New Crossover Operators for Timetabling with Evolutionary Algorithms

Rhydian Lewis and Ben Paechter
Centre for Emergent Computing,
School of Computing, Napier University,
10 Colinton Rd, Edinburgh, EH10 5DT, Scotland, UK.
{r.lewis|b.paechter}@napier.ac.uk

Abstract: *When using an evolutionary algorithm (EA) to optimise a population of feasible course timetables, it is important that the mutation and crossover operators are designed in such a way so that they don't produce unfeasible or illegal offspring. In this paper we present some specialised, problem specific genetic operators that enable us to do this successfully, and use these in conjunction with a steady state EA to solve the University Course Timetabling Problem (UCTP)[1]. We introduce a number of different crossover operators, each of which attempts to identify useful building blocks within the timetables, and investigate whether these can be successfully propagated through the population to encourage the production of high quality solutions. We test the effectiveness of these crossover operators on twenty well-known problem instances and present the results found. Whilst the results are not state-of-the-art, we make some interesting observations on the nature of the various crossover operators and the effects that they have on the evolution of the population as a whole.*

Keywords: *Genetic Algorithms, Evolutionary Computation, Timetabling, Scheduling, Optimisation.*

1. Introduction

Timetabling is NP-hard in almost all variants and is a problem that can take many forms such as exam [3, 8], school [1, 5] and university [2, 4, 6, 7, 10]. Due to various quirks and characteristics of these three types we tend to view them as separate problems. Consequently they are approached and solved in different ways. As well as this, the process of timetabling may also be classified as a *search problem* and/or an *optimisation* problem – the former is concerned with finding a timetable that satisfies all its constraints [1, 9], the latter with forming a timetable that satisfies its hard constraints and minimises an objective function that embeds a set of soft constraints [2, 4, 5, 6, 7, 10].

The International Timetabling Competition organised by the Metaheuristic Network in 2002-3 presented a simplified (but still NP-hard) version of a UCTP along with 20 input instances. A competition was then run to see who could produce the best timetables over all the inputs using just one algorithm. Various metaheuristc and hybrid approaches were presented, some of which can be found in [2] and [4]. Two interesting points about the results of this competition were (1) the best algorithms all used a two-stage approach where feasibility of the timetable was first obtained and then optimality was sought (using various search methods), whilst *always* remaining in feasible search space, and (2) none of the entries to the competition made use of an EA[2]. We therefore propose an EA that starts with a population of feasible timetables and then evolves, whilst always remaining in the feasible search space. There already exist some EAs that deal only with the search space of feasible timetables, most notably by Burke and Weare *et al.* [3, 8]. However their methods are slightly different in that they actually leave unspecified the number of timeslots in a timetable, and instead incorporate this into the objective function (with the aim of reducing it). This research is also directed towards examination timetabling which, as mentioned earlier, is generally considered to be a different problem. Enzhe *et al.* [6] have also presented a two stage EA for a problem similar to UCTP and introduce the idea of *sector-based* crossover, something that we will look at in this paper. However the researchers in this case used their own input and failed to provide comparisons with other works in order to gauge how effective their EA actually was.

---

[1] As defined for the International Timetabling Competition organised by the Metaheuristic Network in 2002-3 – see the competition website at - http://www.idsia.ch/Files/ttcomp2002/

[2] An EA was presented for this problem before the competition [10], which attempted to solve hard and soft constraints at the same time by applying weightings to hard constraints, but it was not particularly successful.

2. Problem Definition.

The input to the problem comprises a set of rooms $R = \{r_1,\ldots,r_m\}$, each with an associated size that represents the maximum number of students it can hold at any one time. We have a set of timeslots $T = \{t_1,\ldots,t_{45}\}$, representing five days of nine slots, and a set of events $E = \{e_1,\ldots,e_n\}$. We are also given a set of students $S = \{s_1,\ldots,s_p\}$, where for each event $e_i \in E$ we have a set $S_i \subseteq S$ which represents the students that will attend e_i. Finally we have a set of features $F=\{f_1,\ldots,f_q\}$. Each room $r_k \in R$ *possesses* a set $F_k \subseteq F$ of these features, whilst each event e_i *requires* a subset $F_i \subseteq F$ of these features. The objective is to assign every event a room and timeslot such that there are no hard-constraint violations and minimal soft-constraint violations. The hard constraints are (1) No student should attend more than one event in any one timeslot, (2) each event should be assigned a room that is large enough to hold all of the attending students, (3) each event should be assigned a room that has *at least* the features it requires and (4) only one event should be scheduled in a room at any one time. In this paper we will say that two events *conflict* if they have one or more common students. Only if all events are scheduled and no hard-constraints are violated is the timetable considered feasible.

The soft-constraints are (1) no student has a class in the last timeslot of the day (the *last-slot* constraint), (2) no student has more than two classes in a row (the *long-intensive* constraint) and (3) no student has just one lesson in a day (the *single-event* constraint). As we only consider feasible timetables in this approach, we refer to the total number of soft-constraint violations as our objective function. The way this value is calculated can be found on the competition web page.

3. The Evolutionary Approach to the Problem

3.1 Chromosome representation.

For our EA we use a direct representation and employ a genetic repair function. An assignment of events to rooms and timeslots is represented as a matrix $X_{m\times 45}$ such that rows represent rooms and columns represent timeslots [4]. In the context of the EA, we refer to each cell as a gene, and an entire matrix as a chromosome. A gene *x(y, z)* in *X* can be blank, or can contain at most one event $e \in E$. If gene *x(y, z)* contains *e*, then event *e* is to be scheduled in room *y* and timeslot *z*; if *x(y, z)* is blank, then room *y* is unoccupied during timeslot *z*. We note that this particular representation disallows the possibility of assigning more than one event to a room in any timeslot allowing us to disregard the 4th hard constraint. Additionally, hard constraints 2 and 3 can be easily avoided by ensuring events are placed on appropriate rows in the matrix. Violations of the remaining constraint are detected by examining the individual columns of the matrix.

3.2 Forming an initial Population.

To construct an initial population of feasible timetables we use a constructive algorithm similar to the method outlined in [2]. To start, we construct a list *L* of all the events in *E*, with each event also owning a list specifying all the *places* to which it can be assigned. The complete set of places *P* is the Cartesian product of the timeslot set and the room set $P = R \times T$. Thus an event $e_i \in E$ will have an attached *place list* $K_i \subseteq P$ that holds all the places where we may feasibly put e_i. Obviously at the outset, we will not yet have assigned any events to places, so each K_i will simply say that e_i can be placed into all suitable rooms in all time slots. To form one feasible timetable we do the following:

1. Determine the event e_i in *L* that has the shortest list K_i. If there is more than one of minimal size, choose between these randomly.
2. Choose a place for this event by doing the following. For each $k \in K_i$, calculate:
   - The total number of events left in *L* that may feasibly be assigned to *k* (i.e. that have *k* in their place list).
   - The number of events currently scheduled in the same timeslot as *k*,
3. Choose the $k \in K_i$ that minimises the total of these two values. If there is more than one minimum choose between these randomly.
4. Update the data structures by inserting e_i into the timetable at the chosen place and removing all instances of this place *k* from the other place lists. Also remove any node from any other event place list that defines a place in the same timeslot as our chosen place whose event conflicts with e_i. Finally, remove e_i from *L*.

5. If L is not empty, we iterate back to step 1 otherwise we have a feasible solution.

If at some point during this process we encounter an event e_i that has an empty place list K_i we consider e_i as *unassignable*. In this case we start the whole process again with a different random seed. For the competition problem instances however, this is rarely necessary.

3.3 Crossover Operators.

The purpose of crossover is to encourage the combining of useful groups of genes (*building-blocks*) into new and hopefully fitter offspring than the preceding generations. However, when considering feasible timetables, it is unclear where these building blocks lie and indeed, what actually constitutes a good one. To investigate this we propose four different methods of crossover, all of which utilise a similar method of 'genetic repair' in order to ensure the feasibility of the resultant offspring. We then describe how we go about transferring these genes in section 3.3.5, the resultant genetic repair operator in section 3.3.6 and the mutation operator in section 3.3.7. For ease of reading, we explain how to make *child1* from *parent1* and *parent2*. To make *child2* we simply reverse the roles of the two parents.

3.3.1. Sector-Based Crossover. The first crossover operator explores the idea that there will be '*sectors*' of a chromosome that will have a strong sub-fitness [6]. That is, areas of the timetable that may have few soft-constraint violations. We start by making an exact copy of *parent1* called *child1*. We then choose randomly a sector in *parent2* and try to insert each gene within the sector of *parent2* into the same gene position in *child1*. Fig. 1(*i*) shows the basic idea. We allow *wraparound* of the sector to allow greater flexibility and to avoid showing unnecessary bias to genes in the centre of the chromosome - see fig 1(*iii*).

3.3.2. Day-Based Crossover. Day-based crossover is a limited version of sector-based crossover based on the observation that none of the three soft-constraints carry across different days - all the events in a day may be considered a sub-timetable with all its soft-constraints contained within it. It might therefore be useful to exploit this fact by trying to insert these sub-timetables into other timetables to try and promote the good ones in the population. Thus we limit the form that a sector can take so that it can only contain entire days' assignments.

3.3.3. **Student-Based Crossover.** For student-based crossover we calculate all the events that a randomly chosen student s is scheduled to take (that is, calculate a set $E' \subseteq E$). We now go through *parent2* and try to insert each event $e \in E'$ into *child1* in the position it holds in *parent2*. See Fig 1(*ii*). We consider the *personal-timetable* of student s to be the timetable that describes only the events that s attends. Some students will have better personal-timetables than others and so it is reasonable to see if these can be propagated through the population during evolution.

3.3.4. Conflicts-Based Crossover. Our final operator follows a similar idea to student-based crossover. This time however we choose an event e at random and calculate the list $E' \subseteq E$ that represents all the events that *conflict* with e. So, similar to student-based crossover, we are identifying collections of genes that are related due to common students, and attempting to promote the good ones in the population. The competition problem instances, just like in real world timetabling situations have events that conflict due to *groups* of students taking both events. Thus we hope to be able to identify building blocks that involve multiple students and therefore arm ourselves with the potential to alter the objective function to a larger degree.

3.3.5. Inserting Genes into child1. To start, *child1* will be an exact copy of *parent1*. The recombination operator of choice will define a set of genes in the *parent2* that we wish to insert into *child1* and the positions we wish to put them. We now attempt to insert these whilst always maintaining feasibility. Going through each gene in turn, there are four things that can happen.

Let $e, g \in E$ such that $e \neq g$ and let (x, y) define the gene's coordinates.

1. *Parent2(x, y) = child1(x, y)*. In this case we do nothing.
2. *Parent2(x, y)* is blank, *child1(x, y)* contains *e*. In this case we make *child1(x, y)* blank, and attempt to find a new place for *e* using the genetic repair function (see section 3.3.6).
3. *Parent2(x, y)* contains *e*, and *child1(x, y)* is blank. In this case we first delete the occurrence of *e* in *child1*. Next, we insert *e* into *child1(x, y)*. If this maintains feasibility of column *y* then we accept the move and end. Otherwise we reset the change and we consider this particular insertion as having failed.

4. *Parent2(x, y)* contains *e*, and *child1(x, y)* contains *g*. In this case we first delete the occurrence of *e* in *child1*. Next, we insert *e* into *child1(x, y).* If this makes column *y* infeasible then we reset the change and end. Otherwise we need to find a new place for *g* using the genetic repair operator (section 3.3.6).

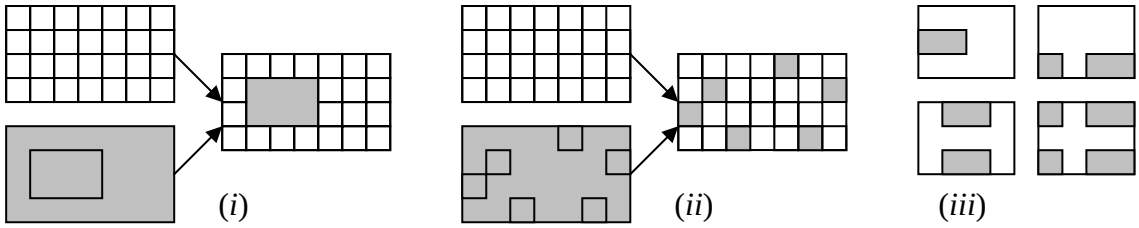

Fig. 1. (*i*) Sector-based crossover, (*ii*) Student and Conflicts-based crossover (*iii*) Varying ways a sector can wraparound.

3.3.6. The Genetic Repair Function. In our methods of crossover we often have a situation during gene transfer where we have an unplaced event *e*. The genetic repair function is responsible for finding a new place for *e* that preserves feasibility. Let the coordinates *(x, y)* indicate the original position of *e*. We find a new position for *e* by searching horizontally along row *x* looking for a blank gene in which to insert *e* without making the timetable infeasible. Note that we already know the room defined by row *x* to be feasible for *e*, so once again we only need to check for violation of hard-constraint 1. If we cannot find a position for *e* in row *x*, we can go on to check any other rows (if there are any), which define other feasible rooms for *e*. We continue this process either until a new place for *e* is found, or until all free feasible places have been tried and we still cannot find anywhere suitable. In the latter case, the process has failed - we reset the timetable to its previous state and move on.

Although seemingly straightforward, this method does invoke one additional problem that needs addressing. Generally a good timetable, according to soft-constraint 1 should not have many (if any!) events scheduled in the last-slots of each day. Using our form of repair, there will be a tendency to want to put *e* into these last-slots. The reasons for this are twofold. (1) The columns which define the last-slots in a good timetable will contain a higher than usual number of blank genes and (2) because there *are* less events in these columns, there is a smaller probability that there will be an event here that will clash with *e*. To produce good offspring by crossover, we really want to show prejudice towards placing events in these columns. We therefore introduce the parameter *ls* (where $0 \leq ls < 1$) that we keep at a relatively low value which defines the probability of us allowing an event to be put in one of these penalised slots each time our repair function encounters one

3.3.7. Mutation Operator. The mutation operator selects two genes *x(a, b)* and *x(c, d)* in the chromosome at random such that $x(a, b) \neq x(c, d)$, and swaps them. We note that mutations may result in violations of hard-constraints 1, 2 and/or 3. If this happens, we reset the swap, and select another two genes, repeating this process until we find a swap that maintains feasibility. As we run the risk of an infinite-loop here, we limit the number of attempted swaps to 1000 (although in practice, with the instances used this was never reached).

4. Experiments and Results

Experiments were carried out to compare the effectiveness of the various crossover operators. The 20 problem instances for the competition were used. In all cases we used population sizes of 100, binary tournament selection with a selection bias of 0.75, and a crossover rate of 0.75. A steady state population was used with an elitist replacement strategy whereby an offspring always replaced the weakest individual in the population. The mutation rate was controlled by two parameters – *nm* and *mr*. Every time we produce an offspring, a loop is performed *nm* times, and in each loop we choose whether to mutate the chromosome or not with a probability *mr*. The parameter *nm* was started at 2 and was increased by 1 every *n* generations (where *n* is the number of events). The parameter *mr* was kept constant throughout at 0.75. Finally, *ls* was set to 0.05. Two sets of control experiments were also performed - the first used no crossover at all (i.e. the population was evolved using mutation reproduction and selection), the second just selected some genes randomly in each of the parents and

transferred them in the usual manner[3]. We call these *mut* and *rand* respectively. Although we spent some time experimenting with various parameter settings, the values used in these experiments should not be assumed optimal in any case.

5. Conclusions and Discussion

#	Init Pop %	Sec. Gens.	Sec. Sco.	Day Gens.	Day Sco.	Stu. Gens.	Stu. Sco.	Conf. Gens.	Conf. Sco.	Mut. Gens.	Mut. Sco.	Rand. Gens.	Rand. Sco.
1	2.4	2746	307	2737	320	2570	293	2225	**288**	2715	318	2209	295
2	2.9	2595	**260**	2589	271	2414	282	2091	266	2571	264	2075	269
3	3.4	2354	**322**	2332	377	2180	339	1869	330	2316	327	1873	327
4	4.2	962	688	956	741	931	692	851	**679**	950	710	859	689
5	2.8	1905	561	1853	571	1772	565	1543	**557**	1844	573	1530	583
6	5.5	954	549	949	637	904	548	814	**532**	936	614	817	537
7	6.1	404	468	537	507	469	456	410	**430**	408	513	374	445
8	4.2	1400	319	1259	365	1084	310	577	**305**	1259	376	415	328
9	4.2	1562	330	1569	373	1487	301	1332	**283**	1556	371	1325	300
10	3.3	2218	332	2217	323	2095	319	1823	316	2196	336	1811	**311**
11	2.9	2260	340	2251	346	2117	329	1846	330	2211	347	1846	**328**
12	2.6	2491	379	2491	370	2338	366	2053	373	2476	385	2040	**350**
13	3.2	1734	475	1727	510	1644	464	1484	**420**	1721	507	1469	433
14	5.9	606	479	612	511	590	506	458	**469**	610	503	542	486
15	3.5	1252	421	842	446	759	412	659	**400**	681	448	574	419
16	5.2	1222	321	1218	324	1155	309	1031	**302**	1211	329	1028	308
17	2.2	1030	**521**	2015	563	1902	545	1669	548	2003	539	1672	542
18	2.4	2772	260	2762	282	2588	255	2247	256	2743	270	2228	**254**
19	6.6	541	596	543	590	527	569	490	**550**	457	584	488	566
20	5.7	765	447	687	505	292	481	277	**424**	578	524	596	442
Total Score		**8375**		**8932**		**8341**		**8058**		**8838**		**8212**	

Table 1: Comparison of the different crossovers and control groups with the 20 competition input instances.

Table 1 displays the results of our experiments for each crossover operator (sector, day, student, conflict, mut. and rand. respectively) and for each problem instance. Ten trials were carried out and we present the score (Sco.) of the best timetable found in each run, averaged over the ten trials. We also present the average number of generations the population was able to evolve for within the competition time limit (Gens.) and the percentage of the total time that it took to produce the initial population (Init Pop %). **Fig. 2** shows a typical run of the EA and the different effects that the crossover operators had upon the objective function within the time limit. It can be seen here that some of the crossover operators allow more generations to be produced within the time limit than others.

The first thing to notice from **table 1** is that conflicts-based crossover (conf.) seems to be on the whole, the more effective crossover method. This is reflected in the number of instances where it produced the best result, and also in the total score for the 20 instances. However this is also the operator that tends to transfer the largest number of genes per crossover so there is the possibility that this recombination operator might just be *chancing* upon good results due to it taking larger steps in the search space than the others (as opposed to propagating useful building bocks). However, this theory is contested by the corresponding results of the second control group (*rand*), which in most cases is outperformed. An interesting observation is that when conflicts-based crossover *is* outperformed it is always with a problem instance where the solution has been able to evolve for a large number of generations within the time limit, and by sector-based crossover or *rand*. We suggest that this is due to the fact that these two operators generally offer more flexibility in what genes can be selected for crossover, and consequently manage to avoid population convergence for longer. This hypothesis is also backed up by the fact that day-based crossover, a considerably *inflexible* operator, performs very badly in most cases. In conclusion then, the selection and propagation of appropriate building blocks does allow us to move into good areas of the search space in less generations (as demonstrated by student and conflicts-based

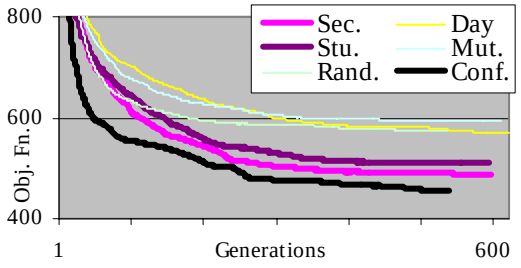

Fig 2. Objective Function Vs Generations for various Crossovers on Instance 19.

[3] As we will see, conflicts-based crossover was usually the best, but this might've simply been occurring because more genes on average were being transferred during crossover than the other crossover operators. To investigate this phenomenon we kept the number of genes selected for crossover with the second control (*rand*) similar to the number of genes selected, on average, with conflicts crossover.

crossover) but it also imposes a certain inflexibility as to what genes we can select, which may cause earlier convergence in some cases (as indicated when the sector or random-based crossover produced the best result). A possible downfall of this approach is that *complete* building blocks of genes can sometimes not be fully injected into offspring due to the failure of the repair operator from time to time. Similarly, the repair operator will also sometimes need to introduce foreign genes in order to keep a timetable complete, which can also have an effect of disrupting other building blocks.

Although these results are not state-of-the-art, we have still made useful observations on the nature of the different crossover operators. With careful consideration of what constitutes a good building block, and by the use of appropriate operators that preserve feasibility we can see that good progress can be made through the search space. We also believe that these results could be substantially improved by the introduction of local search at various points [4, 7] and/or the use some sort of *smart-mutation* (e.g. [9]), although in order not to cloud any of the conclusions that we have drawn here in regards to the various crossover operators, we leave this for later experiments.

Finally, as with the methods described in [3, 6, 8], the approach outlined here depends very much on being able to produce large numbers of diverse, feasible timetables in reasonable time, and to consistently preserve this feasibility during crossover and mutation. The competition problem instances allow us to achieve this. In reality there will of course be instances that are so heavily constrained that this may not be possible. It is envisaged that harder instances would also cause problems with the genetic-repair function because the rate at which the process fails would increase, perhaps to a point where we were stopped from doing anything particularly useful.

References

[1] D. Abramson & J. Abela (1991). A Parallel Genetic Algorithm for Solving the School Timetabling Problem. *Technical report, Division of Information Technology*, C.S.I.R.O, 1991.

[2] H. Arntzen & A. Løkketangen (2003). A tabu search heuristic for a university-timetabling problem. *MIC 2003, Fifth Metaheuristics International Conference* pp02-1, 02-7.

[3] E. Burke, D. G. Elliman, and R. F. Weare (1995). Specialised recombinative operators for timetabling problems. In T. C. Fogarty (ed), *AISB Workshop on Evolutionary Computing*. LNCS 993, pp. 75-85. Springer-Verlag, Berlin, 1995.

[4] M. Chiarandini, K. Socha, M. Birattari, and O. Rossi-Doria (2003). An effective hybrid approach for the university course timetabling problem. Technical Report AIDA-2003-05, FG Intellektik, FB Informatik, TU Darmstadt, Germany, 2003.

[5] A. Colorni, M. Dorigo and V. Maniezzo (1992). A genetic algorithm to solve the timetable problem. Technical report. 90-060 revised, Politecnico di Milano, Italy 1992.

[6] Yu Enzhe & Ki-Seok Sung (2002). A Genetic Algorithm for a University Weekly Courses Timetabling Problem. *International transactions in Operational Research 9* pp 703 – 717, 2002.

[7] B. Paecher, R. C. Rankin, A. Cumming, T. C. Fogarty. Timetabling the Classes of an Entire University with an Evolutionary Algorithm (1998). T. Beck, M. Schoenauer (eds.), *Parallel Problem Solving from Nature PPSN V*. Springer-Verlag, Berlin, 1998.

[8] R. Weare, E. Burke and D. Elliman (1995). A Hybrid Genetic Algorithm for Highly Constrained Timetabling Problems, in Eshelman (ed) *Proceedings of the Sixth International Conference on Genetic Algorithms*, pp. 605-610, Pittsburg, Morgan Kaufmann, 1995.

[9] Peter Ross, Dave Corne, and Hsiao-Lan Fang. (1994). Improving evolutionary timetabling with delta evaluation and directed mutation. In Davidor, Schwefel, and Manner (eds), *Parallel Problem Solving in Nature, PPSN III*. Springer-Verlag, Berlin, 1994.

[10] Rossi-Doria, O., Sampels, M., Birattari, M., Chiarandini, M., Dorigo, M., Gambardella, L.M., Knowles, J., Manfrin, M., Mastrolilli, M., Paechter, B., Paquete, L., Stützle T. (2002) A comparison of the performance of different metaheuristics on the timetabling problem. In Burke and De Causmaecker (eds) *The Practice and Theory of Automated Timetabling IV: 4th International Conference, PATAT 2002*. LNCS 2740. pp. 329-351. Springer-Verlag, Gent 2004.