A Memetic Algorithm for Course Timetabling

Dan Qaurooni
Faculty of Mathematics and Computer Science
Amirkabir University of Technology
Tehran, Iran
dani.qaurooni@aut.ac.ir

ABSTRACT

Course timetabling consists in scheduling a sequence of lectures while satisfying various constraints. In this paper, we develop and study the performance of a memetic algorithm, designed to solve a variant of the course timetabling problem. Our aim here is twofold: to develop a competitive algorithm, and to investigate, more generally, the applicability of evolutionary algorithms to timetabling. To this end, an algorithm is first introduced and tested using a benchmark set. Comparison with other algorithms shows that our algorithm achieves better results in some, but not all instances, signifying strong and weak points. Subsequently, more comprehensive analyses are performed in relation with another evolutionary algorithm that uses strictly group-based operators. Ultimately, empirical results and analyses lead us to question the exclusive use of group-based evolutionary operators for timetabling problems.

Evolutionary Combinatorial Optimization and Metaheuristics

Categories and Subject Descriptors

I.2.8 [**Computing Methodologies**]: Artificial Intelligence—*Problem Solving, Control Methods, and Search*

General Terms

Algorithms

Keywords

Time-tabling and scheduling, Genetic algorithms, Local search, Adaptation/ self-adaptation, Combinatorial optimization

1. INTRODUCTION

The problem of automating the task of timetabling concerns the scheduling and assignment of a set of institutional events to a number of rooms and time slots such that they satisfy specific constraints. Naturally, many variants of the timetabling problem have been proposed and studied through the years. The main source of variation here involves problem constraints, consisting mostly of various limitations, that rule out many potential assignments of events to rooms and time slots. A good survey of the various timetabling subclasses can be found in [11].

Among timetabling subclasses, educational timetabling is arguably the most widely studied and as its name implies, focuses on constraints related to teachers, lecture hours, facilities, program conflicts, room capacities etc. In this paper we focus on one branch of educational timetabling called the University Course Timetabling Problem (UCTP). The constraints of UCTP are divided into two main categories of hard and soft, denoting issues of feasibility and preference, respectively. Hard constraints deal with constraints that should no be ignored, the main type of which concern event clashes that happen when two events that share participants are scheduled to be held at the same time. Violations of such constraints render the timetable infeasible. Unlike hard constraints and its focus on feasibility however, soft constraints opt for a more optimal timetable that better suits the convenience of the institution, the participants, or both.

Due to the importance of the university timetabling problem and its NP-Completeness [13], a diverse array of solution-building approaches have been proposed. Some more recent general surveys of algorithmic performance have been carried out in [25, 18]. Here we focus on the application of metaheuristics which have been widely utilized for solving timetabling problems. For instance, in the framework of Ant Colony Optimization, the authors of [27, 26] use each ant to construct a 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. Simulated annealing (SA) has been implemented in [30, 23, 28] among others. Specifically, [30] reports that the used heuristic improves upon all previous efforts based on SA. Some recent tabu search algorithms are [9, 17]. In particular, [17] reports the application of an Adaptive Tabu Search algorithm which integrates several features such as an original double Kempe chains neighborhood structure, a penalty-guided perturbation operator and an adaptive search mechanism, all of which combine to achieve remarkable results.

More specifically, evolutionary algorithms (EA) have been applied to timetabling problems with varying degrees of success in the works of [15, 24, 12, 22, 19] among others. For example, [15] is among the first to remedy the poor per-

formance of a genetic algorithm compared with other conventional methods by way of a grouping encoding. Also, many EAs have employed separate techniques to make up for their potential failure in dealing with local optima. One such technique, that we will later utilize, is variously dubbed memetic, Lamarckian, hybrid or cultural [21], and enhances the simple transition process of an otherwise evolutionary structure by including problem-specific knowledge such as heuristics, local search operators etc. to boost performance. Memetic algorithms have been applied to timetabling in the works of [10, 14, 19]. In recent years, two International Timetabling Competitions have brought many researchers in timetabling together. An overview of the more recent 2nd event can be found in [20]. Further information on the winning algorithms can be found in [6].
In the next section, a more thorough description of our specific timetabling problem is provided. The remainder of the paper is then organized as follows: in section 3, an outline of our memetic algorithm for university course timetabling is provided. This includes a description of the workings of the algorithm as well as a more focused discussion of the fitness function. In section 4, experimental parameters are provided and the algorithm is put to test. Comparisons with three other algorithms of the literature follow in section 5, which demonstrate the superior performance of our memetic algorithm in two of the three instance sets. The focus of attention, however, is a comparison of our own algorithm in contrast to another evolutionary algorithm, which turn out to be revealing with regards to the applicability of GAs to timetabling. This investigation leads to further tests and the development of a diversity measure that is used to contrast the aforementioned algorithms. Following further analyses, we close in section 6 with conclusions and future directions for possible research.

2. PROBLEM DESCRIPTION

As stated formally in [11], timetabling has four parameters: T, a finite set of times; R, a finite set of resources; M, a finite set of meetings; and C, a finite set of constraints. The problem, then, is to assign times and resources to the meetings so as to satisfy as many constraints as possible. In other words, timetabling is the problem of assigning events to rooms and time slots according to specific requirements imposed by certain constraints. The specific branch of timetabling that we focus on is the university timetabling problem that was first used in the First Timetabling Competition [1]. Here, each problem instance has a set of rooms with predefined sizes, a set of events with attendance specifications for each event, a set of room features, a subset of which is provided by each room and required by each event. The hard constraints of the problem (that have to be satisfied to produce a feasible timetable) include

- H1: no student is required to attend more than one event at any one time;
- H2: only one event is assigned to any room in any time slot;
- H3: all of the features that are required by the event are satisfied by the room, which has an adequate capacity.

A feasible solution to a timetabling problem is an assignment of each event to one room and one of 45 time slots (five days, each with nine time slots) in such a way that none of the aforementioned hard constraints are violated. Although a complete timetabling algorithm attends to both hard *and* soft constraints, we ignore soft constraints and focus on hard constraints that produce feasible or "working" timetables, for reasons that will be explained later in section 4.

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 a two-dimensional array, will pick out either a certain event specified by its number, say n ($r_{ij} = n$), or a vacancy. It is important to note that 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. Also, from the outset, the number of alloted time slots is fixed at the total number of usable time slots (which is 45 in this case).
Within such a framework, a general outline of our memetic algorithm is described in Figure 1. The first phase of the algorithm, denoted by the Initialize method, generates the initial population in a greedy manner, similar to that of [29]. By the time Initialize is done, several (rather sparse) timetables are created, each with a list of events that have not been scheduled.

```
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 $P$.
```

Figure 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.

When initialization is over, the population goes through 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

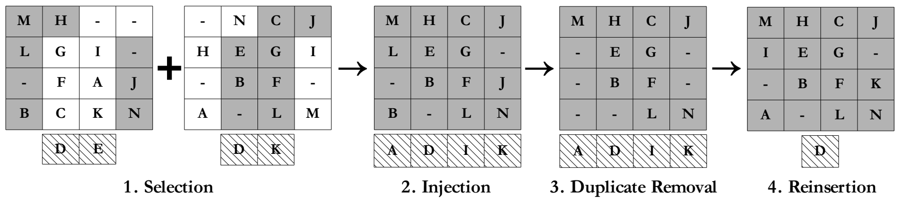

Figure 2: Illustrating the process of recombination as it goes through its four stages. Two parent timetables with hypothetical events ranging from A to N are to be feasibly scheduled in a four by four timetable with sixteen slots. The hatched items below each timetable represent its list of unscheduled events. Selected boundaries are highlighted with gray.

chosen from the intersection of rooms and time slots to form a range.

2. **Injection**: The selected range from one parent is replaced by the corresponding range in 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. Figure 2 illustrates this process with an example.

Next is the time slot-based mutation operator, where a specified number of time slots are randomly selected and emptied out into the unscheduled list. This list is then shuffled and traversed. 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.

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 is based on the concept of a clique from graph theory. A simplified timetabling problem, with its requirements of facility and size relaxed, might be formulated in graph theoretical terms as follows: a timetable T with a set of events E and conflict constraints C is analogous to a graph G with a set of vertices V for its corresponding events, and edges E only between pairs of vertices whose corresponding events conflict with one another. Thus construed, graph G', the complement of graph G, will have edges between vertices that correspond to events that can form a conflict-free group together. In other words, a clique - a subset of the vertices of a graph where each pair is connected by an edge - in graph G', is analogous to a group of events that, other hard constraints allowing, can fit in a single time slot together. Based on this analogy, the conflict-based local search operator searches each timetable with the aim of expanding some cliques of events (at the expense of reducing others).

The second local search operator, illustrated with an example in Figure 3, receives the timetable at a stage when the algorithm has done all it can to improve timetable quality. 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 it has to climb down the local peak and tolerate a few "bad" moves, at least temporarily. With this in mind, the second local search procedure operates in three stages: Slot selection, Shoving, Reinsertion.

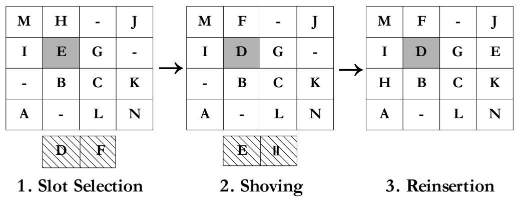

Figure 3: The three phases of the second local search operator, as described in the text. It is assumed that the event marked by the letter D has no feasible assignments in the current setting of the timetable.

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 empty places 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 of 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 (note: students, *not* events) in clash with it, 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 aforementioned steps have been carried out for *all* originally unscheduled events, a new list of unscheduled events emerges. These might be feasibly placed 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 subsection, we discuss how best to complement the algorithm with a suitable fitness function.

3.1 Fitness function

A natural choice for evaluating timetables might be to count the number of unplaced events. However such a fitness function misses the point 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 some detail. Also, for highly constrained problems such as 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 for the fitness function:

1. To take solution structure into account;
2. To learn from experience;
3. To keep computation costs as low as possible.

The optimal trade-off between these criteria adds enough detail to distinguish between dissimilar states while keeping the computation cost affordable in the long run. With this in mind, to address the first criterion in our problem, any possible mismatch between the facilities and capacities of rooms on the one hand, and the requirements of events on the other hand can be taken into account. To address the second criterion, we introduced a new index that we hereon refer to as the shoved index. For each event, the shoved index is initialized to the number of students that are in conflict with it, and is incremented every time the event is shoved by the second local search operator. As a result of this increment, the fitness of an individual that has not scheduled that event decreases. Since this index is then shared among all individuals, when calculating fitness values, this allows for the cumulative "experience" of the population in dealing with a set of events to be taken into account. This, of course, is 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.
The above observations are captured in the equation below:

$$f = \sum_{i=1}^{n} [(1 - P_i) \sum_{j=1}^{m} RF_{E_{i}j}(1 - EF_{ij}) + (C_{R_{E_i}} - A_{E_i}) + (P_i)(S_i + A_i)] \quad (1)$$

While the equation looks formidable, it 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_i says 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 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 of the fitness function, which deals with unscheduled events, S_i represents the total number of times event i has been shoved, while A_i denotes the number of attendants of event i. It is important to note that while 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.

4. EXPERIMENTAL SETUP

Various problem sets have been proposed through the years. Standard benchmark packages for course timetabling include [4, 5, 2, 7]. However, such packages are usually designed with hard *and* soft constraints in mind, hence finding feasibility is rarely difficult. In fact, many of these focus on soft constraints and feasible solutions are found so fast as to render any comparison and analysis meaningless. In such a situation, the benchmarking instances provided in [3] 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, medium, and big size (approximately 200 events and 5 rooms, 400 events and 10 rooms, and 1000 events and 25 rooms, respectively). The instances are called "hard" since, according to [19], 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 PC with a 2.4GHz CPU and 512 MBs of memory. Also, two sets of time limits (30, 200, and 800 seconds, and 200, 500, and 1000 seconds) were imposed for instances of small, medium and big sizes. To check the integrity of our particular implementation and the solutions produced, the test program of [8] was used.
The selection process is rank-based, stochastically preferring timetables with higher fitness values. 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 clique-based 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. The adopted parameter values are chosen to further our analyses.

5. ANALYSIS OF PERFORMANCE

The results of the tests for the 60 instances are demonstrated in Figure 4. Since different time limits have been used by [19] and [28] as their stopping criteria, we had to divide the table into two sets of columns:

- the four columns on the left of the dividing line are subject to the original time limits of 30, 200, and 800 seconds for small, medium and big instances respectively. MA signifies our Memetic Algorithm, GGA is a Grouping Genetic Algorithm, similar to our algorithm in many respects, that we will thoroughly analyze later on in the paper, and H is a single thread local search

Small instances

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)
R	15	15	11	15	17	17	15
E	7	7	1	5	4	6	0

Medium instances

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)	0.123 & 0.38 (0 & 0)	0.123 & 0.36 (0 & 0)	1.84 & 3.07 (0.2 & 0.52)
R	17	16	6	13	18	16	12
E	13	12	0	7	7	6	0

Big Instances

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	150.65 (146)	148.85 (142)	157.3 (145)	168.5 (150)	150.65 (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)	0.55 (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)	138.85 (120)	124.45 (116)	89.9 (76)
18	124.2 (118)	120.55 (107)	194.75 (173)	179.25 (164)	124.2 (118)	118.75 (107)	62.6 (53)
19	209.45 (197)	216.85 (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	40.81 & 67.97 (37.8 & 63.63)	40.02 & 67.48 (36.9 & 63.31)	81.0 & 86.33 (71.5 & 80.3)	62.19 & 78.52 (52.3 & 72.6)	40.52 & 67.54 (37.7 63.37)	39.51 & 66.69 (36.75 & 63.22)	19.89 & 36.91 (16.5 & 31.5)
R	11	13	2	5	12	13	9
E	10	15	0	3	3	5	6

Figure 4: Performance results 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 the 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.

Heuristic algorithm that works by improving the packing of individual time slots. The results for these two algorithms are based on tests that have been performed on a 2.66 GHz pentium processor with 1 GBs of RAM. The interested reader is referred to [19] for more information.

- the three columns on the right, on the other hand, have time limits of 200, 500 and 1000 for small, medium and big instances respectively.[1] HSA refers to the Hybrid Simulated Annealing, defined in [28], which uses a succession of problem relaxation, Kempe chain and graph heuristics, and simulated annealing to tackle the problem. The reported results are based on tests that have been performed on a 3.2 GHz pentium processor.

To compare different aspects of algorithmic performance, mean values, standard deviations and two other measures, labeled R and E in Figure 4 are reported. The mean value 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 algorithmic performance across an entire set. Looking at the general results for the four configurations on the left (which, as noted earlier, are subject to a more strict time limit), it is obvious that the Memetic Algorithm (MA) performs better than the Grouping Genetic Algorithm (GGA) and the Heuristic Algorithm (H) in all three instance sets, with the best performance achieved in the setting $rr = 0.8, ls = 0.5$ in both small and medium sets, and $rr = 0.5, ls = 0.5$ in the big set. In case of the algorithms in the three columns on the right (that are generally allowed more time), the configuration with $rr = 0.8, ls = 0.5$ outperforms HSA for small and medium sets. However, here, HSA bests MA in the big set in terms of mean performance and variability. In all of the above cases, the algorithm with the better mean, also had the lowest standard deviation, signifying the close relationship between the two. The bottom row of each set includes another index, E, which keeps track of the number of instances for which an algorithm has achieved best results (highlighted in gray). To increase clarity, we only counted the cases where best results are not achieved by all participating algorithms. Based on such criteria, both configurations of MA have higher values and indicate better performance.

When focusing on patterns across the three instance sizes, it became clear that other than the significant drop in performance, the behavior of MA in the big instances displayed another significant trait: We observed that while the algorithm actively switched between different alternatives in the solution space in the case of small and medium instances, it failed to do so in the case of big instances. In fact, the runs scarcely displayed any significant improvement after the first 600 seconds or so. This is especially troublesome for big instances such as no. 19, where more than 200 events (20% of the total) remain unplaced. This suggests that MA has a high convergence rate, and while this clearly aids performance in the case of small and medium instances, it is detrimental when it comes to big instances. This suggestion

[1]Although a time limit of 400 seconds is reported for medium instances in [28], 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 seconds.

is further backed by the calculated value of R, the number of instances of each size for which feasibility is achieved in all runs. With regards to R, MA shows better values in all categories. The higher convergence rate might help to explain this, especially in the case of big instances, where HSA has a better average performance than MA. As demonstrated, among the instances where both algorithms are able to find feasibility (instances 5, 9, 10, 11, 16) MA achieves feasibility on more occasions, hence the higher R value.

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 by [19]. Therein, the authors partly blame the destructive tendencies of their recombination operator for the loss of performance as instance size increases. This is partly hinted at by our choice of best performing parameters for the MA. Whereas the configuration which includes a high rate of recombination ($rr = 0.8$) is the best performer in small and medium instances, in the big set, it is replaced by another configuration with much less recombination ($rr = 0.2$).

Despite this, looking at the performance of the MA and the GGA, we think that the difference in their obtained results merits a separate analysis, especially since the two algorithms are structurally similar in many respects. Yet, as observing Figure 4 might have shown, and according to a Wilcoxon signed rank test, the results point to significantly different underlying distributions for these two algorithms. More information about the GGA can be found in [19]. Here we tend to abstract away from irrelevant details and focus on the results of the two algorithms as a device to investigate further points regarding the applicability of various genetic operators to timetabling. To this end, it might be helpful to sketch out a general outline of the group-based genetic algorithm of [19]. Firstly, the algorithm defined therein is based on the conception of timetabling as a grouping problem, which is defined in [16] as a problem where the task is to partition a set of objects U into a collection of mutually disjoint subsets u_i of U, where

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

according to a set of problem-specific constraints that define valid and legal groupings. It is argued that when dealing with grouping problems, traditional genetic operators tend to break up, rather than improve, the building blocks produced in the previous round of the algorithm. This highlights an issue of granularity and is a consequence of the item-oriented (rather than group-oriented) choice of building blocks. Moreover, when focusing on items (as opposed to groups), the search space 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. In light of this, what distinguishes group-based operators is their focus on groups (as opposed to items).

Applying these insights to the case of timetabling and in order to investigate the merits of group-based operators in this case, [19] departs from item-oriented genetic operators ("items" here being individual events) and focuses on time slots as the appropriate building block of the problem. Thus construed, *entire* time slots are subject to evolutionary operators such as recombination and mutation. This is manifested, for instance, in the design of the recombination operator in that, while rather similar in other respects to MA's recombination, the choices for injection points are only limited to the start of each time slot. Also, the operator removes the *entire* time slot containing a duplicate event, and builds new time slots from scratch to accommodate unscheduled events. The GGA also adopts a variable-length solution, meaning that a varying number of time slots are allowed at first. Later on, the algorithm encourages a reduction of the number of used time slots through the use of distance to feasibility as a fitness measure.

With this general outline in mind, we take the analysis of the authors of [19] as a starting point. Based on their obtained results, which were demonstrated in section 4, a hypothesis is developed therein regarding the destructive tendencies of the grouping recombination operator as instance size grows. Subsequent empirical results corroborate this, leading them in the end to posit that group-based operators might have "pitfalls in certain cases." Picking up where they left off, and in order to measure the destructive tendencies of the group-based recombination operator as opposed to our own, we first need to develop a measure of diversity, in the spirit of [19] but in accordance with our operator design. The resulting measure of distance to diversity is calculated using the following formula:

$$f = \sum_{i=1}^{n} \sum_{j=2}^{n} [1 - R_{ij}] \tag{3}$$

This essentially counts the pairings of events that are allowed by the problem formulation (meaning that they do not violate conflict constraints), but have not been "realized" in the current population as a distance to an ideal measure of diversity. In the formula, 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.

To measure the destructive tendencies of each algorithm, the aforementioned grouping algorithm (complete with recombination and mutation operators, its defined heuristics and procedures) was developed and the amount of repair that is required after each application of the recombination operator, and the level of diversity sustained by each recombination operator was measured. The results for one instance of each size are demonstrated in Figure 5. The first thing to notice is that the amount of repair increases with instance size for both algorithms. However a closer look reveals that the relative difference between the measured repair for the two recombinations also increases significantly, since the incurred repair in the case of our operator depends much less on instance size. Quantitatively, the group-based recombination, compared to our recombination, requires $6.25/1.25 = 5$ times more repair for the small instance, $74.87/5.72 \approx 137$ times for the medium instance and $451.7/12.41 \approx 36$ more repair for the big instance.

Regarding the level of diversity, the trends in all three cases demonstrate a quick drop in the level of diversity in the case of our recombination, whereas the group-based recombination maintains its diversity throughout the run. The implications are obvious in light of the "mirror-like" relationship between diversity and required repair that is evident in both algorithms. The calculated correlation between diversity and repair turned out to be -0.99, -0.92, and -0.97 for

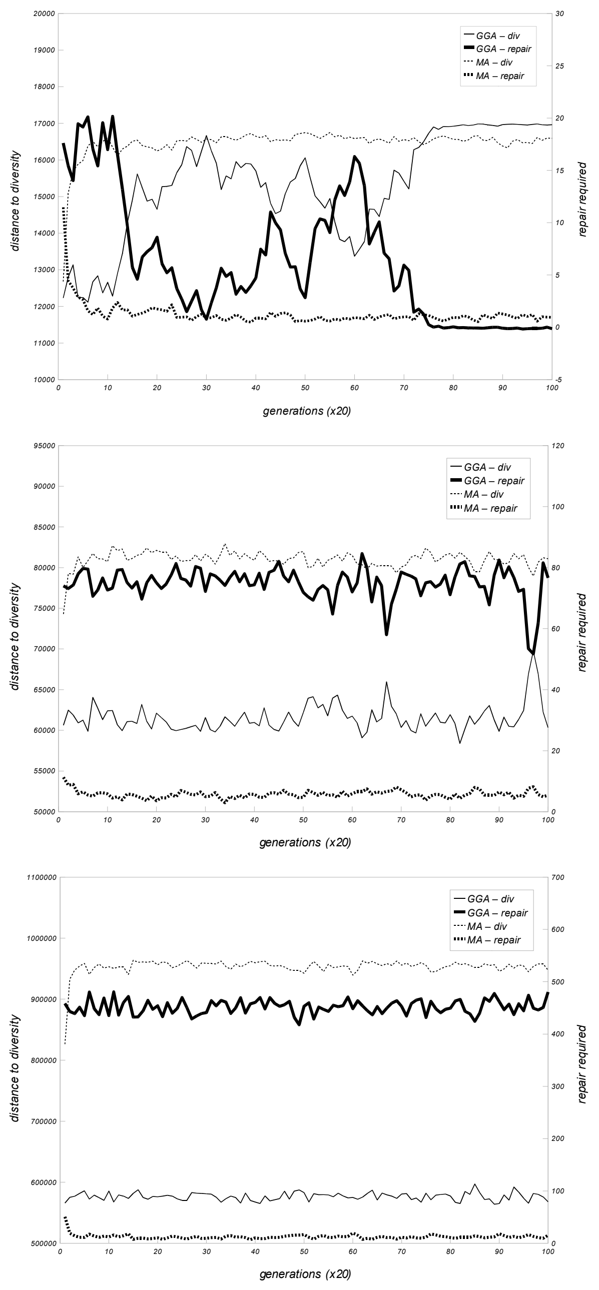

Figure 5: Comparing the performance of the recombination operators of the memetic and group-based algorithms. The results for an instance of small, medium, and big sets has been demonstrated in the top, middle, and bottom diagrams with $rr = 1.0$, $mr = 1$, $ls = 0.0$, and $p = 40$ across 2000 generations. The adopted configurations serve the purpose of abstracting away from other algorithmic aspects and focusing on recombination and not much else.

the small, medium and big instance of our recombination, and −0.99, −0.88, and −0.77 of the group-based recombination. The high correlation value for both operators is to be expected. As groups grow in size (and diversity), the cost of recombination tends to increase, since there is a higher chance that one time slot injected from another timetable shares events with more time slots of the host and this, in turn, escalates the cost. Thus, due to its high correlation with the destructive tendencies of recombination, maintaining a high degree of diversity on the part of the group-based operator works to its detriment by incurring a high level of repair.

Thus, we think the fact that the group-based recombination produces increasingly "disappointing" results as instances get larger, *partly* stems from its reductive commitment to recognizing groups (time slots) as the sole building blocks of the problem, especially in the case of the recombination operator that we studied here. While this might seem like an attractive commitment to make in theory, as demonstrated in Figure 5, it fails to live up to expectations in practice. It might be argued that while, as noted before, the notion 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 by precluding the possibility of two separate lineages evolving groups of events *within* a single time slot. As a third alternative, the level of granularity adopted by MA alternates between arbitrary sets of events (recombination), cliques which might be larger or smaller than the capacity of a time slot (clique-based local search), and entire time slots (mutation).

In the end, we might add that the difference between recombination costs for the two algorithms does not end here. Once the group-based algorithm starts rescheduling the recently displaced events, it uses expensive variable and value heuristics that serve to evaluate assignment choices prior to making them. In contrast, by choosing randomly and shifting much of the burden of evaluation to the fitness function, MA is able to cut costs by relying, for fitness evaluation, on either existing structures that figure in previous stages of the algorithm (i.e. the conflict matrix, the size and facility matrices etc.), or computations that are cost efficient (i.e. computing size mismatches, updating dynamic variables etc.). This results in an increased selection pressure, as well as decreased computation cost per generation. In sum, in the case of the group-based algorithm, not only are a larger proportion of the events subject to repair, but each repair is itself significantly more expensive.

6. CONCLUSION

The idea behind this paper was to investigate the applicability of evolutionary algorithms to a timetabling problem. Accordingly, we have attempted to develop a memetic algorithm to solve a predefined set of timetabling problems. In comparison with three other algorithms noted in the paper, our memetic algorithm performed better in case of small and medium instances, but fell behind in case of big instances. Following a general analysis of algorithmic performance, we focused on the difference between the obtained results of a group-based genetic algorithm and our own. To investigate this, the grouping genetic algorithm and a diversity measure were implemented. The analysis that followed cor-

roborated the hypothesis of [19] regarding the "pitfalls of grouping algorithms". In sum, our analysis attributed the better performance of the memetic algorithm (as opposed to the group-based genetic algorithm) to a number of factors including: implementing a multi-level selection as opposed to a single level group selection, propagating the experience of the algorithm between generations by way of a data structure that tries to identify "tough" events for each problem, using less costly heuristics, and developing a more sophisticated fitness function.
Despite its less destructive operators, the high convergence rate of the memetic algorithm did not fare well in the case of big instances, as noted in the text. Therefore, we might focus 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.

7. REFERENCES

[1] `http://www.idsia.ch/Files/ttcomp2002`, Jan. 2010.

[2] `http://www.cs.qub.ac.uk/itc2007/Login/SecretPage.php`, Nov. 2010.

[3] `http://www.emergentcomputing.org/timetabling/harderinstances`, June 2005.

[4] `ftp://ftp.mie.utoronto.ca/pub/carter/testprob`, Sept. 2010.

[5] `http://iridia.ulb.ac.be/supp/IridiaSupp2002-001/index.html`, Sept. 2010.

[6] `http://www.cs.qub.ac.uk/itc2007/winner/finalorder.htm`, Sept. 2010.

[7] `http://iridia.ulb.ac.be/~msampels/tt.data/`, Sept. 2010.

[8] `http://www.idsia.ch/Files/ttcomp2002/IC\_Problem/checksln.cpp`, Jan. 2010.

[9] C. Aladag, G. Hocaoglu, and M. Basaran. The effect of neighborhood structures on tabu search algorithm in solving course timetabling problem. *Expert Systems with Applications*, 36(10):12349–12356, 2009.

[10] A. Alkan and E. Ozcan. Memetic algorithms for timetabling. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 3, pages 1796–1802. IEEE, 2004.

[11] E. K. Burke, D. de Werra, and J. Kingston. Application to timetabling. In J. Gross and J. Yellen, editors, *Handbook of graph theory and applications*, pages 445–474. CRC press, 2003.

[12] E. K. Burke and J. Newall. A multi-stage evolutionary algorithm for the timetable problem. *IEEE Transactions on Evolutionary Compututation*, 3(1):63–74, 1999.

[13] T. Cooper and J. Kingston. The complexity of timetable construction problems. *Practice and Theory of Automated Timetabling (PATAT)*, pages 283–295, 1996.

[14] C. Cotta and A. Fernàndez. Memetic algorithms in planning, scheduling, and timetabling. *Evolutionary Scheduling*, pages 1–30, 2007.

[15] W. Erben. A grouping genetic algorithm for graph colouring and exam timetabling. *Practice and Theory of Automated Timetabling (PATAT) III*, pages 132–156, 2001.

[16] E. Falkenauer. *Genetic algorithms and grouping problems*. John Wiley & Sons, Inc. New York, NY, USA, 1998.

[17] Z. Lü and J. Hao. Adaptive tabu search for course timetabling. *European Journal of Operational Research*, 200(1):235–244, 2010.

[18] R. Lewis. A survey of metaheuristic-based techniques for university timetabling problems. *OR Spectrum*, 30(1):167–190, 2008.

[19] R. Lewis and B. Paechter. Finding feasible timetables using group-based operators. *IEEE Transactions on Evolutionary Computation*, 11(3):397–413, 2007.

[20] B. McCollum, A. Schaerf, B. Paechter, P. McMullan, R. Lewis, A. Parkes, L. Gaspero, R. Qu, and E. Burke. Setting the research agenda in automated timetabling: The second international timetabling competition. *INFORMS Journal on Computing*, 22(1):120–130, 2010.

[21] P. Moscato and C. Cotta. A gentle introduction to memetic algorithms. *Handbook of Metaheuristics*, pages 105–144, 2003.

[22] N. Pillay and W. Banzhaf. An informed genetic algorithm for the examination timetabling problem. *Applied Soft Computing*, 10(2):457–467, 2010.

[23] P. Pongcharoen, W. Promtet, P. Yenradee, and C. Hicks. Stochastic optimisation timetabling tool for university course scheduling. *International Journal of Production Economics*, 112(2):903–918, 2008.

[24] P. Ross, E. Hart, and D. Corne. Genetic algorithms and timetabling. In *Advances in evolutionary computing*, page 771. Springer-Verlag New York, Inc., 2003.

[25] 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. *Practice and Theory of Automated Timetabling (PATAT) IV*, 2740:329–351, 2003.

[26] K. Socha, J. Knowles, and M. Sampels. A max-min ant system for the university course timetabling problem. *Ant Algorithms*, pages 63–77, 2002.

[27] K. Socha, M. Sampels, and M. Manfrin. Ant algorithms for the university course timetabling problem with regard to the state-of-the-art. *Applications of Evolutionary Computing*, pages 334–345, 2003.

[28] M. Tuga, R. Berretta, and 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.

[29] R. Weare, E. K. Burke, and D. Elliman. A hybrid genetic algorithm for highly constrained timetabling problems. *Department of Computer Science*, 1995.

[30] D. Zhang, Y. Liu, R. M'Hallah, and S. Leung. A simulated annealing with a new neighborhood structure based algorithm for high school timetabling problems. *European Journal of Operational Research*, 203(3):550–558, 2010.