Application of the Grouping Genetic Algorithm to University Course Timetabling

Rhydian Lewis and Ben Paechter

Centre for Emergent Computing, Napier University, Edinburgh EH10 5DT, UK.
{r.lewis|b.paechter}@napier.ac.uk

Abstract. University Course Timetabling-Problems (UCTPs) involve the allocation of resources (such as rooms and timeslots) to all the events of a university, satisfying a set of hard-constraints and, as much as possible, some soft constraints. Here we work with a well-known version of the problem where there seems a strong case for considering these two goals as separate sub-problems. In particular we note that the satisfaction of hard constraints fits the standard definition of a grouping problem. As a result, a grouping genetic algorithm for finding feasible timetables for "hard" problem instances has been developed, with promising results.

1 Introduction

The university course-timetabling problem (UCTP)[1] is the task of assigning the events of a university (lectures, tutorials, etc) to rooms and timeslots in such a way as to minimise violations of a predefined set of constraints. This version of the problem is already well known and in the last few years has become somewhat of a benchmark in a problem area that is notorious for having a multitude of different definitions. Specifically, given a set of events E, the task is to assign every event a room from a set R and timeslot from a set T (where $|T|$=45, comprising 5 days of nine timeslots). The problem is made taxing by the fact that various pairs of events *clash* - i.e. they can't be scheduled in the same timeslots because one or more student may be required to attend them both, making it analogous to the well known NP-hard graph colouring problem. There are also other complications - not all rooms are suitable for each event (it may be too small to accommodate the students or might not have the facilities the event requires), and cases of *double booking* (where a particular room is given more that one event in a timeslot) are strictly disallowed. A violation of any of these three so-called hard-constraints makes a timetable *infeasible*. The total number of possible assignments (timetables) is therefore $(|R|.|T|)^{|E|}$ and it can be easily appreciated that in anything but trivial cases, the vast majority of these contain some level of infeasibility.

In addition to finding feasibility, it is usual in timetabling problems to define a number of soft constraints. These are rules that, although not imperative in their

[1] Defined by B. Paechter for the International Timetabling Competition 2002. More details, and example problem instances are at [19].

satisfaction, should be avoided, if possible, in order to show some consideration to the people who will have to base their working lives around it. In this particular UCTP these are (1) no student should be scheduled to sit more than three events in consecutive timeslots on the same day, (2) students should not be scheduled just one event in a day and (3) events should not be scheduled in the last timeslot of a day.

This UCTP has been studied by Rossi-Doria *et al.* [14] as a means for comparing different metaheuristics. A conclusion of this substantial work is that the performance of any one metaheuristic with respect to satisfying hard constraints and soft constraints might be different; i.e. what may be a good approach for finding feasibility may not necessarily be good for optimising soft constraints. The authors go on to suggest that hybrid algorithms comprising two stages, the first to find feasibility, the second to optimise soft constraints whilst staying in feasible regions of the search space might be the more promising approach. This hypothesis was reinforced when the International Timetabling Competition [19] was organised in 2002 and people were invited to design algorithms for this problem - as it turned out, the best algorithms presented used this two-stage approach [1, 4, 13], using various constructive heuristics to first find feasibility, followed by assorted local improvement algorithms to deal with the soft constraints.

It seems then that we have a case for this two-stage approach, but although there is substantial work pertaining to the optimisation of soft constraints whilst preserving feasibility [1, 4, 11, 13], there is still a major issue of concern: How we can ensure that we have a good chance of finding feasibility in the first place? Indeed, this (sub)problem is still NP-hard [10] and should not be treated lightly. Therefore in "harder" instances, where methods such as those in [1], [4], [11] and [13] might start to fail, some sort of stronger search algorithm is required.

1.1 Grouping Genetic Algorithms and their Applicability for the UCTP

Grouping genetic algorithms (GGAs) may be thought of as a special type of genetic algorithm specialised for *grouping problems*. Falkenauer [9] defines a grouping problem as one where the task is to partition a set of objects U into a collection of mutually disjoint subsets u_i of U, such that $\cup u_i=U$ and $u_i \cap u_j=\emptyset$, $i \neq j$, and according to a set of problem-specific constraints that define valid and legal groupings. The NP-hard *bin packing* problem is a well used example - given a finite set of "items" of various "sizes", the task is to partition all of the items into various "bins" (groups) such that (1) the total size of all the items in any one bin does not exceed the bin's maximum capacity and (2) the number of bins used is minimised (a *legal* and *optimal* grouping respectively).

It was bin packing that was first used in conjunction with a GGA by Falkenauer [7]. Here, the author argues that when considering problems of this ilk, the use of classical genetic operators in conjunction with typical representation schemes[2] (as used for example with timetabling in [3] and [14]) are highly redundant due to the

[2] Such as the standard-encoding representations where the value of the *i*th gene represents the group that object *i* is in, and the indirect order-based representations that use a decoder to build solutions from permutations of the objects.

fact that the operators are *object-oriented* rather than *group-oriented*, resulting in a tendency for them to recklessly break up building blocks that we might otherwise want promoted. Falkenauer concludes that when considering grouping problems, the representations and resulting genetic operators need to be defined such that they allow the *groupings* of objects to be propagated, as it is these that are the innate building blocks of the problem, and not the particular positions of any one object on its own.

With this in mind, a standard GGA methodology is proposed in [9]. There has since been applications of these ideas to a number of grouping problems, with varying degrees of success. Examples include the equal piles problem [8], graph colouring [5, 6], edge colouring [12] and the exam-timetabling problem [6]. To our knowledge, there is yet to have been an application of a GGA towards a UCTP[3], although it is fairly clear that, at least for finding feasibility, it *is* a grouping problem - in this case, the set of events represents the set of objects to partition and the groups are defined by the timeslots. A feasible solution is therefore one where all of the $|E|$ events have been partitioned into $|T|$ feasible timeslots $t_1,\ldots,t_{|T|}$, where a feasible timeslot t_i $1 \leq i \leq |T|$ is one where none of the events in t_i conflict, and where all the events in t_i can be placed in their own suitable room.

Note that soft constraints are not considered in this definition. There are two reasons for this. Firstly, in this UCTP violations of soft constraints (1) and (2) arise as a result of factors such as timeslot ordering and the occurrence of sequences of events with common students across adjacent timeslots. Thus they are in disagreement with the more general definition of a grouping problem [9]. Secondly, if we *were* to take soft constraints into account at this stage they would need to be incorporated into the fitness function (which we'll define in section 2.1). But taking soft-constraints into account, while at the same time searching for feasibility (as used in [3] and [14] for example), might actually have the adverse effect of leading the search away from attractive (and 100% feasible) areas of the search space, therefore compromising the main objective of the algorithm.

2 The Algorithm

Similarly to the work presented in [4], [11], [15] and [17], in our approach each timetable is represented by a two dimensional matrix M where rows represent rooms and columns represent timeslots; thus if $M(a,b)=c$, then event c is to occur in room a and timeslot b. If, on the other hand, $M(a,b)$ is blank, then no event is to be scheduled in room a during timeslot b. In our method, the timeslots are always kept feasible and the number of timeslots in each timetable is allowed to vary. We therefore open new timeslots when events cannot be feasibly placed in any existing one, and the aim of the algorithm is to reduce this number down to the required amount $|T|$ (remembering that in this case $|T|=45$).

[3] We note that UCTPs are generally considered to be different problems to exam timetabling problems due to various differences that we will not go into here, but are detailed in [16] for example.

Fig. 1 shows how we perform recombination to construct the first offspring timetable using parents P_1 and P_2, and randomly selected crossover points $x_{1,\ldots,4}$. To form the second offspring the roles of the parents and the crossover points is reversed. The mutation operator we use follows a typical GGA mutation scheme - remove a small number (specified by the mutation rate *mr*) of randomly selected timeslots from a timetable and reinsert the events contained within them using the rebuild scheme (see below). We also use an inversion operator that randomly selects two columns in the timetable and swaps all the columns between them.

An initial population of timetables is constructed using the recursive rebuild scheme defined below. The same scheme is also used to reconstruct partial timetables that occur during recombination and mutation (with subtle differences regarding the breaking of ties - see table 1. Note too that event selection for mutation-rebuild is also different). The heuristics we use with this scheme are variations on those used in [1] and [11] and have already shown themselves to be powerful with this sort of problem.

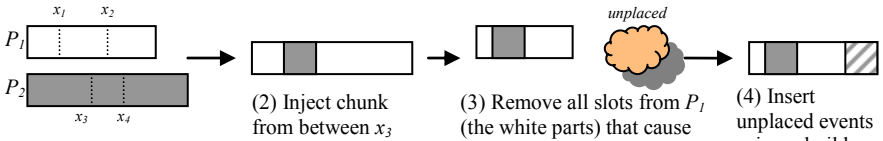
Specifically, the rebuild scheme takes an empty or partial timetable *tt* and a set U of unplaced events. It then assigns all $u \in U$ a room and timeslot to produce a complete timetable, opening new timeslots where necessary. Note that for the initial population generator $U=E$. Let S represent the set of timeslots in *tt* and let P represent the set of *places* in *tt* - i.e. $P=R\times S$.

Rebuild (*tt*, *U*)

1. If $U = \emptyset$ end, else if ($|S|<|T|$) open ($|T|-|S|$) new slots, else open $\lceil |U|/|R| \rceil$ new slots.
2. PlaceEvents (*tt*, *U*).
3. Rebuild (*tt*, *U*).

PlaceEvents (*tt*, *U*)

1. Pick $u \in U$ with the smallest number of possible places to which it can be feasibly assigned in *tt*. Break ties with H_1 (see table 1). For mutation, just choose any u randomly.
2. Pick the feasible place for u in *tt* that the least number of other events in U want. Break ties with H_2 (see table 1).
3. Remove u from U and insert it into *tt* at the chosen place.
4. If there are still events in U with feasible places, go back to step 1.



(1) Copy P_1 and P_2 and choose X-over points

(2) Inject chunk from between x_3 and x_4 at x_1

(3) Remove all slots from P_1 (the white parts) that cause duplicate events. Track the events that become unplaced.

(4) Insert unplaced events using rebuild scheme.

Fig. 1. How recombination is performed in this algorithm

2.1 Judging Criteria and the Fitness Function

When looking at the final output of the algorithm, it seems reasonable to assume that what we are ultimately interested in is the timetable's *distance to feasibility*, if indeed feasibility has not been found. In timetabling this can be measured in various ways such as the level of student inconvenience, the number of broken constraints, the number of extra timeslots used, etc. Of course, what *is* chosen depends first and foremost on user preference. In our approach we choose to use the number of unplaced events. This is calculated by doing the following: Recall that $|T|$ represents the target number of timeslots that we wish to use (i.e. 45), and $|S|$ represents the current number of timeslots. Additionally, let F_i indicate the number of events placed in timeslot i, where $1 \leq i \leq |S|$, and let $|S|$' represent the number of *extra* timeslots being used i.e. $|S|' = (|S| - |T|)$. The distance to feasibility is the total number of events in the $|S|$' timeslots to which F_i is minimal – i.e. the $|S|$' timeslots with the least events in them.

During the algorithm's run however, the distance to feasibility need not be the only measure that we use to determine fitness. Indeed, if other information is present then it makes sense to use it if it is thought that it can help guide the search towards more promising areas of the search space. Consequently, we use a fitness function somewhat akin to the one proposed for graph colouring by Eiben *et al*. [5]: we calculate the number of extra timeslots $|S|$' being used and the distance to feasibility. The fitness function is the total of these two values.

	Heuristic - H_1	*Heuristic - H_2*
Recombination	Choose the event that conflicts with the most others	Choose the place that defines the emptiest slot
Initial population	Choose randomly	Same as above
Mutation	N/A	Choose randomly

Table 1. Showing the various ways that ties are broken in the rebuild scheme for the three genetic operators (described in section 2). Note that H_1 for mutation is not applicable. As explained in the text, for this operator the order that unplaced events are inserted back into the timetable is entirely random.

3 Experimental Analysis

We created sixty test instances using an instance generator, which we separate into three classes: small, medium and large[4]. It is known that all have at least one feasible timetable. These instances were created with no reference to the proposed GGA but are deliberately intended to be troublesome for finding feasibility. This was mainly achieved by simple experiments whereby instances were created and run on two existing constructive algorithms [1, 11] that attempt to use stochastic heuristics to

[4] For the small instances $|E| = 200$ to 225 and $|R| = 5$ or 6. For the medium instances $|E| = 390$ to 425 and $|R| = 10$ or 11. For the large instances $|E| = 1000$ to 1075 and $|R| = 25$ to 28. Other parameters for the instances can be found online at the URL at the end of section 4.

place all events feasibly. With many instances, these algorithms could only place about 80% of events (and sometimes even less) before running out of ideas and getting totally stuck. We therefore tended to take these as the instances to use in our experiments. Indeed, between them these algorithms could not find feasibility in 52 of the 60 instances.

For all the experiments we used a PC Pentium 4 2.40GHz with 512MB RAM. For the evolution scheme, a steady-state population of size *ps* was used: at each step, two parents are selected using binary tournament selection with parameter *ts*. Two offspring are then created with recombination rate *rr*. These are then mutated and inserted back into the population, in turn, over the least fit individual. If there is more than one least-fit individual we choose between these randomly. Also at each step, *ir* individuals are also chosen randomly and inversion is applied. During the run we keep track of the fittest solution found so far according to the fitness function defined in section 2.1. This is the algorithm's final output. In all experiments we set *ps*=50, *ts*=0.9, *mr*=3, and *ir*=4.

For our first set of experiments[5] we tested the algorithm on all sixty problem instances. We introduced time limits of 30, 200 and 800 seconds for the instance sets small, medium and large respectively, and set the recombination rate to 0.5. In this case, even with the strict time limits and the fact that we performed minimal parameter tuning, the algorithm found feasibility in 23 of the 60 instances – fifteen more than the algorithms presented in [1] and [11]. Additionally, there is the obvious advantage that this new algorithm is able to produce a number of different solutions in the same run.

In the experiments we also noticed that, in some cases (10 in small, 3 in medium and 1 in large), solutions were actually found in the initial populations! Although this might lead the prudent reader to suspect that the test instances are suspiciously easy, we argue that if anything this just goes to highlight the strength of our rebuild heuristics. Indeed this could be somewhat expected - similar heuristics have already shown themselves able to find feasibility in one go with other well-known instances [1, 11] and so there is no reason why they shouldn't occasionally do the same here.

Our next experiments attempted to address an issue that we consider to be of particular interest with this algorithm - the consequences that the recombination operator has on the number of new individuals that can be produced within the time limit, and the effects that this has on movement in the search space. It is well acknowledged that the general goal of recombination is to aid the search by allowing useful building blocks from multiple parents to be combined into new, different and hopefully fitter offspring. However, as can be imagined in this algorithm, recombination is more expensive than mutation, so if it is not doing its intended job at an acceptable level then its presence is questionable. If, on the other hand, recombination *is* aiding the search, there will still be a trade-off between the amount of recombination that is used, and the resultant number of new individuals that can be produced within the time limit.

To investigate this, we conducted experiments using different recombination rates with all other parameters, including the time limits, remaining as previously defined.

[5] Full results can be found online at the URL defined at the end of section 4.

For all problem instances we ran five separate trials using recombination rates 0.0, 0.25, 0.5, 0.75 and 1.0.

Table 2 shows clearly that as we increase the recombination rate, the mean number of new individuals produced within the time limit falls. The effect that this characteristic has on the resultant movements in the search space within the time limit is illustrated in fig 2. This graph shows the mean fitness of populations for each of the twenty medium instances, over time. The effect of using a high level of recombination is shown well here - a rate of 1.0 for instance, at least for the first half of the run, gives the slowest progression through the search space per second, whilst a rate of 0.0 offers the most. Clearly, if the time limits for these instances were shorter, then using no recombination would seem the more sensible choice. What is also noticeable however is that the higher rates of 0.75 and 1.0 still seem to be making positive movements in the search space towards the end of the runs, whilst the other three rates (which use less recombination) seem to be levelling off. Indeed, the lines for both 0.75 and 1.0 both cross the other three towards the end of the run and, if the time limit were increased, look like they could go on to make further positive movements.

	Rr=0.0	*Rr=0.25*	*Rr=0.5*	*Rr=0.75*	*Rr=1.0*
Small	24382	19750	14542	11600	7662
Medium	62752	45658	32262	26160	10454
Large	67594	38424	18148	4072	1052

Table 2. Showing the mean number of new individuals the algorithm is able to produce within the specified time limits for the different recombination rates.

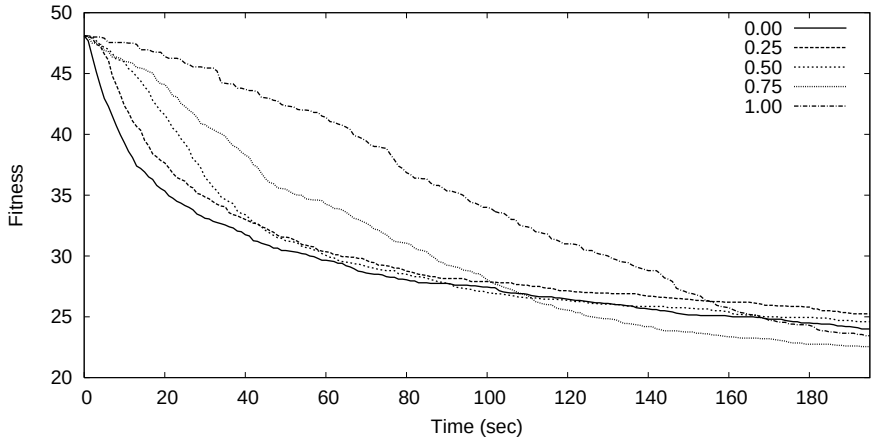


Fig. 2. Mean fitness of populations for the twenty medium instances, per second, for the different recombination rates.

This is exactly what is shown in fig 3. Here, rather than concern ourselves with CPU time, we look at the way the fitness changes according to the number of timetable evaluations performed. This measure is frequently considered instead of CPU time

when looking at the performance of an evolutionary algorithm [3] as evaluation can often be the most expensive operation. Indeed, timetabling problems in particular are prone to this, as often there might be an abundance of different real-world constraints that need to be checked, resulting in very complex and expensive evaluation functions. Although this is not the case for the relatively simple UCTP used here, this still seems a reasonable criterion to study, as it might be the case that as we add extra realism to the problem, the fitness function might become so intricate that the relative expense of the recombination operator becomes negligible. In this graph then, it can be seen quite clearly that as we increase the recombination rate, both the amount of positive movement through the search space per evaluation *and* the quality of the final solution increases (reflected in the steepness of the curve and the lower levelling-off point respectively). It can also be seen that as we decrease the recombination rate, these characteristics lessen in a uniform fashion. Thus it would seem that recombination is indeed doing its intended job.

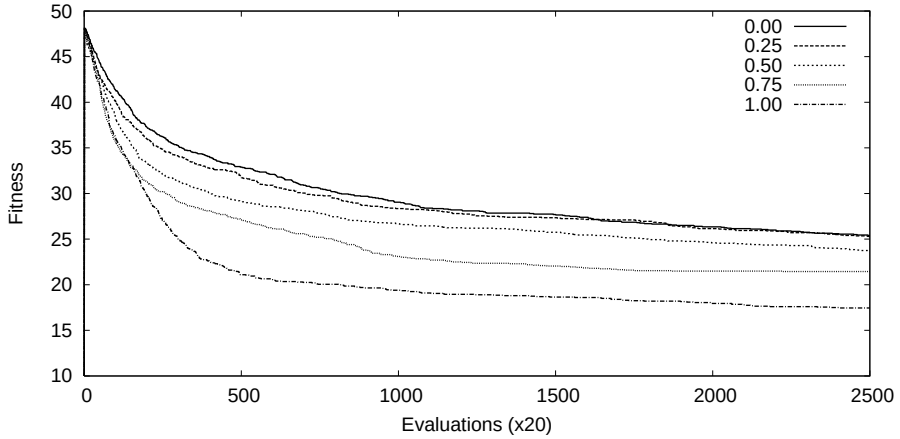


Fig. 3. Mean fitness of populations for the twenty medium instances, per evaluation, for the different recombination rates.

4 Conclusions and Further Work

We have presented an algorithm for university-course timetabling that combines powerful constructive heuristics with GGA methodology. To our knowledge this is the first such algorithm aimed at this problem domain. Our initial experiments with sixty new "hard" problem instances have shown that results are promising with regards to the number of cases where we have found feasibility, although we do not yet claim these results to be state of the art. Further experiments in this paper have shown that the use of recombination does seem to aid the search towards better solutions, but if the time limit imposed is highly restricted it should probably be used in smaller amounts. Finally we round off this paper with some remarks about possible future work and various other issues that might be of interest.

A place where some improvements might be found is the fitness function. Currently the search landscape defined by our function can sometimes seem a little stepped in nature - especially when we are close to finding feasibility. In these cases the jump from near-feasibility to full-feasibility might well be a lucky one. However, other *smoother* fitness functions that concentrate more on favouring solutions with good combinations of events in timeslots might show to improve the search. We have recently been conducting experiments using the fitness function

$$f = \frac{\sum_{i=1}^{|S|} (F_i / |R|)^2}{|S|} \tag{1}$$

where F_i is as defined in section 2.1. Although this is more archetypal of this type of algorithm [6, 7, 8] it actually seems to give significantly worse results than our current function under the same test conditions. Further work might reveal other promising avenues.

Other improvements may also be seen through the introduction of some sort of smart-mutation [3] and/or various local search operators [14]. Whether or not these will improve the algorithm's results is also pending further work.

It is also worth noting that not all cases of UCTPs will have these same classical grouping characteristics as this one. In some problem definitions, all timeslots might not be the same because certain resources might be unavailable in certain predefined timeslots. Secondly, some cases may incorporate hard-constraints that pan across the timeslots such as the specification that one event must take place before another etc. in which case the *ordering* of the timeslots might also become important.

Finally, and perhaps most noticeably, we have not addressed for the time being the important task of optimising the soft constraints. The two-stage approach that we support here dictates that only once feasibility is found should soft constraints be considered. However, whether effective searches can still be made in the more restricted, feasible-only search space for these "hard" instances is yet to be investigated. In the meantime, the sixty problem instances and full tables of results are available at www.emergentcomputing.org/timetabling/harderinstances.htm.

References

1. H. Arntzen and A. Løkketangen. A Tabu Search Heuristic for a University Timetabling Problem. (2003), Proceedings of the Fifth Metaheuristics International Conference MIC 2003, Kyoto, Japan. An older version of the paper is also available at (accessed Dec 2004) http://www.idsia.ch/Files/ttcomp2002/arntzen.pdf
2. E. Burke, D. Elliman and R. Weare, Specialised Recombinative Operators for Timetabling Problems. (1995) In Proceedings of the AISB (AI and Simulated Behaviour) Workshop on Evolutionary Computing, Springer LNCS 993, pp. 75-85.
3. D. Corne, P. Ross, H-L Fang, The Practical Handbook of Genetic Algorithms, Applications, Volume 1. (1995). Edited by Lance Chambers, CRC Press, pp 219-276.
4. M. Chiarandini, K. Socha, M. Birattari, and O. Rossi-Doria. An effective hybrid approach for the university course timetabling problem.(2003) Technical Report AIDA-2003-05, FG Intellektik, FB Informatik, TU Darmstadt, Germany.

5. A.E. Eiben, J.K. van der Hauw, and J.I. van Hemert. Graph Coloring with Adaptive Evolutionary Algorithms (1998). Journal of Heuristics, 4(1):25-46.
6. W. Erben, A grouping Genetic Algorithm for Graph Colouring and Exam Timetabling (2000). Proceedings of the Practice and Theory of Automated Timetabling III, Springer LNCS 2079, pp132-156.
7. E. Falkenauer. A New Representation and Operators for GAs Applied to Grouping Problems (1994). Evolutionary Computation, Vol. 2 Issue 2, Summer 1994 pp123-144.
8. E. Falkenauer. Solving equal piles with the grouping genetic algorithm (1995) Proceedings of the 6th Int. Conf. on Genetic Algorithms. Morgan Kaufmann, pp 492-497.
9. E. Falkenauer. Genetic Algorithms and Grouping Problems (1999). John Wiley and Sons Ltd.
10. M. R. Garey and D. S. Johnson. Computers and Intractability – A guide to NP-completeness. (1979). W. H. Freeman and Company, San Francisco.
11. R. Lewis and B. Paechter. New Crossover Operators for Timetabling with Evolutionary Algorithms (2004). In proceedings of the 5th International Conference on Recent Advances in Soft Computing RASC2004. ISBN 1-84233-110-8, pp189-194. A copy is also available at http://www.soc.napier.ac.uk/publication/op/getpublication/publicationid/7207469
12. S. Khuri, T. Walters and Y. Sugono. A Grouping Genetic Algorithm for Coloring the Edges of Graphs (2000), Proceedings of the 2000 ACM/SIGAPP Symposium on Applied Computing, ACM Press, pp 422-427.
13. P. Kostuch. The University Course Timetabing Problem with a 3-stage approach (2004). In E. Burke and M. Trick (eds.) Proceedings of the 5th International Conference on the Practice and Theory of Automated Timetabling, pp 251-266.
14. O. Rossi-Doria, M. Samples, M. Birattari, M. Chiarandini, J. Knowles, M. Manfrin, M. Mastrolilli, L. Paquete, B. Paechter, T. Stützle. (2002). A comparison of the performance of different metaheuristics on the timetabling problem. In E. Burke and W. Erben (eds.) Proceedings of the Practice and Theory of Automated Timetabling III, Springer LNCS 2740, pp329-351.
15. B. Paechter, H. Luchian, A. Cumming and M. Petriuc. Two Solutions to the General Timetable Problem Using Evolutionary Algorithms (1994). In Proceedings of the IEEE World Congress in Computational Intelligence, pp 300 305.
16. A Schaerf. A Survey of Automated Timetabling. (1995) Centrum voor Wiskunde en Informatica (CWI) report CS-R9567, Amsterdam, The Netherlands. A revised version appeared in Artificial Intelligence Review 13(2), 87-127.
17. K. Socha, J. Knowels, M. Sampels. A MAX-MIN Ant System for the University Course Timetabling Problem (2002). In Dorigo, M., Di Caro, G., Sampels , M. (eds.), Proceedings of the 3rd International Workshop on Ant Algorithms (ANTS'2002), Springer LNCS 2463, pp 1-13.
18. J. M. Thompson and K. Dowsland. A Robust Simulated Annealing Based Examination Timetabling System (1998). Computers and Operations Research 25 pp 637- 648 ISSN 0305-0548.
19. International Timetabling Competition - http://www.idsia.ch/Files/ttcomp2002, accessed Dec 2003.