Metaheuristics for University Course Timetabling

Rhydian Lewis[1], Ben Paechter[1], Olivia Rossi-Doria[2]

[1]Centre for Emergent Computing,
Napier University, Edinburgh EH10 5DT, Scotland.
[2]Dipartimento di Matematica Pura ed Applicata,
Univesita' degli Studi di Padova, via G. Belzoni 7, 35131 Padua, Italy.

Summary. In this chapter we consider the NP-complete problem of university course timetabling. We note that it is often difficult to gain a deep understanding of these sorts of problems due to the fact that so many different types of constraints can ultimately be considered for inclusion in any particular application. Consequently we conduct a detailed analysis of a benchmark problem version that is slightly simplified, but also contains many of the features that make these sorts of problems "hard". We review a number of the algorithms that have been proposed for this particular problem, and also present a detailed description and analysis of an example algorithm that we show is able to perform well across a range of benchmark instances.

1 Introduction to Timetabling

Timetables are ubiquitous in many areas of daily life such as work, education, transport, and entertainment: it is, indeed, 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 overly in abundance, the problem of constructing workable and attractive timetables can often be a very challenging one, even for the experienced timetable designer. However, given that these timetables can often have large effects 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.), their construction is also a problem that we will have to face on a fairly regular basis.

1.1 Timetabling at Universities

In this chapter we will be concerning ourselves with the problem of constructing timetables for universities. The generic university-timetabling problem may be summarised as the task of assigning events (lectures, exams, etc.) to a limited set of timeslots, whilst also trying to satisfy some constraints.

Probably the most universally encountered constraint for these problems is the *event-clash* constraint: if one or more persons are required to attend a pair of events, then these events must not be assigned to the same timeslot. However, beyond this simple example, university timetabling problems, in general, are notorious for having a plethora of different problem definitions in which any number of different constraints can be imposed. These constraints can involve factors such as room facilities and capacities, teacher and student preferences, physical distances between venues, the ordering of events, the timetabling policies of the individual institution, plus many more. (Some good surveys on constraints can be found in [9, 12, 22, 41].) Some problem definitions may even directly oppose others in their criteria for what makes a good timetable. For example, some might specify that we want timetables where each member of staff is given one day free of teaching per week (e.g. [20]). Others, however, might discourage or disallow this. Obviously, which constraints are imposed, as well as the relative importance that each one has, depends very much on each individual university's preference. This, on the whole, makes it difficult to formulate meaningful and universal generalisations about timetabling in general.

One important feature that we do know, however, is that timetable construction is NP-complete in almost all variants [46]. Cooper and Kingston [21], 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 transformations from various well-known NP-complete problems (such as graph-colouring, bin-packing, and three-dimensional matching) to a number of different timetabling problem variants. Even, Itai, and Shamir [28] have also shown a method of transforming the NP-complete 3-SAT problem into a timetabling problem.

Of course, this general NP-completeness implies that whether we will be able to obtain anything that might be considered a workable timetable in any sort of reasonable time 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 there is an abundance of rooms or some extra teaching staff. In these cases, maybe there are many good 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 maybe only a small number of workable timetables – or perhaps none – may exist. (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" problems are encountered, there is an implicit need for powerful and robust heuristic search methods. Some excellent surveys of these can be found in [9, 11, 14, 15, 17, 41].

When looking at the timetabling problem from an operations research point-of-view, the constraints that are imposed on a particular problem tend usually to be classified as either *hard* or *soft*.[1] Hard constraints have a higher priority than soft, and are usually mandatory in their satisfaction. Indeed, timetables are usually only considered feasible if *all* of the hard constraints have been satisfied. Soft constraints, on the other hand, are those that we want to obey only if possible, and more often than not will describe what it is for a timetable to be *good* (with regards to the timetabling policies of the university, as well as the experiences of the people who will have to use it). As can be imagined, most real-world timetabling problems will have their own particular idiosyncrasies, and while this has resulted in a rich abundance of different timetabling algorithms, it also makes it very difficult to compare and contrast them. However, as Schaerf [46] points out, this situation is perfectly understandable given that many people will often be more interested in solving the timetabling problems of their own university rather than spending time comparing results with others.

It is widely accepted, however, that timetabling problems within universities can be loosely 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 similar characteristics (both are usually likely to require a satisfaction of the event-clash constraint, for example), but one common and generally acknowl-

[1] A good review of the many different sorts of constraints that can be encountered in real-world timetabling problems can be found in [22].

edged difference is that in exam timetabling, multiple events can be scheduled into 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 may sometimes allow some flexibility in the number of timeslots being used (see for example [8, 10, 23, 27]).

1.2 Chapter Overview

In this chapter we will primarily concern ourselves with university course timetabling. We will, however, also refer to exam timetabling research when and where it is useful to do so. (Readers more interested in the latter are invited to consult some good texts presented by Burke *et al.* [12, 13], Thompson and Dowsland [49], Cowling *et al.* [24], and Carter [14-16].)

The remainder of this chapter is set out as follows: in the next section we will review some of the most common forms of timetabling algorithm apparent in the literature, and will discuss some possible advantages and disadvantages of each. Next, in section 3, we will give a definition and history of the particular timetabling problem that we will be studying here, and will include a survey of some of the best works proposed for it. In section 4, we will then describe an example algorithm for this problem and will provide a short experimental analysis. Finally, section 5 will conclude the chapter.

2 Dealing with Constraints

When attempting to design an algorithm for university timetabling, one of the most important issues that needs to be addressed is the question of how the algorithm proposes to deal effectively with both the hard constraints and the soft constraints. A survey of the literature indicates that most metaheuristic timetabling algorithms (of which there are many) will generally fall into one of three categories:

1. **One-Stage Optimisation Algorithms**: where a satisfaction of both the hard and soft constraints is attempted simultaneously (e.g. [20, 22, 26, 45]).

2. **Two-Stage Optimisation Algorithms**: where a satisfaction of the soft constraints is only attempted once a feasible timetable has been found (e.g. [5, 18, 19, 31, 32, 49]).

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 (e.g. [8, 10, 27, 37]).

Looking at category (1) first, algorithms of this type generally allow the violation of both hard and soft constraints within the timetable, and the aim is to then search for a timetable that has an adequate satisfaction of both. Typically, the algorithm will attempt this by using some sort of weighted sum function, with violations of the hard constraint usually being given much higher weightings than the soft constraints. For example, in [22] Corne, Ross, and Fang use the following evaluation function: given a problem with k types of constraint, where the penalty weighting associated with constraint i is w_i, and where $v_i(tt)$ represents the number of constraint violations of type i in a timetable tt, quality can be calculated using the formula:

$$f(tt) = 1 / \left(1 + \sum_{i=1}^{k} w_i v_i(tt)\right) \quad (1)$$

In [22], the authors use this evaluation method in conjunction with an evolutionary algorithm, although, one large advantage of this method is that it can, of course, be used with any reasonable optimisation technique (see [26] and [45], for example). Another immediate advantage of this approach is its flexibility: any sensible constraint can be incorporated into the algorithm provided that an appropriate penalty weighting is stipulated in advance (thus indicating its relative importance compared to others).

However, this sort of approach also has some disadvantages. Some authors (e.g. Richardson *et al.* [38]) have argued that this sort of evaluation method 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 in the evaluation function will often critically influence the algorithm's navigation of the search space (and therefore its timing implications and solution quality), there does not seem to be any obvious methodology for choosing the best ones. Some authors (e.g. Salwach [44]) have also noted that a weighted sum function can be problematic, because it can cause a discontinuous fitness landscape, where small changes to a candidate solution can actually result in overly large changes to its fitness.

With regards to timetabling problems, however, it is worth noting that some researchers have tried to circumvent some of these problems by allowing penalty weightings to be altered dynamically during the search. For example, in order to penalise hard constraint violations in his tabu search algorithm for school timetabling, Schaerf [45] defines a weighting value *w*, which 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.

The operational characteristics of two-stage optimisation algorithm for timetabling (category (2)) 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[2].

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 (we no longer need to directly compare feasible and infeasible timetables), meaning that a number of the problems inherent in the use of penalty weightings no longer apply. In practical situations, such a technique might also be very appropriate where finding 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, use of a one-stage optimisation algorithm in this situation could be inappropriate in many cases because, whilst searching for feasibility, the weighted sum evaluation function would always be taking the soft constraints into account. Thus, by making concessions for the soft constraints, the search could suffer the adverse effect of actually being led away from attractive (i.e. 100% feasible) regions of the search space.)

One of the major requirements for the two-stage timetabling algorithm to be effective, however, is that a reasonable amount of movement in the feasible-only search space must be achievable. If the feasible search space of the problem is convex and constitutes a reasonable part of the entire search space, then this may be so. However, if we are presented with a

[2] This could be achieved using neighbourhood operators that always preserve feasibility (c.f. [49]); by using some sort of repair mechanism to convert infeasible individuals into feasible ones (e.g. [32]); or by immediately rejecting any infeasible candidate solutions that crop up during the search. (In evolutionary computation, the latter is sometimes known as the "death penalty" heuristic [36].)

non-convex feasible search space, then searches of this kind could turn out to be extremely inefficient because it might simply be too difficult for the algorithm to explore the search space in any sort of useful way. (In these cases, perhaps a method that allows the search to take "shortcuts" across infeasible parts of the search space might be more promising.)

Lastly, whether this technique will be appropriate in a practical sense also depends largely on the users' requirements. If, for example, we are presented with a problem instance where feasibility is very difficult or seemingly impossible to achieve, then an algorithm of this form will never end up paying any consideration to the soft constraints. In this case, users may 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).

Looking now at category (3), some authors have shown that good timetabling algorithms can also be achieved through the use of more specialised methodologies whereby various constraints of the problem are relaxed in order to try and facilitate better overall searches. For example, in their evolution-based algorithm for exam timetabling, Burke, Elliman, and Weare [8, 10] do not allow the direct violation of any of the problem's hard constraints; instead, they choose to open up new timeslots for events that cannot be feasibly placed into any existing timeslot. The number of timeslots being used by a candidate timetable then forms part of the evaluation criteria. In addition to this, the authors also define an opposing soft constraint that specifies that exams for individual students must be spread out (in order to avoid situations where students are required to sit exams in consecutive timeslots). Because a reasonable satisfaction of this type of constraint will usually rely on there first being an adequate number of timeslots available, the overall aim of the algorithm is to find a suitable compromise between the two objectives.

A second example of this type of approach is provided by Paechter *et al.* in [37]. Here, the authors describe a memetic approach for course timetabling in which an evolutionary algorithm is supplemented by a local-search routine that aims to improve each timetable. In this approach, a constructive scheme is also used and, rather than break any hard constraints, events that cannot be feasibly assigned are left to one side unplaced. Soft constraint violations are also penalised through the use of weightings that can be adjusted by the user during the search.

One interesting aspect of this approach is the authors' use of sequential evaluation: when comparing two candidate timetables, the algorithm deems the one with the least number of unplaced events as the fitter. However, ties are broken by looking at the penalties caused by each of the timetable's soft constraint violations. Thus many of the problems encountered when judging a timetable's quality through a single numerical value alone (as is the case with category (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 a 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 methods.

Concluding this section, it should be clear to the reader that the question of how to deal with both the hard and soft constraints in a timetabling problem is not always easily answerable, yet it is certainly something that we have to effectively address if automated timetabling is to be considered a worthwhile endeavour. As we have noted at various points, the issue of meeting the user's timetabling requirements (whatever these might be) often constitutes an important part in this decision. Indeed, it would seem reasonable to assume that perhaps this is the most important issue, considering that solutions to practical problems will inevitably have to be used by real people. However, it is, of course, also desirable for the algorithm to be fast, reliable and robust whenever possible.

3 The UCTP and the International Timetabling Competition

In the previous two sections, we mentioned that a difficulty often experienced in automated timetabling is that it is not always easy to compare and contrast the performance of different timetabling algorithms. Indeed, many authors often only report results from experiments with their own university's timetabling problem and fail to provide comparisons to others. (In many of these situations we also, of course, have no way of determining if the problem instances used in the experiments were actually "hard" or not, although what actually constitutes a "hard" timetabling instance is still not completely understood). These difficulties in making algorithm comparisons are in contrast to many other problems faced in operations research (such as the travelling salesperson problem and the bin-packing problem) where we often have standardised problem definitions, together with an abundance of different problem instance libraries available for benchmarking algorithms[3].

[3] See, for example, http://www.research.att.com/~dsj/chtsp/index.html or http://www.wiwi.uni-jena.de/Entscheidung/binpp/index.htm for libraries of TSP and bin packing problems respectively.

However, over the past few years a small number of instance sets have become publicly available. In 1996 for example, Carter [16] published a set of exam-timetabling problem instances taken from twelve separate educational establishments from various parts of the world[4]. A number of different studies have now used these in their experimental analyses [8, 16, 23, 48, 49]. More recently, a number of problem instances for course time-tabling have also been made publicly available [1, 2]. It will be these particular collections of problem instances that we will focus our studies upon in this chapter.

3.1 Origins of this Problem Version

The so-called University Course Timetabling Problem (UCTP) was originally used by the Metaheuristics Network[5] – a European Commission funded research project – in 2001-2, but was also subsequently used for the International Timetabling Competition in 2002 [1], of which further details will be given later. The problem, which was formulated by the authors, is closely based on real-world problems, but is also simplified slightly. Although, from the outset, we were not entirely happy about using a simplified problem, we had a number of reasons for doing this. Firstly the problem was intended for research purposes, particularly with regards to analysing what actually happens in algorithms that are designed to solve the problem. (Real problems are often too complicated and messy to allow researchers to properly study these processes.) Secondly, the large number of hard and soft constraints usually found in real-world problems often makes the process of writing code (or updating existing programs to be suitable) a long and arduous process for timetabling researchers. Thirdly, many of the constraints of real-world problems are idiosyncratic and will often only relate to specific institutions, and so their inclusion in a problem will not always be instructive when trying to learn about timetabling in general.

The UCTP therefore offers a compromise: a variety of real world aspects of timetabling are included, yet for ease of scientific investigation, many of the messy fine-details found in practical problems have been removed.

[4] Download at http://www.or.ms.unimelb.edu.au/timetabling/atdata/carterea.tar
[5] http://www.metaheuristics.org/

3.2 UCTP Problem Description

A problem instance for the UCTP consists of a set E of n events to be scheduled into a set of timeslots T and a set of m rooms R, each that has an associated seating capacity. We are also given a set of students S each attending some subset of E. Pairs of events are said to *conflict* when one or more students are required to attend them both. Finally, we are given a set of features[6] F. These are *satisfied* by rooms and *required* by events. In order for a timetable to be *feasible*, every event $e \in E$ must be assigned to a room $r \in R$ and timeslot $t \in T$ (where $|T| \leq 45$, to be interpreted as five days of nine timeslots), such that the following hard constraints are satisfied:

H1. 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);

H2. All of the features required by an event are satisfied by its room, which must also have an adequate seating capacity;

H3. Only one event is put in any room in any timeslot (i.e. no double booking of rooms).

Note that the presence of H1 above makes the task of finding a feasible timetable similar to the well-known NP-hard graph colouring problem. In order to convert one problem to the other, each individual event is considered a node, and edges are then added between any pair of nodes that represent conflicting events. In very basic timetabling problem formulations (e.g. [40]), the task is to then simply colour the graph with as many colours as there are available timeslots. (Indeed, graph colouring heuristics are often used in timetabling algorithms [13, 16, 35, 49]).

However, as we demonstrate in fig. 1, in the case of this UCTP, the presence of H2 and H3 add extra complications because we must now also ensure that for any given timeslot (i.e. colour class) there are adequate and appropriate rooms available. From a pure graph colouring perspective, this means that many feasible colourings might still represent infeasible timetables[7].

[6] In the real world, these features might be things such as audio equipment, computing facilities, wheelchair access, etc.

[7] Note that the presence of the rooming constraints provides us with a lower bound to the underlying graph colouring problem, because a feasible solution can never use less than $\lceil n/m \rceil$ colours (timeslots).

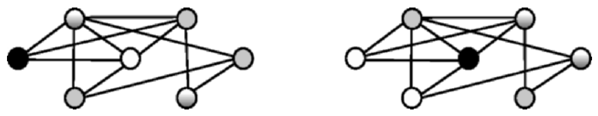


Fig. 1. In this example, both graphs have been coloured optimally. However, in the case of our timetabling problem, if only 2 rooms were available per timeslot then the left graph could never represent a feasible timetable because one of the time-slots would have 3 events assigned to it. The right solution, on the other hand, might represent a feasible timetable, providing that each event can also be granted the rooming features and seating capacity that they require.

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

S1. No student should be required to attend an event in the last timeslot of a day;

S2. No student should sit more than two events in a row;

S3. No student should have a single event in a day.

Note that each soft constraint is slightly different (indeed, this was done deliberately): violations of S1 can be checked with no knowledge of the rest of the timetable, violations of S2 can be checked when building the timetable, and, lastly, violations of S3 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 S1, if a student has a class on the last timeslot of the day, we count this as one penalty point. Naturally, if there are *s* students in this class, we consider this as *s* penalty points. For S2, if one student has three events in a row we give 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.

We consider a timetable to be *perfect* if it is feasible (i.e. has no hard constraint violations) and if it contains no soft constraint violations.

3.3 Initial Work and the International Timetabling Competition

Rossi-Doria *et al.* conducted one of the first studies using this timetabling problem in 2002 [43]. Here, the authors used five different metaheuristic techniques (namely, evolutionary algorithms, ant colony optimisation, iterated local search, simulated annealing, and tabu search) to produce five separate algorithms for the UCTP. In order to facilitate a fair comparison of these algorithms (the main objective of their study), all used the same solution representation and search landscape. In some cases satisfaction of both the hard and soft constraints was attempted simultaneously (in the case of the evolutionary algorithm, for example, a weighted sum function was used to give higher penalties for hard constraint violations). Others, such as the iterated local search and simulated annealing algorithms, used a two-stage approach. Upon completing a comparison of these five metaheuristic algorithms, two interesting conclusions were offered by the authors:

- "The performance of a metaheuristic, 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 minimising the number of soft constraint violations, is a promising direction."

Following this work, the International Timetabling Competition [1] was 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 execution time limit[8]. 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 only available to the organisers) was awarded a prize. The exact criteria for choosing the winner can be found on the competition web site [1].

The twenty problem instances used for the competition consisted of between 200 and 300 students, 350 to 440 events, and 10 or 11 rooms. As usual, the number of timeslots was fixed at 45. Additionally, in 13 of the 20 instances the number of events n was equal to the number of rooms multiplied by 40. This means that, because all instances were ensured to have at least one perfect solution[9], optimal solutions to these instances had

[8] The execution time limit was calculated for a participant's computer by a program that measured various characteristics of that computer during execution. The effectiveness of this benchmarking program was later verified by running the best competition entries on a single standard machine.

[9] In fact, we actually know that there are at least $5! = 120$ perfect timetables, because we note that the soft constraints do not actually span across different days. Thus, we can permute the days of a perfect timetable, and it will still have no soft constraint violations.

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.)

Another important aspect of the competition was the way in which timetables were chosen to be evaluated. The official rules of the competition stated that timetable quality would only be measured by looking at the number of soft constraint violations: if a timetable contained any hard constraint violations, used any extra timeslots, or had any unplaced events, then it would immediately be considered worthless. Participants were then only allowed to submit an entry to the competition if their algorithms could find feasibility on all twenty instances. Given this rule, and also taking into consideration the conclusions of Rossi-Doria *et al.* [43] quoted above, it is perhaps, unsurprising that many of the entrants to this competition therefore elected to use the two-stage timetabling approach mentioned in section 2. Another consequence of the evaluation scheme was that the problem instances were chosen so that feasibility was relatively easy to find.

The competition, which ended in March 2003, eventually saw a total of 21 official entries, plus 3 unofficial entries (the latter 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, some hybrid algorithms, and heuristic construction with backtracking. The winning algorithm was a two-stage, simulated annealing-based algorithm by Philipp Kostuch of Oxford University. Details of this, plus many of the others mentioned above can be found at the official competition web page [1].

3.4 Review of Relevant Research

Since the running of the competition, quite a few good papers have been published regarding this particular timetabling problem. Some of these describe modifications to algorithms that were official competition entries and claim excellent results. Some have gone on to look at other aspects of the problem. In this subsection we now review some of the most notable and relevant works in this problem area.

In [5], Arntzen and Løkketangen describe a two-stage tabu search algorithm for the problem. In the first stage, the algorithm uses a constructive procedure to build an initial feasible timetable, which 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 an assignment might cause. The order in which 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.

Cordeau, Jaumard, and Morales (available at [1]) also use tabu search to try and satisfy the soft constraints in their timetabling algorithm. 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 introducing a partially stochastic parameter α that is then used in the following evaluation function:

$$f(tt) = \alpha h(tt) + s(tt) \tag{2}$$

where $h(tt)$ indicates the number of hard constraint violations in timetable tt, and $s(tt)$ the number of soft constraint violations. During the search, the parameter α helps to control the level of infeasibility in the timetable because if the number of hard constraint violations in tt increases, then α is also increased. Thus, as the number of infeasibilities rises, it also becomes increasingly unlikely that a search space move causing additional infeasibilities will be accepted. The authors claim that such a scheme allows freer movement about the search space.

Socha, Knowles, and Sampels have also suggested ways of applying the ant colony optimisation metaheuristic to this problem. In [47], the authors present two ant-based algorithms – an Ant Colony System and a MAX-MIN system – and provide a qualitative comparison between them. At each step of both algorithms, every ant first constructs a complete assignment of events to timeslots using heuristics and pheromone information, due to previous iterations of the algorithm. Timetables then undergo further improvements via a local search procedure, outlined in [42]. Indeed, the only major differences between the two approaches are in the way that heuristic and pheromone information is interpreted, and in the methodologies for updating the pheromone matrix. However, tests using a range of problem instances indicate that the MAX-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 [1], where good results are reported.

Another good study looking at this problem is offered by Chiarandini *et al.* in [19]. 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 (much of which was done automatically using the *F-Race* method of Birattari *et al.* [6]), their favoured method is a two-stage, hybrid algorithm that actually 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 are 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 concentrates on satisfying the soft constraints and conducts its search only in feasible areas of the search space. It does this first by using variable neighbourhood search and then with 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 search is starting to stagnate). Extensive use of delta evaluation [39] is also made in an attempt 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 also uses simulated annealing as the main construct of his timetabling algorithm, described in [31]. Based upon his winning entry to the competition, this algorithm works by first gaining feasibility via simple graph colouring heuristics (plus a series of improvement steps if the heuristics prove inadequate) and then uses simulated annealing to try and satisfy the soft constraints by first 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 being constructed, efforts are made in order to try and schedule the events into just forty of the available forty-five timeslots. As the author notes, five of the available timeslots will automatically have penalties attached to them (due to the soft constraint S1) and so it could be a good idea to try and eliminate them from the search from the outset. Indeed, the author only allows the extra five timeslots to be opened if feasibility using forty timeslots cannot be achieved in reasonable time. (In reported experiments, the events in nine of the twenty instances were always scheduled into forty timeslots.) Of course, if an assignment to just forty timeslots is achieved, then it is possible to keep the five end-of-day timeslots closed and simply conduct the soft constraint satisfaction phase on the remaining forty timeslots. This is essentially what

Kostuch's algorithm does and, indeed, excellent results are claimed in [31].

Finally, in [35] Lewis and Paechter have proposed a "grouping genetic algorithm" (GGA) that is used exclusively for finding feasible timetables in this UCTP (i.e. the algorithm does not consider soft constraints). The rationale for this approach is that the objective of this (sub)problem may be viewed as the task of "grouping" events into an appropriate number of timeslots, such that all of the hard constraints are met. Furthermore, because, in this case, it is the timeslots that define the underlying building blocks of the problem (and not, say, the individual events themselves) the algorithm makes use of specialised genetic operators that try to allow these groups to be propagated during evolution[10]. Experiments in [35] show that performance of this algorithm can sometimes also be improved through the use of specialist fitness functions and additional heuristic search operators. One negative feature of this algorithm, however, is that whilst seeming to perform well with smaller instances (≈200 events), it seems less successful when dealing with larger instances (≈1000 events). This is mainly due to the fact that the larger groups encountered in the latter cases tend to present much more difficulty with regards to their propagation during evolution. Indeed, experiments in [35] show that in most cases, significantly better results can actually be gained when the evolutionary features of the algorithm (population, recombination etc.) are removed altogether, thus allowing the heuristic-search operator to work unaided. (This heuristic search-based algorithm forms a part of the algorithm that will be described in section 4.3 later.)

4 A Robust, Two-Stage Algorithm for the UCTP

Having reviewed a number of published works that have looked at this standardised version of the UCTP, in this section we will now describe an example two-stage algorithm that, in our experiences, has performed very well with many available benchmark instances for this problem. The feasibility-finding stage (sections 4.2 and 4.3) is particularly successful; with the twenty competition instances, for example, we will see that it is often able to achieve its goal in very small amounts of time. We will also see that it is able to cope very well with a large number of specially made "harder" instances of various sizes. Meanwhile, the second stage of our

[10] The resultant "grouping" genetic operators follow the methodologies used in similar algorithms for other "grouping problems" such as bin packing [29] and graph colouring [25, 27].

algorithm is concerned with the satisfaction of the soft constraints, which is attempted using two separate phases of simulated annealing that will be described in section 4.5.

4.1 Achieving Feasibility: Pre-compilation and Representational Issues

Before attempting to construct a feasible timetable, in our approach we first carry out some useful pre-compilation by constructing two matrices that are then used throughout the algorithm. We call these the *event-room* matrix and the *conflicts* matrix. Remembering that n represents the number of events and m the number of rooms, the Boolean $(n \times m)$ event-room matrix is used to indicate which rooms are suitable for which events. This can be easily calculated for an event i by identifying which rooms satisfy both the seating capacity and the features required by i. Thus if, room j is deemed suitable, then element (i, j) in the matrix is marked as true, otherwise it is marked as false.

The $(n \times n)$ conflicts matrix, meanwhile, can be considered very much like the standard adjacency matrix used for representing graphs. For our purposes, the matrix indicates which pairs of events can and 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 false. As a final step, and following the suggestions of Carter [14], we are also able to add some further information to the matrix. Note that, in this problem, if we have two events, k and l, that do not conflict but can both only be placed into the same single room, then there can exist no feasible timetable in which k and l are assigned to the same timeslot. Thus, we may also mark elements (k, l) and (l, k) as true in the conflicts matrix.

With regards to the way in which an actual timetable will be represented in this algorithm, similarly to works such as [19, 32, 35, 47], we choose to use a two-dimensional matrix where rows represent rooms, and columns represent timeslots. We also choose to place the restriction that each cell in the matrix (i.e. each place[11] in the timetable) can be blank, or can contain *at most* one event. Note that this latter detail therefore actually encodes the third hard constraint into the representation, meaning that it is now impossible to double book a room.

[11] For the remainder of this chapter, when referring to a timetable, a *place* may be considered a timeslot/room pair. More formally, the set of all places $P = T \times R$.

4.2 Achieving Feasibility - The Construction Stage

An initial assignment of events to places (cells in the matrix) is achieved following the steps outlined in the procedure CONSTRUCT in fig. 2. This procedure is also used for completing partial timetables that can occur as a result of the heuristic search procedure, explained in the next subsection. Starting with an empty or partial timetable *tt* and a list of unplaced events U (in the first case $U = E$), this procedure first opens up a collection of timeslots, and then utilises the procedure INSERT-EVENTS that takes events one-by-one from U and inserts them into feasible places in the timetable *tt*. (The heuristics governing these choices are described in Table 1.) The entire construction procedure is completed when all events have been assigned to the timetable (and therefore $U = \emptyset$).

Of course, during this process, there is no guarantee that every event will have a feasible place into which it can be inserted. In order to deal with these, we therefore relax the requirement regarding the number of timeslots being used, and open up extra timeslots as and when necessary. Obviously, once all of the events have been assigned, if the number of timeslots being used $|T|$ is larger than the actual target amount, then the timetable may not actually be considered feasible (in the strict sense), and efforts will, of course, need to be made to try and rectify the situation. Methods for achieving this will be described in section 4.3.

With regards to the heuristics that are used in this construction process (Table 1), it is worth noting that those used for determining the order in which events are inserted are somewhat akin to the rules for selecting which node to colour next in the classical Dsatur algorithm for graph colouring [7]. However, in this case we observe that h_1 also takes the issue of room allocation into account. Heuristic h_1 therefore selects events based on the state of the current partial timetable, prioritising those with the least remaining feasible options. Ties are then broken by h_2, which chooses the event with the highest conflicts degree (which could well be the most problematic of these events). Once an event has been chosen, further heuristics are then employed for selecting a suitable place. Heuristic h_4 attempts to choose the place that will have the least effect on the future place options of the remaining unplaced events [5]. Heuristic rule h_5, meanwhile, is used to encourage putting events into the fuller timeslots, thereby hopefully packing the events into as few timeslots as possible. Finally, h_3 and h_6 add some randomisation to the process and, in our case, allow different runs to achieve different timetables.

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

INSERT-EVENTS (tt, U, l, r)
1. while (∃ e ∈ U with feasible places between timeslots l and r in tt)
2.     Choose an event e ∈ U with feasible places in tt using $h_1$, breaking ties
       with $h_2$, and further ties with $h_3$;
3.     Pick a feasible place p for e using heuristic $h_4$, breaking ties with $h_5$ and
       further ties with $h_6$;
4.     Move e to p;
5. if (U = ∅) end;
6. else
7.     Open $\lceil |U| / m \rceil$ new timeslots;
8.     INSERT-EVENTS (tt, U, r, len(tt));
```

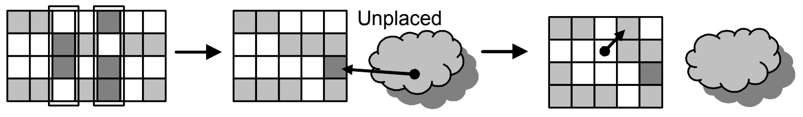
Fig. 2. The procedures CONSTRUCT and INSERT-EVENTS: Here, *tt* represents the current partial timetable and U is a set of unplaced events of cardinality $|U|$. Additionally, len(*tt*) represents a function that returns the number of timeslots currently being used by *tt*; max_timeslots represents the maximum number of timeslots that a timetable can use for it to be considered feasible (i.e. 45), and, as before, m represents the cardinality of the room set.

Table 1. Description of the various event and place selection heuristics used within the procedure INSERT-EVENTS.

Name	Description
h_1	Choose the event with the smallest number of feasible places to which it can be assigned in the current timetable
h_2	Choose the event which conflicts with the largest number of other events
h_3	Choose an event randomly
h_4	Choose the place that the least number of other unplaced events could be feasibly assigned to in the current timetable
h_5	Choose the place in the timeslot with the most events in
h_6	Choose a place randomly

4.3 Reducing the Number of Timeslots with a Heuristic Search Procedure

Although no hard constraints will be violated in any timetable produced by the construction procedure described above, it is, of course, still possible that more than the required number of timeslots will be used, thus rendering it infeasible. We therefore supplement the construction procedure with a heuristic search procedure (originally described in [35]) that operates as follows (see also fig. 3):



(1) Randomly select some timeslots in *tt*, and remove them. Set $i = 0$.

(2) If there are no unplaced events (i.e. $U = \varnothing$) then end; else try to insert the unplaced events into the blank cells in the partial timetable

(3a) Randomly choose a blank and non-blank cell in *tt* and swap; Add 1 to *i*.
(3b) If *tt*'s feasibility is maintained then go back to step (2), else reset the swap.
(3c) If $i \geq$ the iteration limit then end, else go back to step (3a).

Fig. 3. Pictorial description of the heuristic search procedure used for attempting to reduce the number of timeslots used by a timetable.

Given a timetable *tt*, a small number of randomly selected timeslots are first removed (defined by a parameter *rm*, such that between one and *rm* timeslots are chosen randomly). The events contained within these are then put into the list of unplaced events *U*. Steps (2) and (3) of fig. 3 are then applied repeatedly until either *U* is empty, or an iteration limit is reached. If, as in the latter case, upon termination *U* still contains some events, then CONSTRUCT is used to create new places for these. Now, if the resultant timetable is using the required number of timeslots, then the process can be halted (a completely feasible timetable has been found), otherwise further timeslots are selected for removal, and the whole process is repeated.

4.4 Experimental Analysis

As it turned out, the construction procedure described in section 4.2 was actually able to cope quite easily with the twenty problem instances used for the International Timetabling Competition. Indeed, in our experiments feasible timetables using a maximum of 45 timeslots were found straight away in over 98% of trials without any need for opening up additional timeslots or invoking the heuristic search procedure. (Even in cases where the heuristic search procedure *was* needed, feasibility was still always achieved in less than 0.25 seconds of CPU time[12].) We also observed that the construction procedure was often actually able to pack the events into less than the available forty-five timeslots. For example, competition instance-15 required only 41.9 timeslots (averaged across twenty runs), and

[12] These trials, like all experiments described in this chapter, were conducted on a PC under Linux, using 1GB RAM, and a Pentium IV 2.66Ghz processor.

instance-3 required just 41.8. Others, such as competition instances 6 to 9, on the other hand, always required the full forty-five timeslots.

However, although these observations seem to highlight the strengths of our constructive heuristics in these cases, they do not really provide us with much information on the operational characteristics of the heuristic search procedure. For this reason, we therefore conducted a second analysis using an additional set of UCTP instances [2] that have been used in other studies [33-35] and which are deliberately intended to be "hard" with regards to achieving feasibility. These sixty instances are separated into three classes: small, medium, and large (containing approximately 200 events, 400 events and 1000 events respectively). Further details of the instances, including information on how we attempted to ensure their "hardness" can be found at [2] and [33]. Note, however, that each of these instances is known to have at least one feasible solution, and that for some of them there is also a known perfect solution. For the remaining instances, meanwhile, some are known to definitely *not* have perfect solutions[13], whilst, for others, this is still undetermined. See Table 2 below for further information.

In this second set of experiments, we conducted 20 trials per instance, using CPU time limits of 30, 200, and 800 seconds for the small, medium and large instances respectively (these match the time limits used in [35]). We also used parameters $rm = 1$, and an iteration limit of $10000n$. Note that our use of the number of events n in defining the latter parameter allows the procedure to scale with instance size.

Table 2 summarises the results of these experiments and entries that are highlighted indicate problem instances where feasibility was found in every individual trial. Here we can see that in many instances, particularly in the small and medium sets, when timetables using 45 timeslots were not achieved by the construction procedure, the heuristic search operator has successfully managed to remedy the situation within the imposed time limits. Additionally, even with problem instances where solutions using 45 timeslots were not always achieved, we see that the number of timeslots being used generally drops a noteworthy amount within the time limit.

Indeed, our use of the heuristic search procedure is further justified when we compare these results to those achieved by the GGA presented in [35]. From the above table we can see that, with this algorithm, we are

[13] We were able to determine that an instance had no perfect solution when the number of events was greater than $40m$ (where m represents the number of rooms). In these instances we know that at least $(n - 40m)$ events will always have to be assigned to the end-of-day timeslots, thus causing violations of soft constraint S1.

always able to achieve feasible timetables for fifteen, thirteen, and seven instances of the small, medium, and large instance sets respectively. This compares favourably with the work described in [35], where solutions to only eleven, six, and two problem instances were always found. It is also worth pointing out that the results in [35] were also gained after performing considerable parameter tuning with each instance set. Here, on the other hand, the results in Table 2 were gained with very little tuning (beyond our own intuitions), hinting that this algorithm might also be more robust with regard to what instances it is able to effectively deal with.

Table 2. Performance of the Heuristic Search Procedure with the Sixty "Harder" Instances. This table shows, for each instance, the mean and standard deviations of the number of timeslots being used (a) after the initial assignment by the construction procedure (Av. slots. init. ± σ), and (b) at the time limit (Av. Slots. end ± σ). Also shown is the number of timeslots used in the most successful runs (Best). All results are taken from 20 runs per instance and are rounded to one decimal place. Lastly, in column P we also provide some supplementary information about the instances: a "Y" indicates that we know there to be at least one perfect solution obtainable from the instance, an "N" indicates that we know that there definitely isn't a perfect solution, and "?" indicates neither.

	Small (30 seconds)				Medium (200 seconds)				Large (800 seconds)			
#	P	Av. slots. init. ± σ	Av. slots end ± σ	Best	P	Av. slots. init. ± σ	Av. slots end ± σ	Best	P	Av. slots. init. ± σ	Av. slots end ± σ	Best
1	Y	45.8 ± 0.9	44.7 ± 0.5	44	Y	45.8 ± 0.8	44.8 ± 0.4	44	Y	43.9 ± 0.7	43.9 ± 0.7	43
2	Y	45.0 ± 0.0	45.0 ± 0.0	45	Y	47.9 ± 1.4	44.6 ± 0.5	44	Y	49.2 ± 1.6	44.8 ± 0.4	44
3	?	50.0 ± 0.0	44.8 ± 0.4	44	?	47.1 ± 1.4	44.9 ± 0.3	44	Y	46.9 ± 0.8	44.9 ± 0.3	44
4	Y	50.5 ± 1.8	44.4 ± 0.5	44	N	50.4 ± 1.3	44.7 ± 0.5	44	N	52.1 ± 0.9	45.2 ± 0.4	45
5	?	57.6 ± 1.3	45.0 ± 0.0	45	N	51.0 ± 1.5	45.0 ± 0.2	44	N	54.1 ± 1.5	46.0 ± 0.0	46
6	Y	43.3 ± 1.6	43.3 ± 1.6	41	Y	56.8 ± 1.8	45.0 ± 0.2	44	N	59.8 ± 1.7	48.7 ± 0.5	48
7	?	53.0 ± 0.0	44.9 ± 0.3	44	?	62.2 ± 1.6	48.1 ± 0.6	47	N	67.1 ± 1.8	54.0 ± 0.6	52
8	N	55.7 ± 1.2	46.0 ± 0.4	45	Y	58.8 ± 1.4	44.9 ± 0.3	44	N	53.6 ± 1.7	45.0 ± 0.0	45
9	N	64.0 ± 0.0	45.5 ± 0.5	45	?	67.1 ± 1.8	47.8 ± 0.6	47	N	51.1 ± 1.0	45.1 ± 0.3	45
10	N	46.0 ± 0.0	45.0 ± 0.0	45	Y	46.0 ± 1.3	44.7 ± 0.5	44	N	51.7 ± 1.1	46.0 ± 0.0	46
11	Y	44.9 ± 0.4	44.9 ± 0.4	43	Y	60.3 ± 1.7	45.0 ± 0.2	44	N	53.3 ± 1.0	46.0 ± 0.0	46
12	N	45.0 ± 0.0	45.0 ± 0.0	45	?	51.2 ± 1.3	45.0 ± 0.2	44	Y	48.7 ± 1.0	45.0 ± 0.2	44
13	N	60.8 ± 0.9	45.1 ± 0.3	45	Y	63.7 ± 1.6	45.2 ± 0.5	44	Y	51.6 ± 0.7	45.0 ± 0.0	45
14	N	64.1 ± 0.8	46.7 ± 0.9	45	Y	55.4 ± 1.0	44.8 ± 0.4	44	Y	49.1 ± 0.9	45.0 ± 0.0	45
15	Y	45.0 ± 0.0	45.0 ± 0.0	45	N	59.8 ± 1.9	45.0 ± 0.0	45	Y	65.4 ± 1.2	45.6 ± 0.7	45
16	Y	60.9 ± 2.3	44.8 ± 0.4	44	?	75.1 ± 1.5	46.9 ± 0.7	46	Y	63.0 ± 1.3	45.9 ± 0.7	45
17	?	59.0 ± 0.0	45.0 ± 0.0	45	Y	65.6 ± 1.4	44.7 ± 0.5	44	?	88.9 ± 1.4	56.0 ± 1.2	54
18	N	53.2 ± 0.8	45.3 ± 0.5	45	?	89.6 ± 0.5	45.6 ± 0.6	45	?	77.0 ± 1.6	56.3 ± 1.1	54
19	N	73.6 ± 2.8	45.0 ± 0.0	45	N	92.4 ± 1.5	46.0 ± 0.5	45	?	81.7 ± 1.4	61.1 ± 0.8	60
20	N	46.0 ± 0.0	45.0 ± 0.0	45	N	77.7 ± 2.3	46.3 ± 0.6	45	?	76.1 ± 2.3	55.0 ± 0.7	54

4.5 Satisfying the Soft Constraints

Having now reviewed a seemingly effective and robust algorithm for achieving timetable feasibility, in this section we will now move on to the task of satisfying the soft constraints of the UCTP. Similarly to the ideas of White and Chan [51] and also Kostuch [31], our algorithm will attempt to do this in two phases: firstly, by seeking a suitable ordering of the timeslots (using neighbourhood operator N_1 – see fig. 4), and secondly by shuffling events around the timetable (using neighbourhood operator N_2). In both phases we will use simulated annealing (SA) for this task and, as we will see, the second SA phase will generally constitute the lengthiest part of this process. In this algorithm we also make extensive use of delta-evaluation [39], and the algorithm will halt when a perfect solution has been found or, failing this, when a predefined time limit is reached (in the latter case, the best solution found during the whole run will be returned).

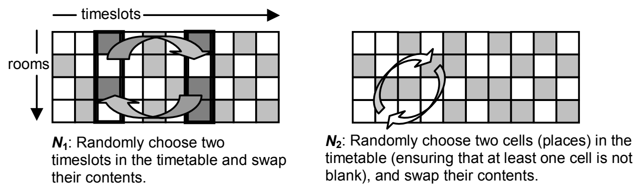

timeslots
rooms
N_1: Randomly choose two timeslots in the timetable and swap their contents.
N_2: Randomly choose two cells (places) in the timetable (ensuring that at least one cell is not blank), and swap their contents.

Fig. 4. The two neighbourhood operators used with the simulated annealing algorithm.

In both phases, SA will be used in the following way: starting at an initial temperature t_0, during the run the temperature t will be slowly reduced. At each value for t, a number of neighbourhood moves will then be attempted. Any move that increases the *cost* of the timetable (i.e. the number of soft constraint violations) will then be accepted with probability defined by the equation $\exp(-\delta/t)$, where δ represents the change in cost. Moves that reduce or leave unchanged the cost, meanwhile, will be accepted automatically.

In the next four subsections we will outline the particular characteristics of these two SA phases. We will then present an experimental analysis in section 4.5.5-6.

4.5.1 SA Phase-1 - Search Space Issues

This phase of SA is concerned with the exploration of the search space defined by neighbourhood operator N_1 (see fig. 4). Note that due to the structure of this timetabling problem (in particular, that there are no hard constraints that depend on the ordering of events), a movement in N_1 will always preserve feasibility.

It is also worth mentioning, however, that often there may be many feasible timetables that are not achievable through the use of N_1 alone. For example, the size of the search space offered by N_1 is $|T|!$ (i.e. the number of possible permutations of the timeslots). However, given that a feasible timetable must always have $|T| \leq 45$, this means that the number of possible solutions achievable with this operator will not actually grow with instance size. Also, 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 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 *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 therefore decided that this phase of SA would only be used as a preliminary step for making quick-and-easy improvements to the timetable. Indeed, this also showed to be the most appropriate response in practice.

4.5.2 SA Phase-1 - Cooling Schedule

For this 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 are beyond the scope of this chapter, but of which more details can be found in [50]. However, it is worth noting that in general SA practice, it is important that a correct value for t_0 is determined: a value 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 might cause the algorithm to be too greedy from the outset and make it 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.

With regards to other features of the cooling schedule, because we only view this phase as a preliminary, during execution we choose to limit the number of temperatures that we will anneal at to a fixed value M. In order to have an effective cooling, this also implies a need for a cooling schedule that will decrement the temperature from t_0 to a value close to zero, in exactly M steps. We use the following cooling scheme:

$$\begin{aligned} \lambda_0 &= 1-\beta \\ \lambda_{i+1} &= \lambda_i + \tfrac{\beta+\beta}{M} \\ t_{i+1} &= t_i - \lambda_{i+1}\tfrac{t_0}{M} \end{aligned} \tag{3}$$

Here, β represents a parameter that, at each step, helps determine a value for λ. This λ-value is then used for influencing the amount of concavity or convexity present in the cooling schedule. Fig. 5 shows these effects in more detail.

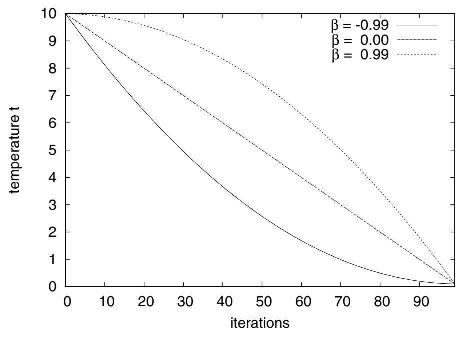


Fig. 5. The effects of the parameter β with the cooling scheme defined in eq. (3). For this example, $t_0 = 10.0$ and $M = 100$.

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 $\beta = -0.99$. The number of neighbourhood moves to be attempted at each temperature was set at $|T|^2$, thus keeping it proportional to the total size of the neighbourhood (a strategy used in many SA implementations [3, 19, 31]).

4.5.3 SA Phase-2 - Search Space Issues

In this 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 neighbourhood operator N_2 is conducted (see fig. 4). However, 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 deal with this fact by immediately rejecting and resetting 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 N_2 can offer the search. Suppose, for the sake of argument, that in a single application of the operator we elect to swap cells p and q:

- 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, only the rooms of the affected events will change;
- If p and q are in the same row, only the timeslots of the affected events will change;
- If p and q are in different rows *and* different columns, then both the rooms and timeslots of the affected events will be changed.

As can be seen, N_2 therefore has the potential to alter a timetable in a variety of ways. In addition, we also note that the number of new solutions (feasible and infeasible) that are obtainable via any single application of N_2 is exactly:

$$\tfrac{1}{2}n(n-1)+nx-1 \tag{4}$$

(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 suggests that for anything beyond very small instances, more time will generally be required for a thorough exploration of N_2's solution space.

4.5.4 SA Phase-2 - Cooling Schedule

For this phase, an initial temperature t_0 is calculated in a very similar fashion to SA phase-1. However, before starting this second SA phase we also choose to reduce the result of this calculation by a factor (c_2/c_1), where c_1 represents the cost of the timetable 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.

In order to determine when the temperature t should be decremented we choose to follow the methodologies used by Kirkpatrick *et al.* [30] and Abramson *et al.* [3] and define 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, we calculate this with the formula: $\eta_{\max}n$ (where $\eta_{\max}$ is a parameter that we will need to tune[14]). 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 with the formula $\eta_{\min}(\eta_{\max}n)$, where $\eta_{\min}$ is in the range (0, 1] and must also be tuned.

To decrease the temperature, we choose to use the traditional geometric scheme [30] where, at the end of each cycle, the current temperature t_i is modified to a new temperature t_{i+1} using the formula $t_{i+1} = \alpha t_i$, where α is a control parameter known 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 when no improvement in cost is achieved for ρ successive values of t (and so the search has presumably become stuck at a local optimum). In order to calculate a suitable temperature to reheat to, we choose to use a method known as "reheating as a function of cost", which was originally proposed by Abramson, Krishnamoorthy, and Dang in [4]. In essence, this scheme determines a reheat temperature by considering the current state of the search; thus, if the best solution found so far has a high cost, then a relatively high reheat temperature will be calculated (as it is probably favourable to move the search to a new region of the search space). On the other hand, if the best solution found so far is low in cost, then a lower reheat temperature will be calculated, as it is probably the case that only small adjustments need to be made. In studies such as [4] and [26] (where further details can also be found) this has shown to be an effective method of reheating with this sort of problem.

4.5.5 Algorithm Analysis – 45 or 40 Timeslots?

For our experimental analysis of this SA algorithm, we performed two separate sets of trials on the 20 competition instances, using a time limit specified by the competition-benchmarking program[15]. For the first set, we simply used our construction and heuristic search procedures (section 4.2 and 4.3) to make any feasible timetable where a maximum of 45 timeslots was being used. The SA algorithm would then take this timetable and operate in the usual way. For our second set, however, we chose to make a slight modification to our algorithm and allowed the heuristic search pro-

[14] Note that our use of the number of events n in this formula keeps the result of this calculation proportional to instance size.

[15] This equated to 270 seconds of CPU time on our computers

cedure to run a little longer in order to try and schedule all of the events into a maximum of just 40 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 N_2 seemed to have when attempting to deal with violations of soft constraint S1: often, when trying to rid a timetable of a violation of S2 or S3, N_2 would do so by making use of an end-of-day timeslot. Or in other words, in trying to eliminate one constraint violation, the algorithm would often inadvertently cause another one. The reasons why such behaviour might occur start to become more evident if we look back at the descriptions of the three soft constraints in section 3.2. Note that S2 and S3 stand out as being slightly different to S1, because if an event *e* is involved in a violation of either S2 or S3, then this will not simply be down 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 S1, then this will be due to it being assigned to one of the five end-of-day timeslots, and has nothing to do with the relative positions of other events with common students to *e*. Thus, given that a satisfaction of S1 depends solely on not assigning events to the five end-of-day timeslots, a seemingly intuitive idea might therefore be to simply remove these five timeslots (and therefore constraint S1) from the search altogether. In turn, the SA algorithm will then only need to consider the remaining 40 (unpenalised) timeslots and only try to satisfy the two remaining soft constraints.

In our case, it turned out that our strategy of allowing the heuristic search procedure to run a little longer worked quite well: using the same experimental set-up as described in section 4.4, with the 20 competition instances the procedure was able to schedule all events into 40 timeslots in over 94% of cases. In the remaining cases (which, incidentally, never actually required more than 41 timeslots) the extra timeslots were labelled as end-of-day timeslots. However, in order to still lend special attention to the task of eliminating S1 violations, we used a slightly modified version of N_2 that would automatically reject any move that caused the number of events assigned to these end-of-day timeslots to increase, but would also eliminate these timeslots if they were ever to become empty during the SA process. In our case, this strategy would always eliminate the remaining end-of-day timeslots within the first minute-or-so of the run.

4.5.6 Results

Table 3 provides a comparison of these two sets of trials using 50 runs on each of the 20 instances. In both cases we used a cooling rate of $\alpha = 0.995$ and $\rho = 30$. Suitable values for $\eta_{\min}$ and $\eta_{\max}$ (the two parameters that we witnessed to be the most influential regarding algorithm performance), on the other hand, were determined empirically by running 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 $\eta_{\max}$ we then performed 20 separate runs on each of the 20 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 40 and 45 timeslots) were then used for our comparison.

It can be seen in Table 3 that when using just 40 timeslots the SA-algorithm is able to produce better average results in the majority of cases (17 out of the 20 instances). Additionally, the best results (from 50 runs) are also produced in 16 of the 20 instances, with ties occurring on a further 2. A Wilcoxon signed-rank test also reveals the differences in results produced in each set of trials to be significant (with a probability greater than 95%). The results gained when using 40 timeslots also compare well to other approaches. For example, had the best results in the Table 3 been submitted to the timetabling competition, then according to the judging criteria, this algorithm would have been placed second (although note that according to a Wilcoxon signed-rank test, there is actually no significant difference between these results and the competition winner, which, incidentally, also reported results that were the best found in 50 runs).

The reasons why we believe the use of just 40 timeslots to be advantageous have already been outlined in the previous subsection. However, it is also worth noting that although the entire search space will be smaller when we are using only 40 timeslots (because there will be $5m$ fewer places to which events can be assigned to) 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 blanks (and therefore just one event) are, in general, more likely to retain feasibility than those involving two, this means that further restrictions will actually be added to a search space where movements are already somewhat inhibited. Considering that that one of the major requirements for the two-stage timetabling approach is for practical amounts of movements in feasible areas of the search space to be achievable (see section 2), there is thus a slight element of risk in reducing the number of timeslots in this way. For these instances, however, the strategy seems to be beneficial.

Table 3. Comparison of the two trial-sets using the 20 competition instances. In each case the average cost, standard deviation, and best cost (parenthesised) from 50 runs on each instance is reported.

Instance #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Using 45 slots with η_{max} = 9 and η_{min} = 0.1	85.9 ± 10.9. (68)	68.5 ± 8.2. (49)	86.9 ± 12.7. (63)	260.1 ± 23.4. (207)	190.1 ± 25.7. (133)	31.5 ± 8.8. (12)	42.3 ± 17.0. (19)	28.3 ± 7.2. (14)	52.2 ± 9.6. (31)	84.9 ± 8.0. (68)
Using 40 slots with η_{max} = 5 and η_{min} = 0.9	86.9 ± 17.6. (62)	53.5 ± 10.2. (39)	95.6 ± 18.8. (69)	231.8 ± 39.5. (176)	147.7 ± 29.5. (106)	22.8 ± 8.4. (11)	23.7 ± 13.3. (5)	22.2 ± 8.6. (10)	41.4 ± 13.7. (22)	91.7 ± 15.3. (70)

Instance #	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**Av.**
Using 45 slots with η_{max} = 9 and η_{min} = 0.1	61.6 ± 9.7. (43)	147.5 ± 16.3. (109)	130 ± 14.1. (101)	107 ± 33.4. (55)	41.5 ± 8.5. (22)	47.2 ± 7.9. (29)	169.3 ± 26.5. (119)	45.9 ± 7.8. (27)	85.5 ± 14.7. (62)	9.5 ± 4.5. (1)	88.8 (61.6)
Using 40 slots with η_{max} = 5 and η_{min} = 0.9	60.6 ± 16.0. (38)	133.8 ± 28.1. (94)	128.2 ± 19.2. (101)	66.3 ± 20.7. (37)	33.2 ± 13.6. (14)	35.8 ± 12.6. (18)	129.4 ± 25.0. (94)	40.8 ± 9.7. (27)	84.9 ± 21.2. (55)	8.6 ± 6.1. (0)	77 (52.4)

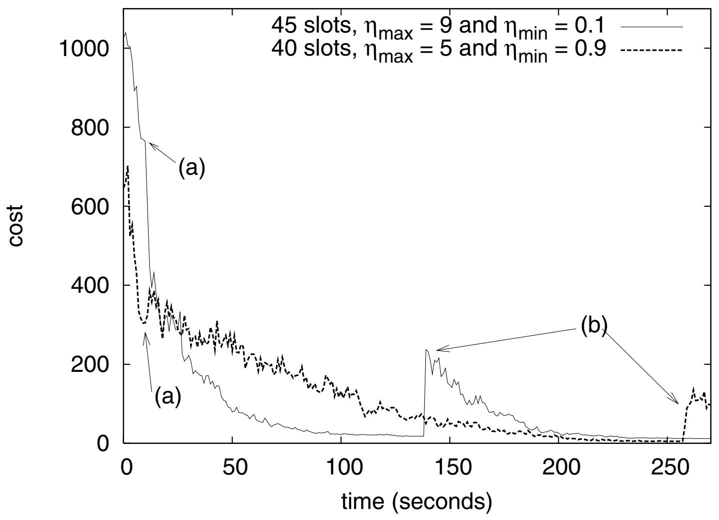


Fig. 6. Two example runs of the SA algorithm on competition instance-20. Points (a) indicate where the algorithm has switched from SA phase-1 to SA phase-2. Points (b) indicate where a reheating has occurred.

Finally, in fig. 6 we show two example runs of the SA algorithm using the parameters defined in Table 3. Here we can observe the 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 it is invoked too late to have a positive effect). Additionally, we can see that the second line (using 40 timeslots) actually starts at a markedly lower cost than the first, because the elimination of all S1 violations 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.

5 Conclusions and Discussion

University timetabling in the real world is an important problem that can often be difficult to solve adequately, and sometimes impossible (without relaxing some of the imposed constraints). In this chapter we have mentioned that one of the most important issues for designers of timetabling algorithms is the question of how to deal effectively with both the hard constraints and the soft constraints, and have noted that when using metaheuristics, this is usually attempted in one of three ways: by using one-stage optimisation algorithms; by using two-stage optimisation algorithms; or by using algorithms that allow relaxations of some feature of the problem.

In this chapter we have given a detailed analysis of the so-called UCTP and have reviewed many of the existing works concerning it. In section 4 we have also provided a description and analysis of our own particular algorithm for this problem. As we have noted, the UCTP was used as the benchmark problem for the International Timetabling Competition in 2002-3. By formulating this problem and then encouraging researchers to write algorithms for it, we have attempted to avoid many of the difficulties that are often caused by the idiosyncratic nature of timetabling, and have provided a means by which researchers can test and compare their algorithms against each other in a meaningful and helpful way.

However, it should be noted that when conducting research in this way we must always be cautious about extrapolating strong scientific conclusions from the results. For example, whilst one timetabling algorithm may *appear* to be superior to another, these differences could be due to mundane reasons such as programming/compiler issues, or the parameters and/or seeds that are used. Superior performance might also simply occur because some algorithms are more suited to the particular constraints of this problem.

It is also worth bearing in mind that whilst the use of benchmark instances may facilitate analysis and comparison of algorithms, ultimately they do not necessarily allow insight into how these algorithms might fare with other kinds of problem instance. For example, in this chapter we have seen that many of the algorithms that have gained good results to the 20 competition instances – including our own – have done so using a two-stage optimisation approach. However, this apparent success could, in part, be due to the competition criteria for judging timetable quality (section 3.3), and also the fact that the instances are fairly easy to solve with regard to finding feasibility. This might therefore lend favour to the two-stage optimisation approach. Indeed, in cases where different judging criteria or different problem instances are used, perhaps some other sort of timetabling strategy would show more value.

In conclusion, when designing algorithms for timetabling, it is always worth remembering that in the real world many different sorts of constraints, problem instances, and even political factors might be encountered. The idiosyncratic nature of real-world timetabling indicates 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.

References

[1] http://www.idsia.ch/Files/ttcomp2002/

[2] http://www.emergentcomputing.org/timetabling/harderinstances

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] P. Cowling, S. Ahmadi, P. Cheng, and R. Barone, “Combining Human and Machine Intelligence to Produce Effective Examination Timetables,” presented at The Forth Asia-Pacific Conference on Simulated Evolution and Learning (SEAL2002), Singapore, 2002.

[25] 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.

[26] 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.

[27] 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.

[28] S. Even, A. Itai, and A. Shamir, “On the complexity of Timetable and Multi-commodity Flow Problems,” *SIAM Journal of Computing*, vol. 5, pp. 691-703, 1976.

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] R. Lewis and B. Paechter, "Finding Feasible Timetables using Group Based Operators," *(Forthcoming) Accepted for publication in the IEEE Trans. Evolutionary Computation*, 2006.

[36] Z. Michalewicz, "The Significance of the Evaluation Function in Evolutionary Algorithms," presented at The Workshop on Evolutionary Algorithms, Institute for Mathematics and Its Applications, University of Minnesota, Minneapolis, Minnesota, 1998.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

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

[47] K. Socha and M. Samples, "Ant Algorithms for the University Course Timetabling Problem with Regard to the State-of-the-Art," in *Evolutionary Com-*

putation in Combinatorial Optimization (EvoCOP 2003), vol. 2611, *Lecture Notes in Computer Science*. Berlin: Springer-Verlag, 2003, pp. 334-345.

[48] H. Terashima-Marin, P. Ross, and M. Valenzuela-Rendon, "Evolution of Constraint Satisfaction Strategies in Examination Timetabling," presented at The Genetic and Evolutionary Computation Conference (GECCO), 2000.

[49] 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.

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

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