The University Course Timetabling Problem with a 3-phase approach

Philipp Kostuch

Oxford University, Department of Statistics
1 South Parks Rd., Oxford, OX1 3TG, UK
kostuch@stats.ox.ac.uk
http://www.stats.ox.ac.uk

Abstract. This paper describes the University Course Timetabling Problem (UCTP) used in the International Timetabling Competition 2003 organized by the Metaheuristics Network and presents a state-of-the-art heuristic approach towards the solution of the competition instances. It is a greatly improved version of the winning competition entry. The heuristic is divided into three phases: at first, a feasible timetable is constructed, then Simulated Annealing (SA) is used to order the thus created time-slots optimally, and finally SA is used to swap individual events between time-slots to improve the solution quality.

1 Introduction

In 2002/03 the Metaheuristics Network organized an International Competition on a problem called University Course Timetabling Problem (UCTP) [1]. The competition sought to promote research in the field of timetabling and to establish a widely accepted benchmark problem for the timetabling community. The UCTP model is a reduction of a real-world timetabling problem encountered at Napier University, Edinburgh, UK. The (artificial) instances used for the competition were produced with an instance generator written by Paechter[1].

1.1 UCTP

The UCTP consists of a set of n events $E = \{e_1, \ldots, e_n\}$ to be scheduled in a set of 45 timeslots $T = \{t_1, \ldots, t_{45}\}$ (5 days in a week, 9 time-slots a day), and a set of j rooms $R = \{r_1, \ldots, r_j\}$ in which events can take place. Additionally, there is a set of students S who attend the events, and a set of features F provided by rooms and required by events. Each student attends a subset of events. A feasible timetable is one in which all events have been assigned a time-slot and a room, so that the following hard constraints are satisfied:

– no student attends more than one event at the same time;

[1] http://www.dcs.napier.ac.uk/~benp

– the assigned room is big enough for all the attending students and posses all the features required by the event;
– only one event is taking place in each room at a given time.

In addition, a feasible candidate timetable is penalized equally for each occurrence of the following soft constraint violations:

– a student has a class in the last slot of the day;
– a student has more than two classes in a row on a given day (one penalty for each class above the first two);
– a student has exactly one class during a day.

Infeasible timetables are worthless and are considered equally bad regardless of the actual level of in-feasibility. The objective is to minimize the number of soft constraint violations in a feasible timetable.

1.2 The Competition

The International Timetabling Competition began on October 1^{st} 2002 with the publication of 10 instances of the UCTP. On March 17^{th} 2003 another 10 instances were released and solutions to all 20 instances had to be submitted by the end of March. Only entries with feasible timetables for all problems were considered. It is known that all instances have a feasible solution without soft constraint violated.

For an entry to be accepted, the algorithm solving the UCTP had to be the same for all instances and results had to be obtained within a given time limit (to be determined for every machine by running a provided benchmark program). Nonetheless, it was allowed to submit different random seeds for different instances, thus enabling the selection of the best run of the algorithm over a range of different seeds.

The results of the competition, descriptions of various entries and the exact rules for determining the winner can be found on the competition web page [2]. The official winner of the competition is an early version of the algorithm described in this paper. At that time, its results ranked second behind a cooperative effort of members of the Metaheuristic Network, which was not allowed to enter the competition.

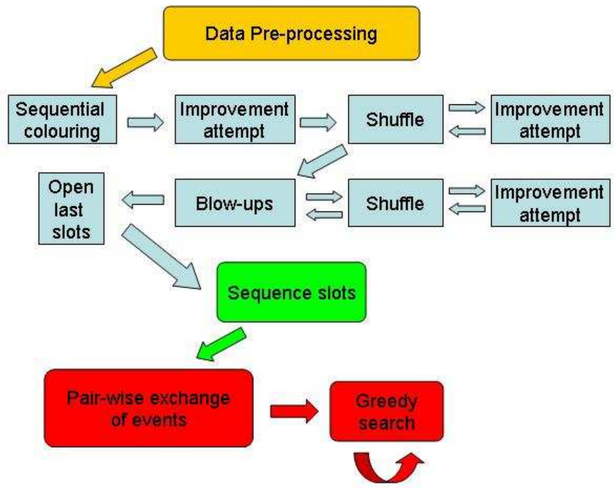
The remaining part of the paper is organized as follows. In Section 2, the algorithm used to solve the UCTP is described in detail. Section 3 presents the algorithm's performance on the benchmark instances and an analysis of some aspects of the algorithm. Finally, the findings are summarized in Section 4.

2 The Algorithm

In this section, the algorithm used to solve the UCTP will be described in detail. At first, some general comments about the internal data structures will be made.

Then, the three phases will be discussed in detail. Phase I of the algorithm tries to construct a feasible initial timetable within 40 time-slots, leaving the 5 penalized end-of-day slots empty. In Phase II, the time-slots created before are sequenced using Simulated Annealing (SA) in order to minimize the objective function value. Once this is done, Phase III swaps individual events between time-slots, always maintaining feasibility. This is first done by a standard SA approach. Then–to obtain the final solution–the same neighbourhood structure is used at lower temperatures with periodical restarts at the best solution found so far. A flowchart representation of the algorithm is given in Figure 1.

Fig. 1. Flow chart of the 3-phase algorithm for UCTP.



2.1 Pre-processing and Data structures

As there is a time limit for finding the solutions, besides a powerful algorithmic concept, efficient data structures and internal representations of the problem are needed to obtain competitive results. While some design choices presented in the following will be helpful for many algorithms, others are tailored more closely to the needs of this particular approach and will therefore not necessarily provide benefits in other settings.

The first step is to reduce the whole information contained in the instance file to a compact format. Besides the generic informations provided (number of students, events and rooms) everything else can be represented in two matrices, the *incidence* matrix and the *event-room* matrix. The incidence matrix describes the hard constraints stemming from student clashes: we imagine a graph where events are nodes and edges indicate that two events share at least one student and can therefore not be scheduled together. The matrix of dimension ($events \times events$) representing this graph is the incidence matrix. The event-room matrix has been compiled from the information about which features are needed by events, which are provided by rooms and whether the room size can accommodate a given event. This leads to a matrix of dimension ($events \times rooms$), indicating which rooms are feasible for a given event.

Before starting to solve the problem, one useful manipulation of the incidence matrix is implemented. As we will accept only feasible moves in Phase III, every move has to be checked for feasibility before evaluating the objective function. There are two possible sources of infeasibility: either a student clash occurs or it is not possible to find suitable rooms for all events scheduled in a given time-slot. The first is computationally easy to detect by looking at the incidence matrix, while the second one needs a maximum matching algorithm to be run on a subset of rows from the event-room matrix. So, it is computationally attractive to shift information to the incidence matrix, if possible.

We observe, that there are a substantial number events with only one room option. Two such events having the same room as their only option can obviously not be scheduled together. Therefore, we search for the subsets of events with the same single room option and introduce edges into the incidence matrix so that these subsets form cliques in the graph, i.e. all possible edges between these events are present. The beauty of this step is that–while not altering the trajectory of the algorithm itself–it speeds up the detection of infeasibilities considerably.

A second manipulation suggested in [3], goes a step further: based on the observation, that some of these subsets of events sharing the same room as their single option have size 40 and the knowledge that a perfect timetable exists, further alterations are implemented. Given such a subset of size 40, we do not only know that these events form a clique in the incidence matrix, but can also conclude that in the perfect solution no other event is assigned to this particular room. This allows us to delete this room as an option for all other events from the event-room matrix. As this may create new single room option events, this step has to be implemented iteratively until there are no further changes. Unfortunately, this does not just speed up the algorithm, but changes its results, as flexibility in the possible assignments is lost. Experiments show that the algorithm presented here does suffer from this step, and it was therefore not used. Nonetheless, for other algorithms it may be beneficial.

Besides these two static matrices, we keep a number of dynamic data structures during the execution. An attendance matrix of dimension ($students \times 45$) that tells us for the current timetable in which time-slots a given student has events. Also, we have a matrix of dimension ($students \times 5$) that keeps track of the number of events per day for a given student. These two will allow us to implement a cheap evaluation of new timetables. To represent the current time-table, we have arrays identifying the time-slot and room of every event, as well as a representation that gives us the events currently placed in a particular time-slot. This last information reduces the computational effort when checking the feasibility of neighbourhood moves.

2.2 Phase I: Feasibility

With these preparations completed, we can enter Phase I, where we try to construct an initial feasible timetable using graph colouring and maximum matching algorithms. In this phase, we will try to assign events to time-slots and within the time-slots to rooms in such a way that no hard constraints are violated. At first, we will attempt to achieve this goal using only time-slots without penalty, thus implicitly catering for the soft constraint involving end-of-day events. If this fails, we open up the yet unused but available slots.

Initial attempt: We start by using a sequential colouring algorithm for the graph given by the incidence matrix. The order in which events are considered for colouring is the degeneracy order [4]. The assignment of a time-slot (which we identify with the colours) to the event under consideration works in the following way: we determine all those slots among the initial 40 slots that are feasible for the event, i.e. there is no colour conflict with an already placed event. Among these options we assign a slot with the minimal number of events in it, provided the number of events in this time-slot does not exceed the number of rooms. Events that cannot be assigned to a time-slot are placed in a pool of unassigned events for further consideration.

After this initial assignment of time-slots, we run a maximum matching algorithm for bipartite graphs [5] for each time-slot. The bipartite graph is the subgraph of the event-room matrix where we only use those rows that correspond to the events assigned to the time-slot under consideration. Events that do not get a room in a maximum matching are removed from the list of events in this time-slot and go into the pool of events that need further consideration.

Improvement attempt: We call the next phase *improvement attempt* and describe it here is full detail. It will be used in unchanged form later again. The idea is to take every unplaced event and scan the 40 existing time-slots: first we check whether there is a colour-conflict with a given time-slot and if there is none whether we can add the event to the time-slot such that a maximum matching algorithm can assign rooms to this event and to all events that were previously in the time-slot. If this is possible, the event is assigned to this time-slot and room and leaves the pool of unassigned events.

There are two mechanisms why this may work at this point:

- events that could not find a time-slot and therefore entered the pool during the colouring phase of the initial attempt may now fit in some time-slots where events were removed during the room assignment phase.
- events removed during the room assignment phase may well fit into other time-slots.

Shuffling: For the events still left, there is no way of placing them unless we change existing assignments. We call the first step in this direction *shuffling*. For a number of iterations we look at every event from the pool of unplaced events and choose a time-slot at random. First we check whether there is a colour-conflict between this element and the chosen time-slot. If this is not the case, we provisionally assign the event to this time-slot and run a maximum matching algorithm. If the number of events that are assigned to a room is one larger than the initial number of events in this time-slot, we have managed to place an unassigned event. Otherwise, we take the one event without room and put it into the pool of unplaced events (note that the number of events with feasible room assignments cannot drop below the initial number). The hope is, that in such a case the newly unassigned event is different from the initially unassigned event. If this is true, we run the procedure called improvement attempt (see 2.2) for this event before placing it in the pool of unassigned events, hoping that it fits into a different time-slot. To increase our chance, that the initially unassigned event is part of the new maximum matching, we randomize the order in which nodes appear in the maximum matching algorithm.

Blow-ups: The events that are still left must be considered as problematic and we have to go one step further in using invasive measures, that change the assignments made so far. The idea here is to force the placement of an unassigned event in an existing time-slot by first removing all events from this time-slot (*blow-up*). Then, we place the unassigned event in the now empty time-slot. Next, we fill up the time-slot with those removed events that do not have a colour-conflict with the initially unassigned event. Then we assign rooms to the events and place those events that had a colour-conflict or that were not given a room in the pool of unassigned events.

This way, we accept changes that may lead to an increased number of unplaced events. After this change, we apply improvement attempt to all unassigned events and afterwards we run a number of repetitions of the shuffling step. After a certain number of blow-up steps (i.e. blow-ups combined with improvement attempt and shuffling), we restart the whole procedure at the assignment that was the best so far in terms of number of unassigned events. This is necessary, as often the number of unplaced events would otherwise just keep growing.

Opening the last slots: Events that are not placed despite the above efforts are distributed over the last five time-slots with the methods described above. If this does not place these events conflict-free in the 5 new time-slots, we could use the shuffling and blow-up steps on all 45 time-slots to overcome problems. In reality however, this proves unnecessary, as the number of events left after the blow-ups are for all instances and all random seeds below 5, which guarantees finding a feasible initial assignment.

2.3 Phase II: Slot Sequence

Having found a feasible assignment which does in the majority of cases satisfy the soft constraint of not using end-of-day time-slots, we turn our attention towards the other two soft constraints, namely students with single events on a day and students with more than two events in a row. In the second phase of the algorithm, we take the time-slots formed during phase one and try to sequence them optimally. The individual time-slots, i.e. the grouping of the events into 45 subsets, are not changed. For this optimization we will use Simulated Annealing (SA) [6].

The solution space are all possible permutations of the time-slots and the neighbourhood move is swapping two entries of the permutation. This formulation automatically preserves feasibility. Improving moves and side steps are always accepted. Deteriorating moves are accepted with a probability

$$P(Acceptance|\Delta, T) = e^{-\frac{\Delta}{T}}$$

where Δ is the size of the deterioration in the objective function and T the current temperature. There are 35 different temperature levels T, starting at $T_{start} = 6.5$ and going down to $T_{end} = 1.5$. The schedule used is a standard geometric cooling, this means

$$T_{n+1} = T_n \times (1 - \delta_t)$$

where $\delta_t \ll 1$ is determined in such a way, that T_{end} is reached after 35 steps. At a given temperature, all possible moves are tried in a deterministic order, independently of the acceptance or rejection of earlier moves. We note that this simulated annealing problem has a connected search space and that the neighbourhoods are equi-sized.

Our selection process differs in two respects from the standard. Firstly, we choose to scan the neighbourhood deterministically instead of randomly. Secondly, we do not react in any way if a move is accepted, i.e. we do not restart the neighbourhood scanning in this new solution. We choose this approach to reduce computation time. We can be sure that every possible pair exchange is considered at least once per temperature without running a large number of random steps. This is acceptable as this phase is only a supportive measure for the main part of the algorithm, which is described in the next section. It is not desirable to spend much computation time in order to improve the result of this phase as it is very likely that by fixing the time-slots in this feasible but arbitrary way, the optimal sequence will have an objective function value far away from the overall achievable optimum of 0.

2.4 Phase III: Exchange of events

This phase forms the main part of the algorithm, both in terms of time consumption as well as impact. The basic idea is again a local search guided by Simulated Annealing. The starting position in the solution space will be the best timetable found by sequencing. The neighbourhood is defined by all timetables where exactly one pair of events has swapped time-slots. In comparison with other neighbourhoods suggested, see for example [7], this is a very simple structure. We only accept moves where this swap does not lead to infeasibilities, be it student clashes or events that cannot be assigned a room within their time-slot.

As we have in most instances managed to put all events into 40 time-slots, this insistence on feasibility while only considering pair swaps, seriously curtails the algorithm's flexibility. After all, there are no events that could be swapped in the 5 end-of-day slots. Instead of complicating the algorithm by introducing another neighbourhood move that changes only 1 event, we introduce 10 so-called *dummy* events. These are additional events without students attending them that can go in all rooms. We place 2 of them in every end-of-day time-slots, so that they can take part in the swapping of events. Apart from keeping the simple neighbourhood structure, this allows us also to control the maximum number of events in end-of-day time-slots. In the final timetable, these dummy events are just removed. These measure has the same flavour as allowing temporary infeasibilities to occur, in the hope that they will be correctable later but allowing the local search in the meantime to discover more attractive regions of the search space.

A second manipulation of the problem deals with *empty* events, i.e. original events that have no students. One could think of this as a flaw in the instance generator, but there is actually a reasonable interpretation for them: they are events whose audience cannot be determined in advance. So, they should not in any way affect the objective function, yet they have to be placed in the timetable. In the original formulation, these events may not be able to go into all rooms. This is for the reminder of the search changed. Of course, this has to be corrected in the final timetable. Those empty events that end up in a room that is infeasible have to be placed somewhere else in the timetable. This is relatively easy as they clash with no other events. They can also go into end-of-day slots without causing penalties. It turns out that this repair of the final timetable was always possible and no timetable had to be discarded because of this. The reason why this helps is that valuable room resources in the not penalized time-slots are not blocked by events that could be placed in end-of-day slots without incurring penalties.

Main Schedule As mentioned above, we use a SA approach to optimize the current timetable with a neighbourhood that is defined by all feasible swaps of two events. Improving moves and side steps are always accepted. Deteriorating moves are accepted with a probability

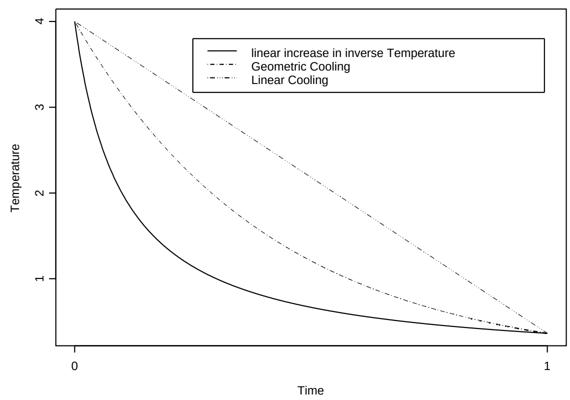
$$P(Acceptance|\Delta, T) = e^{-\frac{\Delta}{T}}$$

where Δ is the size of the deterioration in the objective function and T the current temperature. We start at $T_{start} = 4$ and go down to $T_{end} = 0.36$. The cooling schedule we use between these two temperatures is not the standard geometric cooling. Instead, in our schedule the inverse temperature T^{-1} grows linearly. This leads to

$$T_{n+1}^{-1} = T_n^{-1} + \delta_t \equiv T_{n+1} = \frac{1}{T_n^{-1} + \delta_t} \approx T_n \times (1 - \delta_t \times T_n)$$

So, in contrast to geometric cooling, the cooling factor between two steps depends on the current temperature. The two different schedules plus a linear schedule are shown in Figure 2. This schedule was chosen as preliminary tests showed it to perform better than geometric or linear cooling. Mainly, because a high starting temperature was found to be important for good results and the geometric schedule spends too little time in low temperature regions when used with a high T_{start}.

Fig. 2. Three different cooling regimes for Simulated Annealing



For a given temperature T, all possible moves are tried in a deterministic order, independently of the acceptance or rejection of earlier moves. This makes for $\frac{events^2 - events}{2}$ moves per temperature level. We will call this quantity a *cycle*. The majority of moves in a cycle will be discarded as infeasible. Again, we do not follow the standard SA procedure of selecting neighbours at random.

In contrast to Phase I, it is not practical to preset the number of temperature steps between T_{start} and T_{end}. There are huge differences in time demand from instance to instance for the completion of one cycle. Therefore, we first try to get an on-line prediction of the time demand for a given instance. For a fixed time (roughly 5% of the available time) we run the SA routine at T=0.15 recording how many cycles can be completed in this time. We then determine the number of temperature levels or cycles to be used in this phase, so that at its end 80% of the available time has been used, i.e. 20% remain for the last part of phase III.

Earlier, we have already described how we manipulated the incidence matrix to speed up the detection of infeasible moves by translating room conflicts into student clashes. But swaps that do not cause student clashes have still to be checked for whether rooms can be assigned. As mentioned above, a standard maximum matching algorithm between events and rooms in a time-slot is used. The simple structure of our neighbourhood allows us to save time on this. In order to determine whether we can assign rooms to the old events in a time-slot plus the new one minus the one that has left, we do not start the maximum matching algorithm from scratch. Instead, we start the matching algorithm with the partial assignment of the old events to their rooms. This way, there is at most one augmenting path to be found, which greatly reduces the time demand.

For the actual evaluation of feasible moves, we use a fast Δ-evaluation. Instead of re-evaluating the whole timetable, we make maximum use of the old value plus our knowledge of the neighbourhood structure. First, we determine all the students that are in one of the two events to be swapped, but not in both. Only these students are relevant for the change in the objective function value. We then use their rows in the old attendance matrix, the matrix describing for every student in which time-slot he has events, to determine the change caused by the swap. We only have to look 2 time-slots up and 2 time-slots down in this matrix to decide on the change in penalty from consecutive events. The penalty in single events per day can easily be determined from the matrix giving for every student the number of appearances per day. Changes in end-of-day penalties are straightforward to calculate. If a move is accepted, the data is updated before the next move.

Greedy search As a final step to improve the timetable, we run the SA search as described above with a different schedule until the time limit is reached. The starting temperature is $T_{start} = 0.5$ and the end temperature is $T_{end} = 0.1$ with 100 temperature steps between them. This phase is meant to greedily optimize the best solution found so far, placing emphasis on exploitation instead of exploration. At the end of the schedule, the greedy search is restarted, but not in the current solution but in the best found so far. This is repeated until the time limit is reached.

3 Results

In this section, the performance of the algorithm on the 20 competition instances will be presented. For every instance, 50 runs with different random seeds are recorded. If not indicated otherwise, quantitative results will be given as the mean over these 50 runs.

Table 1 presents the distribution of the computation time over the different phases in percent. Note, that due to rounding errors, the results may not add up to 100%. Those runs that were prematurely aborted, because an optimal solution was discovered within the time limit, have not been considered for in this table. We clearly notice the dominating role that the main schedule of

Phase III plays in the algorithm. We also see, that our online time prediction is reasonably accurate. We wanted to save 20% of the time for the greedy part of Phase III and have managed to design the main schedule accordingly. Phase II, the sequencing of the initial time-slots, takes up only a small portion of the overall time. Interestingly, the variation in the time demand for Phase II between the instances is almost perfectly correlated with the number of students in the instance (correlation value of 0.97). The time demand of Phase I varies greatly among instances, indicating that finding a feasible timetable is not equally hard for the competition instances.

Table 1. Run time distribution over the different phases in percent of total time, for the 20 competition instances

Instance	1	2	3	4	5	6	7	8	9	10
Phase I	3.4	1.5	8.4	6.5	8.6	0.5	0.4	1	0.7	7.9
Phase II	2.5	2.5	2.5	3.6	3.6	3.6	4.2	2.9	2.7	2.5
Phase III Main	74.9	70.7	70.1	72.6	72.7	80.1	80.1	71.8	72.5	72
Phase III Greedy	19.1	25.3	19	17.3	15.1	15.8	15.3	24.3	24.1	17.7

Instance	11	12	13	14	15	16	17	18	19	20
Phase I	5.3	8.4	6.7	2.7	1.6	0.3	12.3	1.5	3.8	0.2
Phase II	2.7	2.5	3.1	4.2	3.6	2.7	3.6	3	3.6	3.3
Phase III Main	72.1	71.2	71	78	79.8	68.5	69.3	80.6	74.8	74.1
Phase III Greedy	20	17.9	19.1	15.1	14.9	28.5	14.8	14.9	17.7	22.4

This claim is also supported by the first row of Table 2. Here, we see the average number of events that could not be placed into the 40 time slots, i.e. that had to go into end-of-day slots. For a number of instances, all runs succeeded in placing all events, but for others there were regularly events left. We note that even for the events with higher average values, such as instances 5 and 17, there were never more than 5 events left. So, finding a feasible timetable was always possible. The time Phase I takes and the average number of end-of-day events is with 0.93 highly correlated, supporting the notion, that variation in time demand for Phase I is due to varying difficulty of successful initialisation.

At this point we note that the competition instances are well-behaved in terms of constrainedness. It is well possible and maybe more realistic to have instances where the generation of feasible initial timetables is harder, especially when using only 40 instead of 45 time-slots. Evidently, the algorithm presented here would have to be modified in order to deal with such a situation, as it depends on the successful completion of Phase I. There is currently work in progress to use Grouping Genetic Algorithms to generate initial timetables for UCTP instances where the presented method fails.

The further rows of Table 2 show the objective function value of the best solution found so far at the end of the respective phase. Phase I values are not yet optimised and represent therefore some kind of random sample from the feasible solution space. Unsurprisingly, the values are again almost perfectly correlated (values of 0.98) with the number of students. This could be expected from a random sample, as all penalties are tied to students, and therefore the more students there are the more penalty we can incur. Phase II manages to cut the objective function value drastically by ordering the time-slots. An interesting observation is that the improvement achieved in this phase is very similar for all instances. The resulting values are highly concentrated around 45% of the starting value.

Evidently, the main schedule of Phase III has the most significant impact on the objective function. The values of the previous phases show no correlation with these values. The greedy part of Phase III can still improve these results, but it seems that its impact is higher on those instances that have already a low penalty value.

Table 2. Results at the end of the individual phases, for the 20 competition instances

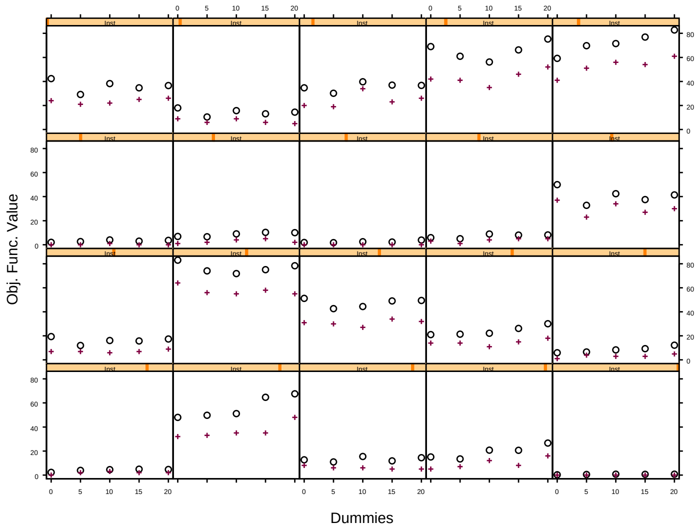
Instance	1	2	3	4	5	6	7	8	9	10
Phase I: end-of-day events	0.3	0.0	2.4	0.6	3.1	0.0	0.0	0.0	0.0	1.8
Phase I	537	493	545	760	846	800	859	634	547	539
Phase II	245	219	262	350	392	347	365	271	249	261
Phase III Main	32	15.9	32.6	61.6	73.8	6.6	11.1	6.5	12.3	36.4
Phase III Greedy	30.2	11.4	31	60.8	72.1	2.4	8.9	2	5.8	35

Instance	11	12	13	14	15	16	17	18	19	20
Phase I: end-of-day events	0.6	2.7	1.2	0.2	0.0	0.0	3.3	0.0	0.3	0.0
Phase I	557	535	662	889	761	582	820	516	786	761
Phase II	259	246	301	368	325	274	387	225	367	330
Phase III Main	16.9	78.1	48.6	27	12	9.7	56.4	14.2	19.6	3.5
Phase III Greedy	12.9	76.3	47.1	22.3	8.4	3.4	54	9.4	16.4	0.5

Before we will discuss the final results in more detail, including the best values found, we take a quick look at the effect of the dummy events introduced in Section 2.4. In Figure 3 we see a panel of all twenty instances. In every panel box, we can see the average performance over 20 runs for a varying number of dummy events, indicated by the circle. The crosses mark the best solution found over 20 runs. We realise that the effect is not homogeneous over the instances. We still try to make some general comments. In general, too many dummy events lead to a loss in performance, clearly so for 20 dummy events, and also to a lesser extend for 15. Since the more dummy events, the more flexibility in the exchange phase, we must introduce another information to understand this result.

The time demand to complete a single cycle of the exchange phase, i.e. trying to exchange all possible pairs of events, increases steeply with the number of dummy events. In other words, as the number of dummy events goes up, the

Fig. 3. Best and average performance for varying number of dummy events, for the 20 competition instances



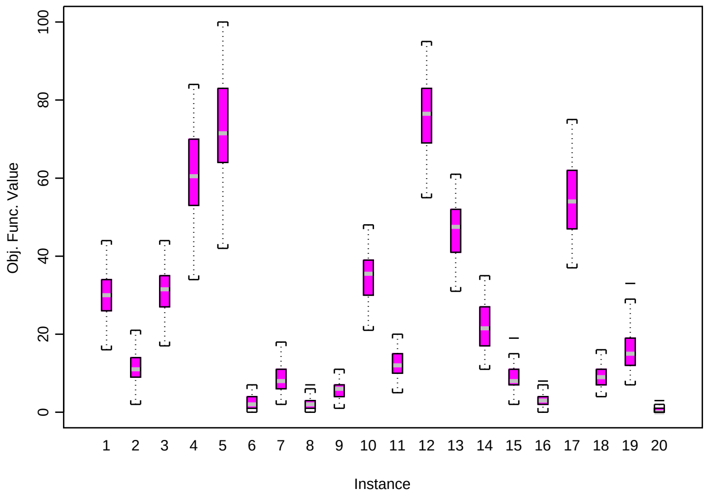
number of cycles that can be completed within the time limit goes down. This effect is very pronounced, as exchanges involving one dummy event are much more likely to be feasible. Therefore, their evaluation is much more likely to involve a fitness function evaluation. In a situation, where this is a complexity driver, this leads to increasing time demand. So, we can say that the gains in flexibility have to be balanced with the losses in speed.

On the other hand, there is a good number of instances, where the total absence of any dummy events will cause a sub-optimal performance. Most notably against this trend is the performance on instance 5. The graphs do indeed suggest that a low number of dummy events, i.e. 5 or 10, will be best for the overall performance, although maybe not optimally for certain instances. As the final choice was based on best and not mean performances, 10 dummy events were used in the final version of the algorithm, although 5 looks better on the mean values.

Figure 3 shows boxplot graphics for the 50 runs per competition instance. It gives us an idea about the variability of the performance for different random seeds. Clearly, the variability increases with the mean values. It also shows us that the best results for the instances are not lucky outliers, but fit very well to the overall distribution of results.

Table 3 presents the best values obtained by different algorithms. The official competition winner is an early version of the 3-phase approach presented here, the best algorithm at the competition is an early version of the Chirandini et

Fig. 4. Boxplots showing the 50 individual runs on the 20 competition instances



al. algorithm [8]. Both algorithms were in the aftermath of the competition substantially improved. This has led to significantly enhanced performances. The results for both up-to-date version are taken from 50 different seeds runs. The 3-phase approach yields the best result achieved on all 20 instances. On 4 of them, it finds an optimal timetable.

4 Conclusion

This paper presents a state-of-the-art algorithm for the University Course Timetabling Problem (UCTP) based on Simulated Annealing. The results are clearly outperforming the next best algorithm known for UCTP. Yet, there is still plenty of scope for further research and improvement. Only for 4 out of 20 instances, a global optimum has been found so far and on some instances, the best objective function value is still far away from 0.

It is shown, that with a straight-forward approach, it is possible to construct initial feasible timetables, that use few or none of the penalized time-slots. Furthermore, a simple neighbourhood seems to suffice in order to achieve excellent results in the optimization phase.

The computational results for the so-called dummy events indicate, that the temporary acceptance of infeasible moves or at least moves that are clearly hurting the objective function, has beneficial effects by increasing the algorithm's

Table 3. Best results from different algorithms, for the 20 competition instances

Instance	1	2	3	4	5	6	7	8	9	10
Official Competition Winner	45	25	65	115	102	13	44	29	17	61
Best Algorithm at the Competition	57	31	61	112	86	3	5	4	16	54
Chirandini et al.	45	14	45	71	59	1	3	1	8	52
3-phase approach	**16**	**2**	**17**	**34**	**42**	**0**	**2**	**0**	**1**	**21**

Instance	11	12	13	14	15	16	17	18	19	20
Official Competition Winner	44	107	78	52	24	22	86	31	44	7
Best Algorithm at the Competition	38	100	71	25	14	11	69	24	40	0
Chirandini et al.	30	75	55	18	8	5	46	24	33	0
3-phase approach	**5**	**55**	**31**	**11**	**2**	**0**	**37**	**4**	**7**	**0**

flexibility. Yet, the amount of such moves or the degree of infeasibility accepted has to be balanced carefully.

Acknowledgements: Philipp Kostuch acknowledges the support of a Marie Curie Training Site fellowship funded by the Improving Human Potential (IHP) programme of the Commission of the European Community (CEC), grant HPRN-CT-2000-00032, as well as the ongoing support of the Rhodes Trust. Special thanks go to Krzysztof Socha for fruitful discussions and valuable tips.

References

1. Rossi-Doria, O., Sampels, M., Chiarandini, M., Knowles, J., Manfrin, M., Mastrolilli, M., Paquete, L., Paechter, B.: A comparison of the performance of different metaheuristics on the timetabling problem. In Burke, E.K., De Causmaecker, P., eds.: Proceedings of PATAT 2002. Volume 2740. (2003)
2. www.idsia.ch/Files/ttcomp2002.
3. Socha, K.: $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System for International Timetabling Competition. Technical Report TR/IRIDIA/2003-30, Université Libre de Bruxelles, Belgium (2003)
4. West, D.B.: Introduction to Graph Theory. 2 edn. Prentice Hall (2001)
5. Galil, Z.: Efficient algorithms for finding maximum matching in graphs. ACM Computing Surveys (CSUR) **18** (1986) 23–38
6. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220** (1983) 671–680
7. Rossi-Doria, O., Blum, C., Knowles, J., Sampels, M., Socha, K., Paechter, B.: A local search for the timetabling problem. In Burke, E., Causmaecker, P.D., eds.: Proceedings of PATAT 2002 - Fourth International Conference on Practice and Theory of Automated Timetabling, KaHo Sint-Lieven, Departament of Industrial Engineering (2002) 124–127
8. Chiarandini, M., Socha, K., Birattari, M., Rossi-Doria, O.: An effective hybrid approach for the university course timetabling problem (2003)