An Empirical Analysis of the Grouping Genetic Algorithm: The Timetabling Case.

Rhydian Lewis and Ben Paechter
Centre for Emergent Computing,
Napier University,
Edinburgh, Scotland, EH10 4DQ.
{r.lewis|b.paechter}@napier.ac.uk

Abstract- A grouping genetic algorithm (GGA) for the university course timetabling problem is outlined. We propose six different fitness functions, all sharing the same common goal, and look at the effects that these can have on the algorithm with respect to both solution quality and time requirements. We also propose an additional, stochastic local-search operator and discover that this too can have large positive and negative effects on the runs. As a by-product of these studies, we introduce a method for measuring population diversity with the GGA model and note that diversity seems to have huge consequences on the cost implications of the algorithm. We also witness that the algorithm can behave quite differently with varying sized instances, introducing scaling-up issues that could, quite possibly, apply to grouping genetic algorithms as a whole.

1 Introduction

The NP-hard problem of university course timetabling involves assigning resources, such as rooms, to the events of a university. Usually, timetabling problems in the literature consist of two sub-problems:

- The production of *feasible* timetables,
- The production of *nice* timetables.

Generally speaking, feasible timetables are ones in which all the events have been assigned the resources they require and which do not ask the impossible of anyone or anything by, say, putting multiple events into the same room at the same time, or asking a student to be in two places at once. A *nice* timetable, on the other hand, is usually (although, it must be noted, not always - see [11]) one that is both feasible and also eases, as much as possible, the burdens of the people who are to be using it. As can be imagined, real-world timetabling problems can vary a great deal from institution to institution and while this has resulted in a rich variety of works addressing different versions of the problem, it also makes it tricky to compare different algorithms in order to identify their individual benefits and drawbacks.

The problem version that we choose to address here has already been widely studied and seems a good benchmark in this field. It was originally formulated for the Metaheuristics Network [16] and was also used for the International Timetabling Competition, run in 2002 [15]. Specifically, the problem involves the assignment of all events to rooms and timeslots in the timetable and, in order to be feasible, these assignments must meet three criteria (the so-called hard constraints). These are:

- Only one event is put in any room in any timeslot,
- Rooms that events are assigned to must be big enough to hold the attending students and have the facilities that the event requires.
- Students do not attend more than one event in any one timeslot.

There are also some (soft) constraints that specify what it is to be a *nice* timetable. These include such things as avoiding students having to attend three or more events in successive timeslots, and stopping students from having only one event in any day.

There have been various algorithms proposed in the literature for this specific problem, most commonly using benchmark instances [15, 17]. Although each has their own strengths and weaknesses, the trend would seem to be that the algorithms that address both sub-problems simultaneously (e.g. through the use of weightings in the evaluation function [12]) are generally outperformed by those that employ a two-stage strategy whereby feasibility is first obtained and then soft constraints are optimised using operators that restrict the search to feasible areas of the search space [1, 2, 6]. However, whilst there are plenty of works proposing algorithms specialising in soft constraint optimisation, there has been less emphasis on producing algorithms that specialise in finding feasibility in the first place. In particular, when considering "harder" problem instances, some of the existing algorithms in the literature could start to fail on this matter. This therefore generates a need for new algorithms that provide more robust searches with regards to finding feasibility.

1.1 Timetabling and Grouping Genetic Algorithms

In [7], Lewis and Paechter designed a *grouping genetic algorithm* (GGA) for constructing feasible timetables. The foundation of this work was based upon the observation that the probelm of finding feasibility in this case, in common with problems such as bin-packing, graph-colouring and bin-balancing, is an example of a so-called *grouping problem* [4]. Grouping problems may be thought of as those where the aim is to arrange a collection of *items* into a number of *groups*, subject to some problem-

specific constraints that define valid and legal groupings. In particular, it is a well-held belief that when using any sort of evolutionary algorithm to address these problems, it is the groups of items themselves (i.e. items in a particular bin, nodes of a particular colour etc.) that represent the underlying building blocks and not the particular states of any of the items individually. Hence representations and genetic operators that allow these groups to be propagated are to be encouraged.

With regards to our timetabling problem, the events are the items and the timeslots represent the groups. Thus, in order for a timetable to be feasible, all events need to be arranged into a predefined number of timeslots such that, (1) no events in a timeslot have any common students (i.e. each timeslot obeys the third hard-constraint), and (2) every event in a timeslot can be assigned to their own room that has the facilities and the seating capacity that the event requires (therefore satisfying the first and second hard-constraints).

Note also that it is usual in many GGAs (e.g. [3]) to consider a particular chromosome as an unordered set of groups - that is, two solutions that define the same groupings will define the same solution, regardless of the group's orderings in the chromosome. However, while this characteristic is also true for our timetabling problem with respect to the hard constraints, it is not so for the soft constraints, making this approach unsuitable for the secondary task of soft constraint optimisation.

Here, as in [7], each timetable is represented by a two dimensional matrix where rows represent rooms and columns represent timeslots. Each timetable of the initial population is built using stochastic, constructive heuristics that attempt to assign all events to a feasible place in the timetable (i.e. cell in the matrix). When this is not possible because there is no feasible place for a particular event, extra timeslots are opened accordingly (extra columns are added to the matrix). The underlying objective of the algorithm is to therefore reduce the number of timeslots being used down to the target amount. GGA operators that follow the basic framework suggested by Falkenauer [4] are also used. The recombination operator works by injecting some timeslots from a timetable p_1 into another timetable p_2, removing timeslots (and the events assigned to these timeslots) from the old part of p_2 that cause duplicates, and then using similar constructive heuristics to insert the resulting unplaced events (see figure 1 for two examples). The mutation operator simply removes a small number of randomly selected timeslots from a timetable and reinserts the events via similar constructive heuristics.

In [7], through the use of these GGA operators and powerful constructive heuristics, the algorithm showed to be very successful – it was able to find feasibility immediately on the benchmark instances used for the International Timetabling Competition [15] (although it must be said that these instances were chosen with soft constraints in mind) and was also very successful at tackling some "harder" instances which were created, and are available for download [17]. In simple tests it was also observed that the recombination operator seemed to add a great deal of power to the search with respect to solution quality, but it was also more computationally expensive than the mutation operator. This presents users with a trade off – if time constraints are particularly tight then recombination should only be used in limited amounts. However, in doing so the algorithm is more susceptible to getting stuck in local optima. Indeed, when recombination was used abundantly and the algorithm given enough time, superior results were nearly always found [7].

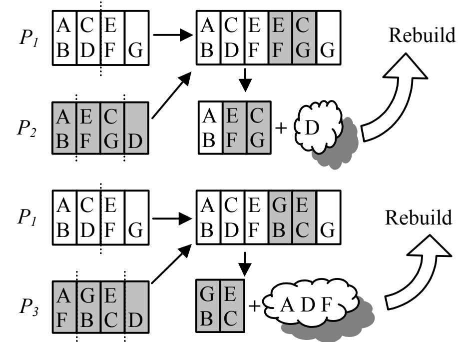

Figure 1. Two examples of recombination showing (1) selection of three crossover points, (2) injection, and (3) elimination of timeslots containing duplicates.

In this paper we conduct a further analysis into this algorithm. We test a number of different fitness functions and look at what effects these can have on solution quality and execution time. Secondly, we also look at the effects that a simple stochastic local-search operator (embedded into the genetic operators) can have on the algorithm.

The remainder of the paper is set out as follows: In section 2 we look at the issue of measuring population diversity with the standard GGA representation and make some remarks on the effects that it can have with the recombination operator. In section 3 we go on to define six fitness functions and analyse their effects on the algorithm's behaviour. Next, in section 4 we introduce the local-search operator. Section 5 concludes the paper.

2 Population Diversities and Recombination

The GGA representation used here admits two important properties: chromosomes are variable in length, and the ordering of groups within the chromosome is irrelevant (with regards to the solutions that they represent). This renders the more traditional methods of calculating population diversity (such as using Hamming distances [9]) as inappropriate.

A suitable measure however seems to be the *substring-count* method of Mattiussi *et al*. [8], which specialises in measuring diversity in populations where individuals are subject to major reorganisations during the evolutionary process. For our, purposes we calculate the diversity of a population P, via the formula:

$$div(P) = \rho\left(\frac{m}{n}\right) \quad (1)$$

where ρ is the population size, m represents the number of *different* groups in the population, and n represents the *total* number of groups in the population. Using this measure, a homogenous population therefore has a diversity of 1.0, and a population of distinct individuals (that is, a population where none of the individuals contain an equivalent grouping of items), will have a diversity of ρ.

Using these ideas, we can also measure the *distance* between two individuals, p_1 and p_2, via the formula:

$$dist(p_1, p_2) = 2\left(\frac{x}{y}\right) - 1 \quad (2)$$

where x is the number of different groups in p_1 and p_2, and y is the total number of groups in p_1 and p_2. Thus, two homogenous individuals will have a distance of zero and two maximally distinct individuals will have a distance of one.

With regards to recombination, it was noticed early on in experiments that the operator tended to take longer at the beginning of the run, and then gradually sped up as the search progressed. Investigations have revealed that when the population is diverse (and hence the average distance between pairs of individuals is large), more events, on average, seem to become unplaced during recombination, therefore increasing the amount of rebuilding needed to be done. In contrast, when the population is nearing convergence, little, if any rebuilding is usually required. We can see this by looking back at figure 1. Here, $dist(p_1, p_2) = 0.5$ and only one event (event D) becomes unplaced. On the other hand, p_1 and p_3 are maximally distinct (that is, $dist(p_1, p_3) = 1.0$) and three of the seven events (events A, D and F) become unplaced. This example, although small, reveals the general pattern displayed by the algorithm and indeed, this correlation is displayed well in figure 2.

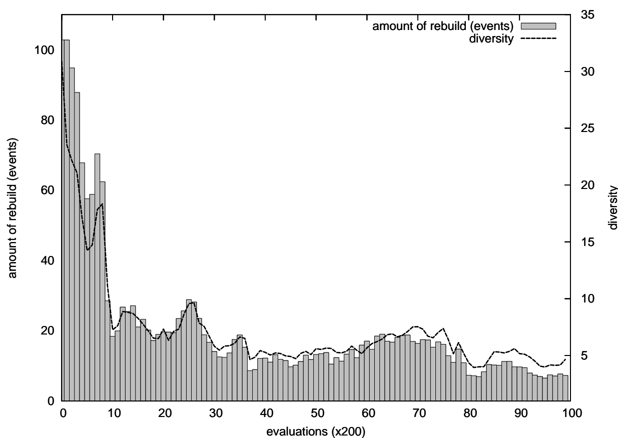

Figure 2. Example run with a medium sized instance (410 events) showing (1) how diversity can change during the evolutionary process (2) the close relationship between this and the amount of rebuilding needed to be done (on average) with the recombination operator. (With a population size 50.)

Here, as a population of timetables is evolved during a run we see, fairly typically, that diversity falls. However, we also see that the average number of events becoming unplaced during recombination (and therefore having to be dealt with by the rebuilding process) mirrors this fall. This correlation signifies an extra issue of algorithmic complexity that we examine further in the following sections.

3 The Effects of Various Fitness Functions

An important aspect of any evolutionary algorithm is the way in which candidate solutions in the population are evaluated. Ideally, a good fitness function should convey meaningful information about a solution and should also encourage the search into promising parts of the search space. For many problems in operational research, a suitable fitness function is suggested naturally by the problem at hand (e.g. the travelling salesman problem [13]). In others it is not so easy. For example, Falkenauer [4] suggests that for the bin-packing problem, the most obvious way of measuring solution fitness is to measure the number of bins being used, with the aim of minimisation. As is noted, from a mathematical point of view this is correct, but in practice it is unsuitable because it leads to a seriously unfriendly search landscape where “a very small number of optimal points in the space are lost in the exponential number of points where this purported cost is just one unit above the optimum. Worse, these slightly sub-optimal points [all] yield the same cost”.

Similar observations can be made with this timetabling problem: the ultimate aim is to arrange all of the events into an acceptable number of feasible timeslots (in our case forty-five). But likewise, we believe that using the current number of timeslots in a particular timetable for a fitness function would be a mistake. From a second (and perhaps more useful) perspective, we might also identify the algorithm’s aim to be to reduce the number of unplaced events to zero (as used, for example, by Paechter *et al.* [11]). In our algorithm, because all events are assigned to timeslots, this figure needs to reflect the minimum number of events that would need to be removed in order to bring the number of timeslots down to the target amount. It can therefore be calculated by working out the number of extra timeslots t' being used in a timetable, and then identifying the t' timeslots with the least events in them, and totalling them up.

In this paper, we will use this latter measure to express a timetables *distance to feasibility*. However, we note that this distance measure might also present an unfriendly search landscape. Consequently we designed and tested four further fitness functions for the evolutionary process.

3.1 Fitness functions for Timetabling

Let t represent the total number of timeslots currently being used in a timetable, t' represent the number of *extra* timeslots being used (i.e. in our case $t'=t-45$), and let r represent the number of rooms available per timeslot. Also, let E_i represent the total number of events currently

assigned to timeslot i, and D_i represent the total *degree* of timeslot i (that is, for each event in timeslot i we calculate the total number of other events in the entire event set that this event clashes with, and D_i is the total of all these values). Additionally, let S_i represent the total number of students attending an event in timeslot i. Finally, let d represent a timetable's distance to feasibility (calculated as described in section 3). The fitness functions we use are as follows. For simplicity's sake, all are made to be maximisation functions:

$$f_1 = \frac{1}{1+t} \quad (3) \qquad f_2 = \frac{1}{1+d} \quad (4)$$

$$f_3 = \frac{1}{1+d+t'} \quad (5) \qquad f_4 = \frac{\sum_{i=1}^{t} (E_i/r)^2}{t} \quad (6)$$

$$f_5 = \frac{\sum_{i=1}^{t} (D_i)^2}{t} \quad (7) \qquad f_6 = \frac{\sum_{i=1}^{t} (S_i)^2}{t} \quad (8)$$

Thus f_1 and f_2 represent the more *obvious* fitness functions already described, while f_3 attempts to combine facets of both – trivially, if two timetables have the same number of extra timeslots, then the one with the least events in these extra ones is probably better (a similar fitness function was used for graph colouring in [5]). A careful examination of fitness functions $f_{4 \ldots 6}$, on the other hand, reveals that what they are attempting to do is to place more emphasis on promoting timetables that contain good *packings* of events into timeslots. They merely differ in the interpretation of what such a packing might be. Consequently, f_4 tries to encourage timetables that have timeslots with high numbers of events in them (similar to the fitness function for bin-packing [4]), and f_5 (as suggested by Erben [3]) prefers timetables that contain timeslots with a high total degrees. Finally f_6 favours timetables that contain timeslots with large numbers of students attending some event in them.

Note also that functions f_2 and f_3 need to know in advance the number of timeslots needed for a timetable to be feasible. If this is undefined, the task of calculating the minimum number of timeslots needed for a given instance is analogous to the problem of calculating the chromatic number in graph colouring, which is NP-complete. In the test instances we used, we have the benefit of knowledge that all instances admit at least one solution using forty-five timeslots. In certain real-world applications however this might be unknown beforehand.

3.2 Experiments and Observations

To investigate the effects of the six fitness functions we designed simple tests where all algorithm parameters were kept the same, identical initial populations for each instance were used, and only the fitness functions were altered. We used a steady state algorithm with population size 50, binary tournament selection, elitist replacement, a mutation rate of 3 and recombination rate 1.0. Note then, that the only *actual* difference between each trial is the criteria used for (1) choosing tournament winners and (2) picking the individuals to replace. Note also that the costs of the scoring functions are roughly equivalent, as all require just one parse of the timetable. The test instances used comprise three sets of twenty instances with the numbers of events approximately 200, 400 and 1000 for the small, medium and large sets respectively. (These can be downloaded at [17].)

Figure 3 shows an example of how the various fitness functions cause the algorithm to behave over time. For the first 200 or so seconds, f_2 shows the quickest movement through the search space but as time progresses, we see that f_5, f_6 and, to a lesser extent f_3 seem to move ahead of the field. The probable reason for this is because at the beginning of the run the population is very diverse and so it is quite easy to pick out the better timetables by looking directly at the distance to feasibility (which is what and f_2 does). However, a point is reached (here at around 200 seconds) where the timetables in the population start to look too similar with regards to f_2's criteria and we are not able to effectively distinguish between them anymore. In this case much of the selection pressure is lost and indeed, other criteria has to be looked at in order to guide the search. This is when f_5, f_6 and f_3 begin to show their strengths - they are able to distinguish between timetables using the same number of timeslots or with the same distance to feasibility, and go on to give a more effective search.

We also note that fitness function f_4 showed slightly disappointing performances. Thus, it would seem that exclusively identifying a *good* timeslot as one with lots of events in it is an oversimplification that does not really aid the search in a satisfactory manner.

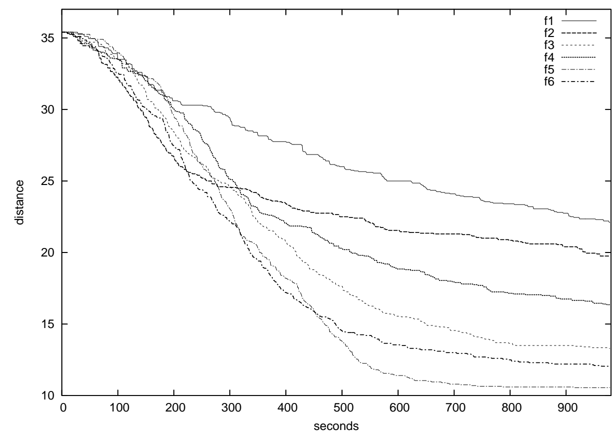

Figure 3. Effects of the six fitness functions over time with the medium instances. As with figures 4, 5, and 8, each line represents the distance to feasibility of the best solution in the population, averaged across the instance set.

Next, we look at the performance of the algorithm with respect to the number of evaluations performed. This sort of performance measure is valid with timetabling because it could be the case that in some applications the evaluation function is the most expensive operation in the algorithm, particularly when dealing with soft constraints. (Consider, for example, an evaluation function that needs to analyse the timetables of every individual student.) With this performance measure, we see a similar pattern

emerging: as figure 4 shows, for the first 15,000 or so evaluations f_2, and then f_3 show the quickest movements, but as the same run is allowed to continue, f_5 (and to a lesser extent f_6 and f_3) go on to find the best results with f_1, f_2 and f_4 eventually producing the worst.

Table 1 also shows some interesting observations of these experiments. As can be seen, when using fitness functions f_4, f_5 and f_6, runs of 100,000 evaluations took considerably less time than runs with the other three fitness functions. Our experiments have led us to believe that this is due to the fitness functions' effects on population diversities during the evolutionary process. As we have already noted in section 2, when a population is diverse, the amount of rebuilding that needs to be done during recombination generally increases, making the entire operation more expensive. Secondly, we have also seen that certain fitness functions seem to reach a point earlier than others where they cannot differentiate effectively between different timetables, causing a reduction or even a cancelling-out altogether of selection pressure. When this sort of transition occurs, obviously there will be less of a push towards convergence and this will cause the recombination operator to remain more expensive for a greater period of the run.

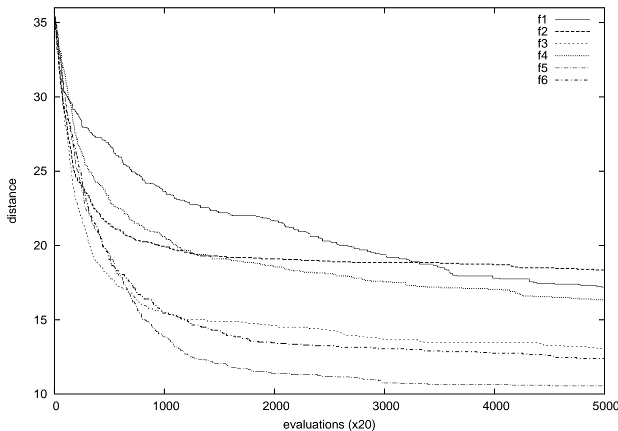

Figure 4. Effects of the six fitness functions Vs number of evaluations with the medium instances.

	f_1	f_2	f_3	f_4	f_5	f_6
Small	241	195	189	171	**164**	167
Med	2335	959	914	751	738	**716**
Large	22872	43242	21040	10102	9524	**9349**

Table 1. Mean time (secs) taken for runs of 100,000 evals. with the six fitness functions**.**

Note that an overly rapid loss of diversity may sometimes be undesirable in an EA as it can lead to a redundancy of the recombination operator and an under-sampling of the search space. However, in the case of this algorithm there is clearly another trade-off because, as already noted, recombination is generally more expensive when the population is diverse. With the small and medium instances, the balance seemed to fall in favour of using f_5 and f_6 which, even though exhibiting tendencies to cause a more rapid loss of diversity, still returned superior results and in less time.

Interestingly, as can be viewed in figure 5, a different behaviour is seen with the large instances. Here, there is not only a dismal performance of f_1 and f_4, but better results (after 100,000 evaluations) clearly come from f_2 and f_3. In this case, it would seem that the trade-off has fallen the other way, and that the sustained diversity due to f_2 and f_3 has allowed better solutions to be found. However, these come at a price; a second glance at table 1 reveals that the runs on average took over twice as long in the case of f_3 (compared to f_4, f_5, and f_6) and over *four* times as long in the case of f_2.

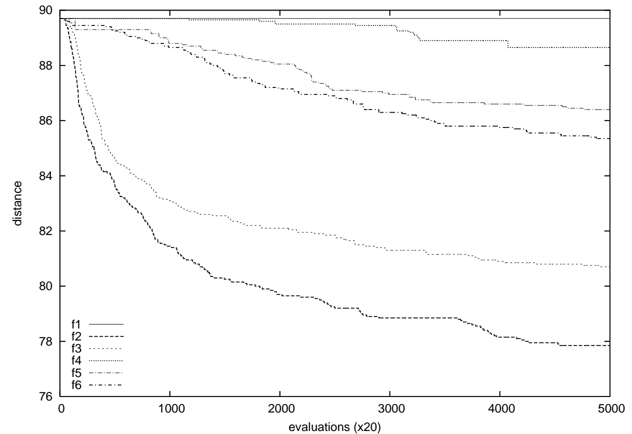

Figure 5. Effects of the six fitness functions Vs number of evaluations with the large instances.

4 The Effects of Stochastic Local-search

It is generally accepted that evolutionary algorithms are very good at coarse-grained global search but are rather poor at fine-grained local-search [14]. Indeed, as the population starts to converge, so the effects of crossover usually lessen until, generally only the mutation operator offers further movements. It is therefore perfectly legitimate and increasingly common (see, for example, [10]) to attempt to enhance an EA by adding some sort of local-search procedure (this is sometimes called a *memetic* algorithm [18]). Thus, a successful marriage can be formed whereby the EA helps move the search into promising regions of the search space, with a local-search method then being utilised to explore *within* these regions.

Figure 6 outlines our local-search operator – taking a list of unplaced events *U* and a partial timetable *tt*, it goes about trying to insert any event from *U* into any unoccupied, feasible place in *tt*. If there is no such place, then the events inside *tt* are randomly shuffled and the process is repeated until either all the events in *U* have been placed or a cut-off point is reached. For our experiments, we decided that this cut-off point should be in some way proportional to the instance size. It is

therefore expressed by the parameter *limit* so that the operation is forced to stop when (*limit* × *e*) iterations of local-search have taken place (where *e* is the total number of events in the particular instance being addressed).

Note that this procedure will do two important things. Firstly, if successful, events from *U* will added to the timetable *tt*, thereby improving (on average) the timeslot packings, and possibly reducing the number of extra timeslots needed to be opened to house the remaining events in *U*. Secondly, because events are randomly shuffled amongst the timeslots, extra diversity will be added to the population.

In our experiments, we used the local-search in conjunction with the mutation operator: as before, each time mutation occurs, a small number of timeslots (and the events contained within them) are removed from the timetable. Now however, the local-search procedure is applied. If at the end of the operation there are still events that have not been inserted into the timetable then these are dealt with by using the original constructive heuristic that, we remember, has the ability to open up extra timeslots.

```
LocalSearch (tt, U, cutOff)
1.  Make a list V containing all the places in tt
    that have no events assigned to them
2.  count = 0
3.  while (U≠∅ and V≠∅ and count<cutOff)
4.      foreach(u∈U and v∈V)
5.          if (u can be feasibly put into v in tt)
6.              Put u into v in tt
7.              Remove u from U and v from V
8.      if (U≠∅ and V≠∅)
9.          done = false
10.         repeat
11.             Choose random event e in tt and v∈V
12.             if(e can be feasibly moved to v in tt)
13.                 Move e to v
14.                 Update V to reflect changes
15.                 done = true
16.             count++
17.         until (count ≥ cutOff or done)
```

Figure 6. The local-search procedure

4.1 Observations

From a practical standpoint, if we are to asses whether the additional local-search operator is worth using at all, then analysing the algorithm's behaviour with respect to the number of evaluations performed is fairly meaningless as more computation will be being performed for each new individual. This means that any experiments must now only be measured with regards to CPU time.

In order to gauge the effects, benefits, and drawbacks of the new operator we performed trial runs using eleven different recombination rates (between 0.0 and 1.0), eleven different settings for *limit* (between 0 and 20) and twelve different population sizes (between 2 and 50) on all sixty instances. As in [7], time limits of 30, 200 and 800 seconds were imposed for the small, medium and large instance sets respectively. Lastly, fitness function f_5 from the previous section was used.

The first thing that we noticed from these experiments was the huge effect that the local-search had on the number of new individuals produced within the time limit. This is illustrated, for two population sizes, in figure 7. Here, we see that the use of local-search, even in small amounts, causes far fewer individuals to be produced within the time limits. We believe that this is almost exclusively to do with the fact that the local-search operator, by adding diversity to the population, keeps the recombination operator consistently expensive throughout the run.

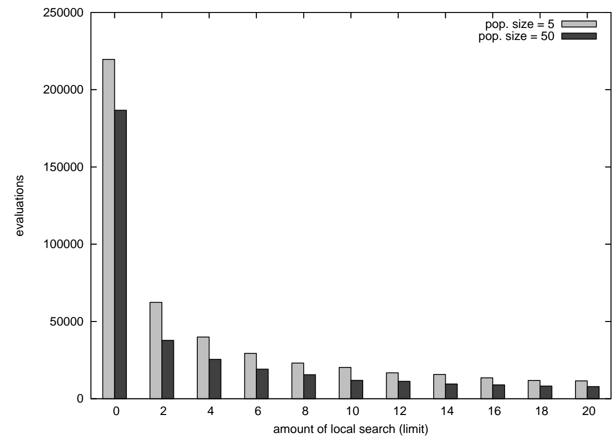

Figure 7. Effects that the local-search operator has on the number evaluations performed within the time limit (using recombination rate 0.5).

We also saw a different response to the parameter settings when dealing with the various instance sets. For the medium instances, the top ten results (on average across the set) all came when using fairly small populations (5 to 15 individuals), with high levels of recombination (between 0.6 and 0.9), and small, but still significant, levels of local-search (in all cases this was *limit* = 2). With the medium instances then, it would seem that both the recombination operator *and* the local-search are integral to the overall search. Additionally, it also seems reasonable to assume that the smaller populations give the most successful runs because the overall potential for diversity is reduced (therefore allowing more individuals to be produced within the time limit) but yet is still kept at high enough levels by the local-search operator.

In contrast, with the large instances we saw the best results being achieved when using *bigger* populations (30 to 50 individuals), with *low* recombination rates (0.0 or 0.1), and little or no local-search (*limit* = 0 or 1). We can only conclude that, in this case, whilst the larger populations help to ensure that more regions of the search space are sampled, recombination is simply too costly to provide enough evolution within the imposed (but still

fair) time limit. Thus using it in small amounts or indeed not at all seems the more promising strategy.

There was no immediate pattern concerning the results of the experiments with the small instances. In many cases the algorithm was able to find a solution in over 80% of the trials. This means that the algorithm probably just finds this instance set too easy, making it difficult to glean any further insights.

Finally, figure 8 shows some example runs illustrating the effects of various parameter settings with the medium instances. As can be seen, in this case the best final results seem to come when we use a suitable balance of both local-search (*limit*=2) and recombination (rate=0.7). However, we also see that the quickest positive movements through the search space (at least for the first 50 or so seconds) actually come when we use no recombination or local-search at all. After this point the run then stagnates. It therefore seems reasonable to investigate if better overall performance can be gained if recombination and local-search are only introduced once the runs using only mutation start to level off.

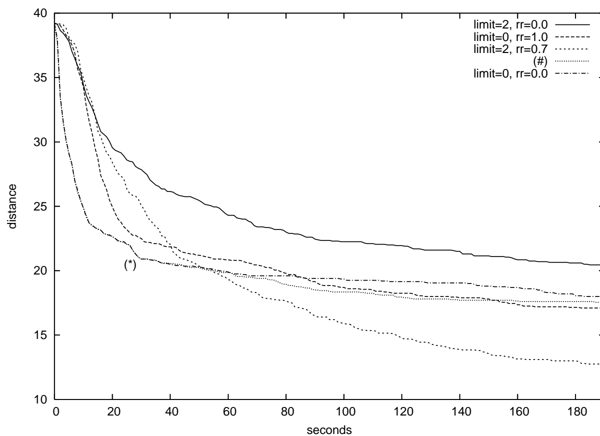

Figure 8. Contrasting the behaviour of the algorithm with different parameter settings using populations of size 10 and the medium instances. Meanings of (#) and (*) are explained in the text.

This is exactly what is done at around the mark (*) on figure 8. Here, we added an additional operator that sensed when the runs' progress started to fade, and then increased the amount of local-search from *limit*=0 to *limit*=2, and the amount of recombination from a rate 0.0 to a rate 0.5. This was done in the hope that the new operators would then take the search further towards a solution. However as the line marked (#) in figure 8 indicates, the benefits of doing this are marginal to say the least. This suggests that for local-search to be given a real opportunity to improve results, it needs to be used from early on in the search.

5 Conclusions

We conclude this paper with the following points:

- We have discussed a way to go about measuring population diversity with the standard GGA representation and discovered, in our case, that this diversity has a very close correlation with the average cost of the recombination operator; i.e. if the population is diverse, and recombination is being used, the rate at which new individuals are formed (per time unit) will usually fall.
- We have seen that the choice of fitness function for this problem can have a large effect both on the overall solution quality and the algorithm's time requirements. In particular, the latter seems almost exclusively due to the effects that the fitness functions have on the diversity of the population. We have also seen that diferent fittness fuctions can be beneficial at different stages of the runs. Future work could show that even more superior results might be found by using different fitness criteria at different points of the run or indeed, by using some sort of sequential evaluation [11].
- We have seen, through a substantial number of experiments, that our stochastic local-search operator *can* improve the search, but should probably be used with some care. Indeed, even when used in small levels, whilst allowing the possibility of further improvement to the timetables, it can also drastically increase and sustain population diversity throughout the run causing the recombination operation to be just too expensive. This might mean that just not enough new individuals are produced within reasonable time. In some situations this problem can be remedied, to a certain extent, by using smaller populations.
- As often noted in this paper, the algorithm seems to behave in different ways with different sized instances; whilst performing extremely well with the small instances and also promisingly with the medium instances, it seems to consistently underachieve when dealing with the large instances. In particular, during runs with these, the overall fitness of the populations did not seem to improve anywhere near as much as should be expected (refer to the *y*-axis of figure 5). A possible explanation for this is the fact that, in our case, we are always interested in finding timetables with forty-five timeslots but the number of events is variable. Thus, as the instance size is increased, the number of rooms being used needs to increase and *not* the number of timeslots. Therefore the genetic operators function on chromosomes whose lengths will remain more or less constant regardless of instance size! Clearly this presents a scalability issue, and we may start to see further unsatisfactory movement through the search space if instance size is further increased (as is common in practical timetabling [11]).

Note that all of these points above may actually extend beyond the application discussed here and could even turn out to apply to the general GGA model as a whole. However, verification of this conjecture awaits further research.

Bibliography

[1] Arntzen, H. and Løkketangen, A. (2003) "A Tabu Search Heuristic for a University Timetabling Problem", Proceedings of the Fifth Metaheuristics International Conference MIC 2003, Kyoto, Japan.

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

[3] Erben, W. (2000) "A Grouping Genetic Algorithm for Graph Colouring and Exam Timetabling" In E. Burke and W. Erben (Eds.) The Practice and theory of Automated Timetabling - PATAT III, Springer LNCS 2079, pp 132-158.

[4] Falkenauer, E. (1999) Genetic Algorithms and Grouping Problems. John Wiley and Sons Ltd 1999.

[5] van Hemert, J., Eiben A., (1996) "Comparison of the saw-ing evolutionary algorithm and the grouping genetic algorithm for graph coloring". Technical Report TR-97-14, Leiden University.

[6] Kostuch P. (2004) "The University Course Timetabling Problem with a 3-stage approach", In E. Burke, M. Trick (Eds.) Practice and Theory of Automated Timetabling - PATAT V, pp 251-266.

[7] Lewis, R. and Paechter, B. (2005) "Application of the Grouping Genetic Algorithm to University Course Timetabling," In G. Raidl and J. Gottlieb (eds) Evolutionary Computation in Combinatorial Optimization, Springer LNCS 3448, pages 144-153.

[8] Mattiussi, C. , Waibel, M. and Floreano, D. (2004) "Measures of Diversity for Populations and Distances Between Individuals with Highly Reorganizable Genomes", Evolutionary Computation, 12 (4), pp. 495-515.

[9] Morrison, W., de Jong, K., (2002) "Measurement of Population Diversity", In P. Collet, C. Fonlupt, J. Hao, E. Lutton, M. Schoenauer (Eds), Artificial Evolution 2001, Springer LNCS 2310, pp 31-41.

[10] Moscato, P and Norman M.G. (1992) "A 'Memetic' Approach for the Traveling Salesman Problem. Implementation of a Computational Ecology for Combinatorial Optimization on Message-Passing Systems", In by M. Valero, E. Onate, M. Jane, J.L. Larriba and B. Suarez (Eds.), Parallel Computing and Transputer Applications, IOS Press, Amsterdam, pp. 187-194.

[11] Paechter, B., Rankin, R., Cumming A., and Fogarty, T. (1998) "Timetabling the Classes of an Entire University with an Evolutionary Algorithm". In A. Eiben, T. Bäck, M. Schoenauer, H. Schwefel (Eds.) Parallel Problem Solving from Nature - PPSN V, Springer LNCS 1498, pp. 865-874.

[12] Rossi-Doria, O., Samples, M., Birattari, M., Chiarandini, M., Knowles, J., Manfrin, M., Mastrolilli, M., Paquete, L., Paechter, B., Stützle, T. (2002) "A comparison of the performance of different metaheuristics on the timetabling problem". In E. Burke, P. De Causmaecker (Eds.) Practice and Theory of Automated Timetabling - PATAT IV, Springer LNCS 2740, pp 329-351.

[13] Tao, G. and Michalewicz, Z. (1998) "Inver-over Operator for the TSP", In T. Baeck, A. Eiben, M. Schoenauer, and H. Schwefel (Eds) Parallel Problem Solving from Nature - PPSN V, Springer LNCS 1498, pp.803-812.

[14] Yao, X. (1996) "An overview of evolutionary computation", Chinese Journal of Advanced Software Research, Allerton Press, Inc., New York, NY 10011, 3(1):12-29.

[15] International Timetabling Competiton - http://www.idsia.ch/Files/ttcomp2002/ accessed March 2005.

[16] The Metaheuristic Network homepage can be found at http://www.metaheuristics.org/ accessed March 2005.

[17] Instances and previous results available at http://www.emergentcomputing.org/timetabling/harderinstances/ accessed March 2005.

[18] Moscato, P. (1989) "On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms," Tech. Rep. Caltech Concurrent Computation Program, Report. 826, California Institute of Technology, Pasadena, California, USA.