating function to help the algorithm escape from local optima; our use of a more flexible and robust neighbourhood operator; our use of the matrix representation; and the use of our own very efficient delta-evaluation function. These features will all be explained in the following sections.

Finally, as before, this algorithm halts either when a perfect solution has been found or, failing this, when a predefined time limit defined by the competition benchmarking program has been reached. In the latter case, the best solution found during the whole run is returned.



***N*$_1$**: Randomly choose two timeslots (columns) in the timetable and swap them.

***N*$_2$**: Randomly choose two places (cells) in the timetable, ensuring that at least one is not blank, and swap their contents.

Fig. 6.5: The two neighbourhood operators used in the SA algorithm.

In both of the SA phases, a very typical SA method will be used: starting at an initial temperature (that we will denote T_0), during the run the temperature T will be altered in some way according to a temperature update rule. At each value of T, a number of neighbourhood moves will then be attempted. Any move that increases the cost of the timetable will be accepted with probability:

$$\exp\left(-\delta / T\right) \qquad (6.1)$$

(where δ represents the change in cost that such a move would cause), whilst any move that reduces or leaves the cost unchanged will be accepted automatically.

In the following four subsections we will now describe the details of the two SA phases. Following these descriptions, in Section 6.2.5 we will then provide some details about how we can also implement an efficient delta-evaluation function for this algorithm.

In Section 6.2.6 we will then present an experimental analysis of this algorithm and discuss some further points arising from this study.

As a final point, it is also worth noting that in the following algorithm description we will see that a number of parameters will be defined, each of which will dictate the various operational characteristics of a run in some way. However, in our case, rather than attempt to empirically confirm optimal settings for all of these, we have decided to fix some of the less influential parameters at values that, according to our initial experiments and intuitions, seem appropriate and/or sensible in these cases. These settings should therefore be considered *design decisions* on our behalf and will be discussed and defined at the appropriate places in the text. The settings for the remaining parameters, meanwhile, will be established using a simple tuning procedure that will be described in Section 6.2.6.

6.2.1 SA Phase-1: Search Space Issues

The first phase of the SA algorithm is concerned with the exploration of the search space defined by neighbourhood operator N_1 (see fig. 6.5). Note that because of the specific structure of this timetabling problem (and in particular, the fact that there are no hard constraints that depend on the ordering of the timeslots within the timetable); a movement in N_1 will *always* preserve feasibility. Also note that an application of N_1 will generally involve multiple events (and thus relatively many students) and is therefore more than likely to cause large, wholesale changes to the timetable's cost in general.

It is also worth mentioning, however, that it is highly unlikely that all feasible timetables will be achievable through a use of N_1 alone. For example, note that the size of the search space offered by N_1 is exactly $s!$ (where s represents the number of timeslots being used in the timetable, and thus $s!$ is the number of different permutations of these timeslots). Indeed, given that a feasible timetable must always have $s \leq t$ (where $t = 45$), this means that the number of possible solutions achievable with this operator will not actually grow with instance size (as is usual), but instead will be fixed at a maximum of 45!.

Additionally, if we were to start this optimisation phase with a timetable in which two events, say i and j, were assigned to the same timeslot, then an application of N_1 would never actually be able to change this fact. Indeed, if the optimal solution to this problem instance required that i and j were, in fact, in *different* timeslots, then an exploration with N_1 would never actually be able to achieve the optimal solution in this case.

Given these issues, it was decided in our case that this first phase of SA should only be used as a preliminary step for trying to make making quick-and-easy improvements to the timetable. This also showed to be the most appropriate response in practice.

6.2.2 SA Phase-1: The Cooling Schedule

For this SA phase, an initial temperature T_0 is determined automatically by calculating the standard deviation in the cost for a small sample of neighbourhood moves. (We used sample size 100). This scheme of calculating T_0 is based upon the physical processes of annealing, which is beyond the scope of this thesis, but of which more details can be found in [110]. However, it is worth noting that in general SA practice, it is very important that a correct value for T_0 is determined. On the one hand, a value for T_0 that is too high will invariably waste run time, because it will mean that the vast majority of movements will be accepted, providing us with nothing more than a random walk about the search space. On the other hand, an initial temperature that is too low could also be detrimental, as it will likely make the algorithm too greedy from the outset, causing it to behave more similarly to a hill-climber, and making it far more susceptible to getting stuck at local optima. In practice, our described method of calculating T_0 tended to allow approximately 75-85% of moves to be accepted, which is widely accepted as an appropriate amount in SA literature [110].

With regards to other features of the cooling schedule for this phase of SA, because, as we have noted, this phase is only viewed as a preliminary, during execution we choose to limit the number of temperatures that are encountered by the algorithm to a fixed value *M*. Of course, in order to have an effective cooling, this also implies a need for a cooling schedule that will decrement the temperature from T_0 down to a value close to zero, in exactly *M* steps. We could, therefore, simply use the following temperature update rule:

$$T_{i+1} = T_i - \left(\tfrac{T_0}{M}\right) \tag{6.2}$$

which would just reduce the temperature at a fixed amount at each stage. However, in practice, we actually found this cooling scheme to be a little unsatisfactory, because the temperature tended to drop too slowly early on in the run (resulting in too much random walk-style movements) and, as a result, did not seem to allow enough of the run to operate at the lower temperatures. We therefore used the following temperature update rule instead:

$$\begin{aligned} \lambda_0 &= 1 - \beta \\ \lambda_{i+1} &= \lambda_i + \tfrac{\beta+\beta}{M} \\ T_{i+1} &= T_i - \lambda_{i+1} \tfrac{T_0}{M} \end{aligned} \tag{6.3}$$

Here, β represents a parameter that, at each step, helps determine a value for λ . The resulting value for λ is then used for influencing the amount of concavity or convexity present in the cooling schedule (fig. 6.6 shows these effects in more detail).

In our experiments, for this phase we set M = 100 and, in order to allow more of the run to operate at lower temperatures, we set β = –0.99. Meanwhile, the number of moves to be attempted at each temperature was fixed at s^2, thus keeping the value proportional to the total size of the neighbourhood (a strategy favoured in many SA implementations).

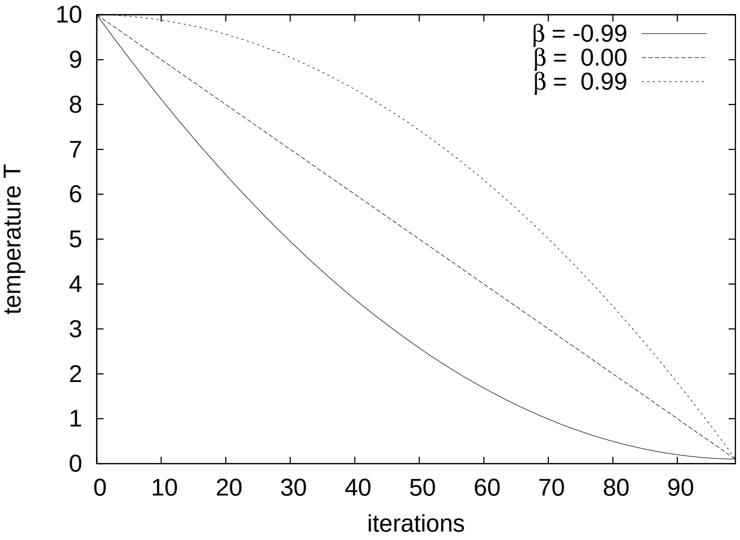


Fig. 6.6: The effects of the parameter β with the cooling scheme defined in eq. (6.3). For this example, T_0 = 10.0 and M = 100. Note also that setting β = 0.0 in this case will produce exactly the same cool as that which would be produced by eq. (6.2).

6.2.3 SA Phase-2: Search Space Issues

In the second and final round of simulated annealing, taking the best solution found in the previous SA phase, an exploration of the search space defined by the neighbourhood operator N_2 is conducted (see fig. 6.5). Note that, unlike neighbourhood operator N_1, moves in N_2 might cause a violation of one or more of the hard constraints. In our approach we therefore choose to immediately reject and reset any move that causes such an infeasibility to occur.

Before looking at how we will tie this operator in with the SA approach, it is first worth considering the large amount of flexibility that an operator such as N_2 can offer the search. Suppose that in a single application of this operator we elect to swap the contents of cells p and q, and for sake of argument, this move will not result in a violation of any of the hard constraints:

- If *p* is blank and cell *q* contains an event *e*, then this will have the effect of moving *e* to a new place *p* in the timetable;
- If *p* contains an event *e* and cell *q* contains an event *g*, then this will have the effect of swapping the places of events *e* and *g* in the timetable.

Additionally,

- If *p* and *q* are in the same column, then only the rooms of the affected events will change (and thus only hard constraint HC_2 will have the possibility of being broken);
- If *p* and *q* are in the same row, then only the timeslots of the affected events will change (and thus only hard constraint HC_1 will have the possibility of being broken);
- Finally, if *p* and *q* are in different rows *and* different columns, then both the rooms and timeslots of the affected events will be changed (and thus both hard constraints HC_1 and HC_2 will have the possibility of being broken).

It can be appreciated, therefore, that the neighbourhood operator N_2 has the potential to alter a timetable in a variety of ways. Additionally, the number of new candidate solutions (feasible and infeasible) that are obtainable via any single application of this operator is exactly:

$$\tfrac{1}{2}n(n-1)+nx-1 \tag{6.4}$$

(where *x* defines the number of blank cells in the timetable). Thus, unlike N_1, the size of the neighbourhood is directly related to the number of events *n*, and therefore the size of the problem. This latter point suggests that for anything beyond very small instances, more time will generally be required for a thorough exploration of N_2's solution space.

6.2.4 SA Phase-2: The Cooling Schedule

For this phase, an initial temperature T_0 is calculated in a very similar fashion to SA phase-1. However, before starting the annealing process we also choose to reduce the result of this particular calculation by a factor (c_2/c_1), where c_1 represents the cost of the timetable that was found *before* SA phase-1, and c_2 the cost *after* SA phase-1. Our reason for doing this is that during our experiments, we observed that an unreduced value for T_0 was often so high, that the improvements achieved during the SA phase-1 were regularly undone at the beginning of the second. Reducing T_0 in this way, however, seemed to allow the second phase of SA to build upon the progress of SA phase-1, thus giving a more efficient run on the whole.

In order to determine when the temperature T should be decremented, in this phase we choose to follow the methods used by Kirkpatrick *et al.* [68] and Abramson *et al.* [9] and make use of two values. The first of these specifies the *maximum* number of feasible moves that can be attempted at any value for T and, in our case, this is calculated using the formula: $\eta_{\max}n$, where $\eta_{\max}$ is a parameter that we will need to tune (note that our use of the number of events n in this formula keeps this value proportional to instance size). However, in this scheme T is also updated when a certain number of feasible moves have been accepted at the current temperature. This value is calculated using the formula $\eta_{\min}(\eta_{\max}n)$, where $\eta_{\min}$ is in the range (0, 1] and, for our purposes, must also be tuned.

Next, with regards to the temperature decrement, we choose to use the traditional geometric scheme [68] where, at the end of each cycle, the current temperature T_i is modified to a new temperature T_{i+1} using the formula:

$$T_{i+1} = \alpha T_i \qquad (1.1)$$

where α is a control parameter that we will refer to as the *cooling rate*.

Finally, because this phase of SA will operate until a perfect solution has been found, or until we reach the imposed time limit, we also make use of a reheating function that is invoked by our algorithm when no improvement in cost is achieved during a number of successive values for T (thus implying that the search has presumably become stuck at a local optimum). In order to calculate a suitable temperature to reheat to, we opt to use the "Reheating as a Function of Cost" method of Abramson, Krishnamoorthy, and Dang [10] that we mentioned previously in Chapter 2. We choose this scheme in particular, because in their paper, the authors showed that this particular method outperformed all other proposed reheat schemes when tackling their particular timetabling problem (similar results are also reported in the work of Elmohamed *et al.* [51]).

6.2.5 Performance Gains via Delta-Evaluation

Finally, before going on to conduct an analysis of this algorithm, it is worth mentioning some points about how this algorithm can be made to run more efficiently during the annealing phases, via the introduction of some computation-saving measures to our evaluation function. Looking at the neighbourhood operators that are used in this algorithm, it should be easy to appreciate that because applications of N_1 and N_2 will only ever alter a small portion of a timetable, it is actually unnecessary (and imprudent) to set about evaluating the whole timetable after every move. Like many algorithms of this type, we therefore chose to make use of delta-evaluation [92], which is the name given to the general method of re-evaluation whereby only the relevant *changes* that have been made to a candidate solution by particular neighbourhood operator are considered. In our case, we can implement an efficient function of this type by making use of the following three facts:

(1) A single application of either N_1 or N_2 will only ever effect the assignments made to at most two of the five days in the timetable. Moreover, because the soft constraints imposed in the UCTP do not carry across different days, at least three of the five days will always be unaffected by a move and will therefore not need to be re-evaluated.

(2) If an application of N_2 involves moving an event to a new room in the same timeslot, or swapping two events in the same timeslot, then the cost of the timetable will not change.

(3) If an application of N_2 involves moving one event to a new timeslot, then only the students taking this single event will be affected (and thus only the personal timetables of these students will need to be recalculated). Meanwhile, if N_2 involves swapping two events in different timeslots, then only students taking just one of these (and not both) will need to have their personal timetables recalculated.

As can be imagined, in most neighbourhood-search algorithms such as this, the re-evaluation of candidate solutions will usually constitute a large part of the algorithm's computation time. Thus, the implementation of efficient delta-evaluation functions has the potential to offer considerable speed-ups, particularly when being used with larger problem sizes. Note, however, that although delta-evaluation is highly suitable with this sort of neighbourhood search-type algorithm, it is perhaps less applicable for use with any sort of evolutionary algorithm where recombination is being used (such as ours, which we mentioned earlier in the chapter). This is because a recombination operation usually results in offspring that are made of by information coming from multiple parents (as well as possibly a repair function). Thus the "changes" that occur in the offspring (compared to the parents) will usually be much larger, making a delta-evaluation process much more complicated and expensive on the whole.

6.2.6 Experimental Analysis

6.2.6.1 Assessing the Effects of Eliminating the End-of-Day Slots

For our experimental analysis of this SA algorithm, we performed two separate sets of trials on the twenty competition instances, again using the time limit that was specified by the competition-benchmarking program. In the first set of trials, we used our IHS algorithm (Section 4.7) to produce any feasible timetable where a maximum of forty-five timeslots was being used. The SA algorithm would then simply take this timetable and operate in the manner described above. For our second set, however, we chose to make a slight modification to the way in which the initial solution was produced by allowing the IHS procedure to run a little longer in order to try and schedule all of the events into a maximum of just forty timeslots (we chose to allow a maximum of 5% of the total runtime in order to achieve this). Our reasons for making this modification were as follows:

When we were designing and testing our SA algorithm, one characteristic that we sometimes noticed was the difficulty that the neighbourhood operator N_2 sometimes encountered when attempting to deal with violations of soft constraint SC_1: often, when trying to rid a timetable of a violation of the other two soft constraints SC_2 or SC_3, N_2 would actually do so by making use of an end-of-day timeslot. Or in other words, by trying to eliminate a violation of one type of soft constraint, the algorithm would inadvertently cause another one.

The reasons why such behaviour might occur start to become apparent if we look back at the descriptions of the three soft constraints in Section 3.1. Note that SC_2 and SC_3 stand out as being slightly different to SC_1, because if an event *e* is involved in a violation of either SC_2 or SC_3, then this will not be simply due to the position of *e* in the timetable, it will also be due to the *relative positions* of the other events that have common students with *e*. By contrast, if *e* is causing a violation of SC_1, then this will be due to it being assigned to one of the five end-of-day timeslots, and will have nothing to do with the relative positions of other events with common students to *e*. Thus, given that a satisfaction of SC_1 depends solely on not assigning events to the five end-of-day timeslots, a seemingly intuitive idea might be to simply remove these five timeslots (and therefore constraint SC_1) from the problem altogether. In turn, the SA algorithm will then only need to consider the remaining forty (unpenalised) timeslots and only try to satisfy the two remaining soft constraints. (Note that this will also have the effect of making the search space smaller, because there will also be $5m$ fewer places/cells to which events can be assigned to in the timetable)

In our experiments, it turned out that our strategy of allowing the IHS procedure to run a little longer worked quite well: using the IHS algorithm described in Section 4.7 together with an iteration limit of $10000n$, trials of twenty runs on each of the twenty competition instances revealed that this procedure was able to produce feasible initial solutions using forty timeslots in over 94% of cases. In the remaining cases, where initial solutions using a maximum of forty timeslots could not be achieved by our methods, our algorithm simply assigned these remaining events into the end-of-day slots. However, in order to still show preference to the task of eliminating SC_1 violations, in these cases we also chose to use a slightly modified version of neighbourhood operator N_2, which would operate using the following steps:

(1) First, randomly choose any non-blank cell *p* in the timetable;

(2) Next, randomly choose any cell *q* in the timetable, ensuring that (a) *q* is not in an end-of-day timeslot, and (b) $p \neq q$.

(3) Swap the contents of cells *p* and *q*;

This operator then, although very similar to the original N_2 operator described earlier, thus allows the prospect of *reducing* the number of events that are assigned to the end-of-day timeslots, but at the same time, does not allow the figure to increase. In our case, this strategy would always eliminate any remaining SC_1 violations within the first minute-or-so of the run and, obviously, once these *had* been gotten rid of, the nature of this modified version of N_2 would mean that these extra timeslots would then be closed off and removed from the problem.

6.2.6.2 Results and Analysis

In Table 6.3 we provide a comparison of these two sets of trials by displaying the average and best results achieved at the time limit in 50 runs on each of the 20 instances. In both cases we used a cooling rate of $\alpha = 0.995$, and specified for the reheating function to be called when 30 successive temperatures were encountered without an improvement in cost. Meanwhile, suitable values for $\eta_{\min}$ and $\eta_{\max}$ (the two parameters that we witnessed to be the most influential in regards to algorithm performance over time), were determined empirically in some initial experiments, whereby we ran the algorithm at 11 different settings for $\eta_{\max}$ (between 1 and 41, incrementing in steps of 4) and 10 different values for $\eta_{\min}$ (0.1 to 1.0, in steps of 0.1). At each setting for $\eta_{\min}$ and $\eta_{\max}$ in these preliminaries, we then performed twenty separate runs on each of the twenty competition problem instances, thus giving a total of 400 runs per setting. The best performing values for $\eta_{\min}$ and $\eta_{\max}$ in both cases (i.e. the settings that gave the lowest average cost of the 400 runs when using forty and forty-five timeslots) were then used in our comparison.

As can be seen in Table 6.3, when considering just forty timeslots, the SA-algorithm is able to produce better average results in the majority of cases (seventeen out of the twenty instances). Additionally, we can see that the *best* results from the fifty runs are also produced in sixteen of the twenty instances in this case, with ties occurring on a further two. A Wilcoxon signed-rank test also revealed that the differences in results produced in each set of trials was significant (with a probability greater than 95%). The results gained when using forty timeslots also compare well to other approaches: for example, had the best results in Table 6.3 (i.e. those using 40 Slots, $\eta_{\max}$ = 5, and $\eta_{\min}$ = 0.9) been submitted to the timetabling competition, then according to the judging criteria, this algorithm would have ranked second. Note, however, that our algorithm actually beats the competition winner in ten of the twenty instances, and according to a Wilcoxon signed-rank test, there is also no significant difference between the two algorithms. (The results reported for the competition winner are also the best results gained from 50 separate runs on each instance.)

Table 6.3: Comparison of the Two Trial-Sets (i.e. Using Forty-Five and Forty Timeslots Respectively) on the Twenty Competition Instances. In Each Case the Average and Standard Deviation in Cost is Reported, as well as the Best Cost (parenthesised) from 50 Individual Runs.

Instance #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Average
45 Slots,	85.9	68.5	86.9	260.1	190.1	31.5	42.3	28.3	52.2	84.9	61.6	147.5	130	107	41.5	47.2	169.3	45.9	85.5	9.5	88.8
$\eta_{\max}$ **= 9, and**	±10.9	±8.2	±12.7	±23.4	±25.7	±8.8	±17.0	±7.2	±9.6	±8.0	±9.7	±16.3	±14.1	±33.4	±8.5	±7.9	±26.5	±7.8	±14.7	±4.5	
$\eta_{\min}$ **= 0.1**	(68)	(49)	(63)	(207)	(133)	(12)	(19)	(14)	(31)	(68)	(43)	(109)	(101)	(55)	(22)	(29)	(119)	(27)	(62)	(1)	(61.6)
40 Slots,	86.9	53.5	95.6	231.8	147.7	22.8	23.7	22.2	41.4	91.7	60.6	133.8	128.2	66.3	33.2	35.8	129.4	40.8	84.9	8.6	77
$\eta_{\max}$ **= 5, and**	±17.6	±10.2	±18.8	±39.5	±29.5	±8.4	±13.3	±8.6	±13.7	±15.3	±16.0	±28.1	±19.2	±20.7	±13.6	±12.6	±25.0	±9.7	±21.2	±6.1	
$\eta_{\min}$ **= 0.9**	(62)	(39)	(69)	(176)	(106)	(11)	(5)	(10)	(22)	(70)	(38)	(94)	(101)	(37)	(14)	(18)	(94)	(27)	(55)	(0)	(52.4)

The reasons why we believe the use of just forty timeslots to be advantageous (instead of forty-five) in this case have already been outlined in Section 6.2.6.1. However, it is worth noting that one potential negative feature of this particular approach is the fact that the removal of the end-of-day timeslots will also have the effect of reducing the number of blanks that are present in the timetable matrix. Indeed, considering that moves in N_2 that involve a blank cell (and therefore just one event) are, in general, more likely to retain feasibility than those involving two events, this means that the removal of blanks will also lead to further restrictions being placed on the algorithm's ability to move about the search space (we remember that there is already a considerable restriction placed upon this algorithm because only moves that preserve feasibility are permitted). Given that one of the major requirements for the two-stage timetabling approach *is* for practical amounts of movement in feasible search space to be achievable (see Chapter 2), there is thus a slight element of risk in reducing the number of timeslots in this way, as in some cases too many restrictions might be imposed. However, in these instances (where, we note that perfect solutions are always achievable) the strategy appears to be beneficial.

Finally, in fig. 6.7 we show two example runs of the SA algorithm using the parameters defined in Table 6.3. Here we can observe the general contributions that both phases of SA lend to the overall search, and also the general effects of the reheating function (although in one case we can see that the latter has been invoked too late to have a positive effect). We can also see that the second line – which uses 40 timeslots – actually starts at a markedly lower cost than the first, because the elimination of all violations of SC_1 in this case has actually resulted in a better quality initial solution. However, note that this line also indicates a slower progression through the search space during the first half of the run, which could well be due to the greater restrictions on movement within the search space that occur as a result of this condition.

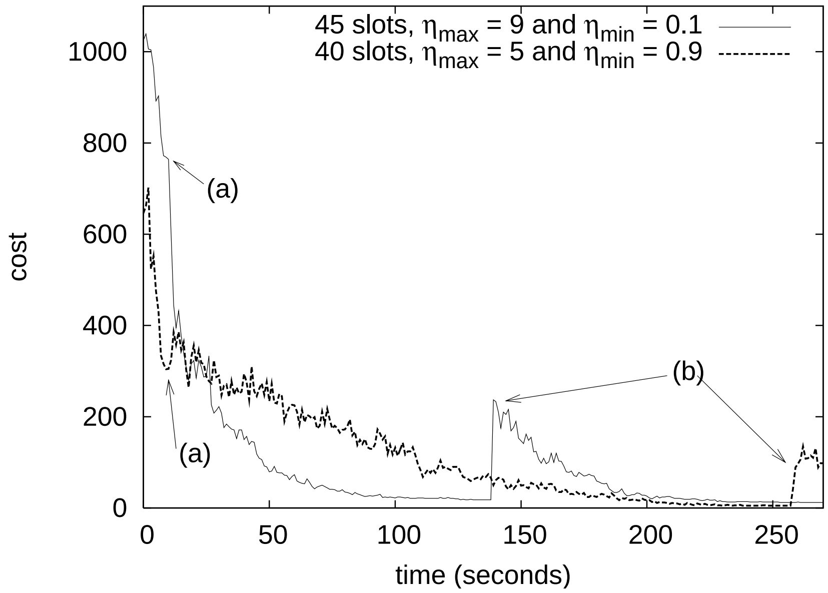


Fig. 6.7: Two example runs of the SA algorithm on competition instance-20. The points marked (a) indicate where the algorithm has switched from SA phase-1 to SA phase-2. Points (b) indicate the point in the runs where a reheating has occurred.

6.3 Conclusions and Discussion

In this chapter we have designed and analysed two separate metaheuristic algorithms for the second stage of the two-stage timetabling strategy. Both have used the same representation and fitness function, but the second of these – our simulated annealing algorithm – has clearly outperformed the proposed evolutionary algorithm in these cases.

As we have noted, we believe that one of the main reasons for this unsatisfactory performance of the EA is the apparent difficulty in the task of producing a recombination operator that can aid the search in a positive manner, whilst at the same time also ensuring that the offspring produced obey the hard constraints of the problem.

With our SA-based algorithm, meanwhile, we have also seen that significant improvements in results can be gained if we choose to place special priority on removing the five end-of-day timeslots from the timetable, thereby eliminating one of the three soft constraints of the problem. However, it is worth reiterating that this strategy will also actually have the effect of tightening the search space in most cases, and it is likely that with other problem instances, particularly those that are more constrained in nature, this approach also has the potential to *decrease* algorithm performance. However, as we have seen, this is not the case with these instances.

As we have mentioned above, the results of the SA algorithm using forty timeslots compare very well with the official entrants to the International Timetabling Competition, although it should be noted that at this point in time, better results for these twenty instances overall, have been reported in papers by both Kostuch in [69] and Chiarandini *et al.* in [34]. However, when making comparisons such as these, although the use of benchmark problem instances and timing programs can be helpful for comparing algorithms, it is also worth bearing in mind the following points when making any overall conclusions. Firstly, as we noted in Chapter 3, the timing program used for the competition (and subsequent works) specifies the time limit in *elapsed* time (or “wall-clock” time), as opposed to CPU time. This means that there is the potential for discrepancies to occur if the computer’s operating system also has to deal with things such as background processes, remote users, the arrival of emails, and so on[17]. (Of course, these factors will also depend on the *type* of operating system being used as well.) Additionally, the use of time (CPU or elapsed) in itself is also open to inaccuracies, because it obviously brings into

[17] In fact, in order to try and eliminate this potential inconsistency with timing, in all of our experiments in this chapter we actually chose to measure algorithm performance using CPU time rather than wall-clock time. In order to work out a suitable CPU time limit, we did the following. First, using the wall-clock time limit specified by the benchmarking program (we will call this wc_1) we took an unoccupied computer (the specification of which was given in Section 4.3), and ran our program for exactly wc_1 CPU seconds. At the end this run we then collected a second value wc_2 which was the amount of wall-clock seconds that this run actually took (which due to background processes etc. would inevitably be more than wc_1). The proportion (wc_1/wc_2) thus gives us an approximation of the amount of CPU time that will be lent to the program during a run on an unoccupied computer, and a suitable CPU time limit for the experiments can be calculated as $wc_1(wc_1/wc_2)$.

question the implementation details of the algorithm, such as the programming language that was used, the type of compiler, and so on.

It is also worth considering that the actual time limit that is used for these comparisons will also have an important effect in the results that are gained. Indeed, although the time limit used for the competition may well be a fair one, it is in essence arbitrary, and it is likely that the relative performances of the various algorithms proposed for the competition (with regards to the best solutions that they can find) will change were it to be lengthened or shortened. For instance, in our SA algorithm if our time limit was lower, then as well as having to use different values for many of the parameters, it could be the case that the first version of the SA algorithm (where we use the full forty-five timeslots) might actually bring better overall results because, as we demonstrate in an example run in fig. 6.7, there will tend to be more freedom of movement in the search space, thus allowing the algorithm to make quicker initial movements (indeed, in fig. 6.7 it is only at around 200 seconds that the advantages of using just forty timeslots seems to become evident). Remember also that the twenty benchmark instances used in these comparisons are all, in essence, quite similar, and it is likely that the relative performances of the different algorithms would change if instances of different sizes and "constrainedness" were also to be considered.

It is also worth mentioning once again that both algorithms described in this chapter, as well as the various other high-performance two-stage algorithms for this problem (reviewed in Chapters 2 and 3) all crucially depend on two things: (a) that feasibility is relatively easy to achieve, and (b) that there is enough freedom in the feasible-only search space for a suitable amount of optimisation to take place. It would appear that the twenty benchmark problem instances that were used for the competition allow this to be achieved. However, with regards to the second requirement, in our case some further reasons as to why our second approach seems able to produce good results can also be demonstrated by looking carefully at the characteristics of our feasibility-preserving neighbourhoods N_1 and N_2 described above:

For purposes of demonstration, let S represent a feasible timetable that is using s timeslots (such that $s \leq t$). As we noted earlier, through the use of N_1 alone we have the potential to produce another (s! – 1) new timetables from S, all of which will also be feasible. Now, given one of these new timetables S', in the vast majority of cases it is likely there will be some (and possibly many) further feasible timetables S'' that are obtainable through an application of N_2. In turn, each of these new solutions will then also have a further (s! – 1) feasible solutions S''' that can produced through a second application of N_1, and so on. Using this simple demonstration, it is thus easy to appreciate that even with these relatively simple neighbourhood operators, the number of different feasible timetables that are obtainable by their application can quickly become very large, and the number of feasible timetables that we have the potential to search through is unlikely to be a trivial amount. Indeed, these numbers are likely to grow at an even greater rate if we were also to consider other feasibility-preserving operators such as the Kempe-chain interchange operators used in some other works [106, 107].

Concluding this chapter, two things should be noted by the reader. First, the IHS algorithm, which we have mainly analysed in Chapter 4, generally seems to be an effective way of achieving feasibility with the UCTP with known problem instances. Second, the SA algorithm discussed in the latter half of this chapter, at least for the twenty competition instances used here, seems to be effective at reducing the number of soft constraint violations, despite being required to navigate in the restricted "feasible-only" search space. However, as we noted in the problem analysis of Chapter 3, the UCTP problem-model is still essentially a simplification, and it is worth remembering that various other sorts of hard constraints that one could encounter in the real world are not presently considered in the model. This issue therefore begs the following questions: What would happen if extra hard constraints *were* to be added to the UCTP? Would this affect our ability to achieve feasibility? Also, what effects would the presence of these extra constraints have on a two-stage timetabling algorithm's capacity to navigate through the feasible-only search space? Would performance increase or decrease? In the next chapter – the penultimate chapter of this thesis – we will endeavour to answer some of these questions.

7: Towards Real-World Timetabling

Up until this point in this thesis, we have spent most of our time analysing the effectiveness of the two-stage timetabling approach on a particular version of the UCTP. From this analysis we have seen that the Iterated Heuristic Search algorithm is generally the most effective of our algorithms for achieving timetable feasibility (Chapter 4); and that by navigating exclusively through feasible areas of the search space, our simulated annealing algorithm, is generally able to satisfactorily reduce the number of soft constraint violations within a feasible solution (Chapter 6). However, at various points in earlier chapters of this thesis we have also made note of the fact that although our chosen timetabling problem-version is useful for scientific research and algorithmic analysis, it is still essentially a simplification of most real-world timetabling problems. In this penultimate chapter, it is perhaps, therefore, an appropriate time to start looking at how other sorts of constraints might be incorporated into this two-stage algorithmic framework. Consequently, we will take a close look at a particularly interesting type of constraint whose absence is arguably the most conspicuous – what we will call the "unavailability constraint" – and will investigate what effects the addition of such constraints might have with respect to finding feasibility (Section 7.2) and for satisfying the soft constraints (Section 7.3). In Section 7.4 we will then round off this chapter by conducting a short discussion on some further hard-constraints and other features that might also occur in some practical university timetabling problems, and we will make some suggestions as to how they might be incorporated into our two-stage timetabling approach in the future.

7.1 Unavailability Constraints

For the purposes of this chapter we can consider an "unavailability constraint" as a type of constraint that specifies that something or someone is not available at some specific time. Typically, unavailability constraints can come in two forms:

(1) **Room-Unavailability Constraints:** such as "room *a* is unavailable for use in timeslot *b*", or "room *c* is unavailable on Thursdays";

(2) **Event-Unavailability Constraints:** such as "event *i* is unavailable in all timeslots except timeslot *b*", or "event *j* is unavailable in all morning timeslots".

The effects that the addition of room-unavailability constraints will have on a timetabling problem are actually quite obvious – if, for example, room *i* cannot be used in timeslot *j*, then this *place* in the timetable (or matrix-cell with our chosen representation) cannot be used by any event in the problem instance and can therefore simply be ignored. Naturally, this means that as the number of room-unavailabilities increases, consequently the number of places that are available in the timetable will *decrease.* (This also implies that if the total number places eliminated by room-unavailability constraints x rises to a point such that $(m \times t - x) < n$, then the problem instance will no longer be solvable, because there will not be enough places for all of the events.)

The issues surrounding the introduction of event-unavailability constraints, meanwhile, seem to be a little more complicated, and so their investigation will therefore constitute the majority of this chapter. First of all, in real-world situations event-unavailability constraints can come in various forms such as "event *i* cannot be assigned to timeslot *j*", "event *i* cannot be assigned to afternoon timeslots", and so on. They can also be stated in the form of *pre-assignments*, such as "event *i* must be scheduled into timeslot *j*", or "event *i* must be scheduled on a Monday". Additionally, they can also be expressed from the point-of-view of the student (and/or lecturer), as in "student *x* is unavailable on Wednesday afternoons", and so on.

As a matter of fact, however, all of the above types of event-unavailability constraint can be specified by a simple $(n \times t)$ Boolean matrix, which can be used to indicate those events that are available and unavailable for each of the individual timeslots. In particular, note that even when considering these constraints from a student's point-of-view, as we have just seen, we can still easily infer the appropriate event-unavailability constraint by simply identifying all of the events that student *x* is required to attend, and by then making these events unavailable in the appropriate timeslots.

Note that the presence of event-unavailability constraints can also provide us with further information with regards to the pre-processing steps that we saw in Chapter 3. For example, if in a particular problem instance a pair of events *i* and *j* have been pre-assigned to different timeslots, then we can actually set elements (*i*, *j*) and (*j*, *i*) in the associated conflicts matrix to be true (i.e. we can add an edge between vertices *i* and *j* in the underlying graph colouring model). This is because we know that these two events cannot now be assigned to the same timeslot in any feasible timetable. In practical situations, adding extra constraints such as this might allow us to determine more accurately just how difficult a particular timetabling problem is. (In his survey paper for exam timetabling, Carter [28] also suggests that if two events have been pre-assigned to the same timeslot, then these can then be merged into one *super*-event that will have the union of conflicts of the originals. However, in the case of the UCTP this step is not wholly appropriate, as we still also need to assign individual rooms to each of these events.)

7.2 Coping with Unavailability Constraints when Achieving Feasibility

In Chapters 4 and 5 we saw that in our algorithms for both the UCTP and graph colouring problem, the grouping representation that was used did not consider the labelling (or ordering) of the groups. Instead, the emphasis was placed – we believe more appropriately – on *how* the items were grouped and into *how many* groups. As we have discussed, when attempting to find feasibility with the UCTP such a representation seems fitting, because the current hard constraints for the UCTP are not concerned with matters pertaining to the timeslots' orderings. However, if we now wish to also incorporate unavailability constraints into our timetabling problem, then it is easy to see that the grouping representation in its pure form is actually no longer appropriate. This is for the obvious reason that constraints such as "event *i* cannot go in timeslot *j*" definitely *are* concerned with the labelling of the timeslots, because it is now necessary to be able to identify exactly which timeslots each of the events can be assigned to. On the face of it, this additional requirement might not seem too much of a problem – all we seemly need to do is to add labels to each of the timeslots. However, in our case this new requirement suggests that our current strategy of opening-up extra timeslots (beyond *t*) in order to deal with the unplaceable events, is perhaps not so appropriate anymore, because each of these additional timeslots, from the point of view of these new constraints, will be meaningless. (Which events should be considered available and unavailable in timeslot (t + 1), for example?)

This factor would seem to suggest, therefore, that when trying to find feasibility in this case, it might be more appropriate to fix the number of timeslots being used to *t* from the outset, thus allowing us to label each timeslot explicitly. However this will, of course, also require us to make modifications to our existing algorithmic model in order to cope with any unplaceable events that we encounter during a run. In the next subsection we will therefore define such modifications, which will then be used in conjunction with our procedure HEURISTIC-SEARCH for the UCTP (see fig. 4.11) In Section 7.2.2 will then perform an experimental analysis of this algorithm using our own instance generator, and will present some results.

7.2.1 Algorithm Description

Our new algorithm for finding feasibility is described by the pseudo-code in figure 7.1. This procedure, that we will call BUILD-COMPLETE-TIMETABLE, takes as arguments an empty timetable *tt* and a list of unplaced events *U*. (Of course, when this procedure is first called *U* will represent the entire set of events, and therefore $|U| = n$). The process then works in the following way. First, an initial solution is constructed using the procedure MAKE-INITIAL-TIMETABLE. This procedure uses the same event and place-selection heuristics as our previous solution builder for the UCTP (Chapter 4)[18]. However, in contrast to our previous methods, in this case when events are encountered that cannot be feasibly placed into *tt*, rather than open extra timeslots, these events are simply kept to one side in the list of unplaced events *U* for consideration later on.

Moving on to the main body of the BUILD-COMPLETE-TIMETABLE procedure, if there are any events remaining in *U* after we have applied MAKE-INITIAL-TIMETABLE, then first-of-all the heuristic search procedure from Chapter 4 (fig. 4.11) is called in order to try and reduce this figure. Upon completion of this step, if *U* is still non-empty, then it is assumed that *tt* will need to undergo some more radical alterations in order to accommodate these remaining events. In our case what we choose to do here is to remove a number of other events from *tt* and put these into a second list *V*. This is achieved by employing the function EXTRACT-SOME-EVENTS, which we will look at in more detail below. By removing events from *tt*, we are, of course, creating extra places in the timetable into which we can hopefully insert some, if not all, of the remaining events in *U*. Thus in our approach the events in *V* are put to one side, and in line 6 of BUILD-COMPLETE-TIMETABLE the heuristic

[18] Note that the constructive heuristics used so far in this thesis (outlined in Table 4.1) are still appropriate here. For example, if a particular event has been pre-assigned to a relatively small number of timeslots, then due to heuristic rule H_1, it will be inserted into the timetable relatively early on in the constructive process.

search procedure is again applied using *U* and the new, emptier version of *tt*. Finally, upon completion of this second phase of heuristic search, the unplaced events in *V* are added to the events (if any) that are still in *U*, this list is randomly permuted, and the process is repeated.

```
BUILD-COMPLETE-TIMETABLE (tt, U,)
1. MAKE-INITIAL-TIMETABLE (tt, U);
2. while ($U \neq \emptyset$ and (time limit not reached))
3.    HEURISTIC-SEARCH (tt, U, itLimit);
4.    if ($U \neq \emptyset$)
5.       V := EXTRACT-SOME-EVENTS (tt, $|U|$);
6.       HEURISTIC-SEARCH (tt, U, itLimit);
7.       Move all events from V into U, and randomly permute U;

MAKE-INITIAL-TIMETABLE (tt, U)
1. Open t new timeslots in tt;
2. while (there are events in U with feasible places in tt between
         timeslots 1 and t)
3.    Choose an event e from U that has feasible places in tt;
4.    Pick a feasible place p for e;
5.    Move e from U to p in tt;

EXTRACT-SOME-EVENTS (tt, q)
1. $V := \emptyset$;
1. for (i := 1 to q)
2.    Randomly choose two events e and g currently assigned to tt;
3.    Move either e or g (according to some heuristic) from tt to V;
```

Fig. 7.1: The BUILD-COMPLETE-TIMETABLE procedure. In this pseudo-code *tt* represents the timetable and *U* and *V* are lists of unplaced events (of lengths $|U|$ and $|V|$ respectively). The function HEURISTIC-SEARCH is already described in figure 4.10 of Chapter 4. As usual *t* represents the target number of timeslots, which in this case = 45.

Looking closely at BUILD-COMPLETE-TIMETABLE in fig. 7.1, it should be noticeable that if we are extracting some events from the timetable in line 5 and then subsequently adding these to the list of unplaced events in line 7, then unless the heuristic search routine in line 6 has managed to transfer all of the events currently in *U* into *tt*, then the overall number of unplaced events will actually *increase*. Of course, with regards to the overall aim of this algorithm, such a situation is undesirable because it will move us further away from our goal of inserting all of the events into the timetable. However, in our case it is hoped that this will only be temporary as, with a bit of luck, the size of *U* will once again decrease when the algorithm then loops back to the heuristic search routine in line 3. (This matter will be examined further in our analysis in the next section.)

Note that our strategy of extracting events from the timetable in order to reinvigorate the search also raises two questions: *how many* events should be extracted; and *which* events should be extracted? To address the first issue, in our case we choose to extract exactly $|U|$ events. This seems an intuitive choice because this will free up exactly the number of extra places in *tt* that are needed to house those events residing in *U*. However it should be noted that this choice is only instinctive, and it is possible that the removal a larger or smaller amount might bring about a better performance in some cases. Meanwhile, the second issue mentioned above also deserves consideration. When choosing which events to extract from *tt*, we could simply make our choices at random. However, it could be that in certain circumstances it might be more favourable to remove some events rather than others. For example, if we were to bias our choices towards removing events that have a low conflicts degree (i.e. events that in many cases will more easy to insert into a timetable) then this may aid the algorithm in the long run, as these events might be more easy to re-insert into *tt* later on. Conversely, however, it could be the case that the removal of events of a *higher* conflicts-degree might be a better strategy, because by removing these more "problematic" events, it may help the algorithm by freeing up the resources needed by the other "problematic" events already residing in *U*. In Table 7.1, we therefore suggest five different heuristic rules that can be used in order to bias our choices of which events to remove from *tt* when applying Extract-Some-Events. Heuristic rule h_1 is simply a random choice. Rules, h_2 and h_3, meanwhile, consider the conflicts-degrees of the events, showing slight biases towards events of high degrees and low degrees respectively. Lastly, heuristic rules h_5 and h_6 consider the number of places (i.e. timeslots and rooms) that are available for each of the events when the timetable is empty; heuristic h_5 biases the extraction of events with higher number of places, and h_6 the opposite.

Table 7.1: The Heuristics used with the Extract-Some-Events Procedure in Fig 7.1. In This Case, All Ties are Broken Using Rule h_1.

Heuristic	Description
h_1	Choose between *e* and *g* randomly.
h_2	Choose the event with the highest degree.
h_3	Choose the event with the lowest degree.
h_4	Choose the event that has the highest number of feasible places in *tt*
h_5	Choose the event that has the lowest number of feasible places in *tt*

7.2.2 Algorithm Analysis

In our experimental analysis of BUILD-COMPLETE-TIMETABLE we will investigate the implications of adding event-unavailability constraints to our timetabling problem. (We remember that this particular type of unavailability constraint is perhaps a little more interesting then the room-unavailability constraint, whose presence simply causes places in the timetable to be eliminated.) In the resultant experiments we will be looking at two main issues:

(1) How this algorithm is generally able to cope with the additional presence of the event-unavailability constraint; and

(2) What effects, if any, the various event-extraction heuristics have on the algorithm's performance overall.

In order to help us investigate the first issue, we implemented our own instance generator. This operated by taking an existing UCTP problem-file, and then simply appended an $(n \times t)$ event-unavailability matrix on to the end. The characteristics of this matrix are controlled by a parameter p defined in the range [0, 1] which is used to control the overall "constrainedness" of the problem instance (with regards to the event-unavailability constraint). In our case this was done by simply going through each element of this new event-unavailability matrix in turn, and marking it false with a probability p, and true otherwise. Thus instances generated with low p-values will generally represent fairly unconstrained instances (with regards to event-unavailabilities) as each of the events will be available in most, if not all, of the t timeslots. Problem instances with a high p-value, meanwhile, will generally represent constrained problem instances, as each event will only be able to be feasibly assigned into a small number of timeslots, if any.

Using this instance generator we chose to perform experiments using ten problem instances from the set of twenty used for the International Timetabling Competition[19]. For each of these ten "root-instances", we then produced ten new problem instances for p-values of 0.0 through to 1.0, incrementing in steps of 0.05. (That is, for each of the 21 values for p considered, we produced 10 different event-availability matrices for each of the 10 problem instances – giving $(21 \times 10 \times 10) = 2100$ problem instances in total.) We then performed ten individual trials with different random seeds on each of these instances, using a CPU time limit of 200 seconds and an iteration limit to $1000n$. (These time and

[19] For consistency we chose to make use of ten instances of equal size (n = 400 and m = 10). Specifically, these were competition instances 2, 3, 4, 8, 10, 11, 12, 13, 18, and 19.

iteration limits are consistent with similar experiments in Chapter 4.) These experiments were then repeated for each of the five extraction-heuristics.

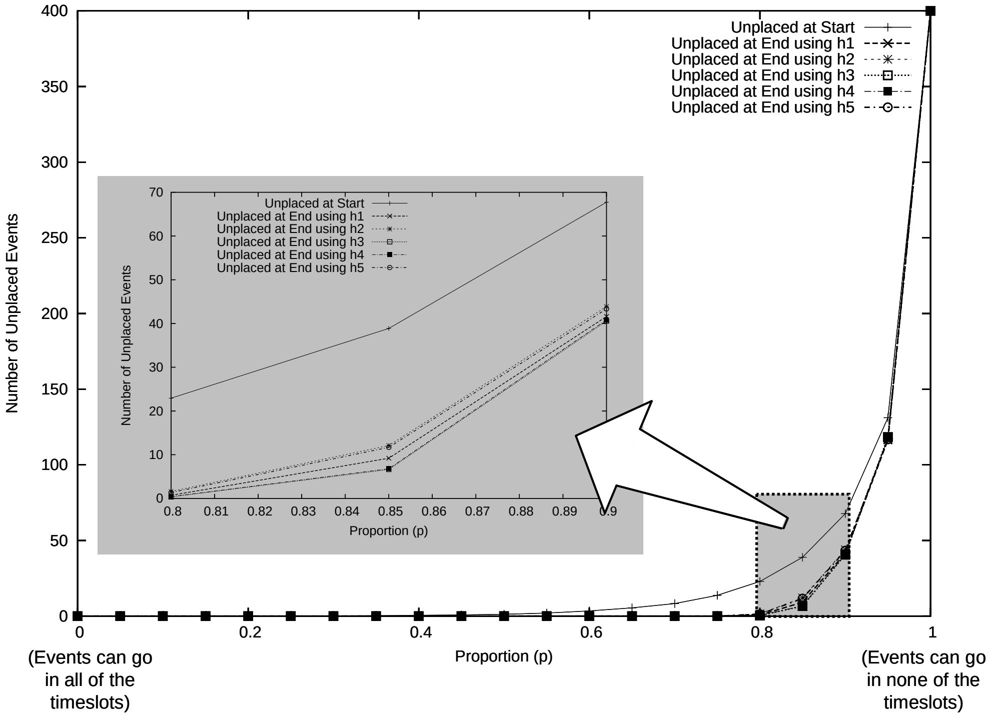


Fig. 7.2: Accuracy of the five algorithm variants for various *p*-values. Inset in grey, we show a magnification of the graph around the values for *p* in which there is a slight difference in performance due to the different extraction heuristics. All points are the averages of 1000 runs (as explained in the text). Also note that when *p* = 1.0, this means that all timeslots are unsuitable for all events, and therefore none of the 400 events can be assigned to the timetable.

In figs. 7.2 and 7.3 we show the results of these experiments with regards to algorithm accuracy and computational effort respectively. In fig. 7.2, for *p*-values 0.0 through to 1.0, two things are shown: (a) the number of unplaced events in the initial solutions produced by Make-Initial-Timetable; and (b) the number of events that remained unplaced upon termination of each of the five algorithm variants. All plotted points are averaged across the 1000 individual runs performed at each *p*-value. In fig. 7.3, meanwhile, we display the average amount of CPU time that it took for the algorithms to find complete, feasible solutions for the same runs. (As in chapter 5, only the runs in which a fully feasible solution were found were considered in the latter's calculation.)

The first thing to notice from fig. 7.2 is that from *p*-values of 0.0 up to 0.75, the algorithm is able to able to find complete, feasible solutions within the time limit in every case. Furthermore, in lower values for *p* (0.0 to approximately 0.4) we can see that this is nearly always achieved at the start of the run. These observations indicate two things. First, they suggest that the constructive heuristics used in our Make-Initial-Timetable procedure are able to cope well with the addition of event-unavailability constraints (when present in moderate amounts); second, they show us that even when some events have been left unplaced by Make-Initial-Timetable, the remaining steps of our algorithm will usually reduce this number by a reasonable amount within the imposed time limit.

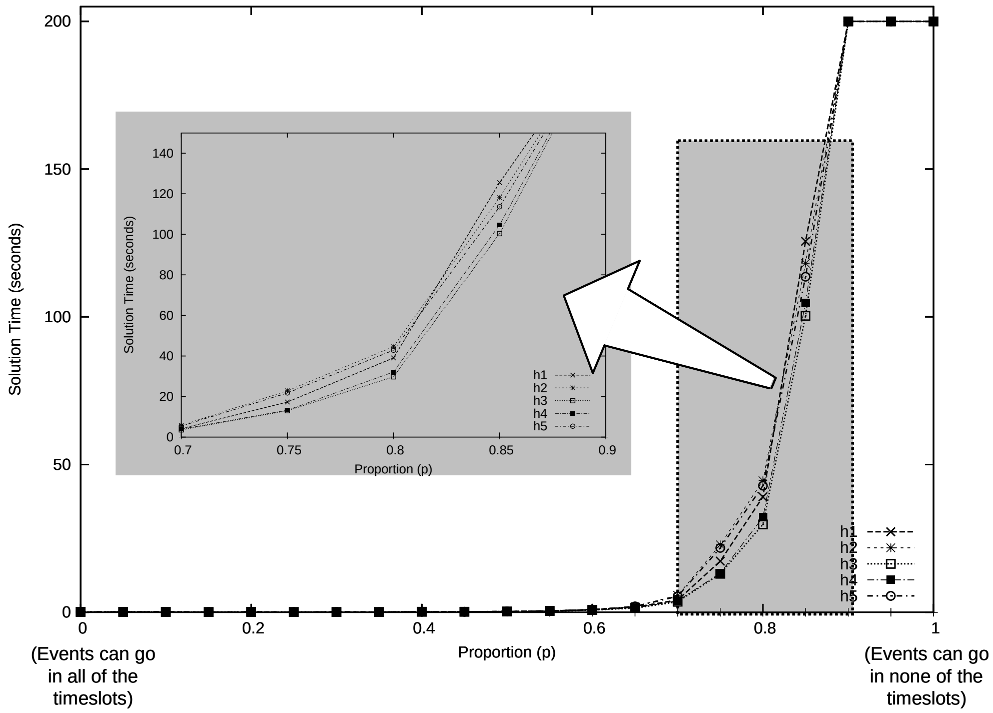


Fig. 7.3: Computational effort of the five algorithm variants for various p-values. Inset in grey, we show a magnification of the graph around the values for p in which there is a slight difference in performance due to the different extraction heuristics. All points are the averages of 1000 runs (as explained in the text).

Also noticeable in both figs 7.2 and 7.3, is that for a small range of p-values (roughly p = 0.75 to 0.85), the heuristics governing the choices of which events to extract from the timetable also seem to have a slight impact on the accuracy and solution times of the algorithm (these characteristics are perhaps slightly more noticeable on the magnifications, which are inset and in grey in the original figures). In particular, we can see that extraction heuristics h_3 and h_4 tend to give a slightly better performance than the three other heuristics. This suggests that by biasing our choices towards removing the less troublesome events (i.e. events with lower conflicts-degrees or events with a larger numbers of feasible places), this will aid the algorithm in the long run, because the extracted events will generally be more easily re-inserted by Heuristic-Search later on. However, it is also worth noting that the differences between the algorithm variants are quite small on the whole, suggesting that it is the successive applications of Heuristic-Search that are doing the majority of the work in general.

Another noticeable characteristic in figs. 7.2 and 7.3 is that for p-values of around 0.9 up to 1.0, none of the algorithm variants is able to find any complete and feasible timetables within the imposed time limit. However, in our present method of instance generation we have not actually concerned ourselves in ensuring that each instance even features an optimal solution; thus it could be the case that these problem instances do not even have feasible solutions available in the first place. Indeed, when p is greater than 0.9, for example, this implies that over 90% of timeslots will be unsuitable for each of the events on average. It is therefore quite possible in these cases that there will be occurrences of events that are marked as unavailable in *all* of the t timeslots, and/or that two conflicting events might be pre-assigned to the same single timeslot (Note that there are various other factors here that might make an instance unsolvable, although we will not go into these here.)

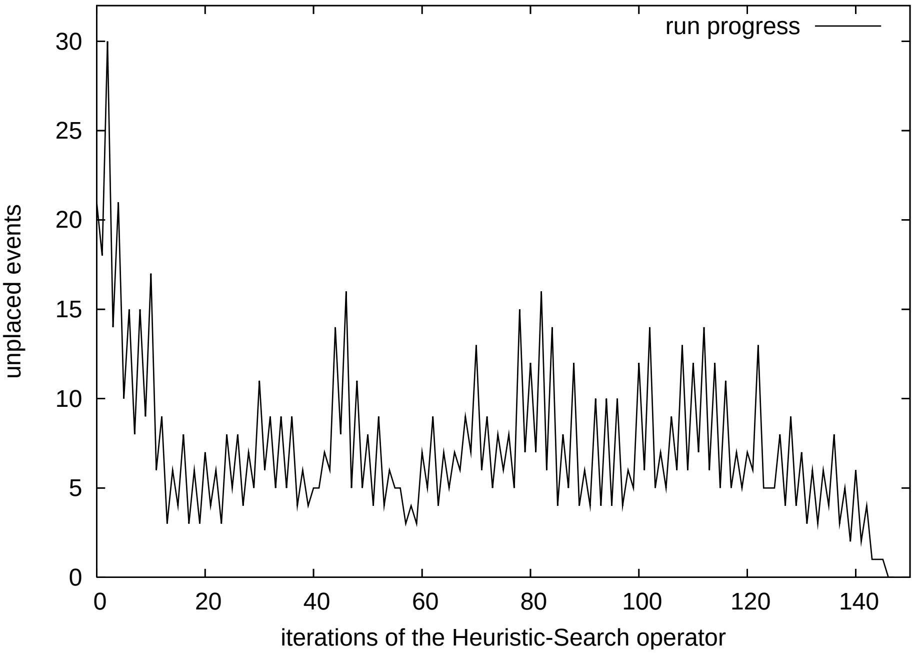


Fig. 7.4: An example run of BUILD-COMPLETE-TIMETABLE. In this figure we show the number of unplaced events that occur after each successive application of the HEURISTIC-SEARCH function (lines 3 and 6 of BUILD-COMPLETE-TIMETABLE, fig 7.1). This particular run was performed using Competition instance-19 with p = 0.8, using extraction heuristic h_3. All other parameters are specified in the text.

In fig. 7.4 we also show an example run of this algorithm with regards to the total number of events that remain unplaced after each successive application of HEURISTIC-SEARCH (i.e. lines 3 and 6 of the BUILD-COMPLETE-TIMETABLE procedure in fig. 7.1). The jagged line in this plot clearly indicates the phenomenon that we alluded to in Section 7.2.1 – i.e. that the total number of unplaced events (those that are held in *U and V*) will often have increased after every second application of the HEURISTIC-SEARCH operator. However, as is demonstrated in this figure, these increases tend to be complemented by the remaining applications of Heuristic-Search (line 3) where decreases in the total number of unplaced events will usually occur. (In this particular case we can see that the number of unplaced events is reduced from 21 down to zero in 145 applications of the operator.)

7.3 Effects of Event-Unavailability Constraints when Satisfying Soft Constraints

As we have noted, the second question that arises when considering the addition of event-unavailability constraints is: what effect do these additional constraints have on our ability to satisfy the soft constraints of the UCTP? And in particular, what are the implications with regards to the ease in which we can perform an adequate navigation in the resultant feasible-only search space? In order to investigate these issues we took our simulated annealing algorithm from the previous chapter and made a small number of modifications so that it would also be able to deal with the event-unavailability constraint. Specifically, these modifications were as follows. First, we replaced our original method of constructing initial feasible timetables with the Build-Complete-Timetable procedure, described above. Secondly, we removed SA phase-1 from the algorithm (i.e. the phase where timeslots were ordered using neighbourhood operator N_1 – described in Section 6.2.1-2), as obviously this process is now not completely appropriate for our purposes. Finally, we also updated our neighbourhood operator N_2 (fig. 6.5) so that, in addition to the original hard constraints of the UCTP, moves that caused a violation of any unavailability constraint would now also be immediately rejected.

Note that the final point of the previous paragraph is likely to be of some importance, because the presence of additional constraints will mean that a greater proportion of N_2 moves will be rejected in general. This, in turn, suggests that movements through the feasible-only search space will also become more restricted – perhaps to a point where an adequate search is not actually achievable.

In order to investigate this issue we chose to implement a second instance generator very similar in style to our previous one, described in Section 7.2.2. However, for purposes of comparison, in this case we also decided that each generated problem instance should have at least one perfect solution obtainable. Our modified problem instance generator therefore operates in the following way. First, an existing UCTP problem instance is taken, together with a corresponding perfect solution. (We remember that the UCTP instance generator, described in Chapter 3, allows the user to specify that a perfect solution should be produced alongside the problem instance.) Next, as before a Boolean ($n \times t$) event-unavailability matrix is then appended to the end of the instance file, with a parameter p again being used to control the proportion of true and false entries. In difference to our previous generator, however, in this case when creating the event-unavailability matrix, we also take special care to ensure that if an event i is assigned to timeslot j in the perfect solution, then element (i, j) in the event-unavailability matrix is also kept false (i.e. event i must not be unavailable in timeslot j). By doing this, it is easy to appreciate that a problem instance generated in this way, though featuring extra constraints, will always feature a perfect solution.

Using this new instance generator, we then performed the following trials. For a given value of p, one hundred problem instances were first generated by producing ten versions of each of the same ten "root-instances" used in Section 7.2.2. (The perfect solutions for these ten competition instances were supplied to us by the competition organisers.) For each one these new problems we then performed ten individual trials with our SA algorithm, giving 1000 individual runs in total (all runs were performed using the same parameters for the SA algorithm as described in Section 6.2.6 (when using forty-five timeslots)). These experiments were then repeated with p-values of 0.0 through to 0.8, incrementing in steps of 0.1. Finally, in order to allow us to observe the algorithm's run-characteristics in their proper context, we also used excess computation time of 1000 seconds of CPU time for each run. Note that we did not consider p-values greater than 0.8 in these experiments because, due to factors that will be explained below, the BUILD-COMPLETE-TIMETABLE procedure was not always able to produce feasible timetables in these cases.

In figure 7.5 we demonstrate the effects that different values for p have on the SA algorithm's ability to make positive movements through the search space over time. As can be seen, as p is increased, then so does the average cost at which the search stagnates. This clearly demonstrates that as the number of event-unavailability constraints is increased, then so does the restrictiveness of the feasible-only search space, thereby reducing the algorithm's ability to make reductions in the number of soft constraint violations. Further results of these experiments are also presented in fig. 7.6. Here we can observe that differing p-values do not seem to have any real effect on the average cost of the initial feasible solutions, but do, however, have a large bearing on the amount that the SA algorithm is then able to reduce this cost over time.

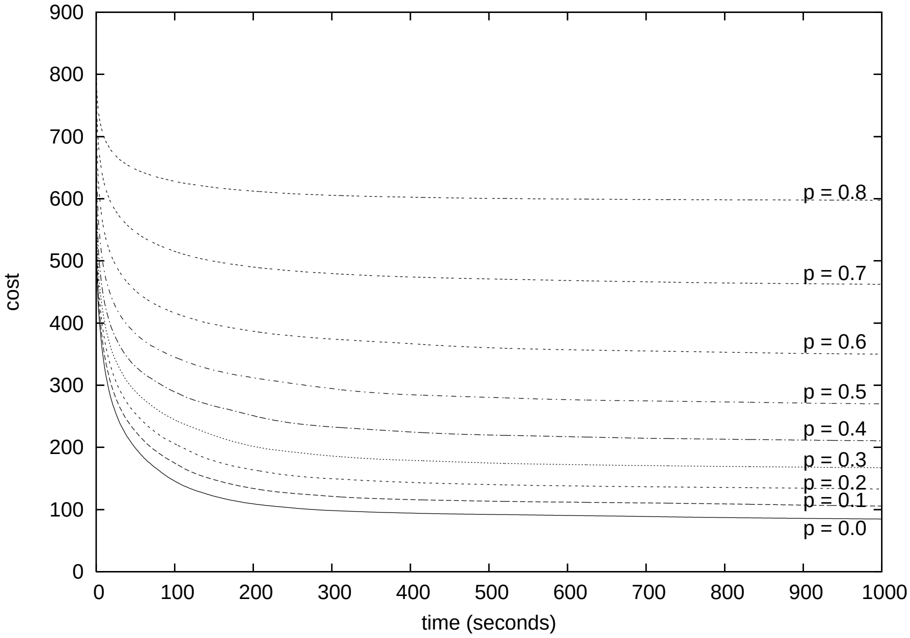


Fig. 7.5: Demonstrating the effects that various *p*-values have on the simulated annealing algorithm's ability to reduce the cost-function over time. Each line represents, at each second, the cost of the best solution found so far averaged across 10 runs on each of the 100 instances (i.e. 1000 runs). Note that the time taken to find an initial feasible solution for this algorithm is not included in the graph; instead (time = 0) may be considered the point at which a feasible solution was first obtained. Note also that every instance considered in these experiments is known to have an optimal cost of zero.

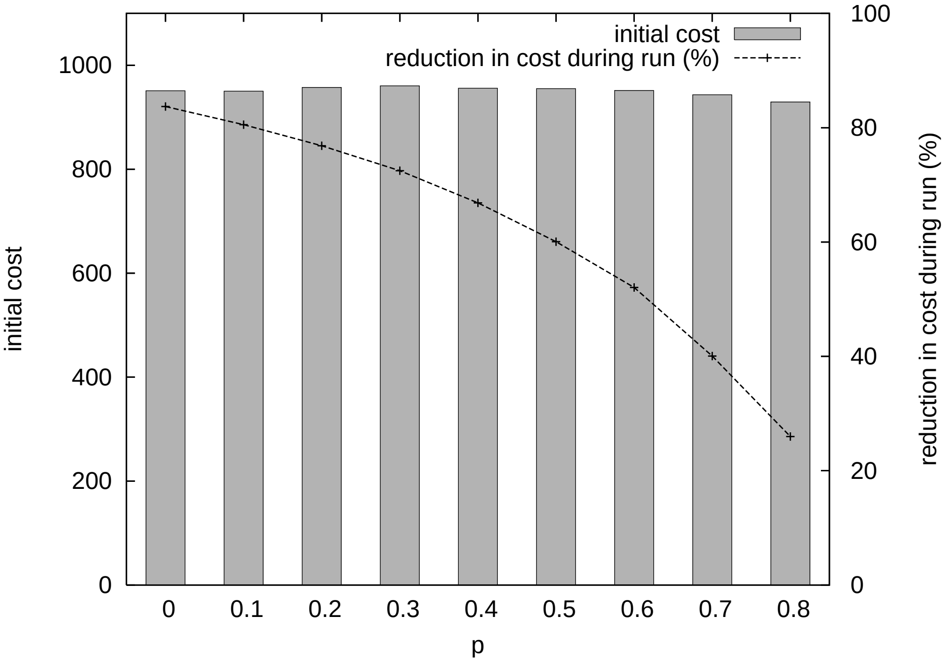


Fig. 7.6: Comparison of (a) the costs of the initial feasible solutions, and (b) the percentage reduction in cost achieved during the run, for various *p*-values. All points are averaged across 1000 runs.

From these experiments, it is also interesting to note that we do not seem to witness the typical "easy-hard-easy"-style phase transition as *p* is increased from 0.0 to 1.0. (When considering solvable instances, "easy-hard-easy" phase transitions have been previously noted in various Operations Research problems such as constraint satisfaction problems [101], graph colouring problems [109], exam timetabling problems [93], and sudoku puzzles [75].) The lack of a phase transition in this case may well be due to the fact that the underlying search mechanism for both the heuristic search and simulated annealing algorithms is the operation of randomly moving events around the timetable whilst not breaking any of the problem's hard constraints. Of course, for higher *p*-values the extra hard constraints that are present in a problem will usually mean that a lower proportion of moves will actually preserve timetable feasibility. This, in turn, will cause the overall search capabilities of both algorithms to drop, resulting in the patterns that we see in the figures. Note that this feature also explains why we were unable to find feasible solutions for instances where $p > 0.8$ in our experiments, despite the fact that solutions to these instances were definitely known to exist.

7.4 Adding further "Real-World" Constraints

In this chapter we have discussed modifications to our two-stage algorithmic approach that allow us to cope with the addition of extra types of hard constraint. In particular we have looked in detail at one of these – the event-unavailability constraint – and have demonstrated through experiments how this constraint can affect our ability to (a) find feasibility, and (b) reduce the number of soft constraint violations when navigating through the resultant feasible-only search space. We have thus made some movements from our original "simplified" university course timetabling model towards problems that are perhaps more applicable to real world scenarios. However, in addition to these constraints, it should be noted that in many practical situations there might also be yet further types of constraints whose satisfaction a particular university might also see as mandatory. As we will see presently, there are also some other possible features of real-world timetabling problems that we have not yet considered. In this final section we will discuss some of these, and will also make some suggestions as to how we might extend our chosen approaches in order to incorporate them.

The first types of constraint that we may wish to consider are those that are concerned with the *relative positioning* of events within the timetable. The following three examples are typical and, rather like the event and room-unavailability constraints considered earlier in this chapter, these also require the timeslots to be explicitly labelled:

(1) **Inter-site travel times**: in some cases, a university might be split across a number of campuses, and students and staff may require some commuting-time in order to travel

from one site to another. Thus, if two events *i* and *j* have common students, but need to take place in different sites, then the constraint "if event *i* is scheduled to occur in timeslot *x*, then event *j* cannot occur in timeslot *x* + 1 if this timeslot is on the same day" might be specified.

(2) **Providing a Lunch-break:** many universities will also want to ensure that all staff and students have the opportunity to eat lunch. Thus constraints such as the following might be imposed: "if a student is attending an event in a 12:00pm timeslot, then he-or-she must not be required to attend an event in a 1:00pm timeslot on the same day, and vice-versa".

(3) **Relative Timing of Events:** universities may also wish to impose other types of constraint on their timetabling problem such as "events *i* and *j* must be assigned to the same/different timeslots", "event *i* must occur before/after event *j*", "events *i* and *j* must take place on the same/different days", and so on.

Note that because these sorts of constraints are concerned with the relative positions of events in the timetable, they are in fact very similar to soft constraints SC_2 and SC_3 that we met in Chapters 3 and 6. One strategy of coping with these might therefore be to simply treat them as additional soft constraints, and try to eliminate as many of them as possible using an algorithm such as our simulated annealing approach (Section 6.2). If, on the other hand, such a strategy is unacceptable, and instead a satisfaction of these constraints is mandatory from the outset, then one approach could be to simply use our basic BUILD-COMPLETE-TIMETABLE procedure as before, and then simply change the criteria for considering which alterations to the timetable *tt* are acceptable. It is also likely that in the presence of these sorts of constraints, other sorts of constructive heuristics might prove useful. For example when selecting a suitable place in which to insert an event *i*, if there is an existing constraint stating that "*i* should appear before events *j*, *k* and *l*" then it might be a good idea to favour the choice of timeslot towards one that occurs earlier in the week. As a second example, if two events were specified as needing to be scheduled on the same day, then when choosing a timeslot for the first of these, common-sense dictates that it may well be useful to show preference towards those days which are also suitable for the second event as well.

Another timetabling feature that has not yet been considered in our algorithms is the occurrence of double-events – that is, events that take place over two (or in some cases more than two) consecutive timeslots on the same day, in the same room. In order to insert a double event into our timetable matrix, such an event obviously needs to occupy two adjacent cells on the same row on the same day. However, these sorts of events could introduce difficulties when shuffling events around the timetable (for instance when we are applying HEURISTIC-SEARCH or our SA algorithm), because in order to be able to move a double-event, an appropriate space in the timetable would also need to be made present. If the timetable were relatively empty perhaps this wouldn't cause too many problems. However, it is likely that in some cases, trying to move double events using our existing types of neighbourhood operators could be quite difficult, and it may be the case that new, more appropriate neighbourhood operators might be needed in order to facilitate this task in a manner that is satisfactory. It is also possible that other constructive heuristics could be defined for dealing with double events as well.

Finally, another possible feature of real-world timetabling problems not yet considered is that some events may not actually require a room (they may take place outdoors, involve trips to off-site locations, and so on). In its current form, our algorithm is not suitable for this feature, because each place (i.e. cell) in the timetable matrix defines an explicit (timeslot × room) pair. One way that we could deal with such a constraint, however, might be to introduce a number of *dummy-rooms* into each timeslot. In effect, each timeslot could have an unlimited number of these dummy-rooms, and any event not requiring a room could then be feasibly assigned to one of these providing, of course, that the remaining hard constraints were not violated in the process. Obviously, if we were to use dummy-rooms, then a restriction would also have to be imposed in order to prevent events that *did* require a room from being inserted into these dummy-rooms, and similarly, we would also have to ensure that normal rooms were not made available to those events that *did not* require rooms.

Note that we are yet to have carried out any of the above suggestions in our research and so, for now, they remain just that: suggestions. Future research will, of course, serve to indicate whether they are actually suitable in a practical sense.

8: Conclusions and Future Research

8.1 Conclusions

In this thesis we have examined various algorithms that constitute parts of the two-stage approach for the university course timetabling problem. We now summarise the main conclusions that can be drawn from this work:

- In Chapter 2 of this thesis we have proposed that metaheuristic-based techniques for timetabling can be classified into three main groups: (1) one-stage optimisation algorithms; (2) two-stage optimisation algorithms; or (3) algorithms that allow relaxations of some feature of the problem. We have concerned ourselves primarily with second of these, and in our studies of the UCTP-version used for the International Timetabling Competition we have seen that this approach is able to work well with a number of benchmark problem instances.
- Our constructive heuristics used for determining the order and locations in which events are to be inserted into the timetable (given in Table 4.1) have shown to be effective with many different instances. We have seen in Chapter 4, for example, that for the twenty competition problem instances, these rules have been able to determine an event-ordering and place-selection strategy that produces fully feasible timetables in nearly all cases, without any need for additional search. To our knowledge, no other set of constructive heuristics proposed for this particular set of problem instances have been able to achieve this. Additionally, in Chapter 7, we have also demonstrated the robustness of these heuristics, as they have been able to cope quite effectively when additional hard constraints have been added to the UCTP (i.e. the “unavailability constraints”), without any extra modifications actually having to be made.

- We have noted that some timetabling problems, in particular those that do not involve having to individually label each of the timeslots, can be considered a type of grouping problem, and can therefore be addressed using Grouping Genetic Algorithms (GGAs). During our analysis of our own GGA for the UCTP, we have introduced a way of measuring diversity and distances between individuals for the general grouping representation. We have also seen how the performance of this algorithm might be improved: first via the use of specialist, fine-grained fitness functions and, also in some cases, by supplementing the GGA with an extra search operator. Additionally, although we have seen that in some cases the use of the standard GGA recombination can enhance the performance of the algorithm, we have also demonstrated conditions whereby the operator might actually start to do more harm than good. Specifically, these conditions are met when there is high population diversity and when the group-sizes are large in size, and we have offered evidence as to why this is so. We have also noted the fact that when the groups *are* larger, the chromosomes, in turn, become proportionally shorter, which might also place limitations on the standard GGA operators in some cases.

- Even though we have discovered ways in which the GGA's performance *can* be improved, in the majority of our experiments with the UCTP we have seen that it has nearly always been outperformed by our much more straightforward Iterated Heuristic Search (IHS) algorithm, which does not make use of a population, selection pressure, or the grouping recombination operator. In particular, the superior performance of the IHS algorithm is most noticeable when dealing with our large UCTP instances where, according to the arguments that we have presented, the GGA operators show the least potential for aiding the search in general. Similar findings to these have also been shown in our experiments with equipartite graph colouring problems in Chapter 5, backing-up our arguments further.

- In Chapter 6 we have presented two algorithms for the second stage of the two-stage timetabling strategy. The first of these – our specialised evolutionary algorithm – has not proved as successful as we had first hoped, mainly due to the difficulties that are encountered when trying to propagate useful and meaningful building-blocks in the population whilst always ensuring that feasibility is maintained. Indeed, our experiments have indicated that the only genetic operator that seems to do anything of consequence in this case seems to be the (blind) mutation operator; however, on its own this operator does not appear to be powerful enough to produce results that are comparable with other algorithms of this type. Meanwhile, the second of our algorithms – our SA-based approach – has proved to be more successful, despite that fact that the majority of its search is conducted using a neighbourhood operator (N_2) that is equivalent to the EA's mutation operator. This observation, together with the results presented in other works [2, 11, 23, 34, 69, 97] suggests that algorithms for the UCTP that are based on some form of neighbourhood-search are perhaps more naturally suited for the second stage of the two-stage approach than algorithms that are based on evolutionary search.

- In Chapter 6 we have also demonstrated how we can improve upon the results returned by the SA algorithm (within the imposed time limits) by treating some soft constraints differently to others. In our case this involves forcing all of the events into just forty of the available forty-five timeslots so that we are then able to remove the five "end-of-day" timeslots from the search altogether. For the twenty competition instances we have seen that such a strategy has allowed performance to improve overall, although we have noted that by doing this, we are also adding further restrictions to the search space, possibly making movements through the search space more difficult in some cases. This latter phenomenon has also been demonstrated in the experiments of Section 7.3. Here, we have shown that as more hard constraints are added to a particular timetabling problem (in this case event-unavailability constraints), this eventually leads to a situation where the movement in the search space is so restricted that the SA algorithm is not able to make satisfactory improvements to the cost function in reasonable amounts of time. Indeed, the fact that we do not witness an "easy-hard-easy" phase transition here (which is what is usually seen when dealing with solvable instances over varying levels of constrainedness) seems to suggest that other sorts of timetabling strategy might be more suitable in these types of situation.

- Finally, in Chapter 7 we have also demonstrated how our IHS algorithm might be modified in order to cope with additional hard constraints such as unavailability constraints. These modifications require us to explicitly label each of the timeslots, thus rendering our strategy of opening-up new timeslots in order to cope with unplaceable events as unsuitable. Our resultant algorithm, which we have called BUILD-COMPLETE-TIMETABLE, has shown to work well on a number of artificially generated instances of various levels of constrainedness. We have also seen that one of the sub-functions of this procedure EXTRACT-SOME-EVENTS, which is used in order to help reinvigorate the search from time-to-time, can improve algorithm performance if it is able to bias its choices of which events to extract towards the less "troublesome" ones, such as those with low conflicts-degrees. However, it should be noted that these differences are only very slight, and indeed, our experiences with this algorithm lead us to believe that it is the HEURISTIC-SEARCH operator that tends to do most of the work in these cases.

As we have seen, in this thesis we have chosen to base our studies on a standardised benchmark timetabling problem-version. This, we believe, has been useful as it has allowed us to avoid some of the negative issues often caused by the idiosyncratic nature of timetabling, and has also provided a means by which we are able to compare our results against other peoples' in a meaningful way. However, it is worth bearing in mind that whilst the use of these sorts of benchmark instances may facilitate the analysis and comparison of algorithms, they do not necessarily allow insight into how these algorithms might fare with other kinds of problem instances – including those from the real-world. It is also worth remembering that while the requirements of this particular UCTP (which state that a timetable with any hard constraint violations is essentially worthless) seem to make the two-stage method the most fitting one, in problems where the requirements are different to these, other timetabling algorithms might show to be more suitable in some cases.

In conclusion, when designing algorithms for timetabling, it is always worth considering that in the real world many different sorts of constraints, problem instances, user-requirements and political factors might be encountered. The idiosyncratic nature of real-world timetabling seems to indicate an advantage to those algorithms that are robust with respect to problem-class changes or to those that can easily be adapted to take account of the needs of particular institutions.

8.2 Discussion and Suggestions for Future Research

Finally, we round off this thesis by making some other general comments arising from this work and also offer some suggestions for future research.

As we have seen, in this thesis we have spent much of our time examining algorithms that constitute part of the two-stage timetabling approach. One worthwhile future endeavour concerning this approach might be to investigate the actual importance of ensuring that a completely feasible timetable is always gained in stage-one. Will an algorithm specialising in the elimination of soft constraints always perform better when presented with a fully feasible timetable? Or will an almost-feasible timetable bring similar results? On a similar theme, it might also be instructive to see if we can identify some features of feasible (or near-feasible) timetables that will give us some indication of how easy it will then be to satisfy the soft constraints. Such studies could well provide us with deeper insights about the two-stage timetabling approach in general.

The scaling-up issues that we have noted when applying the GGA to the UCTP (Chapter 4) also seem to present us with a number of future research-issues. For example, from a practical point-of-view it is not uncommon in universities to have a few thousand or more events that need to be scheduled into a limited number of timeslots, thus presenting some unfavourable practical implications to any timetabling algorithm using a grouping theme. However, a worthwhile future endeavour could be to investigate how complete timetabling problems might be broken up into smaller sub-problems. For example, in [93] it is noted by Ross *et al.* that real-world timetabling problems are often quite *clumped* in their make up: a computing department, for example, might have a set of events that forms a distinct clump largely separated from the events in, say, the psychology department. These departments could have few or no common students, may use different sets of rooms or might even be situated in separate campuses altogether. In these cases, the timetabling problems of these departments may have little bearing on each other and might even be solved independently (see fig. 8.1). As we saw in our literature review in Chapter 2, interesting ideas on the subject of dealing with large problem instances have also been proposed by Burke and Newall [26], where the authors use graph colouring-type heuristics to first break up large sets of events into a number of smaller sub-sets, and then use a memetic algorithm to try and solve each of these in turn.

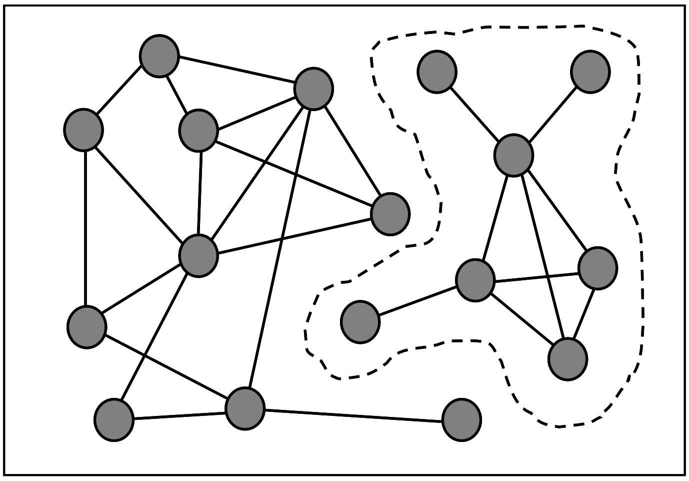


Fig. 8.1: An example of clumping: no vertex in the left sub-graph conflicts with a vertex in the right sub-graph. These two graphs might therefore be coloured separately from one another.

We may also see some general improvements to both the GGA and IHS algorithms for timetabling by making some modifications to the solution construction processes, described in Section 4.2.1. For example, given a list of unplaceable events U, the current function Insert-Events opens up $\lceil |U|/m \rceil$ additional timeslots. As we have noted, this defines a lower bound as to the number of timeslots that are needed to house all of the events in *U*. However, by opening this amount there is no guarantee that additional timeslots will not also need to be opened later on. Indeed, calculating the actual number of timeslots needed for the events in *U* is the same as the NP-hard problem of calculating the chromatic number in graph colouring. On the other hand, opening too few timeslots at this stage (which is what this calculation could do) will also be disadvantageous because it means the algorithm will have to continue to open timeslots later on, adding further cost to the overall process. However, some simple reasoning with respect to problem structure might give us further information. For example, if there are, say, x events in *U* that can only be put into the one same room then it is obvious that at least x extra timeslots will need to be opened for them.

More generally, the observations made in Chapters 4 and 5 regarding group-size also hold implications for applications of the GGA approach in other problem domains. For example, a GGA has been shown to be very successful with bin packing in the work of Falkenauer [55, 58]. However in this work, all of the problem instances that were used in testing were made up of very small groups (generally there were only three items per bin). Thus, according to our arguments, these groups would have been easily propagated during evolution, and would therefore have allowed the GGA operators to perform relatively effectively. However it would, of course, be very interesting to investigate how these same operators also perform with bin packing problem instances that allow larger numbers of items to be placed into each bin. It would also be instructive to see how the GGA approach copes with various other grouping problems that might feature "large" groups such as large exam timetabling problems and graph colouring problems with low chromatic numbers (some results concerning this latter problem can be seen in the work of Eiben *et al.* [50] where a GGA is applied to various graph colouring problems in which the chromatic number $\chi = 3$). Of course, it would also be interesting to see how algorithms based on our IHS approach also perform with these sorts of problems as well.

Pursuing this theme, it would also be helpful to investigate ways in which we might get around the problems that are inherent in the GGA operators. For example, we might be able to discourage some of the destructive behaviour of recombination by imposing some sort of "breeding-rule" on the population such as "parents must have a distance (see Chapter 4, equation (4.3)) between them that is less than 0.5 in order for them to be allowed to combine to form offspring". In the future we might also be able to create new ways in which the building-blocks of these problems might be propagated without these underlying limitations.

Considering now the second stage of the two-stage timetabling approach, there are also various ways in which our evolutionary and simulated annealing algorithms (Chapter

6) might be improved. The EA, for example, might improve if it were to be supplemented with a local-search routine and/or a smart mutation operator. Perhaps other sorts of feasibility-preserving recombination operator might also be suggested in the future that are more conducive to the search than those that were suggested here. Meanwhile, we might also be able to improve the performance of the simulated annealing-based algorithm by making use of some other sorts of cooling schemes (see, for example, the work of Abramson *et al.* [10]) or by incorporating some further specialised neighbourhood operators such as the Kempe and *S*-chain operators used by Thompson and Dowsland in [107].

Another important avenue of future research might be to investigate how the various algorithms described here are also able to cope with real-world problem instances taken from universities and so forth. As we have seen, all our experimental work in this thesis has been conducted using problem instances that have been artificially generated and, although it was intended by the designers of the UCTP instance-generator that these would reflect real-world problems to some degree, it is worth considering that various common and real-world features may not have actually been included in practice. Such an endeavour may involve having to carry out some of modifications that we suggested in Section 7.4, and possibly others although, of course, all of these matters are currently pending further research.

Finally, due to the fact that we were obliged to make our own problem instances for our experiments in Chapter 4, we have been unable to provide comparisons of these algorithms with other approaches. However, these instances can be found on the web at http://www.emergentcomputing.org/timetabling/harderinstances and we invite any other researchers interested in designing algorithms for this problem to download them for use in their own experiments.

Bibliography

[1] http://www.or.ms.unimelb.edu.au/timetabling/atdata/carterea.tar (The Carter Instances for Exam Timetabling)

[2] http://www.idsia.ch/Files/ttcomp2002 (Website of the International Timetabling Competition)

[3] http://www.wiwi.uni-jena.de/Entscheidung/binpp/index.htm (Scholl's Library of Bin Packing Problem Instances)

[4] http://www.cs.ualberta.ca/~joe/Coloring/index.html (Culberson's Graph Colouring Problem Instance Generator)

[5] http://www.research.att.com/~dsj/chtsp/index.html (TSP Instances by DIMACS)

[6] http://www.metaheuristics.org (Website of the Metaheuristics Network)

[7] http://www.emergentcomputing.org/timetabling/harderinstances (Website for the Sixty "Hard" UCTP instances used in Chapter 4)

[8] K. I. Aardel, S. P. M. van Hoesel, A. M. C. A. Koster, C. Mannino, and A. Sassano, "Models and Solution Techniques for the Frequency Assignment Problems," *4OR : Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, vol. 1, pp. 1-40, 2002.

[9] D. Abramson, "Constructing School Timetables using Simulated Annealing: Sequential and Parallel Algorithms," *Management Science*, vol. 37, pp. 98-113, 1991.

[10] D. Abramson, H. Krishnamoorthy, and H. Dang, "Simulated Annealing Cooling Schedules for the School Timetabling Problem," *Asia-Pacific Journal of Operational Research*, vol. 16, pp. 1-22, 1996.

[11] H. Arntzen and A. Løkketangen, "A Tabu Search Heuristic for a University Timetabling Problem," in *Metaheuristics: Progress as Real Problem Solvers*, vol. 32, *Computer Science Interfaces Series*, T Ikabaki, K. Nonobe, and M. Yagiura, Eds. Berlin: Springer-Verlag, 2005, pp. 65-86.

[12] K. J. Batenburg and W. J. Palenstijn, "A New Exam Timetabling Algorithm," Proceedings of BNAIC'03, 2003.

[13] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A Racing Algorithm for Configuring Metaheuristics," presented at The Genetic and Evolutionary Computation Conference (GECCO) 2002, New York, 2002.

[14] P. Boizumault, Y. Delon, and L. Peridy, "Logic Programming for Examination Timetabling," *Logic Program.*, vol. 26, pp. 217-233, 1996.

[15] D. Brelaz, Y. Nicolier, and D. d. Werra, "Compactness and balancing in scheduling," *Mathematical Methods of Operations Research (ZOR)*, vol. 21, pp. 63-73, 1977.

[16] D. Brelaz, "New methods to color the vertices of a graph," *Commun. ACM*, vol. 22, pp. 251-256, 1979.

[17] E. Brown and R. Sumichrast, "A Grouping Genetic Algorithm for the Cell Formation Problem," *International Journal of Production Research*, vol. 39, pp. 3651-3669, 2001.

[18] E. Burke, D. Elliman, and R. Weare, "The Automation of the Timetabling Process in Higher Education," *Journal of Education Technology Systems*, vol. 23, pp. 257-266, 1995.

[19] E. Burke, D. Elliman, and R. Weare, "A Hybrid Genetic Algorithm for Highly Constrained Timetabling Problems.," presented at Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), 1995.

[20] E. Burke, D. Elliman, and R. Weare, "Specialised Recombinative Operators for Timetabling Problems," in *The Artificial Intelligence and Simulated Behaviour Workshop on Evolutionary Computing*, vol. 993, *Lecture Notes in Computer Science*. Berlin: Springer-Verlag, 1995, pp. 75-85.

[21] E. Burke, Y. Bykov, and M. Petrovic, "A Multicriteria approach to Examination Timetabling," in *Practice and Theory of Automated Timetabling (PATAT) III*, vol. 2070, *Lecture Notes in Computer Science*, E. Burke and E. Erben, Eds. Berlin: Springer-Verlag, 2001, pp. 118-131.

[22] E. Burke and M. Petrovic, "Recent Research Directions in Automated Timetabling," *European Journal of Operational Research*, vol. 140, pp. 266-280, 2002.

[23] E. Burke, Y. Bykov, J. P. Newall, and S. Petrovic, "A Time-Defined Approach to Course Timetabling," *Yugoslav Journal of Operations Research (YUJOR)*, vol. 13, pp. 139-151, 2003.

[24] E. K. Burke, D. G. Elliman, P. H. Ford, and R. Weare, "Examination Timetabling in British Universities: A Survey," in *Practice and Theory of Automated Timetabling*

(PATAT) I, vol. 1153, *Lecture Notes in Computer Science*, E. Burke and P. Ross, Eds. Berlin: Springer-Verlag, 1996, pp. 76-92.

[25] E. K. Burke, J. P. Newall, and R. F. Weare, "A Memetic Algorithm for University Exam Timetabling," in *Practice and Theory of Automated Timetabling (PATAT) I*, vol. 1153, *Lecture Notes in Computer Science*, E. Burke and P. Ross, Eds. Berlin: Springer-Verlag, 1996, pp. 241 - 250.

[26] E. K. Burke and J. P. Newall, "A Multi-Stage Evolutionary Algorithm for the Timetable Problem," *IEEE Transactions on Evolutionary Computation*, vol. 3, pp. 63-74, 1999.

[27] M. P. Carrasco and M. V. Pato, "A Multiobjective Genetic Algorithm for the Class/Teacher Timetabling Problem," in *Practice and Theory of Automated Timetabling (PATAT) III*, vol. 2079, *Lecture Notes in Computer Science*, E. Burke and W. Erben, Eds. Berlin: Springer-Verlag, 2001, pp. 3-17.

[28] M. Carter, "A Survey of Practical Applications of Examination Timetabling Algorithms," *Operations Research*, vol. 34, pp. 193-202, 1986.

[29] M. Carter, "A Langarian Relaxation Approach to the Classroom Assignment Problem," *INFOR*, vol. 27, pp. 230-246, 1986.

[30] M. Carter and G. Laporte, "Recent Developments in Practical Examination Timetabling," in *Practice and Theory of Automated Timetabling (PATAT) I*, vol. 1153, E. Burke and P. Ross, Eds. Berlin: Springer-Verlag, 1996, pp. 3-21.

[31] M. Carter, G. Laporte, and S. Y. Lee, "Examination Timetabling: Algorithmic Strategies and Applications," *Journal of the Operational Research Society*, vol. 47, pp. 373-383, 1996.

[32] M. Carter and G. Laporte, "Recent Developments in Practical Course Timetabling," in *Practice and Theory of Automated Timetabling (PATAT) II*, vol. 1408, *Lecture Notes in Computer Science*, E. Burke and M. Carter, Eds. Berlin: Springer-Verlag, 1998, pp. 3-19.

[33] S. Casey and J. Thompson, "GRASPing the Examination Scheduling Problem," in *Practice and Theory of Automated Timetabling (PATAT) IV*, vol. 2740, *Lecture Notes in Computer Science*, E. Burke and P. De Causmaecker, Eds. Berlin: Springer-Verlag, 2002, pp. 233-244.

[34] M. Chiarandini, K. Socha, M. Birattari, and O. Rossi-Doria, "An Effective Hybrid Approach for the University Course Timetabling Problem," *Technical Report AIDA-2003-05, FG Intellektik, FB Informatik, TU Darmstadt, Germany*, 2003.

[35] A. Colorni, M. Dorigo, and V. Maniezzo, "Genetic Algorithms And Highly Constrained Problems: The Time-Table Case," in *Parallel Problem Solving from Nature (PPSN) I*, vol. 496, *Lecture Notes in Computer Science*, H.-P. Schwefel and R. Manner, Eds. Berlin: Springer-Verlag, 1991, pp. 55-59.

[36] A. Colorni, M. Dorigo, and V. Maniezzo, "A genetic algorithm to solve the timetable problem," *Technical Report 90-060 revised, Politecnico di Milano, Italy 1992.*, 1992.

[37] A. Colorni, M. Dorigo, and V. Maniezzo, "Metaheuristics for high-school timetabling," *Computational Optimization and Applications*, vol. 9, pp. 277 - 298, 1997.

[38] T. Cooper and J. Kingston, "The Complexity of Timetable Construction Problems," in *Practice and Theory of Automated Timetabling (PATAT) I*, vol. 1153, *Lecture Notes in Computer Science*, E. Burke and P. Ross, Eds. Berlin: Springer-Verlag, 1996, pp. 283-295.

[39] D. Corne, P. Ross, and H. Fang, "Evolving Timetables," in *The Practical Handbook of Genetic Algorithms*, vol. 1, L. C. Chambers, Ed.: CRC Press, 1995, pp. 219-276.

[40] D. Costa, "A tabu search algorithm for computing an operational timetable," *European Journal of Operational Research*, vol. 79, pp. 98-110, 1994.

[41] P. Cote, T. Wong, and R. Sabourin, "Application of a Hybrid Multi-Objective Evolutionary Algorithm to the Uncapacitated Exam Proximity Problem," in *Practice and Theory of Automated Timetabling (PATAT) V*, vol. 3616, *Lecture Notes in Computer Science*, E. Burke and M. Trick, Eds. Berlin: Springer-Verlag, 2005, pp. 294-312.

[42] J. Culberson and F. Luo, "Exploring the k-colorable Landscape with Iterated Greedy," in *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, vol. 26, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, D. S. Johnson and M. A. Trick, Eds.: American Mathematical Society, 1996, pp. 245-284.

[43] S. Daskalaki, T. Birbas, and E. Housos, "An integer programming formulation for a case study in university timetabling," *European Journal of Operational Research*, vol. 153, pp. 117-135, 2004.

[44] B. Deris, S. Omatu, H. Ohta, and D. Samat, "University Timetabling by Constraint-based Reasoning: A Case Study," *Journal of Operational Reasearch Society*, vol. 48, pp. 1178-1190, 1997.

[45] L. Di Gaspero and A. Schaerf, "Tabu Search Techniques for Examination Timetabling," in *Practice and Theory of Automated Timetabling (PATAT) III*, vol. 2079, *Lecture Notes in Computer Science*, E. Burke and E. Erben, Eds. Berlin: Springer-Verlag, 2001, pp. 104-117.

[46] L. Di Gaspero and A. Schaerf, "Multi-neighbourhood Local Search with Application to Course Timetabling," in *Practice and Theory of Automated Timetabling (PATAT) IV*, vol. 2740, *Lecture Notes in Computer Science*, E. Burke and P. De Causmaecker, Eds. Berlin: Springer-Verlag, 2002, pp. 263-287.

[47] R. Dorne and J.-K. Hao, "A new genetic local search algorithm for graph coloring," in *Parallel Problem Solving from Nature (PPSN) V*, vol. 1498, *Lecture Notes in Computer Science*, A. Eiben, T. Back, M. Schoenauer, and H. Schwefel, Eds. Berlin: Springer-Verlag, 1998, pp. 745-754.

[48] G. Dueck, "New Optimization Heuristics: The Great Deluge Algorithm and the Record-to-Record Travel," *Journal of Computational Physics*, vol. 104, pp. 86-92, 1993.

[49] A. Eiben and J. Smith, *Introduction to Evolutionary Computation*. Berlin: Springer-Verlag, 2003.

[50] A. E. Eiben, J. K. van der Hauw, and J. I. van Hemert, "Graph Coloring with Adaptive Evolutionary Algorithms," *Journal of Heuristics*, vol. 4, pp. 25-46, 1998.

[51] S. Elmohamed, G. Fox, and P. Coddington, "A Comparison of Annealing Techniques for Academic Course Scheduling," in *Practice and Theory of Automated Timetabling (PATAT) II*, vol. 1408, *Lecture Notes in Computer Science*, E. Burke and M. Carter, Eds. Berlin: Springer-Verlag, 1998, pp. 146-166.

[52] E. Erben, "A Grouping Genetic Algorithm for Graph Colouring and Exam Timetabling," in *Practice and Theory of Automated Timetabling (PATAT) III*, vol. 2079, *Lecture Notes in Computer Science*, E. Burke and W. Erben, Eds. Berlin: Springer-Verlag, 2001, pp. 132-158.

[53] S. Even, A. Itai, and A. Shamir, "On the complexity of Timetable and Multicommodity Flow Problems," *SIAM Journal of Computing*, vol. 5, pp. 691 - 703, 1976.

[54] E. Falkenauer, "A New Representation and Operators for GAs Applied to Grouping Problems," *Evolutionary Computation*, vol. 2, pp. 123 - 144, 1994.

[55] E. Falkenauer, "Setting New Limits in Bin Packing with a grouping GA Using Reduction," *C.R.I.F. Technical Report*, 1994.

[56] E. Falkenauer, "Solving equal piles with the grouping genetic algorithm," in *Proceedings of the 6th International Conference on Genetic Algorithms*, L. J. Eshelman, Ed.: Morgan Kaufmann Inc, 1995, pp. 492-497.

[57] E. Falkenauer, "A hybrid grouping genetic algorithm for bin packing," *Journal of heuristics*, vol. 2, pp. 5 - 30, 1996.

[58] E. Falkenauer, *Genetic Algorithms and Grouping Problems*: John Wiley and Sons, 1998.

[59] E. Falkenauer, "Applying genetic algorithms to real-world problems," in *Evolutionary Algorithms*, vol. 111, *The IMA Volumes of Mathematics and its Applications*, L. Davis, K. De Jong, M. Vose, and L. Whitley, Eds. New York: Springer-Verlag, 1999, pp. 65-88.

[60] P. Galinier and H. J-K., "Hybrid evolutionary algorithms for graph coloring," *Journal of Combinatorial Optimization*, vol. 3, pp. 379 - 397, 1999.

[61] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified NP-complete graph problems " *Theor. Comput. Sci.*, vol. 1, pp. 237-267, 1976.

[62] M. R. Garey and D. S. Johnson, *Computers and Intractability - A guide to NP-completeness*, First ed. San Francisco: W. H. Freeman and Company, 1979.

[63] F. Glover, "Tabu Search: A Tutorial," *Interfaces,*, vol. 20, pp. 74-94, 1990.

[64] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA.: Addison-Wesley, 1989.

[65] H.R. Lourenco, O. Martin, and T. Stuetzle, "Iterated Local Search," in *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, Eds. Norwell, MA: Kluwer Academic Publishers, 2002, pp. 321-353.

[66] A. Hertz, "Tabu search for large scale timetabling problems," *European Journal of Operational Research*, vol. 54, pp. 39-47, 1991.

[67] S. Khuri, T. Walters, and Y. Sugono, "A grouping genetic algorithm for coloring the edges of graphs," in *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*. New York: ACM Press, 2000, pp. 422-427.

[68] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by Simulated Annealing," *Science*, pp. 671-680, 1983.

[69] P. Kostuch, "The University Course Timetabling Problem with a 3-Phase Approach," in *Practice and Theory of Automated Timetabling (PATAT) V*, vol. 3616, *Lecture Notes in Computer Science*, E. Burke and M. Trick, Eds. Berlin: Springer-Verlag, 2005, pp. 109-125.

[70] G. Lajos, "Complete University Modular Timetabling using Constraint Logic Programming," in *Practice and Theory of Automated Timetabling (PATAT) I*, vol. 1153, *Lecture Notes in Computer Science*, E. Burke and P. Ross, Eds. Berlin: Springer-Verlag, 1996, pp. 146-161.

[71] J. D. Landa Silva, E. Burke, and M. Petrovic, "An Introduction to Multiobjective Metaheuristics for Scheduling and Timetabling," in *Metaheuristics for Multiobjective Optimisation,* , vol. 535, *Metaheuristic for Multiobjective Optimisation*, X. Gandibleux, M. Sevaux, K. Sorensen, and V. T'kindt, Eds. Berlin: Springer-Verlag, 2004, pp. 91-129.

[72] Y. Leung, Y. Gao, and Z. Xu, "Degree of Population Diversity - A perspective on Premature Convergence in Genetic Algorithms and Its Marcov Chain Analysis," *IEEE Trans. Neural Networks*, vol. 8(5), pp. 1165-1765, 1997.

[73] J. Levine and F. Ducatelle, "Ant Colony Optimisation and Local Search for Bin Packing and Cutting Stock Problems.," *Journal of the Operational Research Society*, vol. 55(12), pp. 705-716, 2003.

[74] R. Lewis and B. Paechter, "New Crossover Operators for Timetabling with Evolutionary Algorithms," presented at The Fifth International Conference on Recent Advances in Soft Computing RASC2004, Nottingham, England, 2004.

[75] R. Lewis, "Metaheuristics can Solve Sudoku Puzzles," *(Forthcoming) Journal of heuristics*, vol. 13, 2007.

[76] R. Lister, "Annealing Networks and Fractal Lanscapes," presented at IEEE International Conference on Neural Networks, San Francisco, 1993.

[77] S. Martello and P. Toth, "Lower bounds and reduction procedures for the bin packing problem," *Discrete Applied Mathematics*, vol. 28, pp. 59-70, 1990.

[78] C. Mattiussi, M. Waibel, and D. Floreano, "Measures of Diversity for Populations and Distances Between Individuals with Highly Reorganizable Genomes," *Evolutionary Computation*, vol. 12, pp. 495-515, 2004.

[79] F. Melicio and J. Caldeira, "Timetabling Implementation aspects by Simulated Annealing," presented at IEEE Systems Science and Systems Engineering, Beijing. Aceite., 1998.

[80] L. Merlot, N. Boland, B. Hughes, and P. Stuckey, "A Hybrid Algorithm for the Examination Timetabling Problem," in *The Practice and Theory of Automated Timetabling (PATAT) IV* vol. 2740, *Lecture Notes in Computer Science*, E. Burke and P. D. Causmaeker, Eds. Berlin: Springer-Verlag, 2003, pp. 207-231.

[81] N. Mladenovic and P. Hansen, "Variable Neighborhood Search," *Comps. in Opns. Res.*, vol. 24, pp. 1097-1100, 1997.

[82] C. Morgenstern, "Algorithms for General Graph Coloring," in *PhD Thesis, Department of Computer Science*: University of New Mexico, 1989.

[83] W. Morrison and K. de Jong, "Measurement of Population Diversity," presented at Artificial Evolution 2001, 2002.

[84] P. Moscato, "On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms," *Tech. Rep. Caltech Concurrent Computation Program, Report. 826, California Institute of Technology, Pasadena, California, USA*, 1989.

[85] P. Moscato and M. G. Norman, "A 'Memetic' Approach for the Traveling Salesman Problem. Implementation of a Computational Ecology for Combinatorial Optimization on Message-Passing Systems," presented at Parallel Computing and Transputer Applications, 1992.

[86] B. Paechter, R. Rankin, A. Cumming, and T. Fogarty, "Timetabling the Classes of an Entire University with an Evolutionary Algorithm," in *Parallel Problem Solving from Nature (PPSN) V*, vol. 1498, *Lecture Notes in Computer Science*, T. Baeck, A. Eiben, M. Schoenauer, and H. Schwefel, Eds. Berlin: Springer-Verlag, 1998, pp. 865-874.

[87] L. Paquete and C. Fonseca, "A study of examination timetabling with multiobjective evolutionary algorithms," presented at 4th Metaheuristics International Conference (MIC 2001), Porto, 2001.

[88] S. Petrovic and Y. Bykov, "A Multiobjective Optimisation Approach for Exam Timetabling based on Trajectories," in *The Paractice and Theory of Automated Timetabling (PATAT) IV*, vol. 2740, *Lecture Notes in Computer Science*, E. Burke and P. De Causmaecker, Eds. Berlin: Springer-Verlag, 2003, pp. 181-194.

[89] N. J. Radcliffe, "Forma Analysis and Random Respectful Recombination," presented at the fourth International Conference on Genetic Algorithms, San Marco CA, 1991.

[90] B. Rekiek, A. Delchambre, and H. Saleh, "Pickup and Delivery Problems: An application of the Grouping Genetic Algorithm," Universite Libre de Bruxelles, IRIDIA technical report TR/IRIDIA/2003-32. 2003.

[91] J. T. Richardson, M. R. Palmer, G. Liepins, and M. Hilliard, "Some Guidelines for Genetic Algorithms with Penalty Functions.," in *the Third International Conference on Genetic Algorithms*, J. D. Schaffer, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 1989, pp. 191-197.

[92] P. Ross, D. Corne, and H.-L. Fang, "Improving Evolutionary Timetabling with Delta Evaluation and Directed Mutation," in *Parallel Problem Solving from Nature (PPSN) III*, vol. 866, *Lecture Notes in Computer Science*, Y. Davidor, H. Schwefel, and M. Reinhard, Eds. Berlin: Springer-Verlag, 1994, pp. 556-565.

[93] P. Ross, D. Corne, and H. Terashima-Marin, "The Phase-Transition Niche for Evolutionary Algorithms in Timetabling," in *Practice and Theory of Automated Timetabling (PATAT) I*, vol. 1153, *Lecture Notes in Computer Science*, E. Burke and P. Ross, Eds. Berlin: Springer-Verlag, 1996, pp. 309-325.

[94] P. Ross, E. Hart, and D. Corne, "Some Observations about GA-Based Exam Timetabling," in *Practice and Theory of Automated Timetabling (PATAT) II*, vol. 1408, *Lecture Notes in Computer Science*, E. Burke and M. Carter, Eds. Berlin: Springer-Verlag, 1998, pp. 115-129.

[95] P. Ross, E. Hart, and D. Corne, "Genetic Algorithms and Timetabling," in *Advances in Evolutionary Computing: Theory and Applications*, A. Ghosh and K. Tsutsui, Eds.: Springer-Verlag, New York., 2003, pp. 755- 771.

[96] O. Rossi-Doria, J. Knowles, M. Sampels, K. Socha, and B. Paechter, "A Local Search for the Timetabling Problem," presented at Practice And Theory of Automated Timetabling (PATAT) IV, Gent, Belgium, 2002.

[97] O. Rossi-Doria, M. Samples, M. Birattari, M. Chiarandini, J. Knowles, M. Manfrin, M. Mastrolilli, L. Paquete, B. Paechter, and T. Stützle, "A Comparison of the

Performance of Different Metaheuristics on the Timetabling Problem," in *Practice and Theory of Automated Timetabling (PATAT) IV*, vol. 2740, *Lecture Notes in Computer Science*, E. Burke and P. De Causmaecker, Eds. Berlin: Springer-Verlag, 2002, pp. 329-351.

[98] W. Salwach, "Genetic Algorithms in Solving Constraint Satisfaction Problems: The Timetabling Case," *Badania Operacyjne i Decyzje*, 1997.

[99] A. Schaerf, "Tabu Search Techniques for Large High-School Timetabling Problems," in *Proceedings of the Thirteenth National Conference on Artificial Intelligence*. Portland (OR): AAAI Press/ MIT Press, 1996, pp. 363-368.

[100] A. Schaerf, "A Survey of Automated Timetabling," *Artificial Intelligence Review*, vol. 13, pp. 87-127, 1999.

[101] B. Smith, "Phase Transitions and the mushy region in Constraint Satisfaction Problems," in *11th European Conference on Artificial Intelligence*, A. Cohn, Ed.: John Wiley and Sons ltd, 1994, pp. 100-104.

[102] K. Socha, J. Knowles, and M. Samples, "A MAX-MIN Ant System for the University Course Timetabling Problem," in *Proceedings of Ants 2002 - Third International Workshop on Ant Algorithms (Ants'2002)*, vol. 2463, *Lecture Notes in Computer Science*, M. Dorigo, G. Di Caro, and M. Samples, Eds. Berlin: Springer-Verlag, 2002, pp. 1-13.

[103] K. Socha and M. Samples, "Ant Algorithms for the University Course Timetabling Problem with Regard to the State-of-the-Art," in *Evolutionary Computation in Combinatorial Optimization (EvoCOP 2003)*, vol. 2611, *Lecture Notes in Computer Science*. Berlin: Springer-Verlag, 2003, pp. 334-345.

[104] N. Srinivas and K. Deb, "Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms," *Evolutionary Computation*, vol. 2, pp. 221-248, 1994.

[105] G. Tao and Z. Michalewicz, "Inver-over Operator for the TSP," in *Parallel Problem Solving from Nature (PPSN) V*, vol. 1498, *Lecture Notes in Computer Science*, T. Baeck, A. Eiben, M. Schoenauer, and H. Schwefel, Eds. Berlin: Springer-Verlag, 1998, pp. 803-812.

[106] J. Thompson and K. Dowsland, "Variants of Simulated Annealing for the Examination Timetabling Problem," *Annals of Operational Research*, pp. 105 -128, 1996.

[107] J. M. Thompson and K. A. Dowsland, "A Robust Simulated Annealing based Examination Timetabling System," *Computers and Operations Research*, vol. 25, pp. 637-648, 1998.

[108] A. Tripathy, "School Timetabling - A Case in Large Binary Linear Integer Programming " *Managment Science*, vol. 30, pp. 1473-1489, 1984.

[109] J. S. Turner, "Almost all k-colorable Graphs are easy to Color," *Journal of Algorithms*, vol. 9, pp. 63-82, 1988.

[110] P. van Laarhoven and E. Aarts, *Simulated Annealing: Theory and Applications*. Reidel, The Netherlands: Kluwer Academic Publishers, 1987.

[111] G. V. von Lazewski, "Intelligent Structural Operators for the k-way Graph Partitioning Problem," in *Forth International Conference on Genetic Algorithms*, R. K. Belew and L. B. Booker, Eds. San Mateo, CA, USA: Morgan Kaufmann, 1991, pp. 45-52.

[112] G. White and W. Chan, "Towards the Construction of Optimal Examination Schedules," *INFOR*, vol. 17, pp. 219-229, 1979.

[113] X. Yao, "An Overview of Evolutionary Computation," *Chinese Journal of Advanced Software Research*, vol. Allerton Press Inc., New York, NY 10011, pp. 12-29, 1996.

[114] E. Yu and K.-S. Sung, "A Genetic Algorithm for a University Wekly Courses Timetabling Problem," *International Transactions in Operational Research*, vol. 9, pp. 703-717, 2002.

[115] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the Strength Pereto Evolutionary Algorithm for Multiobjective optimization," Gloriastrasse 35, CH-8092 Zurich, Switzerland, Tech. Rep. 103, 2001.