Alcom-FT Technical Report Series
ALCOMFT-TR-01-27

New Algorithms for Examination Timetabling

Massimiliano Caramia* Paolo Dell'Olmo †
Giuseppe F. Italiano‡

March 16, 2001

Abstract

In examination timetabling a given set of examinations must be assigned to as few time slots as possible so as to satisfy certain side constraints and so as to reduce penalties deriving from proximity constraints. In this paper, we present new algorithms for this problem and report the results of an extensive experimental study. All our algorithms are based on local search and are compared with other existing implementations in the literature.

1 Introduction

Examination timetabling is an important operations management problem in Universities worldwide and has been extensively investigated (see e.g., [1]–[9], [11]–[15]). Although educational systems are quite different in different countries, the task of scheduling examinations for University courses shares a common framework: the basic problem is to assign examinations to a certain number of time slots in such a way that there are no conflicts (e.g., no student has to take more than one examination at a time). The complexity of this problem depends on the degrees of freedom that students are allowed in their choice of courses: a greater freedom of choices typically increases the difficulty of producing an examination schedule that fits into a certain number of time slots without creating timetable conflicts for some students. There are also other issues, related to the quality of the solution, which must be taken into account: for instance, among

*Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Roma, Italy. Email: caramia@disp.uniroma2.it. Work partially supported by the Italian National Research Council (CNR).

†Dipartimento di Statistica, Università di Roma "La Sapienza", Roma, Italy. Email: dellolmo@pow2.sta.uniroma1.it. Work partially supported by the Italian National Research Council (CNR) under contract n. 98.00770.CT26.

‡Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Roma, Italy. Email: italiano@disp.uniroma2.it. Work partially supported by the IST Programme of the EU under contract n. IST-1999-14186 (ALCOM-FT), by the Italian Ministry of University and Scientific Research (Project "Algorithms for Large Data Sets: Science and Engineering"), and by the Italian National Research Council (CNR).

two different timetables having the same number of time slots, one might prefer the one having examinations spaced out more evenly or the one with lower number of undesirable assignments. For these reasons, timetabling problems make use of certain cost measures to assess the quality of potential schedules. In most cases there is an additional *proximity cost* w_t whenever a student has to take two examinations scheduled t time slots apart. The proximity cost is non-decreasing with the time slot difference, i.e., the closer two conflicting examinations are scheduled, the higher is the penalty induced by their proximity cost.

In general, the objective of examination timetabling is twofold. First of all, one wishes to find a conflict-free solution which schedules all exams in as few time slots as possible. Secondly, an important measure of the quality of a solution is the ability to spread student examinations as evenly as possible over the schedule, i.e., with the minimum cost. These two goals are clearly in contrast: the fewer the time slots, the less the examinations can be spread out evenly, and the more the penalty introduced by proximity costs. We mention that there are also other *side constraints* which might play a role into examination timetabling. For instance, there can be limitations on available classroom capacity or specific classroom availabilities. It is easy to see that examination timetabling problem, even in its basic versions, is $\mathcal{NP}$-complete as it is a generalization of graph coloring [10].

Related Work. Among the extensive literature on examination timetabling (see e.g., [1]–[9], [11]–[15]), we only cite the most recent implementations, which are by Carter *et al.* [7], by Burke *et al.* [2], and by Di Gaspero and Schaerf [9]. Carter *et al.* [7] employed known graph coloring techniques to tackle examination timetabling without side constraints. They solved an underlying graph coloring problem, where examinations were sorted in order of decreasing difficulty according to some criterion: examinations were assigned to time slots without creating conflicts, in the order they were processed. To reduce the length of the schedule obtained, some form of backtracking was used. Five different algorithms, according to the sorting criterion chosen (and referred to as LD, SD, LWD, LE and RO) were implemented. They also performed extensive experiments where only proximity constraints (and their associated penalties) are taken into account. The details of the method are spelled out in [7]. Burke *et al.* [2] designed a *memetic* timetabling algorithm based on evolutionary techniques, which deals exclusively with fixed length timetables. Their algorithm is based on the correct use of a number of evolutionary operators on a population chosen by a selection operator. The experiments in [2] considered both proximity constraints and classroom capacity constraints: i.e., a maximum number of available seats per time slot is given, and thus only a limited number of students can be examined in each time slot. Very recently Di Gaspero and Schaerf [9] proposed a family of algorithms based on several variants of tabu search.

Our Results. Most of the timetabling algorithms proposed in the literature are designed either to minimize the number of time slots (see e.g., [7]) or to minimize the overall penalty using a fixed number of time slots (see e.g., [2]). In this paper, we consider both issues and investigate in more depth the tradeoffs between time slots and penalties: we propose a family of timetabling algorithms based on local search which have quite different behavior with respect to time slots and penalties. We perform an extensive experimental study of our algorithms, and compare them to previous approaches. We try to reproduce the same experimental scenario set up by previous work: namely, we consider proximity constraints, room capacity constraints, and test our implementations on several real-world benchmarks for timetabling examination available in the public domain.

2 Our Timetabling Algorithms

Given a set of examinations E, *conflict graph* G can be defined as follows. There is a vertex in the conflict graph for each examination, and there is an edge between vertex i and vertex j if at least one student is enrolled in both exams i and j. In what follows, we will use interchangeably the term vertex and exam, as there is no danger of ambiguity. All our algorithms for examination timetabling perform local search on this conflict graph, driven by the following variables associated with each exam i.

- t_i : The time slot t_i tentatively assigned to exam i.
- p_i : The priority with which exam i is examined. At the beginning, p_i is initialized with the degree of vertex i in the conflict graph, i.e., exams with higher potential of conflicts are tentatively scheduled first.
- z_i : A penalty given by the sum of proximity costs scored by i in the current timetable. At the beginning, z_i is initialized with the sum of proximity costs given by the case where exam i is scheduled with the minimum distance from the other incompatible exams.

Our algorithm starts with a *greedy_scheduler*, which considers exams by their non-increasing priorities: exam i is allocated in the lowest available time slot it can be assigned to without generating conflicts. At the end, each penalty z_i is updated accordingly to the schedule found. Let $T = max_{i \in E}\{t_i\}$ be the total number of time slots used in the current timetable. After this step, a *penalty_decreaser* tries to improve the quality of the solution without increasing T. To accomplish this task, *penalty_decreaser* attempts to space out more evenly the exams while keeping the same number of time slots. In particular, exams are considered by non-increasing penalty: exam i is scheduled to a different time slot (not exceeding T) when this new assignment decreases the overall penalty of the timetable. If *penalty_decreaser* is not successful, i.e., it is not able to change the schedule, it calls a *penalty_trader*, which will be explained later on. Otherwise, if the initial penalty has been successfully decreased, each penalty z_i is updated according to the new schedule, and *penalty_decreaser* starts again its execution. When no further penalty reduction is obtained, the *greedy_scheduler* is restarted. We remark that, in this attempt to decrease the penalty of the timetable, *penalty_decreaser* could even find a timetable with fewer time slots. If this happens, T is updated accordingly.

When both penalties and number of time slots T can not be changed by the *greedy_scheduler* and the *penalty_decreaser*, we invoke the *penalty_trader*, which tries to trade off penalties for time slots. In other words, we check whether the penalty of the current schedule can be decreased by allocating one additional time slot (i.e., $T+1$). To do this, we have to choose which exams should be moved from their current time slot to $(T+1)$. We define Δ_i as follows: $\Delta_i = \left(\sum_{j \in E:(i,j) \in \mathcal{E}} |(T+1) - t_j|\right) - \left(\sum_{j \in E:(i,j) \in \mathcal{E}} |t_i - t_j|\right)$. The intuition behind Δ_i is the following. Note that $\left(\sum_{j \in E:(i,j) \in \mathcal{E}} |t_i - t_j|\right)$ gives the distance, measured in time slots, between exam i and its neighbors when i is assigned to time slot t_i, and $\left(\sum_{j \in E:(i,j) \in \mathcal{E}} |(T+1) - t_j|\right)$ gives the same quantity when i is assigned to time slot $T+1$. Thus, Δ_i measures the difference between those two distances: as the penalty is non-decreasing with the time slot difference, $\Delta_i > 0$ indicates that the overall penalty can be decreased by moving exam i from time slot t_i to time slot $T+1$. Similarly, $\Delta_i < 0$ indicates that the overall penalty is increased when exam i is moved from time slot t_i to time slot $T+1$. The *penalty_trader* neglects exams with $\Delta_i < 0$ and sorts the remaining exams by non-decreasing values of Δ_i. It then considers exams in this ordered list: it assigns exam i to time slot $T+1$ and deletes its neighbors from the list. This continues until the (stable) set formed by the examinations moved to time slot $(T+1)$ is maximal. Whenever there are no examinations i with $\Delta_i \geq 0$, the *penalty_trader* can not make any penalty improvement: in this case, it repeats the whole process with increasing values of T, say $T+2$, $T+3$, ... When the *penalty_trader* finds a larger T for which the penalty is decreased, the priorities are reassigned as $p_i = 1/t_i$ and the *greedy_scheduler* is invoked again.

We note that our approach is based on the generation of similar solutions: i.e., we try to perturb the current solution with a certain number of time slots, in the attempt of spacing out the examinations more evenly. This has also some drawbacks, however, as the algorithm can get stuck for a long time in particular areas of the solution space. To avoid this, we use a simple but effective *checkpointing* scheme: the algorithm stops at certain steps, releases basically all of its memory, and starts a new local search after this. We now analyze checkpointing in detail. Define an *iteration* as an attempt (not necessarily successful) to change the current state of an examination i (i.e., change either its time slot t_i or its penalty z_i). Roughly speaking, we force a checkpoint after a certain number of iterations: after each checkpoint, our algorithm restarts again a new local search from the current timetable, with a consequent release of memory used. Releasing memory once in a while helps in decreasing the space consumption. This has a positive effect on the running times as well, as algorithms with high space usage tend to be very slow in practice. Moreover, we noticed in our experiments that a properly tuned checkpoint is beneficial on large instances: indeed, it prevents that the portion of the conflict graph examined by local search becomes too large, and forces the algorithm to work on smaller portions of the graph. We implemented two different types of checkpointing: *constant* and *adaptive*. The first is the simpler: the interval at which checkpointing is performed is constant, i.e., checkpointing is performed exactly every ℓ iterations, with ℓ being a properly chosen constant. The second is a bit more complicated, as the checkpointing interval depends on the number of iterations performed, i.e., it grows as the algorithm performs more iterations. Namely, let ℓ_0 be the initial checkpointing interval, and $\alpha > 1$ be a constant. Let $\ell_i = \alpha\ell_{i-1}$, $i \geq 1$. For the first ℓ_1 iterations (i.e., in the range $(0, \ell_1]$), the algorithm performs checkpointing at interval ℓ_0. For the subsequent $(\ell_2 - \ell_1)$ iterations (i.e., in the range $(\ell_1, \ell_2]$), the algorithm performs checkpointing at interval $\ell_1 > \ell_0$. In general, for iterations in the range $(\ell_i, \ell_{i+1}]$, the algorithm performs checkpointing at interval $\ell_i > \ell_{i-1}$. We will analyze in details the differences between these two checkpointing schemes in Section 3.

Unfortunately, the checkpointing scheme can introduce also some drawbacks. Namely, when we release all the memory used due to checkpointing, we are also risking to loose some important information, such as solutions previously explored, or data which could have been helpful in avoiding getting trapped in a local minimum. To solve this problem, we introduce a careful assignment of priorities after a checkpointing, which we call *bridging priorities*. We do this follows. Define a *phase* as the interval between any two successive checkpoints. As the algorithm retains no information of what happened before a checkpoint, it could explore solutions which are exactly the same or very similar to solutions that were generated in previous phases, without being able to notice it. To circumvent this problem the algorithm relies on an appropriate reassignments of priorities to the examinations after each checkpointing: this dynamic reassignment of priorities tries to bridge subsequent phases by bypassing the checkpoints.

The integration of these two features, i.e., checkpointing and bridging priorities, allows one to start successive local searches which are closely interacting, without increasing the space consumption. We now define how priorities are reassigned. Define the *update count* of a vertex v as the number of times v changed its state in the current phase (i.e., since the last checkpoint). Roughly speaking, the update count of an examination measures its active participation during that phase. After a checkpoint and before the update counts are reset, the examination priorities are set as $p_i = 1/update\_count$. With this assignment, a vertex with a low update count (i.e., which was not very active in the last phase) gets a high priority, and thus there is a better chance to get this vertex involved in the next phases. This clearly prevents the algorithm to continue its visit on a same subset of vertices. In our experiments, this particular choice of priorities generated a sequence of phases which were interacting closely with each other: in most cases a phase was building on the priorities and solutions offered by the previous phase, and consistently offered better priorities and solutions to the next phase. We implemented and tested four variants of our algorithm, depending on the type of checkpointing used, and on whether bridging priorities were used or not:

- `ACP`: uses Adaptive Checkpointing and bridging Priorities;

- `AC`: uses Adaptive Checkpointing but not bridging priorities;
- `CCP`: uses Constant Checkpointing and bridging Priorities;
- `CC`: uses Constant Checkpointing but no bridging priorities.

As a result of our engineering efforts, we decided to combine `ACP` and `CC` into a hybrid algorithm, called `ACCP`. `ACCP` works in two steps. In the first step, it tries to obtain a conflict-free solution with as few time slots as possible. This is accomplished by running `ACP` on the input instance. Let T be the number of time slots obtained. Next, `ACCP` tries to reduce the penalty of the solution while keeping at T the number of time slots. This is done by running a modified version of `CC` on the initial input instance. In this version of `CC`, we did not use either the *greedy_scheduler* or the *penalty_trader*, as there was no need to change (i.e., either decrease or increase) the number of time slots in the current solution. Instead of these two functionalities, we used a simpler *slot_swapper*, which in some sense accomplished the same task as the *penalty_trader*, without moving to a higher number of time slots: the *slot_swapper* was executed whenever either the *greedy_scheduler* or the *penalty_trader* were called before. More in details, the *slot_swapper* swaps examinations in time slot t_i with examinations in time slots t_j, for $t_i \neq t_j$, as follows. Define the quantity Δ_{ij} for each $1 \leq i, j \leq T$, $i \neq j$, as $\Delta_{ij} = \sum_{h \in E: t_h = i} \sum_{k \in E:(h,k) \in \mathcal{E}} |t_j - t_k| + \sum_{h \in E: t_h = j} \sum_{k \in E:(h,k) \in \mathcal{E}} |t_i - t_k| - \sum_{h \in E: t_h = i} \sum_{k \in E:(h,k) \in \mathcal{E}} |t_i - t_k| - \sum_{h \in E: t_h = j} \sum_{k \in E:(h,k) \in \mathcal{E}} |t_j - t_k|$. Intuitively, Δ_{ij} relates to the difference in solutions given by swapping exams in time slot t_i with exams in time slot t_j. We take i and j maximizing Δ_{ij} and swap all exams currently assigned to time slot t_i with all the exams currently assigned t_j. If $max_{i,j}\{\Delta_{ij}\} < 0$ the algorithm terminates. After the *slot_swapper* has run, the *penalty_decreaser* is invoked and the algorithm keeps on iterating between these two functionalities until the stopping criterion is met.

As it was already mentioned in the introduction, in many timetabling applications one is concerned with a fixed number of time slots, and thus wishes to space out as evenly as possible (according to the proximity constraints) a given number of exams within these time slots. This can be easily done by feeding `ACCP` with the desired number of time slots. In this special case, the underlying call to `ACP` accomplishes a simpler task. Indeed, rather than spending a lot of efforts in generating a solution with the lowest possible number of time slots, it can just provide the first solution which meets the number of time slots given in input.

3 Experimental results

Data Sets. Benchmarks for examination timetabling are publicly available at `ftp.cs.nott.ac.uk/ttp/Data` and at `ie.utoronto.ca/pub/carter/testprob`. The data sets consist of real-life examples taken from several Canadian Institutions, from the London School of Economics, from King Fahd University of Petroleum, Minerals, Dhahran, from Purdue University, and from the Nottingham University. Each data set is identified by a code, where the first three letters refer to the Institution, F or S stand for the semester (Fall or Spring), and the last two digits to the year. For instance, PUR-S-93 refers to the examinations scheduled in the Spring '93 at Purdue University. In these benchmarks, the number of students ranges from 611 (STA-F-83) to 30,032 (PUR-S-93), and the number of examinations to be scheduled ranges from 81 (HEC-S-92) to 2,419 (PUR-S-93).

Experimental Setup. Our algorithms have been coded in C and the experiments run on a PC with a 500MHz Pentium and 64MB of RAM. For lack of space we report here only the results of two sets of experiments. The first considers only proximity constraints (and their associated penalties) but no side constraints. The results are illustrated in Tables 1, 2 and 3. The second set of experiments considers also room capacity constraints: i.e., a maximum number of available seats per time slot is given, and thus only a limited number of students can be examined in each time slot. Results of this test are reported in Table 4. In each experiment, we collected data about the number of time slots found by a given algorithm, its associated penalty (normalized on the total number of students) and its running times in seconds. The normalized penalty allows one to compare results on instances of different size.

We compared our algorithms with the five algorithms in [7], denoted by `Carter`, the algorithms in [2], denoted by `Burke` and the algorithms in [9], denoted by `Di Gaspero`. We had no access to the codes of Burke *et al.*, Carter *et al.* and Di Gaspero and Schaerf, and thus the data related to `Burke`, `Carter` and `Di Gaspero` are taken verbatim from their papers [2, 7, 9]. This implies that running times are not necessary meaningful, as they relate to different platforms. In order to ensure a fair comparison, we tried to reproduce for our algorithms exactly the same experimental setup reported in [2, 7, 9]: in particular, like the other implementations, we stopped the tests for our algorithms whenever they were not able to improve the solutions found after a certain number of iterations. We remark that other stopping criteria can be easily implemented in our code.

Experimental Analysis. Table 1 lists the results of the same experiments reported on Table 3 of [7]. We start by analyzing the relative behavior of `ACP`, `AC`, `CCP` and `CC` in this set of experiments. Next, we comment on `ACCP`. Finally, we summarize the performance of our algorithms with respect to the algorithms of Carter *et al.* [7].

The first feature that can be observed from columns `CC`, `CCP`, `AC` and `ACP` of Table 1 is that algorithms that produce better solutions (i.e., solutions with a smaller number of time slots) have consistently higher penalties. This seems natural, as solutions with a higher number of time slots have more room to space out examinations evenly. Secondly, it can be easily observed that adaptive checkpointing outperforms consistently constant checkpointing, i.e., the solutions produced by `ACP` (respectively `AC`) have fewer number of time slots than the solutions produced by `CCP` (respectively `CC`). We explain this phenomenon as follows. In the initial stages, adaptive checkpointing tends to create a larger

Data_set		CC	CCP	AC	ACP	ACCP	Carter LD	Carter SD	Carter LWD	Carter LE	Carter RO
CAR-F-92	Slots	32	30	29	**28**	**28**	31	**28**	30	31	32
	Penalty	**6.0**	6.4	6.8	7.0	6.4	–	–	–	–	–
	Time	428.9	398.2	475.9	486.7	559.2	618.3	510.8	911.6	121.2	1143.3
CAR-S-91	Slots	32	32	30	**28**	**28**	32	**28**	30	32	35
	Penalty	**6.8**	7.0	7.2	7.8	7.3	–	–	–	–	–
	Time	82.9	58.4	60.8	70.7	86.3	293.8	75.1	1046.4	210.4	712.7
EAR-F-83	Slots	23	23	23	**22**	**22**	23	24	23	23	24
	Penalty	**40.2**	40.8	42.0	46.8	43.2	–	–	–	–	–
	Time	140.8	104.9	99.6	111.8	150.1	80.2	173.0	156.6	151.8	256.5
HEC-S-92	Slots	18	**17**	**17**	**17**	**17**	**17**	**17**	18	**17**	**17**
	Penalty	**9.2**	10.8	10.8	11.2	10.6	–	–	–	–	–
	Time	24.2	17.8	12.8	10.2	19.9	31.8	17.1	22.8	219.5	83.1
KFU-S-93	Slots	20	**19**	**19**	**19**	**19**	20	**19**	**19**	**19**	**19**
	Penalty	**13.8**	14.2	14.2	15.9	14.0	–	–	–	–	–
	Time	128.2	110.6	110.4	85.9	159.6	120.1	97.2	2396.5	1852.7	2217.0
LSE-F-91	Slots	18	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**	**17**
	Penalty	**9.6**	9.8	10.8	13.4	**9.6**	–	–	–	–	–
	Time	195.8	178.1	145.7	122.8	223.4	78.7	164.6	763.9	786.2	674.5
PUR-S-93	Slots	42	40	38	**36**	**36**	43	38	40	40	42
	Penalty	**3.7**	3.9	4.1	4.8	4.4	–	–	–	–	–
	Time	5874.8	4289.4	4256.0	4564.0	6021.2	31922.2	73338.3	8782.3	9304.9	72028.5
RYE-F-93	Slots	23	22	**21**	**21**	**21**	23	**21**	22	22	22
	Penalty	**6.8**	7.4	7.8	8.0	7.7	–	–	–	–	–
	Time	125.7	102.2	98.4	142.5	225.9	507.2	380.0	419.4	502.5	3467.4
STA-F-83	Slots	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**	**13**
	Penalty	158.2	162.0	168.4	173.8	**150.2**	–	–	–	–	–
	Time	9.2	5.7	5.5	4.0	10.2	6.0	6.0	5.7	6.1	98.9
TRE-S-92	Slots	23	22	21	**20**	**20**	22	21	21	23	22
	Penalty	**9.4**	9.8	10.2	10.8	10.3	–	–	–	–	–
	Time	115.7	95.2	94.5	100.1	214.7	398.8	108.4	1067.9	107.4	836.3
UTA-S-93	Slots	34	32	31	**30**	**30**	33	35	35	34	35
	Penalty	**3.5**	4.2	4.5	5.0	4.4	–	–	–	–	–
	Time	760.5	698.2	685.9	758.2	1023.5	4294.1	2557.2	9223.0	7340.4	17323.1
UTE-S-92	Slots	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
	Penalty	28.4	32.2	38.5	41.2	**24.3**	–	–	–	–	–
	Time	7.2	6.8	5.8	4.2	9.8	6.3	4.7	8.5	9.1	9.9
YOR-F-83	Slots	21	20	20	**19**	**19**	21	**19**	20	20	20
	Penalty	**36.2**	42.2	44.0	45.9	44.2	–	–	–	–	–
	Time	138.7	134.9	120.2	148.2	226.2	649.4	190.4	67.1	516.4	351.5

Table 1: Comparison with the results of Carter *et al.* [7].

number of phases than constant checkpointing, which could be helpful in exploring the solution space. Indeed, we noticed from a profiling that adaptive checkpointing issued a larger number of calls to the *greedy_scheduler* in the initial phases. This made adaptive checkpointing more prone to finding a good solution (with a small number of time slots) before constant checkpointing did. As the algorithm progresses, adaptive checkpointing makes the number of phases decrease significantly (with respect to constant checkpointing): this implies that, as time goes by, adaptive checkpointing works more with the *penalty_decreaser* and the *penalty_trader* rather than with the *greedy_scheduler*, and thus decreases the penalty rather than the number of time slots. The same effect can be noticed for bridging priorities, as the solutions produced by `ACP` (respectively `CCP`) have consistently fewer number of time slots than the solutions produced by `AC` (respectively `CC`). It thus seems that tunneling information through subsequent phases can improve substantially the performance of the *greedy_scheduler*. In summary, among the four implementations, ACP seems to give always the best solution in terms of number of time slots, while CC produces solutions with the highest number of time slots but lowest penalties. The performance of AC and CCP is within the boundaries defined by ACP and CC, with AC producing consistently fewer time slots and higher penalties than CCP. One could thus use ACP if a solution with minimum schedule length is desired or alternatively use CC if time slots are not an issue and a timetabling with low penalty is preferred. Schedules obtained with AC and CCP represent other tradeoffs between these two objectives, with emphasis on time slots for AC and on penalties for CCP. We now comment on the solutions found by ACCP. As expected, the number of time slots found by ACCP coincide with the time slots of ACP, with ACCP having higher running times than ACP (as ACCP runs ACP as a subroutine). As shown in Table 1, the extra work done by ACCP seems extremely useful in reducing the solution penalties. It was somehow surprising for us to notice that often the penalties of ACCP were lower than those reported by AC or CCP (even when AC and CCP produced a higher number of time slots). Furthermore, in the cases where the solutions found by ACP, AC, CCP and CC had the same number of time slots (such as STA-F-83 and UTE-S-92), ACCP was even able to improve on the penalty achieved by CC. One could wonder whether the stopping criterion adopted in this experiment (i.e., stopping whenever a given algorithm is able to improve the solutions found after a certain number of iterations) might give an unfair advantage to CC and ACCP, which tend to have higher running times than CCP, AC and ACP. This does not seem to be the case. Indeed, we performed another experiment on the same data sets, in which we ran each algorithm for exactly the same (CPU) time. As it can be noted from Table 2, which reports the results of this experiment, time slots and penalties are very similar to those obtained in the Table 1, except only for slight improvements on penalties. Another feature that comes out from Tables 1 and 2 is that in all tested benchmark instances ACCP consistently achieves the lowest number of time slots, and that its penalty is the lowest among all the algorithms achieving exactly this lowest number of time slots.

We now go back to Table 1 and compare our algorithms to the algorithms of [7]. As it can be easily seen, all our algorithms, and in particular ACP and ACCP, always find consistently solutions with fewer time slots than the algorithms of Carter *et al.* Note that Carter *et al.* did not report the penalties of the solutions found by their algorithms in this experiment. To give an idea about this, they set up another experiment: for each benchmark instance they selected a fixed number of time slots (taken as the maximum number of time slots found by their algorithms in the previous experiment), and reported the penalties found by their algorithms with this fixed number of time slots. The results of this test are summarized in column Carter of Table 3. The same experiment was performed by Di Gaspero and Schaerf as well, at least for a subset of benchmark instances: the results are reported on the last column of Table 3 (averaged on several executions of their algorithms). In order to make a fair comparison, we organized for ACCP the same experimental setup, and the results are illustrated in

Data_set		CC	CCP	AC	ACP	ACCP
CAR-F-92	Slots	32	30	29	**28**	**28**
	Penalty	**6.0**	6.2	6.8	7.0	6.4
	Time	1120	1120	1120	1120	1120
CAR-S-91	Slots	32	31	30	**28**	**28**
	Penalty	**6.8**	7.0	7.2	7.6	7.2
	Time	172	172	172	172	172
EAR-F-83	Slots	23	23	23	**22**	**22**
	Penalty	**40.0**	40.4	41.5	46.4	43.0
	Time	300	300	300	300	300
HEC-S-92	Slots	18	**17**	**17**	**17**	**17**
	Penalty	**9.2**	10.6	10.7	11.0	10.5
	Time	48	48	48	48	48
KFU-S-93	Slots	20	**19**	**19**	**19**	**19**
	Penalty	**13.6**	14.1	14.2	15.3	14.0
	Time	320	320	320	320	320
LSE-F-91	Slots	18	**17**	**17**	**17**	**17**
	Penalty	**9.6**	9.8	10.5	12.9	**9.6**
	Time	446	446	446	446	446
PUR-S-93	Slots	42	40	38	**36**	**36**
	Penalty	**3.6**	3.8	4.0	4.7	4.4
	Time	12042	12042	12042	12042	12042
RYE-F-93	Slots	23	22	**21**	**21**	**21**
	Penalty	**6.8**	7.3	7.7	8.0	7.7
	Time	452	452	452	452	452
STA-F-83	Slots	**13**	**13**	**13**	**13**	**13**
	Penalty	158.2	162.0	168.2	173.4	**150.2**
	Time	30	30	30	30	30
TRE-S-92	Slots	23	22	21	**20**	**20**
	Penalty	**9.4**	9.7	10.0	10.8	10.3
	Time	430	430	430	430	430
UTA-S-93	Slots	34	32	31	**30**	**30**
	Penalty	**3.5**	4.1	4.4	5.0	4.4
	Time	2046	2046	2046	2046	2046
UTE-S-92	Slots	**10**	**10**	**10**	**10**	**10**
	Penalty	28.4	32.2	36.6	40.0	**24.3**
	Time	20	20	20	20	20
YOR-F-83	Slots	21	20	20	**19**	**19**
	Penalty	**36.0**	42.0	43.8	45.6	44.0
	Time	452	452	452	452	452

Table 2: ACP, AC, CCP, CC, and ACCP executed with a fixed CPU time.

column ACCP. As it can be easily seen, the penalties obtained by ACCP are always substantially lower than the range of penalties achieved either by the algorithms of Carter *et al.* or by the algorithms of Di Gaspero and Schaerf. Moreover, our algorithms look more robust and less sensitive to the problem instance. Namely, CC, CCP, AC, ACP and ACCP seem to maintain the same relative behavior on all input instances: e.g., ACCP and ACP always find the solutions with fewer time slots, and CC the solutions with smaller penalties. The algorithms of Carter *et al.* and by Di Gaspero and Schaerf, on the contrary, seem quite sensible to the problem instance, and algorithms of the same family show quite different behaviors on different input instances. This might be of practical interest, as having to select one among several programs based on the input instance, or having to tune the program parameters as a function of the input data might reveal to be either too cumbersome or impractical.

Finally, Table 4 contains some results on experiments with room capacity constraints on a fixed number of time slots. We report the data obtained from the same experiment set up by Burke *et al.* [2] and reproduced by Di Gaspero and Schaerf: column `ACCP` contains information about the solutions achieved by `ACCP` and columns `Burke` and `Di Gaspero` contain the data reported in [2] and in [9], respectively. To keep consistency with [2] and [9], differently from Tables 1–3, in Table 4 we report the overall penalties found by the algorithms (i.e., penalties are not normalized over the total number of students). The table shows that `ACCP` yields a substantial improvement over the solution quality of `Burke`, and is superior to `Di Gaspero` in many cases.

Acknowledgments

We are indebted to the anonymous referees for their comments on a previous version of this paper. We are also grateful to Andrea Schaerf for sending us a copy of [9].

References

[1] Burke, E., and Newall, J, "A Multi-Stage Evolutionary Algorithm for the Timetabling Problem", *IEEE Transactions on Evolutionary Computation*, 3 (1) (1999) 63–74.

[2] Burke, E., Newall, J, and Weare, R., "A Memetic Algorithm for University Exam Timetabling", in *Proc. of the 1st Int. Conf. on the Practice and Theory of Automated Timetabling*, (1995) 241–250.

[3] Corne, D., Fang, H.-L., and Mellish, C, "Solving the Modular Exam Scheduling Problem with Genetic Algorithms", Tech. rep. 622, Department of Artificial Intelligence, University of Edinburgh, (1993).

[4] Corne, D., Ross, P., and Fang, H.-L., "Evolutionary Timetabling: Practice Prospects and Work in Progress", in *UK Planning and Scheduling SIG Workshop*, (1994).

[5] Carter, M.W., "A Survey of Practical Applications of Examination Timetabling Algorithms", *Operations Research*, 34 (2) (1986) 193–202.

[6] Carter, M.W., Laporte, G., and Chinnek, J.W., "A General Examination Scheduling", *Interfaces*, 24 (3) (1994) 109–120.

[7] Carter, M.W., Laporte, G., and Lee, S.Y., "Examination Timetabling: Algorithm Strategies and Applications", *Journal of the Operational Research Society*, 47 (1996) 373–383.

[8] De Werra, D., "An Introduction to Timetabling", *European Journal of Operational Research*, 19 (1985) 151–162.

[9] Di Gaspero, L., and Schaerf, A., "Tabu Search Techniques for Examination Timetabling Problems", in *Proc. of the 6th Int. Conf. on the Practice and Theory of Automated Timetabling*, (2000).

[10] Garey, M.R., and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-completeness*, W.H. Freeman & Company, San Francisco, (1979).

[11] Johnson, D., "Timetabling University Examinations", *Journal of the Operational Research Society*, 41 (1990) 39–47.

[12] Laporte, G., and Desroches, S., "Examination Timetabling by Computer", *Computers and Operations Research*, 11 (1984) 351–360.

[13] Metha, N.K., "The Application of a Graph Colouring Method to an Examination Scheduling Problem", *Interfaces*, 11 (5) (1981) 57–64.

[14] Thompson, J.M., and Dowsland, K.A., "A Robust Simulated Annealing Based Examination Timetabling System ", *Computers and Operations Research*, 25 (1998) 637–648.

[15] Schaerf, A., "A Survey of Automated Timetabling", *Artificial Intelligence Review*, 13 (2) (1999) 87–127.

Data_set		ACCP	Carter LD	Carter SD	Carter LWD	Carter LE	Carter RO	Di Gaspero
CAR-F-92	Slots	32	32	32	32	32	32	–
	Penalty	**6.0**	7.6	6.6	6.6	6.2	8.2	–
	Time	142.7	637.9	15.0	886.0	47.0	1143.4	–
CAR-S-91	Slots	35	35	35	35	35	35	–
	Penalty	**6.6**	7.9	7.1	7.4	7.6	11.9	–
	Time	34.7	29.2	20.7	67.5	58.3	712.7	–
EAR-F-83	Slots	24	24	24	24	24	24	24
	Penalty	**29.3**	36.4	46.5	37.3	42.3	42.1	49.4
	Time	100.2	24.7	6.0	76.1	242.4	256.5	40.8
HEC-S-92	Slots	18	18	18	18	18	18	18
	Penalty	**9.2**	10.8	12.7	15.8	15.9	20.0	15.4
	Time	11.0	7.4	6.5	22.8	8.7	54.9	13.4
KFU-S-93	Slots	20	20	20	20	20	20	20
	Penalty	**13.8**	14.0	15.9	22.1	20.8	21.0	19.7
	Time	112.8	120.2	559.4	147.3	208.9	977.6	57.2
LSE-F-91	Slots	18	18	18	18	18	18	18
	Penalty	**9.6**	12.0	12.9	13.1	10.5	13.3	16.2
	Time	92.8	50.3	50.5	233.5	48.0	674.5	29.0
PUR-S-93	Slots	42	42	42	42	42	42	–
	Penalty	**3.7**	4.4	4.1	5.0	3.9	–	–
	Time	4018.3	31922.1	22921.9	10601.1	21729.4	>200000	–
RYE-F-93	Slots	23	23	23	23	23	23	–
	Penalty	**6.8**	7.3	7.4	10.0	7.7	11.0	–
	Time	89.4	507.2	403.0	360.0	391.9	3019.2	–
STA-F-83	Slots	13	13	13	13	13	13	13
	Penalty	**158.2**	162.9	165.7	161.5	161.5	184.1	159.8
	Time	6.5	6.0	6.0	5.7	6.1	98.9	10.1
TRE-S-92	Slots	23	23	23	23	23	23	23
	Penalty	**9.4**	11.0	10.4	9.9	9.6	13.0	10.8
	Time	102.8	150.6	89.3	74.1	107.4	468.4	137.4
UTA-S-93	Slots	35	35	35	35	35	35	35
	Penalty	**3.5**	4.5	3.5	5.3	4.3	6.4	5.0
	Time	589.4	1287.8	664.3	5929.7	2705.1	7135.6	1351.0
UTE-S-92	Slots	10	10	10	10	10	10	10
	Penalty	**24.4**	38.3	31.5	26.7	25.8	31.6	30.6
	Time	5.0	6.3	4.7	8.5	9.1	9.9	6.9
YOR-F-83	Slots	21	21	21	21	21	21	21
	Penalty	**36.2**	49.9	44.8	41.7	45.1	46.5	46.2
	Time	125.4	215.7	428.0	271.4	174.5	231.8	32.0

Table 3: Comparison with the results of Carter *et al.* [7] and of Di Gaspero and Schaerf [9].

Data_set		ACCP	Burke	Di Gaspero
CAR-F-92	Slots	40	40	40
	Penalty	**268**	331	443
	Time	80.4	–	–
CAR-S-91	Slots	51	51	51
	Penalty	**74**	81	98
	Time	31.4	–	–
KFU-S-93	Slots	20	20	20
	Penalty	912	974	**597**
	Time	118.2	–	–
TRE-S-92	Slots	35	35	35
	Penalty	**2**	3	5
	Time	222.4	–	–
NOTT	Slots	26	26	26
	Penalty	44	53	**13**
	Time	359.1	–	–
UTA-S-93	Slots	38	38	38
	Penalty	680	772	**625**
	Time	265.1	–	–

Table 4: Comparison with the results of Burke *et al.* [2] and of Di Gaspero and Schaerf [9].