Finding Feasible Timetables using Group-Based Operators

Rhydian Lewis and Ben Paechter, *Members IEEE*

***Abstract*— This paper describes the applicability of the so-called 'grouping genetic algorithm' to a well-known version of the university course timetabling problem. We note that there are, in fact, various scaling up issues surrounding this sort of algorithm and, in particular, see that it behaves in quite different ways with different sized problem instances. As a by-product of these investigations, we introduce a method for measuring population diversities and distances between individuals with the grouping representation. We also look at how such an algorithm might be improved: firstly, through the introduction of a number of different fitness functions and, secondly, through the use of an additional stochastic local-search operator (making in effect a grouping *memetic* algorithm). In many cases, we notice that the best results are actually returned when the grouping genetic operators are removed altogether, thus highlighting many of the issues that are raised in the study.**

***Index Terms*—Diversity, fitness-functions, grouping-problems, timetabling.**

I. Introduction

In the context of a university, a typical timetabling problem generally involves assigning a set of events (lectures, exams, tutorials, lab sessions and so on) to a limited number of timeslots and rooms in such a way as to satisfy a set of constraints. The two most common forms of this problem are exam-timetabling problems and course-timetabling problems, and in reality, the constraints imposed upon these can often be quite similar. However, the crucial difference between them is usually considered to be that in exam timetables, multiple events can take place in the same room at the same time (as long as the seating capacity is not exceeded), whilst in course-timetabling problems, we are generally only allowed one event in a room per timeslot.

In automated timetabling, the constraints for both types of timetabling problem generally tend to be separated into two groups: the hard constraints and the soft constraints. Hard constraints have a higher priority than soft, and will usually be mandatory in their satisfaction. Indeed, timetables will usually only be considered *feasible* if and only if all of the hard constraints of the problem have been satisfied. Soft constraints, meanwhile, are those that we want to obey if possible, and more often than not they will describe what it is for a timetable to be *good* with regards to the timetabling policies of the university concerned, as well as the experiences of the people who will have to use it. Perhaps the most common hard constraint encountered in timetabling is the 'event-clash' constraint. This specifies that if one or more person (or some other resource of which there is only one) is required to be present at two distinct events, then these events *conflict*, and therefore must not be placed into the timetable in such a way that they overlap in time (as obviously such an assignment will result in this person(s)/resource(s) having to be in two places at once). This particular constraint can be found in almost all university timetabling problems and its presence will often cause people to draw parallels between this problem and the well-known graph colouring problem (which we will look at in more detail in section II *B*). Beyond this example constraint, however, a great many other sorts of constraints – hard and soft – can be considered in timetabling. These can involve factors such as event orderings, lecturer and student preferences, the spreading of events, rooming constraints, and so on. Indeed, in the real world most universities will tend to have their own idiosyncratic set of constraints. A good review of the various different constraints that can often be encountered in practice is given by Corne, Ross, and Fang in [1].

Given the wide diversity of constraints that can be imposed on timetabling problems, it should be appreciable that from a research point-of-view, it can often be quite difficult to formulate meaningful and universal generalisations about the field. Further difficulties can also arise when we want to make comparisons between different timetabling algorithms, as often – but perhaps quite understandably – authors will prefer to design algorithms for their own university's timetabling problem, rather than comply with some set of benchmark instances. However, one important generalisation that we can make about timetabling at universities is that the problem is NP-complete in almost all variants. Indeed, in various forms it has been shown to be equivalent to graph colouring, bin packing, and three-dimensional matching in [2], and also 3-SAT in [3].

Many algorithms that have been proposed for solving timetabling problems have used strategies derived from graph colouring. (See, for example, a very early example by White and Chan in [4]; the more recent backtracking algorithm of Carter, Laporte, and Lee in [5]; and also the work of Erben in

Manuscript received August 8th, 2005. Revised Jan. 2006 and June 2006.

The authors are with the Centre for Emergent Computing at Napier University, Edinburgh, Scotland, EH10 4DQ. (Phone: 0131-455-2767; email: {r.lewis|b.paechter}@napier.ac.uk). From September 2006, the first author will be contactable at the Cardiff Business School, University of Wales Cardiff, Colum Drive, Cardiff, Wales, CF10 3EU. (Phone +44(0)292-087-4000).

[6]). Others, meanwhile, have chosen to use methods such as linear algorithms [7], integer programming [8], and constraint-based techniques [9] for their timetabling problems.

Over the last decade-or-so, there has also been a large interest in the application of metaheuristics towards timetabling problems. For instance, many authors, such as Abramson, Krishnamoorthy, and Dang [10, 11]; Elmohamed, Fox and Coddington [12]; and Thompson and Dowsland [13] have chosen to apply simulated annealing to their timetabling problems. In the latter paper, for example, the authors first employ graph colouring heuristics to construct a feasible timetable, and then use various specialised neighbourhood operators (such as Kempe-chain interchanges) in conjunction with simulated annealing to then try and satisfy the soft constraints of the problem.

Other authors such as Schaerf [14], Costa [15], and Hertz [16] have chosen to use the tabu search metaheuristic for timetabling. In particular, Schaerf reports that good results can be gained when phases of tabu search are broken up by periods of local-search, making use of various neighbourhood operators. In this case, the best results from each stage are passed into the next stage, until no further progress can be made. In this study the author also makes use of a weighted-sum evaluation function, using weights in order to penalise violations of hard constraints more heavily than violations of soft constraints. These weights can be changed during the run however, if it is felt that the search is focusing too closely on one particular region of the search space. (Similar techniques are also used in conjunction with a local-search algorithm by Schaerf in [17].)

There have also been many applications of evolutionary and memetic-style algorithms to various different timetabling problems[1], as in the work of Colorni, Dorigo, and Maniezzo [19, 20]; Abramson and Abela [21]; Corne, Ross and Fang [1]; Paechter *et al*. [22]; Erben [6]; and Burke *et al*. [23-26]. In the work of Paechter *et al*., for example, an evolutionary algorithm for course timetabling using an indirect representation is presented, whereby each chromosome contains instructions on how to build timetables of a given instance. Also included is a memetic search operator that attempts to locally improve each timetable, with any improvements that are found then being written back to the chromosome. The authors also use specialised crossover operators for this representation, as well as heuristic mutation operators. A different sort of evolutionary approach – this time for exam timetabling – has also been proposed by Burke, Elliman and Weare in [23, 24]. In this approach, all candidate solutions produced during a run are kept feasible because, rather than break any hard constraints, extra timeslots are opened to accommodate events that have no feasible place in the current timetable. One of the aims of the algorithm, therefore, is to reduce the number of timeslots being used down to a reasonable level, whilst also taking into consideration the imposed soft constraints. Thus, specialist genetic operators are introduced in order to try and accomplish this.

Recently, an interesting application of the ant colony metaheuristic to timetabling has also been made by Socha, *et al*. in [27] and [28]. At each step of this algorithm, each of the ants constructs a complete assignment of events to timeslots using heuristics and pheromone information (the latter which is present due to previous iterations of the algorithm). Timetables are then improved using a local-search procedure, and results are written back to the pheromone matrix for use in the next iteration.

It is also worth noting that as well as the more 'mainstream' metaheuristic paradigms, various other stochastic-based algorithms have also been proposed for timetabling problems, such as hyper-heuristics [29], a GRASP procedure [30], multi-objective techniques [31, 32], and a number of hybrid algorithms [33-36]. Some good survey papers about the field of automated timetabling can also be found in [37-41]. There are also some publicly available problem instances available for exam timetabling at [42] and course timetabling at [43] and [44]. It is the latter two references that contain the problem instances that will be considered in this study.

II. Problem Analysis

A. The Basic Problem

The version of the university course timetabling problem (UCTP) that we study here was originally formulated for the Metaheuristics Network [45] and was also subsequently used for the International Timetabling Competition in 2002 [43]. We have chosen this particular problem because it has been fairly widely studied in the last few years and, as a result, it has become somewhat of a benchmark problem in this field. Note also that although this problem is actually based on real-world timetabling problems, it is slightly simplified. Although this is not ideal, it is arguable that this sort of problem is perhaps more suited for algorithm analysis and comparison, as it does not feature the various idiosyncratic and institution-specific constraints usually found in practical problems.

Each problem instance consists of the following information: firstly, we are given a set of rooms, each with an associated seating capacity. Secondly, we are given a set of events, each with a pre-specified set of attending students. Pairs of events are said to *conflict* if there is one or more student(s) required to attend them both. Finally, we are given a set of room features (that can represent things such as data projectors, IT equipment etc). Each event *requires* a subset of these features and each room *satisfies* a subset of these features.

The primary aim of this problem is to assign every event a room and one of a fixed-number of timeslots (in this case forty-five, comprising five days of nine timeslots) in such a way that none of the following hard constraints are violated:

1. No student is required to attend more than one event at

[1] See the survey of Ross, Hart and Corne [18] for a good overview of evolutionary computation and timetabling.

any one time (or in other words, conflicting events should not be assigned to the same timeslot);
2. Only one event is put in any room in any timeslot (i.e. no double-booking of rooms);
3. All of the features required by the event are satisfied by the room, which has an adequate capacity.

A timetable that schedules all events and obeys all of these constraints is considered feasible. The total number of possible assignments (timetables) is $(t*r)^e$ (where t = the number of timeslots, r = the number of rooms, and e = the number of events). In anything but trivial cases, the vast majority of these assignments will probably contain some level of infeasibility.

B. Comparison to Graph Colouring

It is worth noting immediately (as it will certainly help with various explanations later on), that the presence of the first hard constraint above makes this problem similar to the well-known graph colouring problem. In order to convert one problem to the other, individual events are considered nodes, and edges are added between any two nodes that represent conflicting events. In very basic timetabling problems (e.g. [46]) the task is to simply colour the graph in as many colours as there are available timeslots. However, in our case, the presence of the second and third hard constraints adds extra complications: now it does not merely suffice for collections of non-conflicting events to be grouped together into timeslots; instead, we must also ensure that every event in a timeslot can be assigned to its own feasible room. From a graph-theoretic point-of-view, this means that many feasible colourings might still actually represent infeasible timetables (see fig. 1 for a simple example).

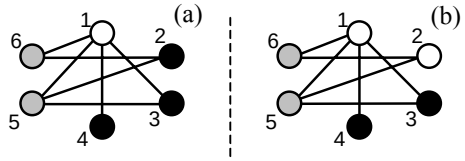


Fig. 1. In this small example both graphs represent optimal colourings. However, in the case of this timetabling problem, if only two rooms were available per timeslot, then graph (a) could definitely not represent a feasible solution, as either event 2, 3 or 4 could not be assigned a room. Graph (b), on the other hand, could represent a feasible timetable although, of course, this would also depend on room capacities and features of the problem instance.

C. Soft Constraints and the Two-Stage Approach

For this particular timetabling problem, there are also three soft constraints. These are as follows:

1. No student should attend an event in the last timeslot of a day;
2. No student should sit more than two classes in a row;
3. No student should have a single class in a day.

Looking at the majority of studies carried out on this problem, it would seem that a popular method for dealing with both the hard and soft constraints is to employ a two-stage algorithm, the methodology of which can be summarised as follows: in the first stage, the soft constraints are disregarded and only the hard constraints of the problem are considered (i.e. only a feasible timetable is sought); next, assuming feasibility has been found, attempts are then made to try and reduce the number of soft constraint violations, using operators that only allow feasible areas of the search space to be navigated. The popularity of this approach could partly be due to the substantial work of Rossi-Doria *et al.* [35] who, in an early study of this problem, concluded that the performance of any one metaheuristic with respect to satisfying hard constraints and soft constraints might be different: that is, what might be a good approach for finding feasibility may not necessarily be so effective when considering the soft constraints.

Indeed, the merit of this two-stage approach was reinforced when the International Timetabling Competition [43] was run in 2002 and people from all over the world were invited to design algorithms for this problem. As it turned out, the best algorithms presented (according to the competition criteria) made use of this two-stage approach, utilising various constructive heuristics to first find feasibility, followed by assorted local improvement algorithms (such as simulated annealing, tabu search, and local-search) to then deal with soft constraint violations. Since then, a number of papers stemming from this work have been published and excellent results have been claimed using this basic two-stage approach. (These include the hybrid algorithm of Chiarandini *et al*. [33]; the simulated annealing-based methodology of Kostuch [36]; the tabu search-based methodologies of Arntzen and Løkketangen [47] and Cordeau *et al*. [48]; and the 'Great Deluge' approach of Burke *et al.* [49].)

It must be noted, however, that the problem instances (available at [43]) used for the competition and subsequent works are actually quite easy to solve with regards to finding feasibility, because they were created mainly with soft constraints in mind. This was because the competition rules stated that for entrants to qualify, their algorithms had to find feasibility on all instances (those that could not achieve this were disqualified). Naturally, this has meant that the majority of work so far has pertained to soft constraint satisfaction. However, this still leaves a major issue of concern: How can we ensure that we have a good chance of finding feasibility when 'harder' instances of this problem are encountered? Indeed, the problem of finding a feasible timetable is, of course, NP-hard and should not be treated lightly. Thus we believe that there are justifications and needs for a more powerful search algorithm that specialises in finding feasible timetables, which can also cope across a wide range of problem instances. An algorithm looking to achieve just this is one of the main aims of this paper.

D. Grouping Genetic Algorithms and the UCTP

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

$$\cup u_i = U \text{ and } u_i \cap u_j = \varnothing,\ i \neq j. \quad (1)$$

As well as this, in grouping problems there are also usually some problem-specific constraints that define valid and legal groupings, and examples include such well-known problems as graph colouring, the frequency assignment problem, bin balancing, and the bin packing problem. Indeed it was the latter problem that was first addressed via a GGA by their creator, Emanuel Falkenauer, in [50].

In [50] and [51], Falkenauer convincingly argues that when considering grouping problems, the so-called 'traditional' genetic operators and representations can actually be highly redundant, not least due to the fact that the operators are *item-oriented* rather than *group-oriented*. The upshot is a general tendency for these operators to recklessly break up the building blocks that we might otherwise want promoted. As an example, consider the traditional item-based encoding scheme, where a chromosome such as 31223 represents a solution where the first item is in group three, the second is in group one, the third is in group two, and so on. (This has been used, for example, with timetabling in [1] and [35].) First of all, when used with a grouping problem, such a representation goes against the principle of minimum redundancy (see [52]) because, given a candidate solution using n groups (in the chromosome above, for example, n = 3), there are actually another (n!)–1 possible chromosomes that will represent the same grouping of items. This means that the size of the search space will be much larger than it needs to be. Next, if we were to make use of a 'traditional' recombination operator with this encoding, we would generally see context dependant information being passed out of context and, as a result, the offspring would rarely resemble either of their two parents (with respect to the solutions that they represent). For example, let us apply a standard two-point crossover to two chromosomes: 3|12|22 crossed with 1|23|12 would give, as one of the offspring, 32322. Firstly, this offspring no longer has a group 1 and, depending on the problem being dealt with, this may mean that it is invalid. Secondly, it could be agued that this operation has resulted in nothing more than a near random jump in the search space, going against the general aim of a recombination operator.

Similar observations can also be made with a standard mutation operator with this encoding and also with the typical genetic operators that work with permutation based encodings, such as the partially mapped crossover of Goldberg [53] (see [51], pages 85-96, for a more detailed description).

These arguments lead to the following conclusion: When considering grouping problems, it is essentially the groups themselves that are the underlying building blocks of the problem and not, for example, the particular states of any of the items individually. Thus, representations and genetic operators that allow these groups to propagate effectively during the evolutionary process are a promising approach. With this in mind, a standard GGA scheme has been proposed by Falkenauer in [50] and [51], and there have since been applications of this basic technique to the bin balancing (or equal-piles) problem [54], graph colouring [6, 55], edge colouring [56] and exam-timetabling [6], each with varying degrees of success.

It is fairly clear from the problem description in this section that the UCTP considered here also constitutes a grouping problem. In this case, the 'items' are the events themselves, and the 'groups' are defined by the timeslots. Thus, in order for a timetable to be feasible, the events need to be grouped into t timeslots (we remember that, in this case, t = 45) such that all of the hard constraints are satisfied.

In the next section we will describe an EA that uses this grouping theme to tackle the UCTP. The remainder of this paper is then set out as follows: in section IV we will provide some general details of the experimental set-up used in this study and, using this framework, will go on to look at the effects that the various genetic operators seem to have on the quality of the search in section V. Next, in section VI, we will go on to introduce a new way of measuring population diversities for this sort of representation, and will make some other general comments regarding this sort of algorithmic approach. In section VII we will then attempt to improve the algorithm: firstly, through the use of some new fitness functions and, secondly, via the introduction of a stochastic local-search operator (making, in effect, a grouping *memetic* algorithm). In particular, we will examine the good *and* bad effects that this operator can have, and will also consider the consequences of removing the grouping genetic operators altogether. Finally, in section VIII we will outline the main conclusions of this study and, in section IX, we will discuss some possible future research issues raised by these conclusions.

III. The Algorithm

A. Representation and Solution Construction

For this algorithm, each timetable is represented by a two dimensional matrix where rows represent rooms and columns represent timeslots [57]. Each place in the timetable (i.e. cell in the matrix) can be blank, or contain *at most* one event. Note that this latter detail thus allows us to disregard the second hard constraint of this problem. Additionally, in this approach we choose to not allow any event to be inserted into a place where it causes a violation of either of the two remaining hard constraints. Instead, extra timeslots are opened (i.e. columns are added to the matrix) in order to handle events that cannot be feasibly assigned to any place in the current timetable. (Similar schemes have also been used in other methodologies: see [6], [23], [24], and [27], for example.)

As we will see, a scheme for constructing full solutions from empty or partial solutions is vital in this sort of algorithmic approach, not only for building members of the initial population, but also for use with the grouping genetic operators (described below). The procedure *Construct* for achieving this is outlined in fig. 2, and takes, as arguments, an empty or partial timetable *tt* and a non-empty list of currently unplaced events *U*. Using the sub-procedure *Build*, events are

then taken one by one from U (according to some heuristics, defined in Table I), and inserted into places in the timetable that are between timeslots x and y and that are feasible. Events for which there are no feasible places are ignored. Eventually then, U will be empty (in which case we will have a complete timetable that may or may not be using the required number of timeslots), or it will only contain events that cannot be inserted anywhere in the current timetable. In the latter case, a number of new timeslots are opened, and the process is repeated on these new timeslots. The number of timeslots that are opened is calculated in line 6 of the *Build* procedure in fig. 2. Note that the result of this calculation represents a lower bound, because we know that a maximum of r events can be assigned to one particular timeslot, and therefore *at least* $\lceil |U| / r \rceil$ extra timeslots will be needed to accommodate the remaining events in U.

In order to form an initial population, the construction procedure is called for each individual. At each step, an event is chosen according to heuristic H_1 with ties being broken by H_3[2]. Next, a place is chosen for the event using heuristic H_4, with ties being broken by H_5 and further ties with H_6. By using H_4, we are making the seemingly sensible choice of choosing the place that will have the least effect on the future place options of the remaining unplaced events. Meanwhile, the use of heuristics H_3 and H_6 (random choices) in the initial population generator provides us with enough randomisation to form a diverse initial population.

```
Construct (tt, U)
1. if (len(tt) < t)
2.     Open (t – len(tt)) new timeslots;
3. Build (tt, U, 1, len(tt));

Build (tt, U, x, y)
1. while (∃ events in U with places in tt
          between timeslots x and y)
2.     Take an event e from U that has feasible
       places in tt;
3.     Pick one of these places and insert e;
4. if (U = ∅) end;
5. else
6.     Open ⌈|U| / r⌉ new timeslots;
7.     Build (tt, U, y, len(tt));
```

Fig. 2. The Construction Procedure: describing how a partial or empty timetable is converted back into a complete timetable. In this pseudo-code *tt* represents the current timetable and U is a list of unplaced events of length $|U|$. Additionally, t represents the target number of timeslots, len(*tt*) indicates how many timeslots are currently being used by *tt*, and r indicates the number of rooms.

(As an aside, it is worth mentioning that we also implemented and tested a second construction scheme that worked by opening timeslots in *tt* one by one, on the fly as soon as any event in U, due to preceding insertions, became unplaceable. However a detailed comparison of the two schemes revealed that the quality of the individual timetables produced by this second method was usually worse, and the cost of this process was significantly higher. This second issue is particularly important because, as we will see in the next section, a reconstruction scheme is also an integral part of the grouping genetic operators. We believe that this greater extra expense is due to the fact that, whilst looking for places for events in U, the *whole* timetable (which would be continually growing) needs to be considered, whilst in our construction scheme, while U remains non-empty, the problem is actually being split into successively smaller sub-problems.)

TABLE I.
THE VARIOUS EVENT AND PLACE SELECTION HEURISTICS USED WITH THE CONSTRUCTION PROCEDURE IN FIG 2

Heuristic	Description
H_1	Choose the event with the smallest number of possible places to which it can be feasibly assigned in the current timetable.
H_2	Choose the event that conflicts with the highest 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.

B. The Genetic Operators

Because, in this case, we have decided to consider the individual timeslots as the principal building blocks of the problem (section II), it follows that appropriate genetic operators should be defined so that these 'groups of events' can be propagated effectively during the evolutionary process. We chose to use the standard GGA recombination methodology (proposed in [50, 51]) modified to suit our particular needs and representation.

Fig. 3 depicts how we go about constructing the first offspring timetable using parents p_1 and p_2 with four randomly selected crossover points. A second offspring is constructed by switching the roles of the parents and the crossover points. What is important to note about this operator is that it allows the offspring to inherit complete timeslots (the structures that we consider to be the underlying building blocks of the problem) from *both* parent timetables.

Note that as a result of stage-two of recombination, there will be duplicate events in the offspring timetable. This problem could be corrected, for example, by going through the timetable, and removing all duplicate events from the timeslots that came from p_1. However, although such an operation would result in a valid and complete offspring timetable, it is likely that the offspring would actually be poor in quality because it would almost certainly be using more timeslots than either of the two parents, thus going against the general aim of the algorithm. We therefore choose to use the

[2] Note that H_1 is somewhat akin to the rule for selecting which node to colour next in Brélaz's classical Dsatur algorithm for graph colouring [58], although this particular heuristic also takes the issue of room allocation into account. This basic rule therefore selects events based on the state of the current timetable, prioritising those with the least remaining feasible options.

additional step of *adaptation* [59] to try to circumvent this issue. This process is described in stage-three of fig. 3 and, indeed, the same procedure has also been used with GGAs applied to bin packing in [50] and [51], and graph colouring in [6] and [55]. Finally, in stage-four of recombination, events that become unplaced as a result of the adaptation step are reinserted using the construction procedure (fig. 2) with heuristic H_1 being used to define the order in which events are reinserted (breaking ties with H_2 and any further ties with H_3). Places for events are then selected using the same heuristics as the initial population generator.

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

Finally, we also make use of an inversion operator. Similarly to the other GGAs already mentioned, this works by selecting two timeslots in a timetable at random, and then simply reversing the order of the timeslots contained between these. Note that inversion does not, therefore, alter the number of timeslots being used, or indeed the packings of events into these timeslots. However, it may assist recombination if promising timeslots are moved closer together, as this will improve their chances of being propagated together later on.[3]

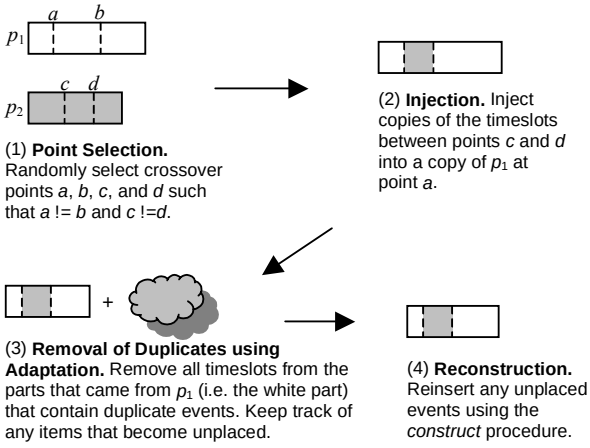


Fig. 3. The four stages of recombination: point selection, injection, removal of duplicates using *adaptation*, and reconstruction. Note that in order to construct the second offspring, copies of the timeslots between points *a* and *b* in p_1 are injected into a copy of p_2 at point *c*.

C. A Preliminary Fitness Measure

Finally, in this algorithm, we need a way of measuring a timetable's quality. In our case, since we are only interested in finding feasibility, a suitable measurement need only reflect the timetable's *distance-to-feasibility*. In general timetabling, this can be measured by taking various factors into consideration such as the number of broken constraints, the number of unplaced events, and so on. Of course, what *is* chosen should depend on the representation being used, and on user and/or algorithmic preference. As we stated earlier, in this algorithm we explicitly prohibit the violation of hard constraints and, instead, open up extra timeslots as and when needed. We could therefore simply use the number of extra timeslots as a distance-to-feasibility measure. However, such a method is likely to hide useful information because it would not tell us anything about the number of events packed into these extra timeslots. We therefore use a more meaningful measure that we calculate by carrying out the following steps. Let t represent the target number of timeslots and s represent the current number of timeslots being used in a timetable:

1. Calculate the number of extra timeslots t' being used by the timetable (where $t' = s - t$)
2. Identify the t' timeslots with the least events in them
3. Total up the number of events in these t' timeslots.

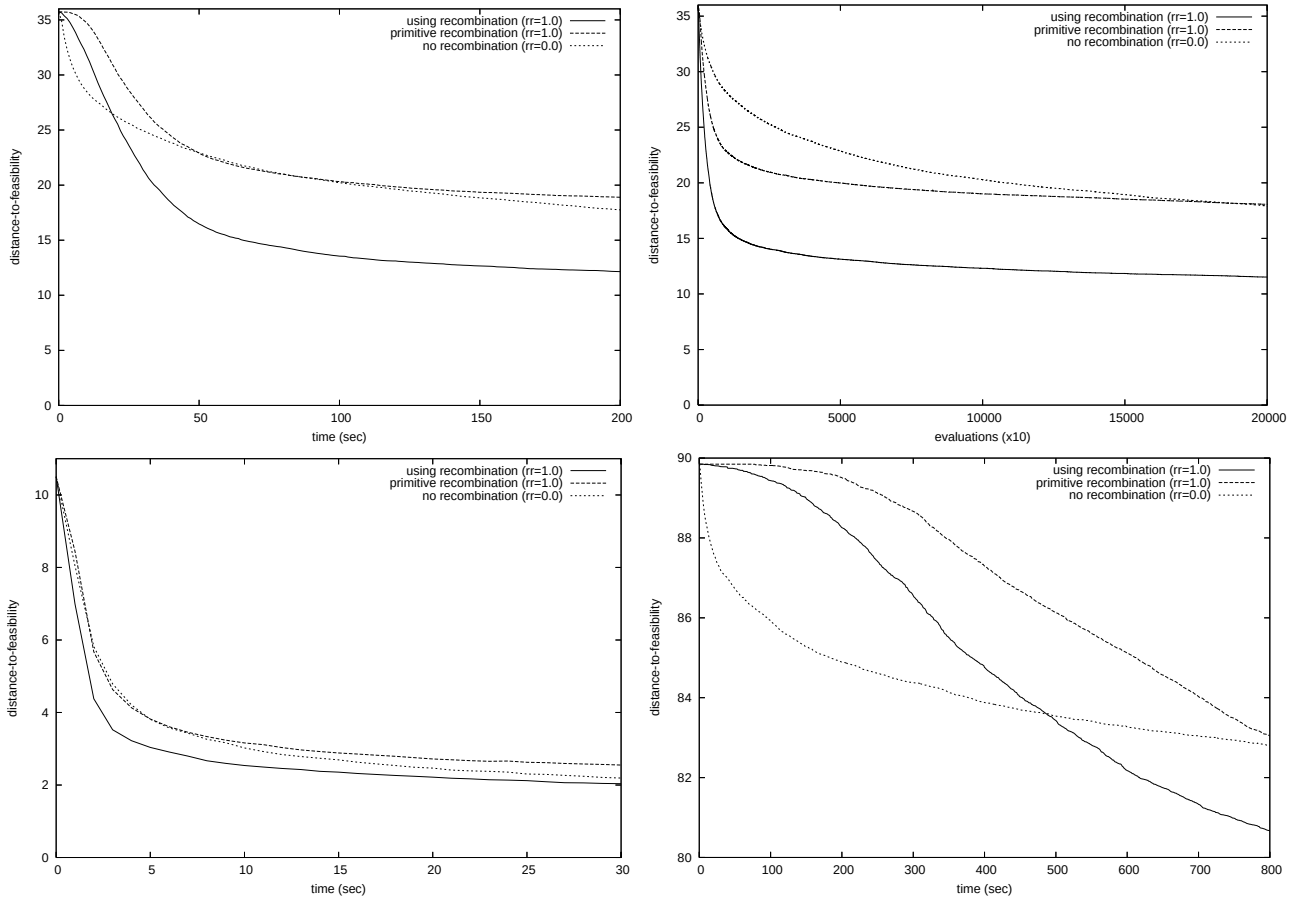
We may also think of this measure as the minimum number of events that would need to be removed from the timetable in order to bring the number of timeslots down to the required amount. Obviously, a fully feasible timetable has a distance-to-feasibility of zero.

IV. Experimental Set-Up

As it turned out, our initial tests showed that many existing benchmark instances (on the web and otherwise) could be easily solved by this algorithm. For example, feasible solutions were usually found in initial populations when using the instances at [43].[4] Although this highlights the strength of our construction scheme, unfortunately it tells us very little about the other characteristics of the algorithm. We therefore set about making some new problem instances. These were created with no reference to this algorithm, but were deliberately intended to be troublesome for finding feasibility. This was achieved by simple experiments whereby instances were created and run on two existing constructive algorithms, reported in [47] and [60]. Only instances that both of these algorithms struggled with were considered for inclusion in the instance set. Indeed, given excess time, these two algorithms were generally unable to place around 20% to 40% of the events. All in all, we made three sets of twenty instances: the small set, the medium set and the large set, with sizes of approximately 200 events and 5 rooms, 400 events and 10 rooms, and 1000 events and 25 rooms respectively. Further details, including a description of how the instances were generated, plus the instances themselves, can be found online at [44]. Note that all instances have at least one solution where

[3] It is probably worth noting that we could have actually implemented a uniform grouping crossover operator here as opposed to the two-point variant explained above. However, in this case we decided to keep all the genetic operators within the GGA design-guidelines specified by Falkenauer in [51].

[4] Note, however, as we mentioned in section II, these benchmark instances were created with soft constraints in mind.



Figs. 4 (a), (b), (c), and (d). (Top left, top right, bottom left, and bottom right respectively). Showing the behaviour of the algorithm with and without recombination over time (except for (b), which is with respect to evaluations). The meaning of 'primitive recombination' is explained in the text. Figs (a) and (b) show runs with the medium instances, (c) with the small instances, and (d) with the large instances. Each line represents, at each second, the distance to feasibility of the best solution in the population, averaged across 20 runs on each of the 20 instances (i.e. 400 runs) using $mr = 2$, $ir = 4$, and $\rho = 50$.

the events can be feasibly packed into the target number of forty-five timeslots.

We also imposed certain time limits on our algorithm that we considered to be fair for these sizes of problem. These were 30, 200 and 800 seconds of CPU time for the small, medium and large sets respectively, on a PC under Linux, using 1GB RAM and a Pentium IV 2.66Ghz processor.

For all experiments, a steady-state population (of size ρ) using binary tournament-selection was used: at each step of the algorithm, two offspring are produced by either recombination or replication[5] (dictated by a recombination rate *rr*). Next, the offspring are mutated (according to the mutation rate *mr* – see section III), and finally reinserted into the population, in turn, over the individuals with the worst fitness. If there is more than one least-fit individual, a choice between these is made at random. Additionally, at each step a few (*ir*) random individuals in the population are selected to undergo inversion.

V. The Effects of Recombination

For our first set of experiments, we looked at the general effects of the recombination operator by comparing runs using recombination (with a rate $rr = 1.0$) against runs that used none ($rr = 0.0$). Results are depicted in figures 4(a)-(d). If we look first at the results for the medium instances in fig. 4(a), we can see that after an initial lag period of around 20 seconds (where using no recombination provides quicker movements through the search space) a use of our recombination operator benefits the search significantly.[6]

Note however, that such a simple comparison on its own is not completely fair because, as the reader may have noticed, the heuristics used for reconstruction with our recombination operator are different to those used with mutation (and therefore it might be the heuristics doing the work, and not the fact that the recombination operator is successfully combining useful parts of different solutions). Thus, we also include a third line that again uses recombination with a rate 1.0, but also uses the more primitive reconstruction heuristics used by the mutation operator (this line is labelled 'primitive recombination' in the figures). In fig. 4(a) we can see that, in this case, the presence of this primitive recombination

[5] An offspring made via replication is simply a copy of its first parent.

[6] In this paper, we use the word 'significant' to indicate that a Wilcoxon signed-rank test showed that results found at the time limit came from a different underlying distribution with probability ≥ 95%.

operator actually seems to hinder the search with regards to time. However, it might also make sense to observe this behaviour from a second perspective: in timetabling, due to the large number of possible constraints that can be imposed on a particular problem, it can often be the case that the evaluation function might actually become the most costly part of the algorithm, particularly when soft constraints are also being considered. If we now look at these same runs, but with regards to the number of evaluations[7] (fig. 4(b)), we can see that according to this criterion, use of this more primitive recombination, up until around 150,000 evaluations, is clearly beneficial. We also see once more that the more advanced recombination operator provides the best search. This difference was also significant.

With the small and large instance sets, meanwhile, we noticed different behaviours. In the case of the small instances, the algorithm generally performed well across the set, and seemed quite insensitive to the various parameter settings (and whether recombination was being used or not). Indeed, although fig. 4(c) indicates a *slightly* better search when using recombination, this difference was small and not seen to be significant. As a matter of fact, in our trials, optimal solutions were regularly found to over half of the instances within the time limit, making it difficult to draw any interesting conclusions other than the fact that performance of the algorithm with these instances was generally quite good.

Finally, with the large instances, yet another type of behaviour was observed. Looking at fig. 4(d), we can see that the use of recombination in these cases seems to drastically slow the search. Indeed, no benefits of recombination can actually be seen until around 500 seconds. Clearly, if we were using shorter time limits, the operator might therefore hinder rather than help. Secondly, if we look at the scale of the *y*-axis in fig. 4(d), we can see that, in fact, only small improvements are actually being achieved during the entire run; indeed, considering all problem instances used in the tests are known to have at least one optimal solution (with respect to the hard constraints) these improvements are disappointing.

The above observations immediately suggest that instance size is an important factor in the run characteristics of this algorithm in terms of both timing implications and general progress made through the search space. In the next section we will present some ideas as to why this might be so.

VI. Scaling-up Issues with the GGA

A. Measuring Diversity

Before describing some of the possible scaling-up issues of this algorithm, it is first necessary to introduce a diversity measure for the grouping representation. Because the concepts that will be described in this section apply to grouping problems as a whole, we will use the more generic terms 'groups' and 'items' in our descriptions, as opposed to 'timeslots' and 'events', which, of course, only apply to timetabling problems.

As we have discussed, the grouping representation admits two important properties: chromosomes are variable in length (in that they can contain varying numbers of groups), and the ordering of the groups within the chromosomes is irrelevant with regards to the overall solution being represented. Unfortunately, these characteristics mean that many of the usual ways of measuring population diversity, such as Hamming distances [61] or Leung-Gao-Xu diversity [62], are rendered inappropriate. Additionally, in the case of this timetabling problem, we believe that it would be misguided to use diversity measures based on population fitness information (such as the standard deviation etc.), because in our experiences it can often be the case that minor changes to a timetable might actually result in large changes to its fitness and, inversely, two very different timetables can often have a similar fitness.

We believe that a suitable diversity measure for this representation, however, can be obtained from the 'substring-count' method of Mattiussi, Waibel, and Floreano, recently presented in [63]. In the grouping representation, it is worth considering that each group can only occur *at most once* in any particular candidate chromosome (otherwise the solution would be illegal because it would contain duplicates). Given a population P, a meaningful measurement of diversity might therefore be calculated via the formula:

$$div(P) = \rho\left(\frac{m}{n}\right) \tag{2}$$

where ρ is the population size, m is the number of different groups in the population, and n is the total number of groups in the population. Using this measurement, a homogenous population will therefore have a diversity of 1.0, whilst a population of entirely distinct individuals (where none of the individuals have an equivalent grouping of items) will have a diversity of ρ.

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

$$dist(p_1, p_2) = 2\left(\frac{x}{y}\right) - 1 \tag{3}$$

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

B. Diversity, Recombination and Group Size

In our experiments, we often noticed that evolution was slow at the beginning of a run, but then gradually sped up as the run progressed. These characteristics were particularly noticeable with the larger instances where we saw new individuals being produced at a somewhat unsatisfactory rate for quite a large proportion of the run. Investigations into this

[7] In this paper, each time a new individual is produced, we consider this as one evaluation.

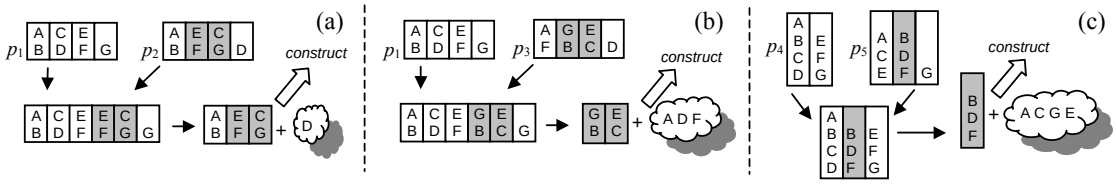


Fig. 5. Demonstrating how diversity and group size can influence (1) the amount of reconstruction needed, and (2) the number of groups that are lost, using the standard grouping recombination operator.

matter revealed that this was due to the fact that the recombination operator usually tended to be more costly at the start of a run and then gradually became less expensive as evolution progressed. Further investigations revealed that this added expense seemed to be influenced by two factors: population diversity, and the sizes of the groups in the candidate solutions.

Fig. 5 shows three examples of the grouping recombination operator in order to illustrate these concepts. In fig. 5(a), the distance between candidate solutions p_1 and p_2 is comparatively small (i.e. 2*(6/8)–1 = 0.5) and only one of the seven items becomes unplaced during the recombination operation. In fig. 5(b) however, although the number of groups and items being injected is the same as fig. 5(a), the distance between p_1 and p_3 is larger (i.e. 2*(8/8)–1 = 1.0); consequently, the duplicate items resulting from the injection are spread across more of the groups, meaning that a greater number of the groups coming from p_1 need to be eliminated. This also means that more items have to be dealt with by the reconstruction process, making the overall procedure more expensive.

Next, in fig. 5(c) we demonstrate the effects that larger groups have on the recombination operator. In this case, the size of the problem may be considered the same as the previous two examples, because we are still only dealing with seven items. However, this time the size of the groups is larger and, as can be seen, the injection of a group from p_5 into p_4 causes a high proportion of the items to become unplaced during stage three of recombination.

In fact, figs. 5(b) and 5(c) depict cases of what we will term a *unilateral recombination*: after the injection stage, all of the groups coming from the first parent have contained a duplicate and have therefore been eliminated. Thus, the resultant offspring does not actually end up containing building blocks from both parents (as is usually desirable), but will instead be made up of some groups from the second parent, with the rest having to be formed, from scratch, by the reconstruction process. In this sense, it might be considered more of a macro-mutation operator than anything else.

In order to further illustrate these concepts, consider figures 6(a)-(c), which show details of an example run of our GGA with a small, medium, and large problem instance respectively. In all three figures it can be seen that as evolution progresses, the level of diversity in the populations generally falls. This, of course, is typical of an evolutionary algorithm. We also see in these figures that the proportion of items (events) becoming unplaced during recombination mirrors this fall very closely, thus highlighting the strong relationship of the two measurements. However, the other noticeable characteristic in these figures is the way in which high levels of both of these measures are sustained for longer periods when the instance sizes (and therefore the number of events/items per timeslot/group) are larger. Looking at fig. 6(c), for example, we can see that no real drop in either measurement actually occurs until around 180,000 evaluations and, up until this point, over half of the items (events) are becoming unplaced, on average, with every application of the recombination operator.

We believe that this latter phenomenon is caused by the fact that because in this case the groups *are* larger, the potential for losing the groups coming from the first parent is increased (as illustrated in fig. 5(c)). We may therefore view this as a more *destructive* recombinative process. Of course, not only does this make the operation more expensive (because a greater amount of reconstruction has to be performed), it also means that it is more difficult for the GGA to successfully combine and pass on complete groups from one generation to the next. Thus, it would seem that, in these cases, the recombination operator is indeed becoming more of a macro-mutation operator, and a key component of the evolutionary algorithm might be being compromised.

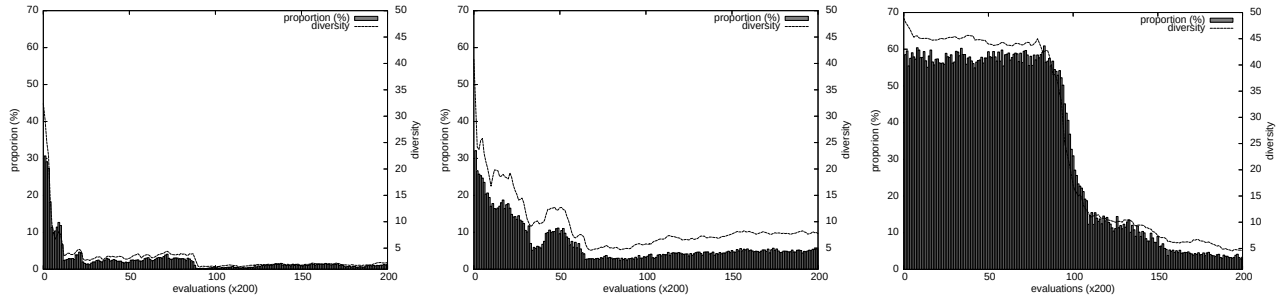
It is also worth noting that in cases where the recombination operator *is* behaving in a more destructive manner, this will generally mean that more groups in an offspring will occur as a result of the reconstruction process (as opposed to being inherited from an ancestor). Unfortunately, however, this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

C. Group Size and Chromosome Length

As a final point, if we refer back to fig. 3 (which demonstrates the standard GGA recombination operator applied to this timetabling problem), we note that the total number of possible values that can be chosen for crossover points c and d (that is, the total number of group-combinations that can be selected for injection from the second parent into the first parent) is exactly:

$$\frac{s(s-1)}{2} \tag{4}$$

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



Figs. 6 (a)-(c) (Left to right.): Example runs with a small, medium, and large instance respectively, demonstrating (1) the close relationship between diversity and the amount of reconstruction needing to be done with the recombination operator, and (2) the differing ways that the two measures vary during the runs as a result of the different sized timeslots/groups. (All runs using rr = 1.0, ir = 4, mr = 2, ρ = 50.) . In the figures, the amount of reconstruction being done with the recombination operator is labelled *Proportion (%)* – this refers to the percentage of events that are becoming unplaced, on average, with each application of the recombination operator.

$$\frac{s!}{m!(s-m)!} \tag{5}$$

where m is the number of groups chosen to be removed from the chromosome, and s represents the length of the chromosome being mutated.

However, unlike in many forms of evolutionary computation where chromosome length is defined by the size of the problem instance being dealt with (e.g. [1], and [64])[8]; with this sort of algorithm and representation, chromosome length is actually defined by the number of groups being used. Of course, this means that when dealing with instances that use larger groups of items, chromosomes will be proportionately shorter, and the values returned by equations (4) and (5) will, in turn, be lower.

The implications of these facts are particularly apparent with this timetabling problem. Here, the number of timeslots being used defines chromosome length, and it is the number of events and rooms that defines problem size. However, given that our aim is to feasibly arrange the events into forty-five timeslots, this means that the lengths of the chromosomes will remain more or less constant, regardless of the problem size. Unfortunately, in practice this means that an increase in instance size will not only cause the timeslots to be larger (resulting in the unfavorable situations described in the previous subsection), it also implies that the potential for the genetic operators to provide sufficient exploration of the search space might also be more limited.

VII. Improving the Algorithm

Having now introduced a GGA for the UCTP, and having also highlighted some general issues relating to this type of approach, in this section we will now look at two separate ways that we might go about improving the algorithm: firstly, through the use of some new fitness functions and, secondly, via the introduction of an additional local-search operator [65].

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

A. Fitness Function Analysis

A central aspect of any evolutionary algorithm is the way in which candidate solutions in the population are evaluated against each other. Ideally, a good fitness function must convey meaningful information about the quality of a candidate solution and should also encourage the search into promising areas of the solution space. For many problems in operational research, a suitable measure is suggested naturally by the problem at hand (e.g. the travelling salesman problem [64]). In others, it is not so easy. For example, in [51] Falkenauer looks at the bin packing problem and suggests that while the most obvious way of measuring a candidate solution's fitness is to just calculate the number of bins being used (with the aim of minimisation), this is actually unsuitable because it will likely lead to a very inhospitable fitness landscape where 'a very small number of optimal points in the search space are lost in the exponential number of points where this purported cost is just one above this optimum. Worse, these slightly sub-optimal points [all] yield the same cost.' This could, for example, lead us to a situation where we might have a diverse population, but all members appear to have the same fitness. In this situation, not only would selection pressure be lost, but also if all the scores were indeed one away from the optimum, any move from near-feasibility to full-feasibility would be more or less down to chance.

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

$$f_1 = \frac{1}{1+s} \tag{6}$$

$$f_2 = \frac{1}{1+d} \tag{7}$$

$$f_3 = \frac{1}{1 + d + (s - t)} \quad (8) \qquad f_4 = \frac{\sum_{i=1}^{s} (E_i / r)^2}{s} \quad (9)$$

$$f_5 = \frac{\sum_{i=1}^{s} (C_i)^2}{s} \quad (10) \qquad f_6 = \frac{\sum_{i=1}^{s} (S_i)^2}{s} \quad (11)$$

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

Function f_3 is basically the same as Eiben *et al.*'s fitness function for graph colouring in [55]. It uses the observation that if two timetables have the same value for d, then the one that uses the least number of extra timeslots is probably better and, similarly, if two timetables have the same number of extra timeslots, then the one with the smallest value for d is probably better.

Functions f_4, f_5, and f_6, meanwhile, judge quality from a different viewpoint and attempt to place more emphasis on the individual timeslots. Thus, timetables that are made up of what are perceived to be promising timeslots (i.e. good *packings* of events) are usually favoured because their fitness will be accentuated by the squaring operations. The three functions differ, however, in their interpretations of what *defines* a good packing: function f_4 tries to encourage timetables that have timeslots with high numbers of events in them, and is similar to the fitness function suggested by Falkenauer for bin packing [51]; function f_5, meanwhile, uses the well-known heuristic from graph colouring [58] that recommends colouring as many nodes (events) of high degree as possible with the same colour (this function was originally proposed by Erben in [6]); finally, function f_6 attempts to promote timetables that contain timeslots with large total numbers of students attending some event in them – following the obvious heuristic that if many big events are packed into one timeslot, then other smaller (and presumably less troublesome) events will be left for easier packing into the remaining timeslots.

As a final point, it is worth noting that functions f_2 and f_3 need to know in advance the target number of timeslots. If this is undefined, the task of calculating the minimum number of timeslots needed to accommodate the events of a given instance is equivalent to calculating the chromatic number in graph colouring. However, computing the chromatic number is itself, an NP-hard problem. In practical course timetabling, however, this detail is probably unimportant because it is typical for the university to specify the target number of timeslots in advance.

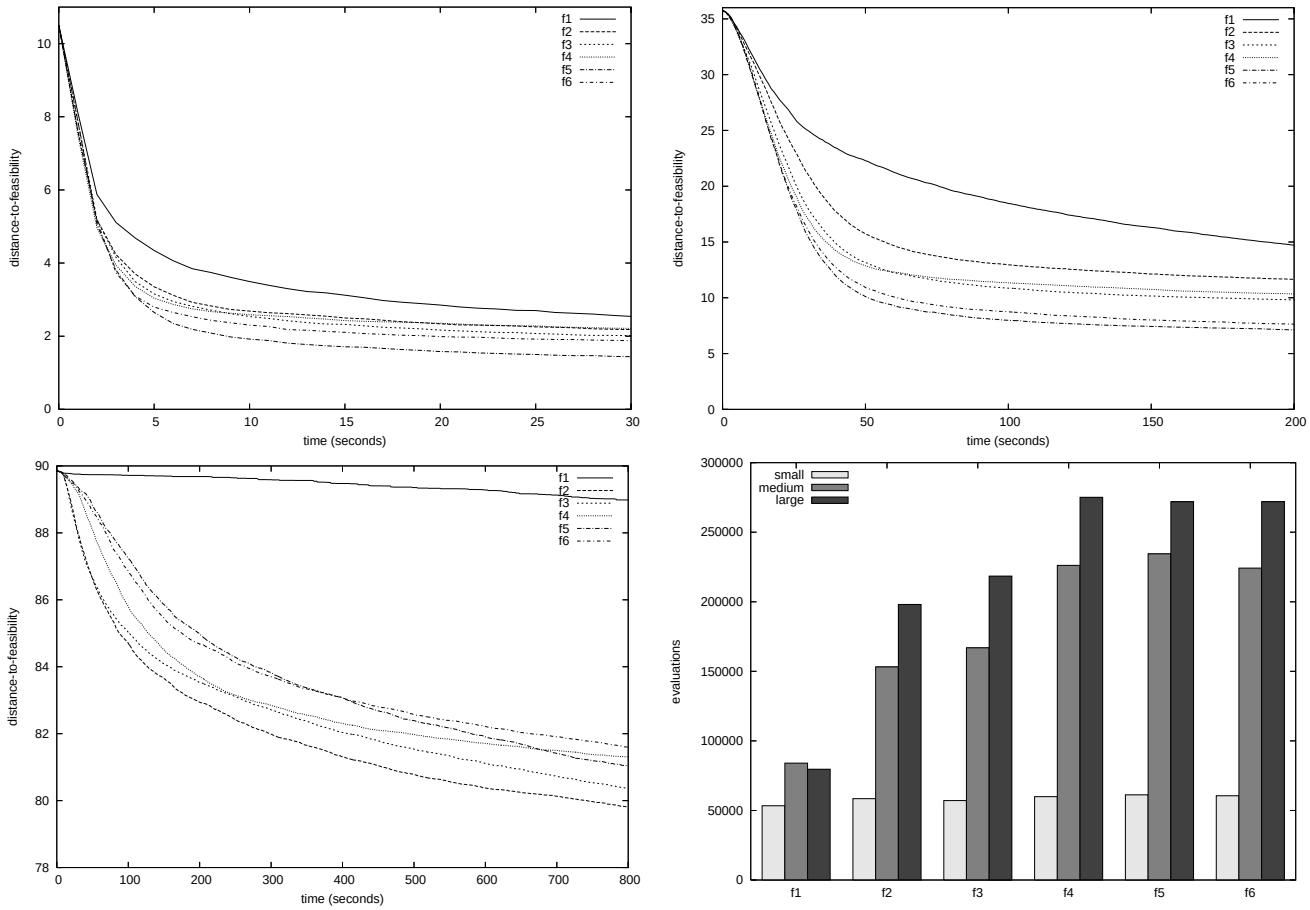
B. Experimental Observations

To investigate the effects of these six fitness functions, we performed tests using the same steady-state population scheme as before (section IV), and simply altered the fitness functions for each set of trials. Note then, that the only *actual* difference between the trials is the criteria used for choosing tournament winners, and picking the individuals to replace. Note also, that the computational costs of the fitness functions are roughly equivalent, as all require just one parse of the timetable.

Figures 7(a)-(c) show how the algorithm responds to the six fitness functions over time with the different instance sets. If we first draw our attention to figures 7(a) and (b), we can see that with regards to the small and medium instances, f_5, and then f_6, clearly give the best searches (on average), with respect to both the speed of the search *and* the best solutions that are found, with f_1 and then f_2, as expected, providing the worst. We believe this is due to the reasons mentioned above: when using f_5 and f_6 (and to a lesser extent, f_3 and f_4) the algorithm is able to distinguish between solutions that, according to f_1 or f_2, might be valued the same. Thus, selection pressure is able to remain for a greater duration of the run. Furthermore, it would appear that the heuristic criteria that f_5 and f_6 use to make these distinctions (described above), is indeed conducive to the search. Indeed, in both cases, the improvements that f_5 and f_6 provided were significant.

Interestingly, we see that the algorithm responds differently to the fitness functions when considering the large instances (fig. 7(c)). As before, we see that f_1 is clearly the worst, but we also see that the characteristics of the remaining five fitness functions is now more or less reversed, with f_2 providing the best performance. This could be because the squaring functions used with f_4, f_5, and f_6 cannot accentuate the characteristics of a good timeslot as much as when used with the other instances (which have fewer events per timeslot). However, most importantly one has to look again at the scale of the y-axis of fig. 7(c) to appreciate that the algorithm, again, actually performs fairly badly with *all* of the fitness functions. Additionally, if we ignore f_1, the differences between the other five fitness functions were not actually seen to be significant.

Fig. 8 also shows some intriguing results of these experiments. As can be seen, when considering the medium and large instance sets, the number of evaluations performed within the time alters drastically depending on which fitness function the algorithm is using. (This pattern also emerges with the small instances, but the distinction is more difficult to make in the figure.) This, we believe, is due to the concepts described above: because the more fine-grained fitness functions (f_4, f_5, and f_6) are able to distinguish between timetables that the other three fitness functions might see as identical, selection pressure remains for a greater part of the run. This, in turn, means that diversity falls at a higher rate for a longer period, resulting in a less expensive recombination process. For the same reason, the most coarse-grained fitness function f_1 also performs by far the fewest evaluations within the time limits. Note that an overly rapid loss of diversity may



Figs. 7(a)-(c) (top-left, top-right, and bottom-left respectively), and figure 8 (bottom-right). Figures 7 (a)-(c) show the effects of the six different fitness functions over time with the small, medium, and large instances respectively. Each line represents, at each second, the distance to feasibility of the best solution in the population, averaged across 20 runs on each of the 20 instances (i.e. 400 runs), using $\rho = 50$, $rr = 1.0$ (0.25 with 7(c)), $mr = 2$, and $ir = 4$. Fig. 8 shows the number of evaluations performed within the time limits (specified in section IV), when using the various fitness functions, and the different sized instances.

sometimes be undesirable in an EA as it might lead to a redundancy of the recombination operator and an under-sampling of the search space. However, in the case of this algorithm there is clearly a trade-off because, as noted, a high amount of diversity can cause recombination to be both expensive and destructive. With the small and medium instances, the trade-off seems to fall in favour of using f_5 and f_6 which, although exhibiting tendencies to lose diversity at a quicker rate, still both return superior results and in less time.

As a final point, it is worth considering some practical implications of these fitness functions. In some real world timetabling problems there may be some events that every student is required to attend (such as a weekly seminar or assembly). Clearly, such an event must be given its own exclusive timeslot, as all other events will clash with it. However, fitness function f_4 will, unfortunately, view this as an almost empty (or badly packed) timeslot and will penalise it, therefore possibly deceiving the algorithm. Fitness functions f_5 and f_6, however, will reward this appropriately. Conversely, f_5 for example, as well as being shown to be less favourable with the larger instances, currently bases its judgement of a timeslot's quality on the total degree of the events within it (with higher values being favoured). However, this criterion is only a heuristic, and it is possible that counter examples could be encountered.

C. *Introducing Local-Search Techniques*

It is generally accepted that EAs are very good at coarse-grained global search, but are rather poor at fine-grained local-search [66]. It is therefore perfectly legitimate (and increasingly common) to try to enhance an EA by adding some sort of local-search procedure. This combination of techniques is commonly referred to as a *memetic* algorithm (see the work of Moscato [67, 68], for example), and the underlying idea is that the two techniques will hopefully form a successful partnership where the genetic operators move the search into promising *regions* of the search space, with the local-search then being used to explore *within* these regions.

Looking at some other algorithms from this problem domain, both Dorne and Hao [69], and Galnier and Hao [70] have shown how good results can be found for many graph colouring instances through the combination of these techniques. In both cases, specialised recombination operators were used, with tabu search then being utilised to eliminate

cases of adjacent nodes having the same colour. Rossi-Doria *et al*. have also used similar techniques for timetabling in [35], where the more global operators (such as uniform-crossover) are complemented by a stochastic first-improvement local-search operator which, as one of its aims, attempts to rid the resultant timetables of any infeasibility.

As a matter of fact, it turns out that none of these three methods are actually suitable for our algorithm, because they are intended for eliminating *violations* of hard constraints, which in our described methodology, we explicitly disallow. Indeed, a suitable local-search procedure in this case should, instead, be able to take a timetable with no hard constraint violations, and somehow find a timetable that is hopefully better, but still with no violations. With regards to other grouping-based algorithms that have used of this sort of representation, but have also made use of an additional local-search technique, Falkenauer's hybrid-GGA [71], and Levine and Ducatelle's ant algorithm [72] (both for bin packing) were both reported to return substantially better results when their global-search operators were coupled with a local-search method. Their techniques were inspired by Martello and Toth's dominance criterion [73] and worked by taking some unplaced items, and then attempting to swap some of these with items already in existing bins so that (a) the bins became more full, but (b) the number of items in the bin stayed the same (i.e. each item was only replaced by a bigger item).

However, even though such dominance criterion does not strictly apply in our timetabling problem, we can still try to define a similar operator that will attempt to improve the packings of events into the timeslots. Such an operator is defined in fig. 9 and, by taking a list unplaced events and a partial timetable (U and tt respectively), it operates by repeatedly trying to take events from U and insert them into free and feasible places in tt.

```
LocalSearch (tt, U, limit)
1. Make a list V containing all the places in tt that
     have no events assigned to them;
2. i = 0;
3. while (U ≠ ∅ and V≠∅ and i < limit)
4.    foreach(u∈U and v∈V)
5.       if (u can be feasibly put into v in tt)
6.          put u into v in tt;
7.          remove u from U and v from V;
8.    if (U≠∅ and V≠∅)
9.       repeat
10.         choose random event e in tt and v∈V;
11.         if(e can be feasibly moved to v in tt)
12.            move e to v;
13.            update V to reflect changes;
14.         i++;
15.      until (i ≥ limit or e has been moved to v)
```

Fig 9. The local-search procedure: In this pseudo code, *tt* represents a partial timetable, *U* is a list of unplaced events, and *limit* represents the iteration limit of the procedure.

Note that this operator will have two important effects. First, while not allowing the number of events contained in *tt* to decrease, if it *is* successful then events will be taken from *U* and added to *tt*, thereby improving its overall timeslot packings. Second, because the events and free spaces within *tt* will be randomly shuffled amongst the timeslots (lines 9-15 of fig. 9), diversity will be added to the population.

In our experiments, we used the local-search operator in conjunction with our mutation operator. As before, each time a mutation occurs, a small number of timeslots are randomly selected and removed from the timetable. The events in these timeslots now make up the list of unplaced events U and the local-search procedure is applied. If U is non-empty when the iteration limit is reached, then the construction scheme (figure 2) is used to insert the remaining events. (Here, the value for *limit* – that is, the iteration limit of the procedure – was made proportionate to the size of the instance being solved. Thus, we used the parameter l such that $limit = l*e$; where e is the number of events in the problem instance.)

D. Experimental Observations

With regards to algorithm performance, the introduction of this operator now presents another trade-off: a high amount of local-search might not allow enough new individuals to be produced within the time limit (and will presumably result in too little global search), whilst too little local-search might result in an inadequate exploration of the search-space regions that the global operators have brought us to.

To investigate these implications, we empirically carried out a large number of trials on the three instance sets, using various different recombination rates *rr*, settings for l, mutation rates *mr*, and population sizes ρ. In all trials, fitness function f_5 from the previous section was used, as well as the same steady-state population scheme described in section IV.

The first thing that we noticed in these experiments was the dramatic effect that the use of local-search had on the number of new individuals produced within the time limits. This is illustrated for the three instance sets in fig. 10. Here, we see that the introduction of local-search, even in small amounts, causes a dramatic decrease in the number of new individuals produced within the time limits. Although we believe that this is partly due to the obvious fact that the local-search procedure is adding extra expense to the mutation operator, we believe that the main reason is due to the fact that, because the local-search operator helps to maintain diversity in the population, this therefore causes the recombination operator to remain more expensive for a greater part of the run. As fig. 10 shows, this is especially so for the medium and large instances. Secondly, in these experiments we also saw the GGA respond differently to the various parameter settings when dealing with the different instance sets. A short summary of these differences now follows (example runs can also be seen in figures 11(a)-(c)):

- With the small instances, the best parameter settings generally involved using small populations with high amounts of local-search and a very small (but still present) amount of recombination. (The best results were gained

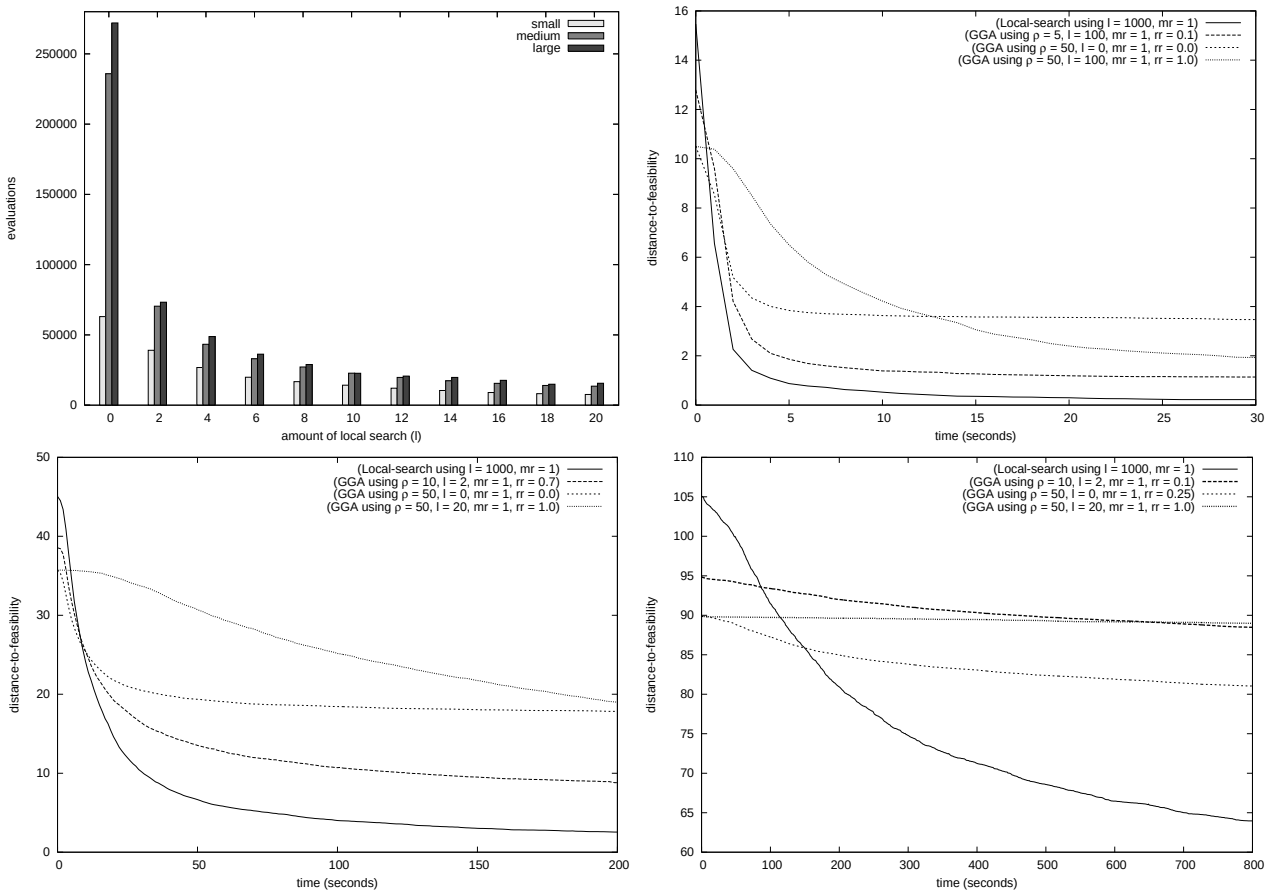


Fig. 10 and figs 11 (a)-(c). Figure 10 (top-left) shows the influence that various amounts of local-search have on the number of evaluations performed within the time limits for the different instance sets (using $\rho = 50$, $rr = 0.5$, $mr = 2$, and $ir = 4$). Figs. 11 (a)-(c) (top-right, bottom-left and bottom-right), show (1) the effects of various parameter settings with small, medium, and large instances respectively; and (2) a comparison between these and the local-search algorithm described in section VII. Each line represents, at each second, the distance to feasibility of the best solution in the population, averaged across 20 runs on each of the 20 instances (i.e. 400 runs). Note, because different population sizes are being used, the lines may not start at the same point on the *y*-axis.

when using $\rho = 5$, $l = 100$, $mr = 1$ and $rr = 0.1$.)

- With the medium instances, the best parameter settings for the GGA involved using small populations, with small (but still present) amounts of local-search, and a fairly high rate of recombination. (The best results were given by the parameters $\rho = 10$, $l = 2$, $mr = 1$ and $rr = 0.7$.) An increase in any of these parameters caused the search to become much slower, particularly for increases in l, which would simply maintain too much diversity in the population, thus keeping the recombination operator destructive and expensive. On the other hand, decreases in any of these parameters tended to cause an earlier stagnation of the search.
- With the large instances, the best results of the GGA were gained when using big populations with small amounts of recombination, and no local-search. (The best results were given using $\rho = 50$, $l = 0$, $mr = 1$, and $rr = 0.25$.) In particular, runs that used both local-search *and* recombination always provided disappointing searches because the diversity offered by the local-search would generally cause the recombination operator to do more harm than good. Thus, the best results were generally gained when we ensured that many regions of the search space were sampled (by using larger populations) with the majority of the downhill movements then being attempted by the less destructive mutation operator.

E. The Contribution of the GGA Operators

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

We dealt with this question by implementing a new algorithm that operated by making just one initial timetable (in the same way as described in section III), and then by repeatedly applying the mutation operator incorporating local-search until the time limit was reached. In the following explanations, this algorithm will be referred to as the 'local-search algorithm', and comparisons of this and the GGA are provided in figs 11(a)-(c).

Considering the large problem-instances first (fig. 11(c)), the presence of the local-search algorithm now allows us to

view, in context, some of the negative effects of the GGA operators. As can be seen, the local-search algorithm – which, we note, does not use a population, recombination, or selection pressure – clearly gives the best results. (Statistical tests showed these differences to be significant.) This may well be due to the various issues raised in section VI – not least the observation that when the groups in candidate solutions are large, the genetic operators seem to be less beneficial to the overall search. Note that there are also some important differences between these two algorithms. Firstly, because the GGA requires that a population of individual timetables be maintained, computation time generally has to be shared amongst the individual members. In the case of the local-search algorithm this is not so. Additionally, the GGA operators of replication and recombination generally have to spend time copying chromosomes (or part of chromosomes) from parent to offspring, which in the space of an entire run, could amount to a consequential quantity of CPU time. Again, with the local-search algorithm, this is not necessary. Differences such as these might thus offer advantages to the local-search algorithm where more effort can be placed upon simply trying to improve just the one timetable. Indeed, if the local-search operator is not particularly susceptible to getting caught in local optima (as would seem to be the case here) then this may well bring benefits.

Finally, moving our attention to figs. 11(a) and 11(b), we can see that the local-search algorithm also seems to outperform the GGA when dealing with the small and medium problem-instances. Indeed, in our experiments the differences in both cases were also seen to be significant, thus demonstrating the superiority of the local-search algorithm in these cases as well. This superiority, presumably, is due to the same factors as those described in the previous paragraphs. However, it is worth noting that the differences in results do seem to be less stark than those of the large instances, hinting that the GGA is perhaps able to be more competitive when the timeslots are smaller in size. This also agrees with the arguments size given in section VI. (The interested reader can find a complete breakdown of these results at [44].)

VIII. Conclusions

In this paper, we have examined a number of different issues regarding GGAs and their applicability to the university course timetabling problem. We now summarise the main findings of this work:

- We have taken a well-studied version of the UCTP and noted that the task of satisfying the hard constraints is a type of grouping problem. Consequently, using the guidelines suggested by Falkenauer [51], we have designed a GGA for the problem that combines standard GGA operators with powerful construction heuristics. We have observed that recombination can aid the evolutionary search in some cases, whilst in others, depending on the run time available, it might be more of a hindrance.
- We have introduced a way of measuring population diversity and distances between pairs of individuals for the grouping representation. We have seen that diversity and group size can influence (a) the overall expense of the recombination operator, and (b) the ease in which building blocks are combined and passed from one generation to the next. We have also noted that there may be other issues with this type of representation, due to the fact that larger groups will cause chromosomes to become proportionally shorter in length. While we still believe that it is indeed the groups that encapsulate the underlying building blocks of this type of problem, in this study we have thus highlighted areas where the propagation of these building blocks can be problematic.
- We have examined ways that the performance of this GGA might be improved through the introduction of a local-search operator and a number of different fitness functions. In particular, we have seen that, in some cases the more *fine-grained* fitness functions (such as f_5 and f_6) can produce significantly better results, but in other instances this is not so. We have also seen that the introduction of a stochastic local-search operator to this GGA *can* improve results, but probably needs to be used with care, as in some cases its use can mean that (a) not enough new individuals are produced within the time limits, and (b) the added diversity that it brings can cause the recombination operator to be too destructive and expensive.
- Finally, given such improvements, in the vast majority of cases we have seen that this GGA is still actually outperformed by our more straightforward local-search algorithm, which does not make use of a population, selection pressure, or the grouping recombination operator. The superior performance of this algorithm is particularly marked in the large problem-instances where, due to the larger groups, we believe the GGA operators display the least potential for aiding the search. This backs up our hypotheses regarding the pitfalls of the GGA approach in certain cases.

Due to the fact that we were obliged to make some new instance sets for this problem, we have been unable to provide comparisons of these algorithms with other approaches. However, these instances can be found on the web at [44] and we invite any other researchers interested in designing algorithms for timetabling to download them for use in their own experiments.

IX. Discussion

Finally, we round off this paper by making some other general comments on the findings of this work, and offer some suggestions for future research.

One of the main themes in this paper has been the observation that the GGA does not seem to perform well as the timetabling instance size (and therefore timeslot size) increases. From a practical point of view, this scaling-issue is particularly important, as it is not uncommon in universities to have a few thousand or more events that need to be scheduled into a limited number of timeslots (see the problem instances

used in [22], for example). On the face of it, this might present some unfavourable practical implications for any algorithm using a grouping theme. However, a worthwhile future endeavour could be to investigate how complete timetabling problems might be broken up into smaller sub-problems. For example, it has been noted [46] that real-world timetabling problems are often quite *clumped* in their make up: a computing department, for example, might have a set of events that forms a distinct clump largely separated from the events in, say, the psychology department. These departments could have few or no common students, may use different sets of rooms or might even be situated in separate campuses altogether. In these cases, the timetabling problems of these departments may have little bearing on each other and might even be solved independently from each other (see fig. 12).

Interesting ideas on the subject of dealing with large problem instances have also been proposed by Burke and Newall in [26]. Here, the authors use graph colouring-type heuristics to break up large sets of events into a number of smaller sub-sets, and then use a memetic algorithm to try and solve each subset individually. Tests by Burke and Newall indicate that this method of problem decomposition can offer both shorter run times and improved solution quality in some cases.

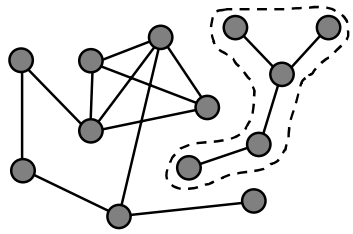


Fig. 12. An example of *clumping*: the nodes (events) in the left sub-graph conflict with none of the nodes in the right sub-graph. These two graphs might therefore be coloured separately from each other.

Another noticeable characteristic of the algorithms discussed here is the fact that, for the moment, no soft constraints are considered. This was never our aim for the algorithm, and indeed good algorithms that specialise in soft constraint violations can be found elsewhere (see, for example [33], [35], [36], [47], [49], and [57]). However, there might be additional complications if we *were* to attempt their inclusion. For instance, constraints such as 'students should not have to attend more than two events in consecutive timeslots' clearly depend on the *ordering* of the timeslots – a factor that is not considered here. Additionally, if we *were* to include soft constraint satisfaction in the GGA's aims we would probably also need to add suitable penalty measures (possibly through the use of weights [1, 14, 17]) to the fitness function. However because the chief aim of the algorithm is to find feasibility, such a modification might actually have the adverse effect of sometimes leading the search away from attractive (and 100% feasible) regions of the search space.

On the other hand, the incorporation of other sorts of soft constraints might present fewer difficulties. Consider, for example, the first soft constraint in section II. If we were to try and reduce the target number of timeslots from forty-five down to forty, this would actually mean that candidate timetables would be deemed feasible when the number of timeslots being used fell below forty-five, but also the total number of violations of this soft constraint would also fall as the number of timeslots being used approached forty.

Another worthwhile endeavour involving the general two-stage timetabling approach might also be to investigate the actual importance of ensuring that a completely feasible timetable is always gained in stage one. Will an algorithm specialising in the elimination of soft constraints always perform better when presented with a fully feasible timetable? Or will an almost-feasible timetable perform equally well? On a similar theme, it might also be instructive to see if we can identify some features of feasible (or near-feasible) timetables that will give us some indication of how easy it will be to then satisfy the soft constraints.

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

Finally, it is worth considering that many of the factors discussed here, in particular those relating to the scaling-up issues surrounding the GGA, might not just apply to this problem, but also to the general GGA model as a whole. For example, a GGA has been shown to be very successful with bin packing in the work of Falkenauer [51]. However, the problem-instances used here were only actually made up of very small groups (generally there were only three items per bin). It would thus be interesting to see how these same operators performed with bin packing problem-instances that allowed larger numbers of items to be placed into each bin, or indeed how other GAA applications tend to deal with these kinds of problem instances.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous referees who provided helpful comments on earlier versions of this work.

REFERENCES

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

[2] T. Cooper and J. Kingston, "The complexity of timetable construction problems," in *Practice and Theory of Automated Timetabling* (*PATAT*) *I*, (Lecture Notes in Computer Science, vol. 1153) E. Burke and P. Ross Eds. Berlin, Germany: Springer-Verlag, 1996. pp 283-295.

[3] 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.

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

[5] 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.

[6] E. Erben, "A Grouping Genetic Algorithm for Graph Colouring and Exam Timetabling," in *Practice and Theory of Automated Timetabling* (PATAT) *III* (Lecture Notes in Computer Science vol. 2079) E. Burke and W. Erben Eds. Berlin, Germany: Springer-Verlag, 2000, pp. 132-158.

[7] E. A. Akkoyunlu, "A Linear Algorithm for Computing the Optimum University Timetable," *The Computer Journal*, vol. 16(4), pp. 347-350, 1973.

[8] S. Daskalaki, T. Birbas, and E. Housos, "An Integer Programming Formulation for a Case Study in University Timetabling," *European Journal of Operational Research*, vol. 153, pp. 117-135, 2004.

[9] B. Deris, S. Omatu, H. Ohta, and D. Samat, "University Timetabling by Constraint-based Reasoning: A Case Study," *Journal of the Operational Research Society*, vol. 48(12), pp. 1178-1190, 1997.

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

[11] 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.

[12] 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*, (Lecture Notes in Computer Science, vol. 1408) E. Burke and M. Carter Eds. Berlin, Germany: Springer-Verlag, 1998. pp 92-114.

[13] 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.

[14] A. Schaerf, "Tabu Search Techniques for Large High-School Timetabling Problems," in *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (AAAI'96), Portland (OR), USA: AAAI Press/MIT Press, 1996, pp. 363-368.

[15] D. Costa, "A tabu search algorithm for computing an operational timetable," *European Journal of Operational Research*, vol. 76, pp. 98-110, 1994.

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

[17] A. Schaerf, "Local Search Techniques for Large High-School Timetabling Problems," *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, vol. 29(4), pp. 368-377, 1999.

[18] P. Ross, E. Hart, and D. Corne, "Genetic Algorithms and Timetabling," in *Advances in Evolutionary Computing: Theory and Applications*, A. Ghosh and S. Tsutsui Eds. New York: Springer-Verlag, 2003, pp. 755-772.

[19] A. Colorni, M. Dorigo, and V. Maniezzo, "Genetic Algorithms And Highly Constrained Problems: The Time-Table Case," in *Parallel Problem Solving from Nature* (*PPSN*) *I* (Lecture Notes in Computer Science, vol. 496), H. Schwefel and R. Manner Eds. Berlin, Germany: Springer-Verlag, 1990, pp 55-59.

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

[21] D. Abramson and J. Abela, "A Parallel Genetic Algorithm for Solving the School Timetabling Problem," *Tech. Rep*., Division of Information Technology, C.S.I.R.O., c/o Dept. of Communication & Electronic Engineering, Royal Melbourne Institute of Technology, PO BOX 2476V, Melbourne 3001, Australia, 1991.

[22] 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* (Lecture Notes in Computer Science, vol. 1498), T. Baeck, A. Eiben, M. Schoenauer, and H. Schwefel Eds. Berlin, Germany: Springer-Verlag, 1998, pp 865-874.

[23] E. Burke, D. Elliman, R. Weare "A Hybrid Genetic Algorithm for Highly Constrained Timetabling Problems" in *Genetic Algorithms: Proc. of the Sixth International Conference* (ICGA95), L. Eshelman Ed. San Francisco, CA: Morgan-Kaufmann, 1995, pp 605-610.

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

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

[26] E. Burke and J. Newall, "A Multi-Stage Evolutionary Algorithm for the Timetable Problem," *IEEE Transactions in Evolutionary Computation*, vol. 3(1), pp. 63-74, 1999.

[27] K. Socha and M. Samples, "Ant Algorithms for the University Course Timetabling Problem with Regard to the State-of-the-Art," in *Evolutionary Computation in Combinatorial Optimization* (EVOCop) *III* (Lecture Notes in Computer Science vol. 2611) M. Dorigo, G. Di Caro, and M. Sampels Eds. Berlin, Germany: Springer-Verlag, 2003, pp 334-345.

[28] K. Socha, J. Knowles, and M. Sampels, "A MAX-MIN Ant System for the University Course Timetabling Problem," in *Proceedings of the Third International Workshop on Ant Algorithms* (ANTS02) (Lecture Notes in Computer Science vol. 2463) M. Dorigo, G. Di Caro, and M. Sampels Eds. Berlin, Germany: Springer-Verlag, 2002, pp 1-13.

[29] E. Burke, G. Kendall, and E. Soubeiga, "A Tabu-Search Hyperheuristic for Timetabling and Rostering," *Journal of Heuristics*, vol. 9(6), pp. 451-470, 2003.

[30] S. Casey and J. Thompson, "GRASPing the Examination Scheduling Problem," in *Practice and Theory of Automated Timetabling* (*PATAT*) *IV*, (Lecture Notes in Computer Science, vol. 2740) E. Burke and P. de Causmaecker Eds. Berlin, Germany: Springer-Verlag, 2003, pp 232-244.

[31] 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*, (Lecture Notes in Computer Science, vol. 3616), E. Burke and M. Trick Eds. Berlin, Germany: Springer-Verlag, 2004, pp 294-312.

[32] M. Carrasco and M. Pato, "A Multiobjective Genetic Algorithm for the Class/Teacher Timetabling Problem," in *Practice and Theory of Automated Timetabling* (PATAT) *III* (Lecture Notes in Computer Science vol. 2079) E. Burke and W. Erben Eds. Berlin, Germany: Springer-Verlag, 2000, pp. 3-17.

[33] M. Chiarandini, M. Birattari, K. Socha, and O. Rossi-Doria, "An effective hybrid algorithm for university course timetabling," *Journal of Scheduling*, vol. 9(5), pp. 403-432, 2006.

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

[35] 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*, (Lecture Notes in Computer Science, vol. 2740) E. Burke and P. de Causmaecker Eds. Berlin, Germany: Springer-Verlag, 2003, pp 329-351.

[36] P. Kostuch, "The University Course Timetabling Problem with a 3-Phase approach," in *Practice and Theory of Automated Timetabling* (*PATAT*) *V*, (Lecture Notes in Computer Science, vol. 3616), E. Burke and M. Trick Eds. Berlin, Germany: Springer-Verlag, 2004, pp 109-125.

[37] M. Carter, "A Survey of Practical Applications of Examination Timetabling Algorithms," in *Operations Research*, vol. 34(2), pp. 193-202, 1986.

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

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

[40] A. Schaerf, "A Survey of Automated Timetabling," *Artificial Intelligence Review*, vol. 13(2), pp. 87-127, 1999.

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

[42] M. Carter. [Online] Available: ftp://ie.utoronto.ca/mwc/testprob/ Accessed June 2005.

[43] B. Paechter. [Online] Available: http://www.idsia.ch/Files/ttcomp2002/. Accessed June 2005.

[44] R. Lewis. [Online] Available: http://www.emergentcomputing.org/timetabling/harderinstances Accessed June 2005.

[45] C. Blum and C. Manfrin [Online] Available http://www.metaheuristics.org/. Accessed June 2005.

[46] 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* (Lecture Notes in Computer Science vol. 1153) E. Burke and P. Ross Eds. Berlin, Germany: Springer-Verlag, 1996, pp 309-325.

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

[48] J. Cordeau, B. Jaumard, and R. Morales, "Efficient Timetabling Solution with Tabu Search," [Online]. Available http://www.idsia.ch/Files/ttcomp2002/jaumard.pdf Accessed June 2005.

[49] E. Burke, Y. Bykov, J. Newall, S. and Petrovic, "Time-Predefined Approach to Course Timetabling," *Yugoslav Journal of Operations Research* (YUJOR), vol. 13(2), 2003, pp 139-151.

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

[51] E. Falkenauer, *Genetic Algorithms and Grouping Problems*: New York, NY: John Wiley and Sons, 1998.

[52] N. J. Radcliffe, "Forma Analysis and Random Respectful Recombination," in *Proceedings of the Fourth International Conference on Genetic Algorithms*, R. Belew and L. Booker Eds. San Marco CA: Morgan Kaufmann, 1991, pp. 222-229.

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

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

[55] A. E. Eiben, J. K. van der Hauw, and J. I. van Hemert, "Graph Coloring with Adaptive Evolutionary Algorithms," *Journal of Heuristics*, vol. 4(1), pp. 25-46, 1998.

[56] S. Khuri, T. Walters, and Y. Sugono, "A grouping genetic algorithm for coloring the edges of graphs," in *Proceedings of the 2000 ACM Symposium on Applied computing*, NY, USA: ACM Press, 2000, pp 422-427.

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

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

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

[60] R. Lewis and B. Paechter, "New Crossover Operators for Timetabling with Evolutionary Algorithms", in *Proceedings of the Fifth International Conference on Recent Advances in Soft Computing* (RASC 2004), A. Lotfi Ed. Nottingham, England, 2004, pp. 189-195

[61] W. Morrison and K. de Jong, "Measurement of Population Diversity," in *Selected Papers from the 5th European Conference on Artificial Evolution* (EA) *2001* (Lecture Notes in Computer Science vol. 2310), P. Collet, C. Fonlupt, J.-K. Hao, E. Lutton, and M. Schoenauer Eds. Berlin, Germany: Springer-Verlag, 2002, pp 31-41.

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

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

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

[65] R. Lewis and B. Paechter, "An Empirical Analysis of the Grouping Genetic Algorithm: The Timetabling Case", in *Proceedings of the 2005 IEEE Congress on Evolutionary Computation* (CEC) vol. 3, 2005, pp 2856-2863.

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

[67] P. Moscato and M. G. Norman, "A 'Memetic' Approach for the Travelling Salesman Problem. Implementation of a Computational Ecology for Combinatorial Optimization on Message-Passing Systems," in *Proceedings of the International Conference on Parallel Computing and Transputer Applications*, Amsterdam, NL: IOS Press, 1992, pp 187-194.

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

[69] R. Dorne and J.-K. Hao, "A New Genetic Local Search Algorithm for Graph Coloring," in *Parallel Problem Solving from Nature* (PPSN) *V*, (Lecture Notes in Computer Science vol. 1498), A.E. Eiben, T. Baeck, M. Schoenauer and H-P. Schwefel Eds. Berlin, Germany: Springer-Verlag, 1998, pp 745-754.

[70] P. Galinier and H. J-K., "Hybrid evolutionary algorithms for graph coloring," *Journal of Combinatorial Optimization*, vol. 3(4), pp. 379-397, 1999.

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

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

[73] S. Martello and P. Toth, "Lower bounds and reduction procedures for the bin packing problem," *Discrete Applied Mathematics*, vol. 28(1), pp. 59-70, 1990.