Timetabling the Classes of an Entire University with an Evolutionary Algorithm

Ben Paechter, R. C. Rankin, Andrew Cumming and Terence C. Fogarty

Napier University, Edinburgh, Scotland
benp@dcs.napier.ac.uk

Abstract. This paper describes extensions to an evolutionary algorithm that timetables classes for an entire University. A new method of dealing with multi-objectives is described along with a user interface designed for it. New results are given concerning repair of poor recombination choices during local search. New methods are described and evaluated that allow timetables to be produced which have minimal changes compared to a full or partial reference timetable. The paper concludes with a discussion of scale-up issues, and gives some initial results that are very encouraging.

1. Introduction

Napier University uses a timetable that was produced by an evolutionary algorithm incorporating a local search. The system timetables 100% of classes and optimises them according to twelve competing objectives.

Timetabling the classes of a University involves finding timeslots for the events such that each event can have the resources (rooms, students, and lecturers) that it requires, and so that constraints on the relative timing of events are maintained. This process produces feasible timetables. In addition to producing feasible timetables, we want to produce timetables that are 'good' measured against some criteria. The production of feasible timetables involves satisfying the hard constraints of the problem. The production of good timetables involves satisfying as many of the soft constraints as possible.

The Napier University problem involves placing 2000 events into 45 timeslots and 183 rooms, and optimising the timetables of 700 lecturers and 1000 student groups. The number of ways to put 2000 events into 45 timeslots is 45^{2000}. Clearly, the vast majority of these timetables are infeasible, because some hard constraint is broken. The problem then becomes how to find good timetables in a search space that contains very few feasible timetables.

There have been several attempts to solve this type of problem with evolutionary algorithms; some examples of these can be found in [1], [2], [4], [5], [6], [7], [8], [10] and [18]. The method used here is distinguished by its use of local search to deal mainly with hard constraints and genetic operators to solve mainly soft constraints. Others have used evolutionary algorithms combined with local search in other ways.

For example, in [3] Burke et al. describe a system that timetables examinations using a local search, in this case the local search is used mainly to solve soft constraints.

2. Summary of Previous Work

The algorithm described here was originally presented in [11] where the principle of using an indirect representation was established. Refinements were made in [12]. In [13] the idea of timeslot suggestion lists was first presented and suitable recombination operators were defined. Directed and targeted mutation were addressed in [14]. In [15] the advantages of local search and Lamarckian writeback were clearly shown, and results for a large real problem were shown to give a considerable improvement over manual methods.

In order to solve the feasibility problem, a local search is employed which searches from a point in the search space specified by each new chromosome to a point with greater feasibility. The result of this is that the evolutionary algorithm can now search through the smaller space of feasible and nearly feasible timetables for timetables that are good.

An indirect representation is used which codes for how a timetable will be produced by the local search engine. The representation is split into two parts.

The first is a permutation that specifies the order in which the events should be considered when trying to fit them in to the timetable. When building an unseeded population the permutation is initialised using a heuristic which ensures that the more difficult-to-place events are considered first.

The second part of the representation specifies a number of suggested timeslots for each event (normally there are two suggestions, one coming from each parent). When building an unseeded population, the suggested timeslots for an event are assigned randomly from the list of possible timeslots for that event (those times when the event could take place if there were no other events to consider).

The search proceeds as follows: events are considered in the order specified by the permutation. For each event an attempt is made to place the event in the primary suggested timeslot. If this fails (because some hard constraint would be broken by doing so) then the other suggested timeslots are tried in order. If none of the suggested timeslots is possible, then other timeslots are tried according to a problem specific heuristic that examines the timeslots which do not incur a penalty first.

If at the end of this process the event has not been placed, it is considered unplaced (which attracts a fitness penalty) and the next event in the permutation is considered.

If an event is placed then the timeslot used is written back into the chromosome as the primary suggested timeslot. The timeslot that was occupying this position (if different) is moved into the second position (and any others are shuffled down). This writing back of the local search results makes the algorithm Lamarckian.

For each event a child inherits its primary timeslot suggestion from one parent and its secondary timeslot suggestion (if more than one suggestion is stored) from the other. The operator is based on multi-point recombination, and as the chromosome is traversed there is an equal chance at every point that the parent contributing the primary suggestion will be switched. When using more than two suggested timeslots, the subsequent suggestions are taken alternately from each parent. This operator conforms with the concepts from Forma Theory of *respect* and *assortment* [16]. The permutation is inherited from one parent only.

Three mutation operators are used; all work on the primary suggested timeslot for one event. The first operator is a blind mutation that randomly reassigns the primary suggested timeslot to some other possible timeslot. This mutation operator ensures that all parts of the search space are reachable.

The other two mutation operators make use of problem specific knowledge to direct the mutation in a way that may be useful. The first is *selfish mutation* which involves an event "stealing" the timeslot used by another event. The second is *co-operative mutation* which involves an event finding another event with which it can swap primary timeslot suggestions, to the possible advantage of both. The directed mutation operators also make changes to the permutation to ensure that during the local search the events end up getting the timeslots they expect. These operators are very useful, particularly in the later stages of a run and when used in conjunction with *targeted mutation.*

Targeted mutation allows mutations of the chromosome to be targeted at those parts of the chromosome that code for parts of the phenotype that attract penalties on evaluation. During evaluation, a score is kept for each event of the extent to which that event detracts from the fitness of the whole timetable. When calculating the chance that the genetic material for an event will mutate, each event has a base chance of mutating, an amount is then added to this which is directly proportional to the degree to which this event detracts from the fitness of the whole timetable. Targeted mutation has little effect, if any, when used with blind mutation alone, but a significant effect when used with directed mutation operators, particularly in the early stages of a run.

Ross et al. in [17] have described other mutation operators for timetabling that use problem specific knowledge. There they are used mainly to solve hard constraints in an algorithm that does not employ a local search. Mutation operators which use problem specific knowledge were described earlier by Eiben et al. in [9].

After recombination and mutation a child may be less feasible than either of its parents. This decrease in feasibility will often be repaired by the local search mechanism undergone by all new chromosomes.

3. User Interface for the Evaluation of Multi-Objectives

During the local search hard constraints are never broken. This means that the resulting timetables never have broken hard constraints, but that some events may remain unplaced. The number of unplaced events can then be reduced by treating the constraint *all events must be placed* as a soft constraint. This soft constraint is then considered along with the other soft constraints. The breaking of soft constraints is measured using *problem measures*. Each problem measure counts the number of occurrences of some problem with the timetable, so we want to reduce the values of the problem measures.

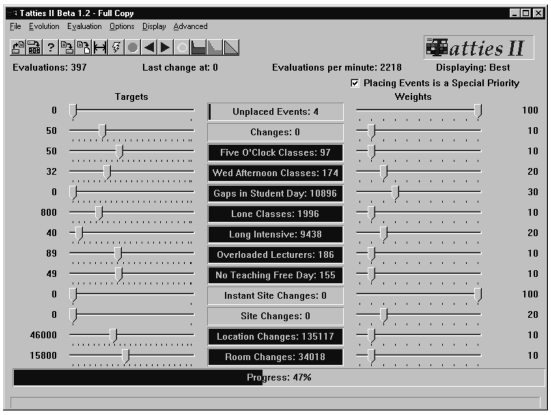

Fig. 1. The User Interface

The way in which the quality of a timetable is measured reflects the fact that users have targets for individual problem measures. Once the target for a problem measure has been met the user does not wish the algorithm to waste more effort reducing the value further. Users also care more about reaching the targets for some problem measures than they do for others. If users are only allowed to specify weights then they tend to change these as the run progresses when certain problem measures reach acceptable levels or levels beyond which they know no improvement is possible. If a run is going to take more than a few minutes then the user either has to sit and watch the evolution in case a weight needs to be changed, or has to accept that the algorithm may waste time optimising something that cannot or need not be optimised further.

In order to take account of this, a user interface has been designed that allows the user to specify (and change during the course of a run) a target t and a weight w, for each of the twelve problem measures: see Figure 1. Targeted mutation rates are then calculated so as not to try to improve problem measures beyond the target and the evaluation function is then constructed so as to give no extra benefit to a chromosome that reduces a problem measure below the target.

In order to evaluate a chromosome we need to know how much progress it has made towards each of the problem measure targets. In order to measure the progress towards a target we have to define the start point s. This is approximated by examining 200 random chromosomes and taking the worst score that occurs on that problem measure (users can provide other values for s if so required). If a problem measure has the value v then the progress on that problem measure p can then normally be calculated by: $p = \max(0, (v - t) / (s - t))$. If p has the value 0 then the target has been met. The progress of the algorithm over all problem measures P can be calculated as the weighted average over all values of p.

The user can also specify that placing events is a special priority. If this is the case then the comparison of two timetables is done sequentially. First the number of unplaced events is considered. If one timetable has fewer unplaced events then it is considered the better timetable. Only if the number of unplaced events is the same (or both timetables have reached the target for unplaced events) is the progress on the other problem measures considered.

4. Experiments with Timeslot Suggestion Lists

Experiments have been conducted to measure the effect of having different numbers of timeslot suggestions within the chromosome. With one suggestion, only one parent contributes to the placing of that event. With two suggestions there is a back-up timeslot from the other parent. When we have three or four suggestions then information from grandparents is stored (as a result of the recombination operator).

Two experiments were conducted. In the first the value of *P* that could be achieved in a set elapsed time was measured, given targets of zero on all problem measures. In the second the time taken for *P* to reach 0.05 (within 95% of the overall target) was measured for targets reflecting those commonly used by users. For each experiment real data for a single large department was used with 10 problem measures. Results were averaged over 50 runs. The same initial 50 populations were used for each set of runs. The results can be seen in figures 2 and 3. The chart in figure 2 is truncated at a point thought to be around the optimum value of *P*.

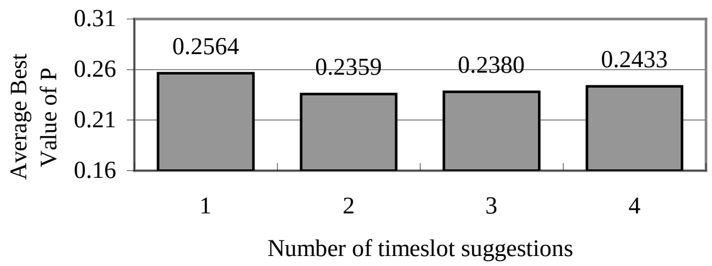

Fig. 2. Progress Achieved Over Given Time

Elapsed time is used as a measure in these experiments since the time taken to perform an evaluation depends on the chromosome being evaluated and the method being used. Hence, measuring against the number of evaluations would produce erroneous results. All experiments in this paper were carried out on a dedicated 200MHz Pentium Pro computer.

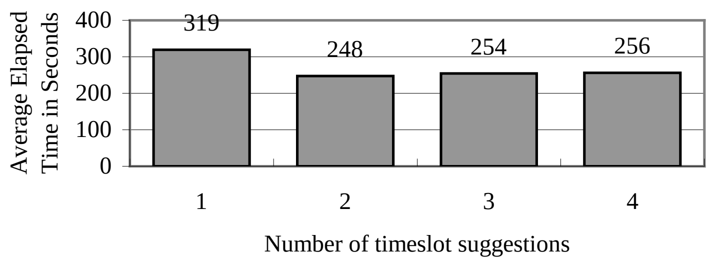

Fig. 3. Time Taken for Given Progress

For each experiment a paired Student's T-test shows a very highly significant (probability greater than 99.99%) improvement using suggestions from both parents, and no significant difference using suggestions from grandparents. For the first experiment we can say with 99% confidence that the average value of P with one suggested timeslot is 0.2564 ± 0.0069 and with two suggested timeslots is 0.2359 ± 0.0087. For the second experiment we can say with 99% confidence that the average time with one suggested timeslot is 319 ± 28 and with two suggested timeslots is 248 ± 23.[1]

These results are as we might expect, as storing suggestions from both parents is equivalent to repairing "mistakes" made by the recombination operator. If the "wrong" parent was chosen to contribute a particular timeslot, then this is rectified by the search mechanism. The lack of significant difference for increasing numbers of suggested timeslots is probably because any benefit from storing the timeslots suggested by grandparents is counteracted by the extra processing overhead required.

5. Using Reference Timetables and Minimising Change

A requirement that is often made of scheduling problems is that the solution should be as close as possible to some reference schedule. The reference schedule may have been produced manually, or may have been produced automatically from incomplete or partially different data (for example when tracking changes to data). This is a feature that is often required of university class timetabling systems.

An initial population can be seeded with the reference timetable and small variations on it. This feature alone allows processing to begin before all the data has been collected as runs with new data can be seeded with the results of previous runs. Of course there is a danger here that what was a global optimum may become a local optimum of the new data, so care is needed in deciding when is it better to start a run from scratch.

The seeding is done by first finding all the events that are common to both the present timetable data and the reference timetable. The primary timeslot suggestion for each of these events is then copied from the reference timetable. All but one of the chromosomes then undergoes a mutation in order to add some variety to the population. Where all the events are common to both timetables, and the reference timetable is the result of a previous run, the permutation can also be copied. Where the reference timetable is feasible given the present timetable data, the local search will not change it and the reference timetable will exist in the initial population.

Where it is also necessary to minimise changes, the number of changes to the reference timetable can be counted and treated as a problem measure. Hence changes can be minimised in the same way that the breaking of other soft constraints is minimised.

---

[1] Note that for the second experiment, results from one initial population were ignored. This was because the result for using one suggested timeslot was over 6000 seconds. The algorithm had become trapped in a local minimum at P=0.06. It was considered that this value was such an outlier that we could safely conclude that this was a very "unlucky" run and that more could be learned by not considering it.

When minimising changes a modified search algorithm is used: events are considered in the order specified by the permutation. For each event an attempt is made to place the event in the primary suggested timeslot. If this fails (because some hard constraint would be broken by doing so) then the event is considered unplaced and the next event in the permutation is considered. When all events have been considered a further attempt is then made to place the unplaced events. First, the secondary suggested timeslot is tried, then the other timeslots according to the heuristic that examines the timeslots which do not incur a penalty first.

This type of search is designed to minimise the "domino" effect of changing an event's timeslot when conducting the search. The difference between this method and the standard method is that with this method all events get to try their primary suggested timeslot before any gets to try other timeslots. There is no chance that an event unable to use its slot will cause an avalanche of changes by "stealing" another event's timeslot.

Table 1. Comparisons of Methods to Minimise Change

% Change	Standard Search	Modified Search
Number of Changes is not a Problem Measure	54.5±1.8	48.3±1.1
Number of Changes is a Problem Measure	47.6±1.3	43.9±0.9

The following experiment was designed to test the modified search algorithm and the effect of treating changes to the reference as a problem measure. A real timetable data set was optimised to produce a reference timetable. The data set was then changed in a number of ways so that about 18% of the events could no longer be placed in the timeslots specified by the reference timetable. These events now had to find new slots and in doing so some would have to displace other events. The new data set was then optimised and changes to the reference were counted. The system was allowed to run until all the events had been placed. The results, over 50 runs, can be seen in Table 1. The figures are percentage change from the reference timetable and are given with 99% confidence intervals.

The results clearly show that each of the two methods gives an improvement, both individually and together. A Student's paired T-test shows an extremely high significance in the improvements (greater than 99.9999% for each comparison pair).

While the modified search keeps the number of changes to a reference timetable lower, it makes the algorithm less effective when it is not necessary to stay close to a reference. The *Progress Achieved Over Given Time* experiment produced a result of P=0.2656±0.008 for the modified search compared with P=0.2359±0.0087 for the standard search – clearly a worse result (99% confidence intervals). The *Time Taken for Given Progress* experiments produced a result of 646±550 seconds for the modified search compared with 248±23 for the standard search – over twice the average time for the same result and a much increased spread. This is partly surprising since reducing the "domino" effect of changes to the chromosome reduces epistasis. The reduction in performance may be due to the fact that since an unplaced event does not have the chance to "steal" another event's timeslot, the unplaced events tend to stay the same with each evaluation and the search is restricted to a smaller part of the search space. It may be possible to rectify this by increasing the amount of directed mutation. Further work is required to investigate this area.

6. Issues of Scale-Up

One of the crucial questions asked by researchers looking at search algorithms is “Will my algorithm scale up from test problems to large real world problems?”. A solution that does not scale up to solve problems in the real world is of little practical use. Our experience has been that our algorithm has scaled up well from initial test data, to data for a whole department, and finally to data for a whole institution. However, the way the solution scales up is something that requires further investigation, in order that general rules can be learned.

Providing real data for scale-up experiments is difficult. The first problem is how to provide data sets of different sizes but similar nature. The second is how to measure the relative performance on data sets that have different ranges of problem measures. The extent to which the algorithm approaches the optimum might be an appropriate measure, but unfortunately, for real world data, the optimum is not known.

Table 2. Scale-Up Results

Number of Events	Elapsed Time (seconds)
74	47±5
155	270±109
307	591±205
587	926±149

The following experiment is not perfect but can give us at least an idea of how the algorithm scales up as the search space grows exponentially. A real data set was used and smaller subsets of this were produced. For each data set the room availability was adjusted so that 91% utilisation of rooms was required for each. Each of the data sets was given a target for each problem measure that was 80% of the approximated worst case value s. The time taken for all events to be placed and for P to reach 0.05 was then measured over 25 runs. The results and 95% confidence intervals are shown in Table 2 and Figure 4.

Because of the difficulty in constructing reliable experiments we are careful not to make strong claims about the results, but the results are very encouraging. Further experiments are required to confirm them. Some of the factors that may contribute to the favourable scale-up figures observed are discussed below:

Firstly, increasing the size of the problem increases the number of rooms available. The heuristic which initialises the ordering permutation ensures that difficult to place events are considered first, at least in the early stages of a run. This means that these events have a greater choice of accommodation in larger runs.

Secondly, timetabling problems partially partition into departments, programmes and levels. There is not a total partition since all partitions are connected by conflicting requirements for resources. However, the algorithm can still make good use of implicit parallelism to work on several partial partitions at once. In the test data (and most real world timetabling problems) larger problems have a larger number of partial partitions and so the degree of implicit parallelism employed increases as the problem grows.

Finally, Ross et al. have shown in [19] that phase transitions exist for evolutionary algorithms applied to timetabling problems. They showed that for some problems that were not completely partitioned, as the number of constraints increased the problem got harder to solve until a particular point when it started to get easier. This work is not completely relevant because it examined an evolutionary algorithm without local search and artificial rather than real data. It also examined increases in constraints rather than increases in events. It is possible however that phase transitions may be playing a part in the results observed. Further work is required in this area.

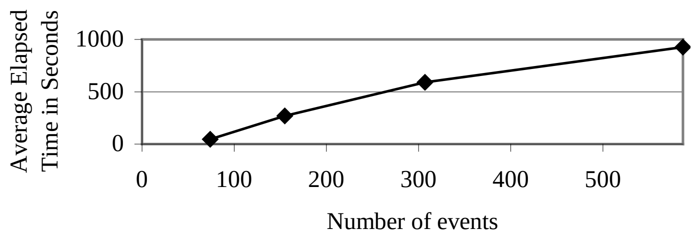

Fig. 4. Scale-Up Graph

7. Conclusions

We have described a new method for treating the optimisation of multi-objectives that fits with the way the user works, and we have described a user interface which facilitates it. We have shown that using back-up timeslots from the other parent can allow the search mechanism to repair "mistakes" made during recombination, and that this gives a significant improvement in results. We have defined two methods for dealing with optimisation relative to a reference timetable, and have shown that each of these gives a significant improvement. Finally we have shown that initial studies on the scalability of our approach to this problem are very encouraging.

References

1. Burke, E. K., Elliman D., and Weare, R., "A Genetic Algorithm for University Timetabling" AISB Workshop on Evolutionary Computing, Leeds, 1994.
2. Burke, E. K., Elliman, D. and Weare, R., "Specialised Recombinative Operators for Timetabling Problems", Proceedings of the AISB Workshop in Evolutionary Computing, Springer-Verlag Lecture Notes in Computer Science Series No 993, Heidleberg, 1995.
3. Burke, E. K., Newall, J. P. and Weare, R. F. "A Memetic Algorithm for University Exam Timetabling", Practice and Theory of Automated Timetabling, Burke and Ross Eds. Springer Verlag, 1996.

4. Caldeira, J. P., and Agostinho, C. R., “School Timetabling Using Genetic Search”, Practice and Theory of Automated Timetabling, Toronto, 1997.
5. Carter, M. and Laporte, G. “Recent Developments in Practical Course Timetabling”, Practice and Theory of Automated Timetabling, Toronto, 1997.
6. Colorni, A., Dorigo M., Maniezzo, V. “Genetic Algorithms and Highly Constrained Problems: The Time-Table Case”. Parallel Problem Solving from Nature I, Goos and Hartmanis (eds.) Springer-Verlag, Heidelberg, 1990.
7. Corne, D., Ross, P. and Fang, H., “Fast Practical Evolutionary Timetabling” Proceedings of the AISB Workshop on Evolutionary Computing, Springer-Verlag Lecture Notes in Computer Science Series No. 865, Heidelberg, 1994.
8. Corne, D., and Ogden, Rev. J., “Evolutionary Optimisation of Methodist Preaching Timetables”, Practice and Theory of Automated Timetabling, Toronto, 1997.
9. Eiben, A.E., Raue, P. E. and Ruttkay, Z. “Heuristic Genetic Algorithms for Constrained Problems”. Working Papers for the Dutch AI Conference, Twente, 1993.
10. Mamede, N. and Renta, T, “Repairing University Timetables Using Genetic Algorithms and Simulated Annealing”, Practice and Theory of Automated Timetabling, Toronto, 1997.
11. Paechter, B., Luchian, H., and Cumming, A., "An Evolutionary Approach to the General Timetable Problem", The Scientific Annals of the "Al. I. Cuza" University of Iasi, special issue for the ROSYCS symposium 1993.
12. Paechter B., Luchian H., Cumming A., and Petriuc M., “Two Solutions to the General Timetable Problem Using Evolutionary Methods”, The Proceedings of the IEEE Conference of Evolutionary Computation, 1994.
13. Paechter, B., Cumming, A., Luchian, H., ”The Use of Local Search Suggestion Lists for Improving the Solution of Timetable Problems with Evolutionary Algorithms.”, Proceedings of the AISB Workshop in Evolutionary Computing, Springer-Verlag Lecture Notes in Computer Science Series No 993, Heidleberg, 1995.
14. Paechter, B., Cumming, A., Norman, M., and Luchian, H., “Extensions to a Memetic Timetabling System”, Practice and Theory of Automated Timetabling, Burke and Ross Eds. Springer Verlag, 1996.
15. Paechter, B., Rankin, R. C. and Cumming A, “Improving a Lecture Timetabling System for University Wide Use”, Practice and Theory of Automated Timetabling, Toronto, 1997.
16. Radcliffe, N. J. “Forma Analysis and Random Respectful Recombination” Proceedings of the Fourth International Conference on Genetic Algorithms”, Morgan-Kaufmann, 1991.
17. Ross, P., Corne, D., and Fang, H., “Improving Evolutionary Timetabling with Delta Evaluation and Directed Mutation”, Parallel Problem Solving from Nature III, Springer-Verlag, Heidelberg, 1994.
18. Ross, P. and Corne, D. “Comparing Genetic Algorithms, Simulated Annealing, and Stochastic Hillclimbing on Several Real Timetable Problems”, Proceedings of the AISB Workshop in Evolutionary Computing, Springer-Verlag Lecture Notes in Computer Science Series No 993, Heidleberg, 1995.
19. Ross, P., Corne D. and Terashima H., “The Phase Transition Niche for Evolutionary Algorithms in Timetabling” Practice and Theory of Automated Timetabling, Burke and Ross Eds. Springer Verlag, 1996.