Applied Soft Computing 13 (2013) 2504–2514

Contents lists available at SciVerse ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Course timetabling using evolutionary operators

Danial Qaurooni [a,*], Mohammad-R. Akbarzadeh-T [b]

[a] Department of Computer Science, Amirkabir University of Technology, Tehran, Iran
[b] Center of Excellence on Soft Computing and Intelligent Information Processing, Ferdowsi University, Mashhad, Iran

ARTICLE INFO

Article history:
Received 13 September 2010
Received in revised form 21 October 2012
Accepted 24 November 2012
Available online 26 December 2012

Keywords:
Combinatorial optimization
Scheduling and timetabling
Evolutionary operators
Memetic algorithms
Grouping genetic algorithms

ABSTRACT

Timetabling is the problem of scheduling a set of events while satisfying various constraints. In this paper, we develop and study the performance of an evolutionary algorithm, designed to solve a specific variant of the timetabling problem. Our aim here is twofold: to develop a competitive algorithm, but more importantly, to investigate the applicability of evolutionary operators to timetabling. To this end, the introduced algorithm is tested using a benchmark set. Comparison with other algorithms shows that it achieves better results in some, but not all instances, signifying strong and weak points. To further the study, more comprehensive tests are performed in connection with another evolutionary algorithm that uses strictly group-based operators. Our analysis of the empirical results leads us to question single-level selection, proposing, in its place, a multi-level alternative.

1. Introduction

The essence of timetabling is scheduling a set of events, where scheduling typically consists of allotting rooms and time slots to events, subject to certain specified constraints. Varying sets of constraints constitute the gamut of timetabling subclasses that are designed to address different theoretical and real-world applications, ranging from terminal scheduling to nurse rostering. A general survey of these various subclasses is available in [1]. One such subclass, educational timetabling, deals with scheduling problems in schools and universities, with constraints pertaining to teachers, lecture hours, facilities, program conflicts, room capacities, etc. Educational timetabling itself consists of three major variants, namely the university course timetabling problem (UCTP), the exam timetabling problem (ETP), and the high school timetabling problem (HTP). These have constraint-related differences. For instance while, in the ETP, events can take place in the same room and time slot as long as they satisfy all other constraints, in the UCTP, only one event can occupy a certain room at a specified time slot. Also, while the length of the UCTP scheduling period (and hence the number of time slots) is constant, since no more than a certain number of time slots can fit in a week, ETP might be able to benefit from some level of flexibility in this area. In this paper we focus on the UCTP, but in light of the similarity in problem structure and constraint sets, the other variants may be subject to some of the issues discussed here with little loss of generality.

The constraints of the UCTP are divided into two main categories of hard and soft, denoting issues of feasibility and preference, respectively. Any violation of a hard constraint results in an infeasible timetable. Most frequent among these violations is when two events that share participants are scheduled to be held at the same time, resulting in a so-called event clash. Clearly, a timetable containing such clashes is unacceptable. Soft constraints, on the other hand, are more like considerations that, when taken into account, can yield timetables that are more accommodating for the institution, the participants, or both. Thus we might formulate a typical soft constraint to discourage the scheduling of more than two classes in a row for any student.

The NP-Completeness of the timetabling problem, has been explicitly established by Even et al. in [2], making the task of tackling the problem particularly attractive and challenging in equal measures for a host of algorithmic approaches. General surveys of the performance of algorithms can be found in [3–5]. In this paper, we focus on course timetabling problems whose solutions conform to a general evolutionary framework.

Metaheuristics have been widely utilized for solving timetabling problems. In the framework of swarm intelligence, ant colony optimization algorithms have been proposed, such as [6,7] which use ants to construct complete assignment of events to time slots using heuristics and pheromone information. Timetables are then improved using a local-search procedure, and the pheromone matrix is updated accordingly for the next iteration. Specifically, Socha et al. compare and analyze two different ant systems in [6], and go on to compare the MMAS (the better-performing ant system) with other algorithms, concluding that on the large instances, MMAS outperforms competitors. Among the more novel

* Corresponding author.
E-mail address: dani.qaurooni@aut.ac.ir (D. Qaurooni).

1568-4946/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.asoc.2012.11.044

approaches, honey-bee mating optimization has also been applied to both exam and course timetabling in [8]. The authors test their algorithm against Carter and Socha benchmark sets and reportedly achieve "competitive if not better" results. The popular simulated annealing (SA) heuristic has been implemented in [9–11] among others. Specifically, [10] uses a two-phased SA algorithm with a few heuristics and a new neighborhood structure that swaps between pairs of time slots, instead of two assignments. The authors report that, when tested against two standard benchmark sets, the new neighborhood heuristic improves the performance of SA. Tabu search is also applied in [12,13]. In [13], an adaptive tabu search is used to integrate an original double Kempe chains neighborhood structure, a penalty-guided perturbation operator and an adaptive search mechanism, to achieve more efficient results.

More specifically, evolutionary algorithms (EAs) have been applied to timetabling problems with varying degrees of success in the works of [14–19] among others. For example, [14] is among the first to remedy the poor performance of a genetic algorithm, compared with other conventional methods, by way of a grouping encoding. Generally, however, EAs employ other techniques to make up for their potential failure in dealing with local optima. For instance, [17] takes advantage of domain-specific heuristics in an otherwise evolutionary structure to boost performance. Working on real-world, rather than artificial, problems, Beligiannis et al. use an evolutionary algorithm to solve timetabling problems in Greek high schools in [18].

As a subclass of EAs, memetic algorithms have been used to solve timetabling in [19–24] among others. As one of the earliest applications, in [20], Burke et al. use one of two mutation operators, namely "light" and "heavy", to reschedule a random selection of events or disrupt whole time slots of events, respectively. The evolutionary process incorporates hill climbing too. Alkan and Ozcan have used a number of different one-point and uniform crossover operations, a weighted fitness function, and again a hill climbing procedure to reduce violations in [21]. A recent successful application of a memetic algorithm to laboratory timetabling is [23] where student preferences are also taken into account.

While standard benchmark sets for timetabling exist [25–27], it has been argued in [5] reasonably that "there is confusion in the field" in this regard, to the point that performing meaningful comparisons or reproducibility might be compromised. In part to overcome such obstacles, two international timetabling competitions have been held recently. The latter consisted of examination and course timetabling problem sets. McCollum et al. provide an overview of this event in [28]. Information on the winning algorithms can also be found in [29].

In general though, the standard benchmark packages are usually designed with hard *and* soft constraints in mind, hence disregarding soft constraints to focus on feasibility is hardly a challenge. In fact, many mainly focus on soft constraints, to the point that feasible solutions are found so fast as to render any comparison and analysis meaningless. In such a situation, the benchmarking instances provided in [30] address the issue of how different algorithms would fare when the focus is shifted to feasibility. This is achieved through a careful choice of a deliberate subset of a larger set of problem instances that have proven to be particularly difficult, but have at least one feasible solution. The first substantial use of this package was in relation to the work of [19], which implemented two algorithms. One of these mainly used grouping genetic operators, while the other implemented a local search. Both algorithms were augmented by local search. The average results of these two algorithms were later improved in the context of a simulated annealing algorithm [31]. In this paper, we will restrict our attention to these "hard instances", along with a discussion of the performance of the three algorithms that have made use of it.

The paper is organized in the following manner: A detailed problem description is given in the next section, followed by the outline of our memetic algorithm in Section 3, where we go over details about general algorithm structure and operator design. Section 4 includes a more focused discussion of the fitness function. In Section 5, experimental parameters are provided and the proposed algorithm is put to test. Comparisons with three other algorithms follow, demonstrating the superior performance of our memetic algorithm in two of the three instance sets. The focus is then narrowed down to evolutionary algorithms, with an analysis of operator design and development of new measures and further tests. We close by making conclusions and suggesting possible directions for future research in Section 8.

2. Problem description

Stated formally, the particular timetabling problem we study here, initially formulated for the First Timetabling Competition [32], has four parameters: T, a finite set of times; R, a finite set of rooms; E, a finite set of events; and C, a finite set of constraints, and concerns assigning times and rooms to the events so as to satisfy the required constraints. These constraints are divided into two categories of hard and soft. Generally speaking, hard constraints denote mandatory requirements that, when satisfied, produce "working" timetables. Soft constraints, on the other hand, cater to the preferences of the teachers, students, etc. Although an optimal solution attends to both hard *and* soft constraints, our algorithm ignores soft constraints and focuses on hard constraints for reasons that will be explained in Section 5.

In mathematical terms, a binary-valued function $h : S \to 0, 1$ can be associated with each hard constraint. For each solution $s \in S$, the function is defined by

$$h(s) = \begin{cases} 1 & \text{if } s \text{ does not satisfy the constraint} \\ 0 & \text{otherwise} \end{cases}$$

Let S be the set of all solutions to a given timetabling problem. A feasible solution is any solution $s \in S$ that satisfies all hard constraints. Thus, the objective function might be formulated as

$$F(S) = \sum_{i=1}^{n} h_i(S) \tag{1}$$

where hard constraints are given by $h_1, h_2, \ldots, h_n$. Our particular formulation includes the following hard constraints:

- H1: no student is permitted to attend more than one event at any one time;
- H2: only one event is scheduled for any room and any time slot;
- H3: all of the features required by the event should be satisfied by the room, which has an adequate capacity.

Aside from general problem description, it might be useful to discuss a problem-solving perspective that will influence our solution-building later on, namely the idea of interpreting timetabling as a grouping problem. In [33], Falkenauer defines grouping problems as those where the task is to partition a set of objects U into a collection of mutually disjoint subsets u_i of U, where

$$\bigcup u_i = U \quad \text{and} \quad u_i \cap u_j = 0, i \neq j \tag{2}$$

according to a set of problem-specific constraints that define allowable groupings. Familiar cases where this criterion holds include bin packing, graph coloring and timetabling. A corresponding grouping genetic algorithm (GGA), devised by Falkenauer is based on the notion that, while traditional genetic operators are well-suited for

a typical non-grouping problem and we can judge the quality of an assignment independent of other assignments, these same operators fail in the case of grouping problems, and the quality of any assignment in such a problem depends on other assignments.

In light of this issue, Falkenauer argues that when dealing with grouping problems, traditional, item-oriented genetic operators such as recombination and mutation tend to break up, rather than improve, the building blocks produced in previous rounds of the algorithm. Consequently, what distinguishes group-based operators from their traditional item-based counterparts is the focus on groups as opposed to individuals. Additionally, group-based operators are more efficient since, when focusing on items (as opposed to groups), the search space of a grouping problem grows much larger than it has to be, as a result of an encoding that makes the algorithm expend time on solutions it has already examined, but that it fails to recognize.

For instance, in the case of our timetabling problem, events constitute the "items" of the grouping formulation and determining whether a student is scheduled to attend two events at the same time (denoted by constraint H1 above), requires checking the other assignments of the same time slot. Thus a recombination operator that focuses on *individual*, rather than *groupings* of events misses the point. Lewis and Paechter apply these insights to the case of timetabling in [19] and, in order to investigate the merits of group-based operators, depart from item-oriented genetic operators and focus on time slots as the appropriate building block of the problem. Thus construed, *entire* time slots are subject to genetic operators such as recombination and mutation. We will examine the effects of these decisions on solution quality later on. The important thing to note for now is that our algorithm utilizes a grouping formulation and we will turn to that in the next section.

1) $P \leftarrow$ **Initialize**(p).
2) **Evaluate**(P).
3) **While no feasible timetable is found:**
   a) $P \leftarrow$ **NextGeneration**(P, rr, mr, lr).
   b) **Evaluate**(P).

Initialize(p)

1) **For** p **timetables:**
   a) $T \leftarrow$ **CreateTimetable**().
   b) **ShuffleEvents**(E).
   c) **ShuffleSlots**(T, S).
   d) **For each event** $e \in E$**:**
      i) **Choose a random slot** s **from** S.
      ii) **If** (s **is a feasible slot for** e)
      $T(s) \leftarrow e$.
   e) $P \leftarrow P \cup \{T\}$.
2) **Return** P.

Fig. 1. The basic structure of the algorithm. p is the desired population size, to be created by the method Initialize. The function NextGeneration creates new individuals at each generation through the application of the three operators of recombination, mutation and local search, whose parameters are passed by *rr*, *mr*, and *lr*, respectively.

3. Algorithm layout

We construct solution timetables in the form of a two-dimensional matrix with rows representing rooms and columns representing time slots. Thus, the intersection of room i and time slot j, in the array R, will pick out either a certain event specified by its number, say n ($r_{ij} = n$), or a vacancy. Note that, other than keeping an array of unscheduled events for each timetable, this approach does not allow any relaxations regarding problem constraints. Therefore, an event is allowed a certain place in the timetable only if it violates none of the hard constraints defined in Section 2. Moreover, we do not allow any relaxations regarding the number of time slots, meaning that from the outset, the number of allotted time slots is fixed at the total number of usable time slots (which is 45 in this case). This is in contrast to some other timetabling algorithms, notably in [19], that relax the constraint on the number of time slots at first, and later discard the extra ones.

Our algorithm belongs to the family of memetic algorithms (MAs). The idea of combining local search with genetic algorithms, variously characterized as memetic, Lamarckian, hybrid or cultural algorithms [34], enhances the simple transition process that happens in a genetic algorithm by including problem-specific knowledge such as heuristics, local search techniques, etc. The idea is to guide local search by the ability of genetic algorithms to explore dark corners of design space and as such the strength of such a search is best harnessed when the exploratory capabilities of a genetic algorithm is complemented by the exploitative power of local search in a successful trade-off.

Within such a framework, the structure of the algorithm is described in Fig. 1. The first phase, denoted by the method Initialize, generates the initial population. Our primary objective here is to create diversity. To this end, the procedure first shuffles both lists of slots and events for each timetable using the Fisher–Yates [35] shuffle algorithm. After choosing an item from each list, the algorithm checks for compliance between the characteristics of the chosen slot and the requirements of the candidate event, and an assignment is made in case they match. Thus, our initialization procedure is greedy in nature, similar to that used in [36]. By the time the method Initialize is done, several rather sparse timetables are created, each accompanied by a list of events that have not been scheduled.

When initialization is over, the population goes through the three stages of recombination, mutation, and local search repeatedly. Beginning with recombination, there are four key steps in the process: Selection, Injection, Duplicate Removal and Reinsertion.

1. **Selection**: Random beginning and end boundaries are chosen from the intersection of rooms and time slots to determine a range.
2. **Injection**: The range of slots defined in the first step are replaced in one parent using the corresponding selection from the other.
3. **Duplicate removal**: Any possible duplicates that might have arisen are removed from the host parent.
4. **Reinsertion**: Unscheduled events are picked at random and reinserted back into the timetable. In case an event can be scheduled in multiple candidate slots, one is chosen at random.

Finally, for constructing the second offspring, parents reverse their roles. Fig. 2 illustrates this process with an example.

Next is the mutation operator, where a specified number of time slots are randomly selected and emptied out entirely into the unscheduled list. This list is then shuffled and traversed and at each step, for the unscheduled event of choice, a search for a feasible slot is performed. If more than one slot is available, the tie is broken randomly.

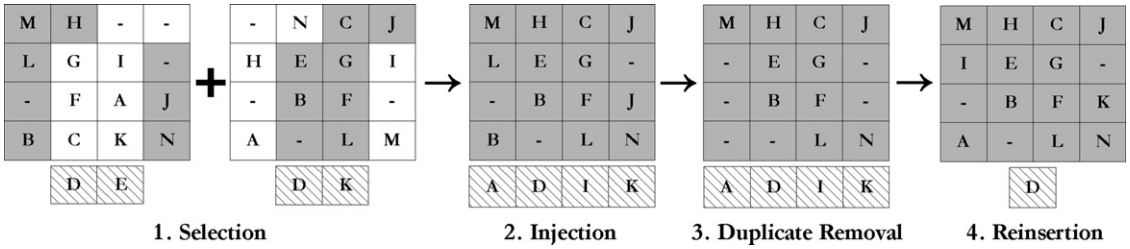

Fig. 2. Illustrating the process of recombination as it goes through its four stages. Two parent timetables with hypothetical events ranging from A to N are to be recombined. The hatched items below each timetable represent its list of unscheduled events. Selected boundaries are highlighted with gray.

At this stage, the part of the genetic operators is over and the timetable is subject to a local search procedure, implemented by two operators. The first one simply attempts to change the grouping of events by searching for alternative random slots for an already scheduled event. Possible alternative slots make up independent sets of nodes in the corresponding clash graph. However, since finding maximal conflict-free groupings (independent sets of nodes) is an NP-Complete problem in its own right, the process is restricted to exchanging events between realized groupings in the current timetable.

By the time the timetable is passed on to the second local search operator, the algorithm has done all it can to improve the packing of events. Hence no feasible, unoccupied slots exist for the events on the unscheduled list. This situations captures a common local optimum, since such a relational structure for the timetable cannot yield a solution and the timetable is bound to be modified if it is to improve at all. This means that, to escape local optima, it has to climb down the local peak and tolerate a few "bad" moves, at least temporarily. Since the operator makes use of a neighborhood definition that is large in size, it might be considered a very large-scale neighborhood search [37]. As illustrated with an example in Fig. 3, the second local search procedure operates in three stages: Slot selection, Shoving, and Reinsertion.

1. **Slot selection**: The list of the unscheduled events is shuffled and the timetable is searched for possible slots to schedule them. Since, as noted earlier, no feasible slots are available in the timetable at this point, the search looks for slots that are either feasible but occupied, or are in conflict with other events that have been scheduled in the same time slot.
2. **Shoving**: For the chosen event, the "best" potential place is picked. "Best" here designates a heuristic that prefers a slot that has the fewest number of students in clash with it (note: students, *not* events). This is commonly known as the Least Constraining Value (LCV) heuristic. Before inserting an unscheduled event, the designated slot is vacated together with the slots containing events that are in conflict with the replacing event.
3. **Reinsertion**: After the above steps have been carried out for *all* originally unscheduled events, a new list of unscheduled events emerges. These might be feasibly scheduled in other vacant slots in the timetable. Hence, as a final step, an exhaustive search is performed to fit in as many of these events as possible.

This concludes the description of the algorithm. In the following section, the fitness function and its parameters are discussed.

4. Fitness function analysis

A natural choice for evaluating timetables might be to count the number of unscheduled events. However, as demonstrated with an example in Fig. 4, such a fitness function is too simplistic since it fails to differentiate between similar-looking solutions that nevertheless have different relational structures. Therefore, other than counting unplaced events, it is desirable for the fitness function to distinguish between timetables in other ways, so that it can maintain selection pressure for greater durations. Also, for highly constrained problems (as is the case for many timetabling instances) it would be desirable if the algorithm could learn from its experience so that it could evaluate solutions more dynamically. All in all, the following criteria should be satisfied:

1. Take solution structure into account;
2. Learn from experience;
3. Keep computation costs as low as possible.

The optimal trade-off between the first two criteria above adds enough detail to distinguish between relationally dissimilar states while keeping the computation cost affordable. To address the first criterion, any possible mismatch between the facilities and capacity of every room on the one hand, and the requirements of its scheduled event on the other hand can be taken into account. To address the second criterion, we introduced a new parameter based on the intuition that events that wind up on the unscheduled list more often, are harder to schedule and should be given a higher priority. To implement this, for each event, a weight adaptation scheme [38] has been implemented by way of an index which is first initialized to the original number of conflicting students. For every event, this number is equal to the number of students that take part in another event, which, if scheduled in the same time slot with the current event, will cause a conflict. The index is also incremented every time the event is shoved by the second local search operator. Since this parameter is then shared among all individuals of the population, it can serve as an estimate of how troublesome scheduling a specified event generally is. This functions to gradually aid the algorithm in setting priorities, because when calculating fitness values, it allows the fitness function to penalize timetables that have higher-priority events on their list of unscheduled events. This way, the cumulative "experience" of the population in dealing with a set of events is taken into account.

The above observations are captured in the equation below:

$$f = \sum_{i=1}^{n} [p_{e_i}(\sum_{j=1}^{m} rf_{e_i j}(1 - ef_{ij}) + (c_{r_{e_i}} - a_{e_i})) + (1 - p_{e_i})(w_i + a_{e_i})] \quad (3)$$

The function simply implements the ideas noted above and essentially consists of two parts that deal with properties of scheduled and unscheduled events, respectively. Throughout the equation, n denotes the total number of events. In the first half, which deals with scheduled events, the boolean p_{e_i} indicates whether event i is scheduled in the current timetable, m denotes the total number of features, the boolean $rf_{e_i j}$ says whether the room for event i has feature j, and the boolean ef_{ij} says whether event i itself requires feature j or not. To conclude the mismatch calculation, the difference between the capacity of the room occupied

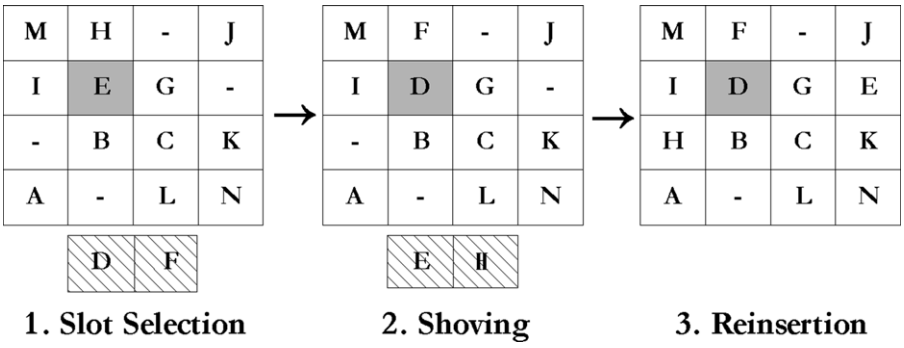

Fig. 3. The three phases of the second local search operator, as described in the text. It is assumed that the event marked by the letter D has no feasible slot before the local search operator begins.

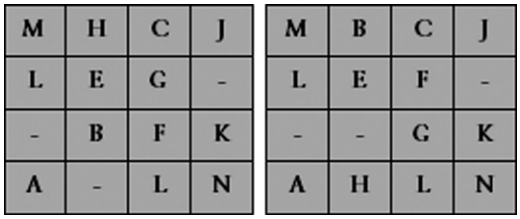

Fig. 4. Two timetables that have the same scheduled events, but should be evaluated differently because of their structural differences.

by event i and the size of the event, captured by $c_{r_{e_i}} - a_{e_i}$, is also added. Turning to the second half, which deals with unscheduled events, w_i represents the weight adaptation index for event i, while a_i denotes the number of attendants of event i.

Although the equation calculates a distance of some sort, where lower values are deemed more desirable, a feasible solution will probably not yield a distance of zero. The reason is that while the second part of the equation evaluates to zero once all events have been scheduled, the first part will most probably yield a positive value to account for size and facility mismatches that might occur even in a feasible timetable.

5. Experimental setup

As noted earlier, the benchmarking instances provided in [30] address the issue of how different algorithms would fare when the focus is shifted to feasibility. These include three sets of 20 "hard" instances of small (approximately 200 events and 5 rooms), medium (400 events and 10 rooms), and big (1000 events and 25 rooms) size. The instances are called "hard" since Lewis and Paechter report in [19] that they are a deliberate subset of a larger set that have been "troublesome for finding feasibility". Still, all of these have at least one feasible solution, which means events can potentially fit in the 45 allotted time slots while satisfying the hard constraints noted earlier in Section 2. More information regarding how particular instances were generated can be found in [19].

For proper comparison with other algorithms, tests were performed on a 2.66 GHz CPU with 1 GB memory. Also, two sets of time limits (one set with 30, 200, and 800 s, and another with 200, 500, and 1000 s) were imposed for comparisons with three other algorithms across instances of small, medium and big sizes. To check the integrity of our particular implementation and the solutions produced, the test program provided in [39] was used.

The selection process is rank-based, stochastically preferring timetables with higher fitness values. The adopted parameter values have mostly been chosen deliberately to further our analyses. The recombination rate (rr) specifies the percentage of individuals produced using the recombination operator. The remainder of each generation is populated by making copies of the highest ranked individuals. The offspring that are produced using recombination then go through the mutation process, which applies to a number of time slots specified by the mutation rate (mr). The first local search operator is applied to all offspring produced by recombination. The local search rate (lr) designates the number of individuals, again selected based on their ranks, that are subject to the second local search operator described earlier.

6. Analysis of performance

Since different time limits have been used in [19] and [31] as their stopping criteria, we had to divide the table into two sets of columns.[1] MA is the memetic algorithm introduced in this paper, GGA is a grouping genetic algorithm based on the grouping formulation of [19] noted earlier and similar to our algorithm in many respects, H is a single thread local search heuristic algorithm that works by improving the packing of individual time slots, and HSA refers to the hybrid simulated annealing, defined in [31], which uses a combination of problem relaxation, Kempe chain and graph heuristics, and simulated annealing to tackle the problem.

The results for GGA and the H algorithm are based on tests that have been performed on a 2.66 GHz Pentium processor with 1 GB of memory, while those of HSA are based on a 3.2 GHz Pentium processor (memory information for HSA's configuration has not been provided by [31]) and finally MA, as noted earlier, was tested on a 2.66 GHz CPU with 1 GB memory. Figs. 5–7 show the average and minimum values for these four algorithms in the case of the 20 instances of size small, medium and big, respectively. The following statistical parameters were also calculated:

[1] Although a time limit of 400 s is reported for medium instances in [31], the reported results therein do not comply with this time limit and the author informed us in a personal correspondence that the correct time limit for the medium instances was 500 s.

	30 seconds				200 seconds		
no.	MA (rr = 0.5, ls = 0.5)	MA (rr = 0.8, ls = 0.5)	GGA	H	MA (rr = 0.5, ls = 0.5)	MA (rr = 0.8, ls = 0.5)	HSA
1	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
2	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
3	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
4	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
5	0 (0)	0 (0)	1.05 (0)	0 (0)	0 (0)	0 (0)	0 (0)
6	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
7	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
8	1.45 (0)	0.95 (0)	6.45 (4)	1 (0)	0.6 (0)	0.4 (0)	1.9 (0)
9	0.5 (0)	0.1 (0)	2.5 (0)	0.15 (0)	0.05 (0)	0 (0)	3.85 (0)
10	0 (0)	0 (0)	0.1 (0)	0 (0)	0 (0)	0 (0)	0 (0)
11	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
12	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
13	0 (0)	0 (0)	1.25 (0)	0.35 (0)	0 (0)	0 (0)	1 (0)
14	1.85 (0)	2 (0)	10.5 (3)	2.75 (0)	0.8 (0)	0.8 (0)	5.95 (3)
15	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
16	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
17	0 (0)	0 (0)	0.25 (0)	0 (0)	0 (0)	0 (0)	0 (0)
18	0.15 (0)	0.25 (0)	0.7 (0)	0.2 (0)	0 (0)	0 (0)	0.45 (0)
19	0 (0)	0 (0)	0.15 (0)	0 (0)	0 (0)	0 (0)	1.2 (0)
20	0.25 (0)	0.7 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
AVG & STD	0.21 & 0.51 (0 & 0)	0.2 & 0.5 (0 & 0)	1.14 & 2.66 (0.35 & 1.1)	0.22 & 0.63 (0 & 0)	0.07 & 0.22 (0 & 0)	0.06 & 0.2 (0 & 0)	0.67 & 1.57 (0.15 & 0.67)
CV	2.42 (0)	2.5 (0)	2.33 (3.14)	2.86 (0)	3.14 (0)	3.33 (0)	2.34 (4.46)
Minimum Feasibility	20	20	18	20	20	20	19
Maximum Feasibility	15	15	11	15	17	18	15
Best Performer	17	17	11	15	18	20	14

Fig. 5. Performance results, for the small set, in comparison with three other algorithms. For each instance, an average of 20 runs is reported, with the best result noted in parentheses. Other than those denoted, parameters for the algorithm were fixed at: $mr = 1$, $p = 40$. The parameters for GGA are reported as $rr = 0.1$, $mr = 1$, $ls = 100$, $p = 5$ for small, $rr = 0.7$, $mr = 1$, $ls = 2$, $p = 10$ for medium, and $rr = 0.25$, $mr = 1$, $ls = 0$, $p = 50$ for big sets respectively.

- the **mean** which is taken to be indicative of general quality of performance, while **standard deviation** might quantify how much the mean value is to be trusted in representing performance *across* each set of instances;
- the **coefficient of variation** which can provide a normalized measure when focusing on dispersion independent of the mean;
- **minimum/maximum feasibility** that count the number of instances of each size where feasibility is achieved in at least one run/ all runs;
- **best performer** that keeps track of the number of instances for which each algorithm has achieved best results (highlighted by gray in the table) when compared with other algorithms.

Looking at the general results for the small set, it is obvious that MA ($rr = 0.8$, $ls = 0.5$) outperforms others on average. In fact, if allowed more time, the optimal setting can outperform others in all instances, as demonstrated by the best performer count. The values for standard deviation and minimum/maximum feasibility also show better results for this algorithm. The performance of MA ($rr = 0.5$, $ls = 0.5$), the H algorithm, HSA and GGA follow respectively. Note that the performance of HSA, despite being allowed significantly more time, is only better than that of GGA among the more time-constrained algorithms and compares poorly with the other two. HSA and GGA are also the only two algorithms that fail to achieve minimum feasibility for instance no. 14. For MA, the added 170 s of the less-constrained time limit yields slight improvements in all of the instances where it failed to achieve maximum feasibility in the first 30 s.

Turning to the results for the medium set, we can see that MA ($rr = 0.8$, $ls = 0.5$) outperforms others according to mean and standard deviation values again. Here, however the difference between the calculated values for the two variants of MA is too small to merit any substantial explanation. The H algorithm and GGA follow in the shorter time limit, and HSA comes after in the longer time limit. Similar to the case of small instances, comparing the performance of MA subject to the two time limits shows slight, robust improvements with added time.

Finally, the results for the big set are a little more complicated. In the case of the 800 second time limit on the left, the trend from the small and medium sets continued and MA ($rr = 0.5$, $ls = 0.5$) performed better than GGA and the H algorithm based on the mean, feasibility and performance measures. However, turning to the less-constrained configurations on the right, it is clear that HSA performs better than MA in terms of average and standard deviation values by a large margin. HSA's superior performance is especially evident in individual instances where none of the algorithms have been able to achieve feasibility.

To explain the significant drop in performance of MA, we tried to look at its behavior and observed that while the algorithm actively switched between different alternatives in the solution space in the case of small and medium instances (resulting in better exploration and improved results), it failed to do so in the case of big instances. In fact, the runs scarcely displayed any significant improvement after the first 600 s or so, waiting instead for chance mutations

	200 seconds				500 seconds		
no.	MA (rr = 0.5, ls = 0.5)	MA (rr = 0.8, ls = 0.5)	GGA	H	MA (rr = 0.5, ls = 0.5)	MA (rr = 0.8, ls = 0.5)	HSA
1	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
2	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
3	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
4	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
5	0 (0)	0 (0)	3.95 (0)	0 (0)	0 (0)	0 (0)	0 (0)
6	0 (0)	0 (0)	6.2 (0)	0 (0)	0 (0)	0 (0)	0 (0)
7	2.2 (1)	2.55 (1)	41.65 (34)	18.05 (14)	1.3 (0)	1.2 (0)	4.15 (1)
8	0 (0)	0 (0)	15.95 (9)	0 (0)	0 (0)	0 (0)	0 (0)
9	2.25 (0)	1.6 (0)	24.55 (17)	9.7 (2)	1.15 (0)	1.15 (0)	4.9 (0)
10	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
11	0 (0)	0 (0)	3.2 (0)	0 (0)	0 (0)	0 (0)	0 (0)
12	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
13	0 (0)	0 (0)	13.35 (3)	0.5 (0)	0 (0)	0 (0)	0.5 (0)
14	0 (0)	0 (0)	0.25 (0)	0 (0)	0 (0)	0 (0)	0 (0)
15	0 (0)	0 (0)	4.85 (0)	0 (0)	0 (0)	0 (0)	0.05 (0)
16	0.45 (0)	0.45 (0)	43.15(30)	6.4 (1)	0 (0)	0.05 (0)	5.15 (1)
17	0 (0)	0 (0)	3.55 (0)	0 (0)	0 (0)	0 (0)	0 (0)
18	0 (0)	0 (0)	8.2 (0)	3.1 (0)	0 (0)	0 (0)	6.05 (0)
19	0 (0)	0.2 (0)	9.25 (0)	3.15 (0)	0 (0)	0.05 (0)	5.45 (0)
20	0 (0)	0 (0)	2.1 (0)	11.45 (3)	0 (0)	0 (0)	10.6 (2)
AVG & STD	0.25 & 0.68 (0.05 & 0.22)	0.24 & 0.66 (0.05 & 0.22)	9.01 & 12.78 (4.7 & 10.0)	2.62 & 4.88 (1.0 & 3.1)	0.12 & 0.38 (0 & 0)	0.12 & 0.36 (0 & 0)	1.84 & 3.07 (0.2 & 0.52)
CV	2.72 (4.4)	2.75 (4.4)	1.41 (2.12)	1.86 (3.1)	3.16 (0)	3 (0)	1.66 (2.6)
Minimum Feasibility	19	19	15	16	20	20	17
Maximum Feasibility	17	16	6	13	18	16	12
Best Performer	19	18	6	13	19	18	12

Fig. 6. Performance results, for the medium set, in comparison with three other algorithms. For each instance, an average of 20 runs is reported, with the best result noted in parentheses. Other than those denoted, parameters for the algorithm were fixed at: $mr = 1$, $p = 40$. The parameters for GGA are reported as $rr = 0.1$, $mr = 1$, $ls = 100$, $p = 5$ for small, $rr = 0.7$, $mr = 1$, $ls = 2$, $p = 10$ for medium, and $rr = 0.25$, $mr = 1$, $ls = 0$, $p = 50$ for big sets respectively.

that might or might not result in climbing down a local optimum. This hints at a convergence rate that is too high. The suggestion is further backed by the maximum feasibility parameter, whose value for MA ($rr = 0.5$, $ls = 0.5$) is higher than HSA. In fact, this is the only case among our tests where the best performer (HSA here) does not display maximum feasibility. The difference between minimum and maximum feasibility for HSA in big instances ($15 - 9 = 6$) is the highest in all our tests, and points in the same direction.

But convergence rate might not be the whole story. The issue of scaling-up in genetic algorithms in the context of the same problem set has previously been addressed in [19]. The results of that study are represented by GGA and H algorithms, and, based on those results, a hypothesis is developed therein regarding the destructive tendencies of the grouping recombination operator as instance size grows. Subsequent empirical results corroborate this, leading the authors to posit that group-based operators might have "pitfalls in certain cases." The notion that recombination might be increasingly destructive as instance size grows is partly hinted at by our choice of MA's best performing parameters across the three sets: ($rr = 0.8$) is a solid performer in the case of small and medium instances, but when we get to the big instances, it is replaced by much less recombination ($rr = 0.2$). Nevertheless, as the results show, there is remarkable difference between the performance of the two algorithms and according to a Wilcoxon signed ranked test, they have significantly different underlying distributions. In order to carry out further analysis in this regard, we will first provide a more detailed outline of GGA as described in [19], and will also attempt to pick up the analysis where Lewis and Paechter left off.

7. Group-based recombination and timetabling

Within the framework of an evolutionary process, [19] adopts a variable-length solution, meaning that a varying number of time slots are allowed at first. This is in contrast to our MA that has a fixed number of time slots from the beginning. Later on, GGA imposes a reduction of the number of used time slots, discarding extra time slots through the use of distance-to-feasibility as a fitness measure. Thus the fitness function can be calculated as the number of events whose corresponding time slots have to be removed, to prevent the timetable from exceeding the allowed number of time slots. More information about GGA can be found in [19]. Here we tend to abstract away from irrelevant details and focus on the recombination operator as a device to investigate further points regarding the applicability of genetic operators to timetabling.

GGA is based on the conception of timetabling as a grouping problem, as defined earlier, and in this particular case the authors designate *entire* time slots as the groups which should be subject to evolutionary operators. The recombination operator goes through the four stages of Point Selection, Injection, Removal of Duplicates using Adaptation, and finally Reconstruction, and is similar to the recombination operator discussed earlier in some aspects. Nevertheless, time slots are judged to be the appropriate building blocks of the problem, so, for instance, Point Selection in GGA is limited to the start of each time slot only.

The choice of time slots has more serious repercussions though. When explaining the workings of MA's recombination operator, we noted that when duplicates arise and have to be removed, the algorithm goes about searching the entire timetable and

	800 seconds				1000 seconds		
no.	MA (rr = 0.5, ls = 0.2)	MA (rr = 0.5, ls = 0.5)	GGA	H	MA (rr = 0.5, ls = 0.2)	MA (rr = 0.5, ls = 0.5)	HSA
1	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
2	0 (0)	0 (0)	0.7 (0)	0 (0)	0 (0)	0 (0)	0 (0)
3	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
4	0 (0)	0 (0)	32.2 (30)	20.5 (8)	0 (0)	0 (0)	0 (0)
5	0.6 (0)	0 (0)	29.15 (24)	38.15 (30)	0.6 (0)	0 (0)	1.1 (0)
6	67.6 (58)	69.05 (54)	88.9 (71)	92.3 (77)	67.6 (58)	66.6 (52)	8.45 (5)
7	150.6 (146)	148.85 (142)	157.3 (145)	168.5 (150)	150.6 (146)	148.05 (142)	58.3 (47)
8	0 (0)	0 (0)	37.8 (30)	20.75 (5)	0 (0)	0 (0)	0 (0)
9	0 (0)	0 (0)	25 (18)	17.5 (3)	0 (0)	0 (0)	0.05 (0)
10	1.2 (0)	0.6 (0)	38 (32)	39.95 (24)	0.8 (0)	0.7 (0)	1.25 (0)
11	0.2 (0)	0 (0)	42.35 (37)	26.05 (22)	0 (0)	0 (0)	0.35 (0)
12	0 (0)	0 (0)	0.85 (0)	0 (0)	0 (0)	0 (0)	0 (0)
13	0 (0)	0 (0)	19.9 (10)	2.55 (0)	0 (0)	0 (0)	0 (0)
14	0 (0)	0 (0)	7.25 (0)	0 (0)	0 (0)	0 (0)	0 (0)
15	0 (0)	0 (0)	113.95 (98)	10 (0)	0 (0)	0 (0)	0 (0)
16	0 (0)	0 (0)	116.3 (100)	42 (19)	0 (0)	0 (0)	2 (0)
17	139.55 (120)	127.3 (117)	266.55 (243)	174.9 (163)	139.6 (120)	124.45 (116)	89.9 (76)
18	124.2 (118)	120.5 (107)	194.75 (173)	179.25 (164)	124.2 (118)	118.75 (107)	62.6 (53)
19	209.45 (197)	216.8 (207)	266.65 (253)	247.35 (232)	206.6 (195)	214.5 (207)	127 (109)
20	122.8 (117)	117.7 (111)	183.15 (165)	164.15 (149)	121.8 (117)	117.35 (111)	46.7 (40)
AVG & STD	40.78 & 67.99 (37.8 & 63.63)	40.04 & 67.45 (36.9 & 63.31)	81.0 & 86.33 (71.5 & 80.3)	62.19 & 78.52 (52.3 & 72.6)	40.56 & 67.58 (37.7 63.37)	39.52 & 66.69 (36.75 & 63.22)	19.83 & 36.94 (16.5 & 31.5)
CV	1.66 (1.68)	1.68 (1.71)	1.06 (1.12)	1.26 (1.38)	1.66 (1.68)	1.68 (1.72)	1.86 (1.90)
Minimum Feasibility	14	14	5	7	14	14	15
Maximum Feasibility	11	13	2	5	12	13	9
Best Performer	13	18	2	5	12	14	15

Fig. 7. Performance results, for the big set, in comparison with three other algorithms. For each instance, an average of 20 runs is reported, with the best result noted in parentheses. Other than those denoted, parameters for the algorithm were fixed at: $mr = 1$, $p = 40$. The parameters for GGA are reported as $rr = 0.1$, $mr = 1$, $ls = 100$, $p = 5$ for small, $rr = 0.7$, $mr = 1$, $ls = 2$, $p = 10$ for medium, and $rr = 0.25$, $mr = 1$, $ls = 0$, $p = 50$ for big sets respectively.

removing duplicates as it spots them. GGA's recombination operator does not deal with the issue in a similar way: it removes the *entire* time slot containing a duplicate event, and builds new time slots from scratch to accommodate any unscheduled events. Thus the two duplicate removal procedures have delicate differences. The rationale behind the design of MA's operator is that once duplicates are removed, news slots are opened up and this means new opportunities for unscheduled events to fit in, which in turn, results in fewer unscheduled events if we have a fixed number of time slots (like MA), or fewer extra time slots in the case of a variable-length solution (like GGA). Through considering the newly-opened slots, MA refuses to limit itself to evolving only time slots, and might even branch out to evolve separate lineages *within* a single time slot. GGA's commitment to evolving time slots from scratch, on the other hand, keeps it from "digging inside" time slots and improving existing time slots from which duplicates have been removed. Hence, in the case of GGA, removing individual duplicate events while keeping the time slot can result in offspring that, as Lewis and Paechter rightfully point out, "would actually be poor in quality because it would almost certainly be using more time slots than either of the two parents." With this theoretical backdrop, we decided to compare the performance of the two recombination operators.

To measure the destructive tendencies of each algorithm, the aforementioned GGA (complete with recombination and mutation operators, heuristics and fitness function) was developed. As a measure of how much these recombination operators help/hinder the search process, we decided to quantify the amount of repair by calculating the net number of events that have been removed after each application of the recombination operator. Construed this way, a positive amount of repair means that the operator has, in sum, removed events from the timetable, while a negative repair value means that the operator has, in sum, been successful in scheduling more events than it has removed. Keeping track of unscheduled events is straightforward in the case of MA, since it entails counting the number of events that occupy the unscheduled list. In the case of GGA, since it does not keep such a list, the number of events occupying all the "extra time slots" is calculated instead, where "extra time slots" are judged to be the ones that are least-densely packed.

To further the analysis, the level of diversity sustained by each recombination operator is also measured based on the following observations:

- potentially, every set of two events we might pick can coexist on a time slot, unless this coexistence is disallowed by conflict constraints. Thus subtracting from possible two-by-two event pairings, all pairings that include conflicting events, yields a theoretical limit for the number of legal event pairings;
- diversity might be operationalized in terms of the diversity of event pairings, that is by measuring the degree to which *possible* event pairings are "realized" in the current population;

A distance-to-diversity measure is calculated using the following formula:

$$f = \sum_{i=1}^{n} \sum_{j=2}^{n} [1 - r_{ij}] \quad (4)$$

where the total number of events is denoted by n, and r_{ij} equals 1 when events i and j are in conflict or there exists a time slot in the population that contains both events, and 0 otherwise.

The results of our measurements for a particularly hard instance of each size are demonstrated in Fig. 8. Note that, the two recombination operators have been stripped down to their bare minimums, and various algorithmic parameters (i.e. initial population count, operator rates, etc.) have been equalized. Also, as their fitness criteria, both algorithms count the number of events that remain unscheduled. For the fixed-sized formulation of MA, this means simply counting the events on the unscheduled list, but for the varying-sized formulation of GGA, as noted earlier, this amounts to counting the number of events, whose corresponding time slots have to be removed, to prevent the timetable from exceeding the allowed number of time slots. Controlling for variables such as parameter rates and fitness evaluation enables us to make meaningful comparisons between the two operators.

Looking at Fig. 8, the first thing to notice is that, as expected, the amount of required repair increases with instance size for both algorithms. This is to be expected, since as time slots grow in size, the chance that a time slot, injected from another timetable, might share events with more time slots of the host timetable increases and this, in turn, escalates the cost of the recombination operator. However a closer look reveals that the relative difference between the measured repair for the two recombinations also increases significantly, and the incurred repair in the case of our operator depends much less on instance size. Quantitatively, as illustrated by Fig. 9, compared to our recombination, the group-based recombination requires $6.25/1.25 = 5$, $74.87/5.72 \approx 137$, and $451.7/12.41 \approx 36$ times more repair for the three instance sizes, respectively.

Regarding the level of diversity, the values for MA are higher in all three instances, indicating higher distance-to-diversity and consequently lower average population diversity. To be sure, due to their different initialization methods, the two algorithms embark on vastly different paths from the beginning but, except for the small instance where its level of diversity drops on average in the course of the subsequent generations, GGA's already higher level of diversity does not decline on average with time in the case of the medium and big instances. Arguably, the implications of maintaining a high/low level of diversity depend, in part, on the amount of repair it incurs on the operator, which is obvious in light of the "mirror-like" relationship between diversity and required repair, evident in Fig. 8, for both algorithms. The high correlation between diversity and repair, as demonstrated in Fig. 9, can be explained by reviewing the process by which any repair can increase diversity. The amount of required repair essentially determines how much work the algorithm has to do from scratch again, and since both algorithms go about doing this using essentially random processes, it is only natural that they work to increase diversity, as any randomization process is required to. Thus, the high level of repair puts GGA at a disadvantage in two ways: a temporary one (i.e. by incurring the cost of repair), but also a perpetual one (i.e. by maintaining a high level of diversity in the subsequent generation).

In sum, we think the difference in the obtained results of MA and GGA, at least partly, stems from the reductive commitment of GGA to recognizing time slots as the only appropriate building block of the problem. As noted earlier, the intuition behind grouping formulations was to be able to pass on meaningful structures

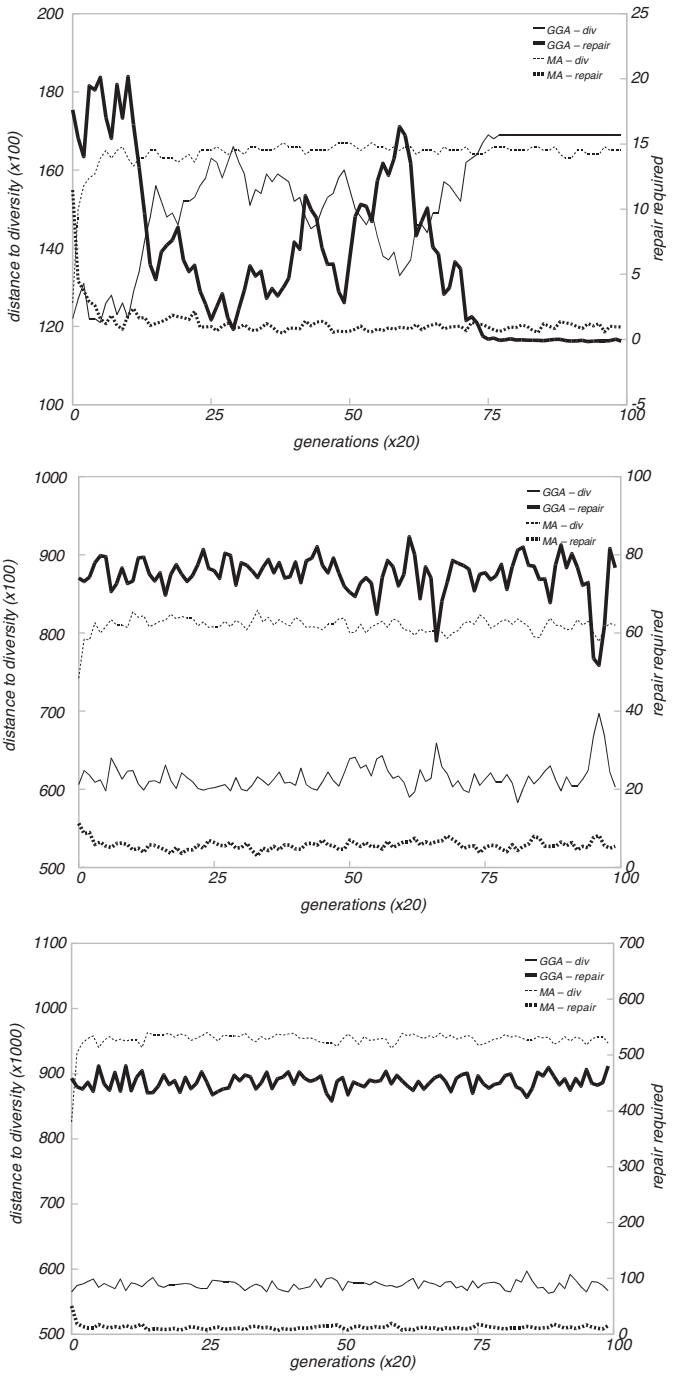

Fig. 8. Comparing the performance of the memetic and group-based recombination operators. The results for an instance of small, medium, and big sets has been demonstrated in the top, middle, and bottom diagrams with $rr = 1.0$, $mr = 1$, $ls = 0.0$, and $p = 40$ across 2000 generations. The adopted configurations serve the purpose of abstracting away from other algorithmic aspects and focusing on recombination.

from one generation to another, but the high amount of repair GGA suffers from, is a direct consequence of the aforementioned commitment to evolving time slots only. Hence it can be argued that while the idea of evolving optimal event assignments is meaningless (since the optimality of any single assignment depends in part on other assignments), evolving entire time slots might impose an artificial restriction on the grouping of events, as discussed earlier. We propose instead, that the level of granularity alternate between arbitrary groupings of events in the recombination, local search and mutation operators.

		GGA			MA		
		Distance To Diversity	Required Repair	Correlation Coefficient	Distance To Diversity	Required Repair	Correlation Coefficient
s.14	mean	15340.07	6.25	-0.99	16462.68	1.25	-0.99
	std	1426.64	5.97		440.27	1.2	
m.20	mean	61550.11	74.87	-0.88	80986.6	5.72	-0.92
	std	1679.92	5.22		1035.62	1.23	
b.5	mean	576511.21	451.7	-0.77	953152.98	12.41	-0.97
	std	6449.57	13.41		14112.51	4.73	

Fig. 9. Comparing the distance-to-diversity and the required repair, for the recombination operators of GGA and MA across 2000 generations for three instances. The correlation coefficient between the values of these two parameters is also displayed.

8. Conclusion

The idea behind this paper was to investigate the applicability of evolutionary algorithms to the course timetabling problem. Accordingly, we have attempted to develop a memetic algorithm to solve a predefined set of benchmark problems. In comparison with three other algorithms noted in the paper, our memetic algorithm performed better in the case of small and medium instances, but fell behind one of the algorithms in the case of big instances. Therefore, we might be able to continue this research by focusing on solving a big problem ($n \approx 1000$) by dividing it into constituent subproblems ($n \approx 500$) that might be better handled by the algorithm. This of course, makes a number of assumptions regarding problem constraints and solution landscape that might not always be viable. Nevertheless, the algorithm's success in dealing with medium-sized problems ($n \approx 400$) makes this a worthy future endeavor.

Following a general analysis of algorithmic performance, we focused on the difference between the obtained results of our own algorithm and a group-based genetic algorithm. We implemented the group-based algorithm together with a diversity measure. The analysis that followed corroborated the hypothesis of [19] regarding the "pitfalls" of the grouping algorithm. We argued that an exclusive focus on evolving time slots, in effect hinders the transmission of evolved structures from one generation to the next, and attributed the better performance of the memetic algorithm to using a multi-level selection scheme in general, and a recombination operator that did not limit itself to groups, in particular.

References

[1] E.K. Burke, D. de Werra, J. Kingston, Application to timetabling, in: J. Gross, J. Yellen (Eds.), Handbook of Graph Theory and Applications, CRC Press, 2003, pp. 445–474.
[2] S. Even, A. Itai, A. Shamir, On the complexity of time table and multi-commodity flow problems, in: 16th Annual Symposium on Foundations of Computer Science, 1975, pp. 184–193.
[3] E.K. Burke, S. Petrovic, Recent research directions in automated timetabling, European Journal of Operational Research 140 (2002) 266–280.
[4] O. Rossi-Doria, M. Sampels, M. Birattari, M. Chiarandini, M. Dorigo, L. Gambardella, J. Knowles, M. Manfrin, M. Mastrolilli, B. Paechter, et al., A comparison of the performance of different metaheuristics on the timetabling problem, in: Practice and Theory of Automated Timetabling (PATAT) IV, 2740, 2003, pp. 329–351.
[5] R. Lewis, A survey of metaheuristic-based techniques for university timetabling problems, OR Spectrum 30 (2008) 167–190.
[6] K. Socha, M. Sampels, M. Manfrin, Ant algorithms for the university course timetabling problem with regard to the state-of-the-art, Applications of Evolutionary Computing (2003) 334–345.
[7] K. Socha, J. Knowles, M. Sampels, A max–min ant system for the university course timetabling problem, Ant Algorithms (2002) 63–77.
[8] N.R. Sabar, M. Ayob, G. Kendall, R. Qu, A honey-bee mating optimization algorithm for educational timetabling problems, European Journal of Operational Research 216 (2012) 533–543.
[9] D. Abramson, M. Krishnamoorthy, H. Dang, Simulated annealing cooling schedules for the school timetabling problem, Asia Pacific Journal of Operational Research 16 (1999) 1–22.
[10] D. Zhang, Y. Liu, R. M'Hallah, S. Leung, A simulated annealing with a new neighborhood structure based algorithm for high school timetabling problems, European Journal of Operational Research 203 (2010) 550–558.
[11] P. Pongcharoen, W. Promtet, P. Yenradee, C. Hicks, Stochastic optimisation timetabling tool for university course scheduling, International Journal of Production Economics 112 (2008) 903–918.
[12] C. Aladag, G. Hocaoglu, M. Basaran, The effect of neighborhood structures on tabu search algorithm in solving course timetabling problem, Expert Systems with Applications 36 (2009) 12349–12356.
[13] Z. Lü, J. Hao, Adaptive tabu search for course timetabling, European Journal of Operational Research 200 (2010) 235–244.
[14] W. Erben, A grouping genetic algorithm for graph colouring and exam timetabling, in: Practice and Theory of Automated Timetabling (PATAT), III, 2001, pp. 132–156.
[15] P. Ross, E. Hart, D. Corne, Genetic algorithms and timetabling, in: Advances in Evolutionary Computing, Springer-Verlag, Inc., New York, 2003, pp. 755–771.
[16] E.K. Burke, J. Newall, A multi-stage evolutionary algorithm for the timetable problem, IEEE Transactions on Evolutionary Computation 3 (1999) 63–74.
[17] N. Pillay, W. Banzhaf, An informed genetic algorithm for the examination timetabling problem, Applied Soft Computing 10 (2010) 457–467.
[18] G. Beligiannis, C. Moschopoulos, G. Kaperonis, S. Likothanassis, Applying evolutionary computation to the school timetabling problem: the Greek case, Computers & Operations Research 35 (2008) 1265–1280.
[19] R. Lewis, B. Paechter, Finding feasible timetables using group-based operators, IEEE Transactions on Evolutionary Computation 11 (2007) 397–413.
[20] E. Burke, J. Newall, R. Weare, A memetic algorithm for university exam timetabling, Practice and Theory of Automated Timetabling (PATAT) (1996) 241–250.
[21] A. Alkan, E. Ozcan, Memetic algorithms for timetabling, in: The 2003 Congress on Evolutionary Computation. CEC'03, volume 3, IEEE, 2003, pp. 1796–1802.
[22] C. Cotta, A. Fernàndez, Memetic algorithms in planning, scheduling, and timetabling, Evolutionary Scheduling (2007) 1–30.
[23] L. Agustín-Blas, S. Salcedo-Sanz, E. Ortiz-García, A. Portilla-Figueras, Á. Pérez-Bellido, A hybrid grouping genetic algorithm for assigning students to preferred laboratory groups, Expert Systems with Applications 36 (2009) 7234–7241.
[24] D. Qaurooni, A memetic algorithm for course timetabling, in: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, ACM, 2011, pp. 435–442.
[25] M. Carter, ftp://ftp.mie.utoronto.ca/pub/carter/testprob, 2010.
[26] O. Rossi-Doria, M. Sampels, M. Birattari, M. Chiarandini, M. Dorigo, L. Gambardella, J. Knowles, M. Manfrin, M. Mastrolilli, B. Paechter, L. Paquete, T. Sttzle, http://iridia.ulb.ac.be/supp/IridiaSupp2002-001/index.html, 2010.
[27] B. McCollum, http://www.cs.qub.ac.uk/itc2007/Login/SecretPage.php, 2010.
[28] B. McCollum, A. Schaerf, B. Paechter, P. McMullan, R. Lewis, A. Parkes, L. Gaspero, R. Qu, E. Burke, Setting the research agenda in automated timetabling: the second international timetabling competition, INFORMS Journal on Computing 22 (2010) 120–130.
[29] B. McCollum, http://www.cs.qub.ac.uk/itc2007/winner/finalorder.htm, 2010.
[30] R. Lewis, http://www.emergentcomputing.org/timetabling/harderinstances, 2005.
[31] M. Tuga, R. Berretta, A. Mendes, A hybrid simulated annealing with kempe chain neighborhood for the university timetabling problem, in: 6th IEEE/ACIS International Conference on Computer and Information Science, IEEE, 2007, pp. 400–405.
[32] B. Paechter, http://www.idsia.ch/Files/ttcomp2002, 2010.
[33] E. Falkenauer, Genetic Algorithms and Grouping Problems, John Wiley & Sons, Inc., New York, NY, USA, 1998.
[34] P. Moscato, C. Cotta, A gentle introduction to memetic algorithms, in: Handbook of Metaheuristics, Springer, 2003, pp. 105–144.

[35] D. Knuth, The Art of Computer Programming, volume 2, Addison-Wesley, Boston, 1998.
[36] R. Weare, E.K. Burke, D. Elliman, A Hybrid Genetic Algorithm for Highly Constrained Timetabling Problems, Department of Computer Science, 1995.
[37] R.K. Ahuja, J.B. Orlin, D. Sharma, Very large-scale neighborhood search, International Transactions in Operational Research 7 (2000) 301–317.
[38] B. Craenen, A. Eiben, Stepwise adaption of weights with refinement and decay on constraint satisfaction problems, in: L. Spector, E. Goodman, A. Wu, W. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, E. Burke (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001), Morgan Kaufmann Publishing, San Francisco, CA, 2001, pp. 291–298.
[39] R. Lewis, B. Paechter, http://www.idsia.ch/Files/ttcomp2002/IC_Problem/checksln.cpp, 2010.