An Order Based Memetic Evolutionary Algorithm for Set Partitioning Problems

Christine L. Mumford

School of Computer Science, Cardiff University, 5 The Parade, Cardiff CF24 3AA, United Kingdom
C.L.Mumford@cs.cardiff.ac.uk

1 Introduction

Metaheuristic algorithms, such as simulated annealing, tabu search and evolutionary algorithms, are popular techniques for solving optimization problems when exact methods are not practical. For example, the run times required to obtain exact solutions to many common combinatorial problems grow exponentially (or worse) with the problem size, and as a result, solving even a relatively modest sized problem of this type may require many centuries of computation time, even on the fastest computer of the day. Such problems are known collectively as NP-Hard, and include the travelling salesman problem (TSP), which is probably the best known problem in the class. The present study will concentrate on a type of NP-Hard combinatorial problem known as the *set partitioning problem*. If we have n objects to partition into m sets, in such a way that each object must be assigned to exactly one set, it follows that there are m^n different ways that the n objects can be assigned to the m sets, for a straightforward unconstrained problem. It is instructive to note that every time the problem size of the set partitioning problem is increased by one object, the corresponding run time for an exhaustive search algorithm will increase by a factor of m, and thus the run time grows exponentially as the number of objects – n – increases. While it is true that much better exact methods than exhaustive search have been developed for most NP-Hard problems, the 'growth factor' remains exponential for the run time, and no one in history has so far managed to change that.

The main focus of the current Chapter is a new hybrid (or memetic) evolutionary algorithm specifically developed to solve set partitioning problems. This technique incorporates useful solution improvement heuristics into an evolutionary framework. New genetic operators have been devised to ensure that parent solutions are able to contribute useful features to their offspring, and a simulated annealing cooling schedule has been adopted to help maintain a balance between quality and diversity within the population of candidate solutions. The effectiveness and versatility of the new hybrid algorithm will be demonstrated by applying variations of the technique to hard literature benchmarks taken from three different set partitioning applications: graph coloring, bin packing and examination timetabling.

It is well known among researchers that effective evolutionary algorithms are notoriously difficult to devise for many types of problems, and set partitioning is particularly challenging, for reasons we will address later. At its best, an evolutionary algorithm will exploit its population structure to explore different parts of the search space for a problem simultaneously, combining useful features from different individuals during reproduction and producing new offspring solutions that may be better, on occasions, than either of that offspring's parents. At its worst, an evolutionary algorithm will take a lot longer than competitive methods to achieve very little. For most other methods the search is propagated through a single focal point. On the other hand, evolutionary algorithms progress from a population of points. The population has to 'earn its living', otherwise it becomes a burden rather than a bonus.

With the limitations and idiosyncrasies of evolutionary algorithms in mind, we will critically evaluate the various components of the new hybrid approach in the present study. In particular, we will endeavor to ensure that every part of the algorithm is making a useful contribution to its overall performance. The main goal is to present a new, and reasonably generic, order based framework that can be applied, with minimum adaptation, to a wide range of set partitioning problems. Although the exact choice of objective or fitness function will very likely depend on the specific problem, it is envisaged that problem specific heuristics and costly backtracking will largely be avoided. Throughout the Chapter a tutorial approach is adopted, to aid newcomers to the field, and the main aspects of the algorithms and operators are illustrated using simple examples and carefully designed diagrams. However, it is hoped that the more experienced researcher will also find the Chapter of interest.

The remainder of the Chapter is organized as follows. We begin with introductory Sections on evolutionary algorithms, some of which may be safely skipped by the more knowledgeable reader. Section 2 outlines the historical development of evolutionary computing, and Section 3 introduces the main elements of a 'standard' genetic approach to problem solving. Section 4 describes the general features of an order based genetic algorithm and, together with Section 5 – on steady-state GAs – lays the foundations for the approach used for set partitioning in the present Chapter. Section 6 introduces the three test problems: graph coloring, bin packing, and timetabling, and this is followed by Section 7 which presents the reader with the main points that motivated the present study. The next Section covers the grouping and reordering heuristics of Culberson and Luo [9]. These heuristics are used in the present study to improve the effectiveness of the new crossover operators, and also provide useful local search capability in their own right. Section 9 details the main features of the new memetic approach, covering all the main aspects of the new genetic simulated annealing algorithm (GSA). The Section also describes new crossover operators and defines the fitness functions that are used, as well as introducing the simulated annealing cooling schedule. Results for the literature benchmarks are presented in Section 10 and this is followed by a Chapter summary. Finally, URL links are provided to all the test data in the Resources Appendix.

2 A Brief History of Genetic Algorithms

Several groups of researchers, working independently in the 1950s and 60s, developed optimization techniques inspired by evolution and natural selection. Rechenberg [37] introduced *evolution strategies* and used the method to optimize real-valued parameters for designing aerofoils. Fogel, Owen and Walsh [17] developed *evolutionary programming*, a technique that they used to evolve finite state machines. Most of these early techniques were developed with specific applications in mind, however. In contrast, John Holland made an extensive study of adaptation in the natural world and used his insight to create a sound theoretical framework from which his *genetic algorithms (GAs)* emerged [22]. Since these early days, interest in evolutionary-inspired algorithms has grown year by year, and numerous variants have appeared on the scene, some of them very different from anything conceived by Rechenberg, Fogel or Holland. For example, in the early 1990s, John Koza proposed *genetic programming*, an evolutionary style technique for evolving effective computer programs to undertake particular tasks.

Other popular paradigms to have been derived from the more generic approach include artificial life [28], evolvable hardware [21], ant systems [12] and particle swarms [26], to name but a few. In addition, there are many examples of hybrid (or memetic) approaches where problem specific heuristics, or other techniques such as neural networks or simulated annealing, have been incorporated into a GA framework. Indeed, we shall make use of specialized heuristics and also simulated annealing to improve our results in the present Chapter. Thus, due to the growth in popularity of search and optimization techniques inspired by natural evolution during the last few decades, it is now common practice to refer to the field as *evolutionary computing* and to the various techniques as *evolutionary algorithms*. Inevitably, though, due to the overwhelming influence of John Holland, the term 'genetic algorithm' is frequently used interchangeably with the more generic term.

3 A Generic Genetic Algorithm

As suggested above, there is no rigorous definition of the term 'genetic algorithm' that everyone working in the field would agree on. There are, however, certain elements that GAs tend to have in common:

1. a population of chromosomes encoding (in string form) candidate solutions to the problem in hand,
2. a mechanism for reproduction,
3. selection according to fitness, and
4. genetic operators.

The *chromosomes* in the population of a GA usually take the form of strings, which may be encoded in binary, denary (that is, base-10), or in some other way. As an example, let us consider a simple optimization problem. Suppose we wish to maximize the following function:

$$f(x_1, x_2) = {x_1}^2 - {x_2}^2 + x_1 x_2 \tag{1}$$

where x_1 and x_2 are both integers that can vary between 0 and 31. If we use a 5-bit binary representation for x_1 and x_2, our GA strings would need to be 10 bits long. Thus, using this representation the string 0010111101 would encode $x_1 = 00101$, and $x_2 = 11101$, or 5 and 29 respectively in denary.

The *mechanism for reproduction* may consist simply of duplicating strings from the population. However, the choice of strings for reproduction is usually biased by some measure of *fitness* associated with each member of the population. The term 'fitness' refers to some estimate of quality allocated to each population member whereby 'better' individuals are assigned higher values than poorer individuals. In the above optimization problem the objective function – $f(x_1, x_2)$ – may be used directly as a fitness function. In other situations the allocation of fitness values may not be so straightforward: when solving a minimization problem, for example, or when dealing with qualitative data.

An essential feature of a successful GA implementation is that the average fitness value of a population should increase over time, reflecting an improving trend in the population. To facilitate this improvement we must somehow ensure that superior individuals have a better chance to contribute to future generations than do individuals with poorer fitness values. Probably the most popular way to drive this improvement is to use *selection probabilities* to bias the choice of parents for the next generation. Converting fitness values into probabilities for selection is usually a straightforward matter involving some simple arithmetic. Random numbers can then be generated and the parents of the next generation selected in accordance with a probability distribution derived from the individual fitness values of the population members. Due to the obvious analogy with a popular casino game, this process is widely known as *roulette wheel selection.*

Evolution cannot proceed by selection and reproduction alone, of course. It is essential that a GA is capable of occasionally producing new individuals that are better than their parents. In order to achieve this a mechanism to effect change is needed, and this is the role of *genetic operators.* Holland describes three types of genetic operators: *crossover*, *mutation* and *inversion.* Examples to illustrate all of these are given in Figure 1.

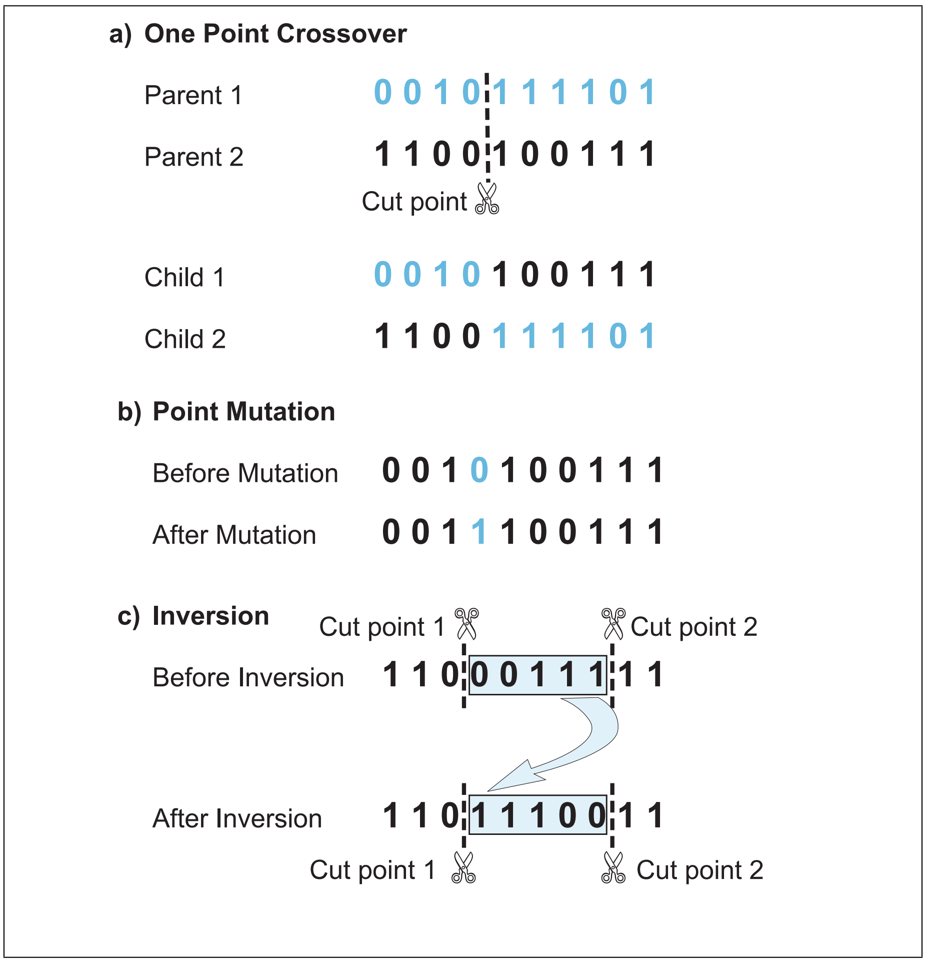


Fig. 1. Examples of Genetic Operators

One point crossover, shown in Figure 1(a), is a process involving two parental strings and begins with an alignment of the strings. A cut point is then chosen at random, and the parental material following the cut point is exchanged between the parents, giving rise to two children. Variants of simple crossover include two point and multi-point crossover where more cut points are selected. Point mutation is illustrated in Figure 1(b). For a binary string a randomly selected bit is simply flipped. For denary or other encodings suitable mutation operators are chosen that produce very small changes. An example of inversion can be seen in Figure 1(c). Here two cut points are selected at random on a single string, and the sub-string between the cut points is then inverted and replaced. Inversion, however, is not commonly used in practice for simple bit strings and denary chromosomes. Following reproduction, each genetic operator is applied with its own predetermined probability.

A simple generic GA is outlined in Algorithm 1. Adapting the generic model to make it effective for a particular problem, however, usually requires considerable effort, as well as a large measure of good luck! First, it is necessary to devise a suitable representation so that candidate solutions can be satisfactorily encoded as chromosomes. For many applications it is not immediately obvious how this can be done, and simple bit- or denary-valued strings may not be appropriate. Second, when deviating from a standard bit string or denary representation, special genetic operators may be needed, to avoid the production of infeasible offspring, say. Another potential difficulty is choosing a suitable fitness function for a given problem. Selection bias towards the better individuals needs to be strong enough to encourage 'survival of the fittest', but not so strong that all variability is quickly lost from the population (note: loss of diversity early on in the execution of a GA is often referred to as *premature convergence*). Without variability, nothing new can evolve. Finally, tuning the GA and determining the best values for various parameters – such as crossover and mutation rates, population size and stopping criteria – can be a very time consuming process.

Algorithm 1 A Generic Genetic Algorithm (GA)

```
Generate $N$ random strings {$N$ is the population size}
Evaluate and store the fitness of each string
**repeat**
  **for** $i = 1$ to $N/2$ **do**
    Select a pair of parents at random {The selection probability is in direct
    proportion to the fitness}
    Apply crossover with probability $p_c$ to produce two offspring
    **if** no crossover takes place **then**
      Form two offspring that are exact copies of their parents
    Mutate the two offspring at a rate of $p_m$ at each locus
    Evaluate and store the fitness for the two offspring
  Replace the current population with the new population
**until** stopping condition satisfied
```

For further reading on genetic algorithms, I recommend the following introductory texts: [20], [32] or [33].

4 Order based GAs

In Section 3 we saw how chromosomes can be encoded, as bit strings or decimal coded lists, and used to directly represent the variables of a problem. For many combinatorial problems, however, the random processes involved in assigning values to the variables make direct representations rather prone to constraint violations, and as a result a GA can waste a vast amount of time evaluating infeasible solutions. Consider a set partitioning problem, which involves the assignment of each available item to exactly one set whilst strictly adhering to any problem specific constraints. For example, with the bin packing problem various sized items are placed in a minimum number of equal sized bins. However, it is likely that assigning items to bins at random may result in some bins becoming overfull. The use of heavy penalty values to discourage the survival of illegal solutions is an approach favored by some researchers, while others prefer to use an heuristic repair mechanism to reduce or eliminate constraint conflicts, following the initial assignment of the GA. Yet another alternative is to use an order based approach with a greedy decoder. This approach will entirely avoid the issue of infeasibility. Starting with an arbitrary permutation of items, a greedy decoder will sequentially assign legal values to all the items in the list. Consider the graph coloring problem (GCP). This involves finding a minimum set of colors for the vertices of a given graph, so that no two adjacent vertices have the same color. If, for example, a GCP instance has n vertices, then order based chromosomes representing potential solutions will consist of permuted lists of the integers $\{1, 2, 3, \cdots, n\}$. A decoder will start with the first vertex on the list and work through assigning, to each vertex in turn, the first available color from an ordered set (in other words, each color is identified by an integer label, 0, 1, 2, 3, ...), that is possible without causing conflicts.

Figure 2 illustrates possible encodings for a legally colored 12 node graph, with Figure 2(b) and (c) showing direct and order based representations, respectively, for the example coloring in Figure 2(a). It is easy to visualize how disruptive genetic operators could be if applied to a direct representation such as that shown in Figure 2(b). An order based representation, on the other hand, will always produce a legal coloring when used in conjunction with a greedy decoder.

Unfortunately, standard crossover and mutation operators are not appropriate for order based representations, because such operators tend to destroy the permutation property and produce infeasible solutions. The problem with crossover is illustrated in the example below, which shows the production of infeasible offspring with duplicated and deleted values, following application of a two point crossover.

$$\begin{array}{ll} A & = 8\ 7\ 6 \mid 4\ 1\ 2 \mid 5\ 3 \\ B & = 2\ 5\ 1 \mid 7\ 3\ 8 \mid 4\ 6 \end{array}$$

Producing:

$$\begin{array}{ll} A' & = \mathit{8}\ \mathit{7}\ 6 \mid \mathit{7}\ \mathit{3}\ \mathit{8} \mid 5\ \mathit{3} \\ B' & = \mathit{2}\ 5\ \mathit{1} \mid \mathit{4}\ \mathit{1}\ \mathit{2} \mid \mathit{4}\ 6 \end{array}$$

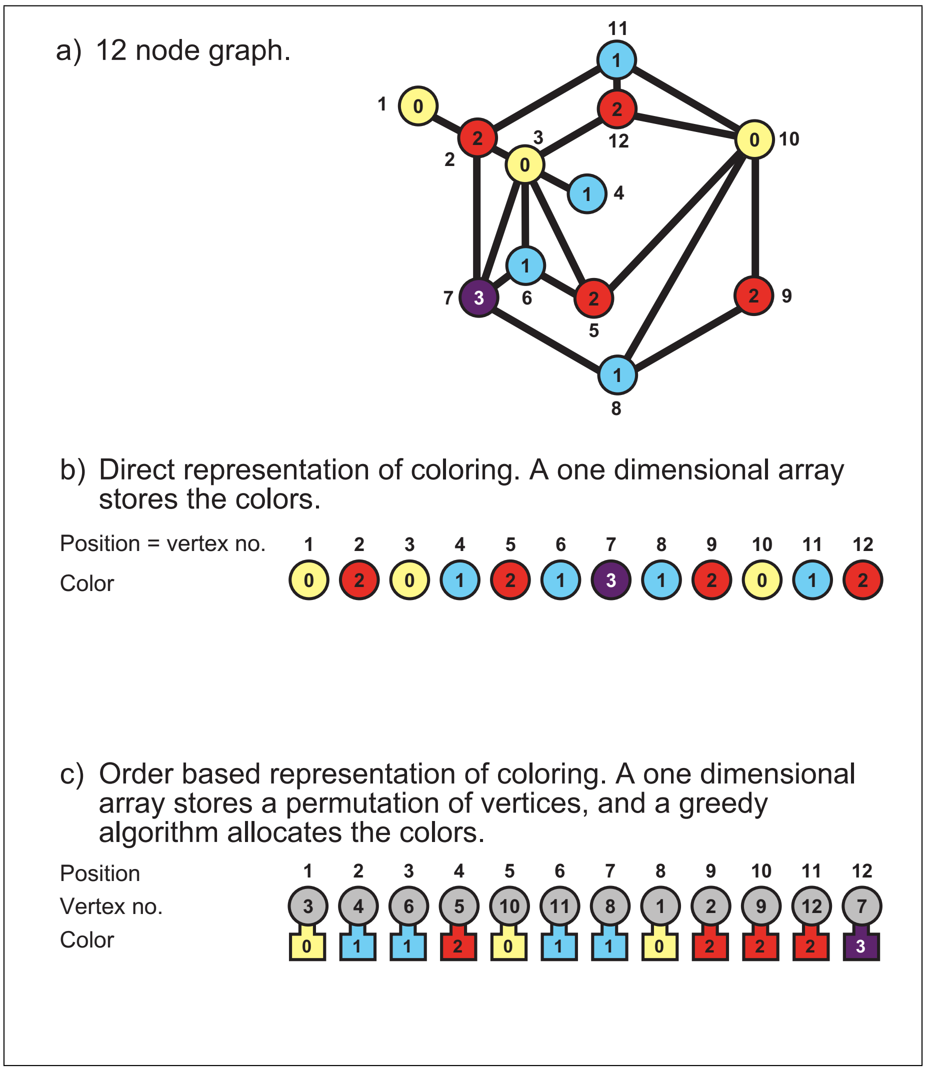

a) 12 node graph.
b) Direct representation of coloring. A one dimensional array stores the colors.
Position = vertex no.
Color
c) Order based representation of coloring. A one dimensional array stores a permutation of vertices, and a greedy algorithm allocates the colors.
Position
Vertex no.
Color

Fig. 2. A direct and order based representation of a coloring for a 12-node graph

Point mutation also produces duplications and deletions. On the other hand, Holland's inversion operator respects the permutation property, and can indeed prove useful as a reordering operator.

The best known crossovers designed for permutations are probably partially matched crossover (PMX) [20], order crossover (OX) [10] and cycle crossover (CX) [35]. OX and CX are explained below along with another more recent example – merging crossover (MOX) [1]; a description of PMX is omitted because it is less relevant to the present study than the other examples. We will discuss the shortcomings of applying the simple order based approach to set partitioning problems in Section 6.

Order Crossover (OX)

This operation always produces legal permutations. It starts by selecting two crossing sites at random:

$$\begin{array}{ll} A & = 8\ 7\ 6\ |\ 4\ 1\ 2\ |\ 5\ 3 \\ B & = 2\ 5\ 1\ |\ 7\ 3\ 8\ |\ 4\ 6 \end{array}$$

Values from the middle segment of one parent are then deleted from the other to leave holes. For example values 4, 1 and 2 will leave holes, marked by '*H*' in string *B*:

$$B' = \mathrm{H}\ 5\ \mathrm{H}\ |\ 7\ 3\ 8\ |\ \mathrm{H}\ 6$$

In one version of OX, the holes are filled with a sliding motion that starts from the beginning of the string.

$$B' = 5\ 7\ 3\ |\ \mathrm{H\,H\,H}\ |\ 8\ 6$$

The substring from string *A* is then inserted into string *B*. The final result of this cross and the complementary cross is:

$$\begin{array}{ll} A' & = 6\ 4\ 1\ |\ 7\ 3\ 8\ |\ 2\ 5 \\ B' & = 5\ 7\ 3\ |\ 4\ 1\ 2\ |\ 8\ 6 \end{array}$$

Cycle Crossover (CX)

The cycle crossover operator ensures that each position in the resulting offspring is populated with a value occupying the same position in one or other of the parents. As an example, suppose we have strings *A* and *B* below as our two parents:

$$\begin{array}{ll} A & = 8\quad 7\quad 6\quad 4\quad 1\quad 2\quad 5\quad 3\quad 9\quad 10 \\ B & = 2\quad 5\quad 1\quad 7\quad 3\quad 8\quad 4\quad 6\quad 10\ 9 \end{array}$$

We now start from the left and randomly select an item from string A. Suppose we choose item 6 from position 3, this is then copied to position 3 of the offspring we shall call A':

$$A' = - \quad - \quad 6 \quad - \quad - \quad - \quad - \quad - \quad - \quad -$$

In order to ensure that each value in the offspring occupies the same position as it does in either one or other parent, we now look in position 3 of string B and copy item 1 from string A to the offspring:

$$A' = - \quad - \quad 6 \quad - \quad 1 \quad - \quad - \quad - \quad - \quad -$$

Next we look in position 5 of string B and copy item 3 from string A:

$$A' = - \quad - \quad 6 \quad - \quad 1 \quad - \quad - \quad 3 \quad - \quad -$$

Examining position 8 in string B we find item 6. This completes the cycle. We now fill the remaining positions in A' from string B thus:

$$\begin{array}{llllllllll} A' = 2 & 5 & 6 & 7 & 1 & 8 & 4 & 3 & 10 & 9 \\ B' = 8 & 7 & 1 & 4 & 3 & 2 & 5 & 6 & 9 & 10 \end{array}$$

The offspring B' is obtained by performing the complementary operations.

Merging Crossover (MOX)

Merging crossover (MOX) was devised by Anderson and Ashlock [1] for use on graph coloring problems. Initially two n element parents are randomly merged into a single $2n$ element list. The first occurrence of each value in the merged list gives the ordering of elements in the first child, and the second occurrence in the second child. MOX is illustrated in Figure 3. Anderson and Ashlock point out the following property of MOX: if an element, a precedes another element b in both parents, then it follows that a will precede b in both children.

Mutation Operators for Permutations

As happens with crossover, standard mutation will produce duplications and deletions in a chromosome, if an order based representation is used. Fortunately, a number of alternatives have been devised. The simplest of these was

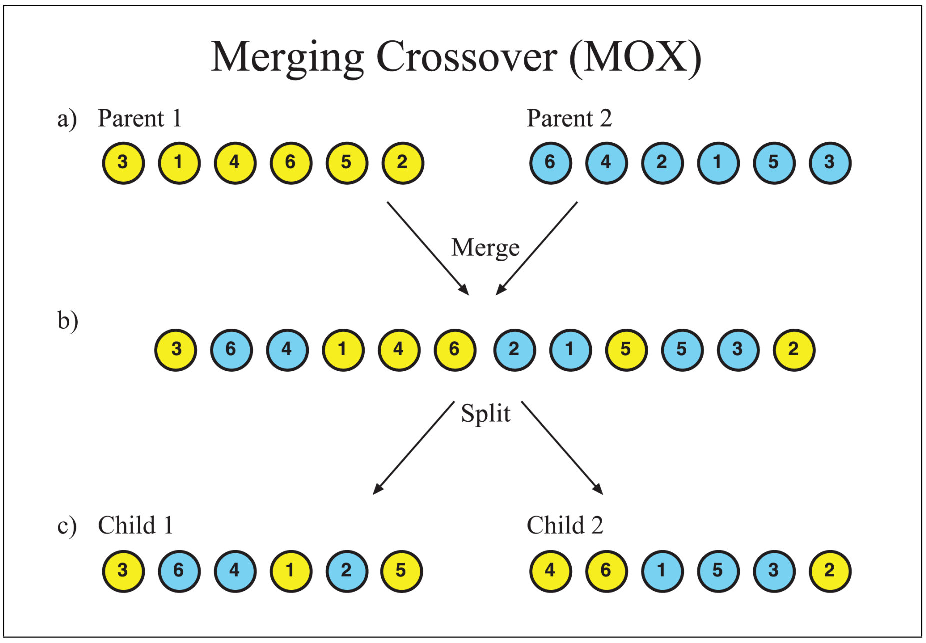


Fig. 3. MOX Crossover [1], used as a basis for the new MIS crossover

named *position based mutation* by Davis [11]. This operation, also known as *insertion mutation*, simply involves selecting two values at random from a permutation list, and placing the second before the first. Another straightforward mutation suitable for order based representations is *order based mutation* [11] or *swap mutation*, which selects two values at random and swaps their positions. Davis also promotes an idea he calls *scramble sublist* [11], in which a substring is selected stochastically and its contents randomly shuffled. In the present work, however, Holland's inversion operator is used extensively as a mutation. This operator seems to be particularly effective for certain set partitioning problems. As we shall see later when we look at the grouping and reordering heuristics of Culberson and Luo [9], the act of reversing a list (or sublist) can have a positive effect on the result, following the application of the greedy decoder.

5 A Simple Steady-State GA

For convenience we will use a simple steady-state GA as a framework for our present study. There are few parameters to set using this approach: no global fitness function is used, for example, thus roulette wheel selection is avoided, and so is the need to manipulate and scale the fitness values. Tuning the fitness function to get the selective pressure just right can be very difficult, and getting it wrong can prove a disaster. The present author favors simple pairwise comparisons to identify whether one individual is better than another, using the result to determine who shall live and who shall die. With this approach we are only concerned with *whether* one individual is better than another, and not with *how much*. The crossover rate in this simple GA is always applied at 100%, and mutation (when used) at one per individual. This one-size-fits-all approach will allow us to concentrate our efforts on representational issues, genetic operators and performance measures (fitness values), substantially reducing the tuning requirements. Other parameters are not quite so easy to standardize as selection, crossover, and mutation rates, however. For example, the population size and stopping criterion are best adjusted to suit the type and size of problem. The simple steady-state GA is outlined in Algorithm 2.

Algorithm 2 A Simple GA

```
Generate $N$ random strings {$N$ is the population size}
Evaluate the performance measure for each string and store it
Apply local search to the offspring {optional}
repeat
  for all strings in the population do
    Each string, in turn, becomes the first parent
    Select a second parent at random
    Apply crossover to produce a single offspring
    Apply mutation {optional}
    Apply local search to the offspring {optional}
    Evaluate the performance measure for the offspring
    if the offspring passes its performance test then
      Then it replaces its weaker parent in the population
    else
      the offspring dies
until stopping condition satisfied
```

At the start of the procedure a population of N random strings is generated. Once the initial population is created, the individual members are evaluated, according to some performance measure (or fitness value). Within the main generation loop, each member of the population is selected in turn and paired in crossover with a second individual, selected (uniformly) at random. The performance measure of the resulting single offspring is then compared to that of its weaker parent. In the simplest version of this algorithm, the new offspring replaces its weaker parent if it is better, otherwise it dies. Later on in the Chapter, when the simulated annealing schedule is introduced, the conditions upon which a new offspring is accepted will be relaxed, in an attempt to maintain diversity within the population. A simple stopping condition is applied throughout the present work whereby the GA runs for a fixed number of generations specified in advance, a generation being defined as N trials of crossover, one led by each member of the population in turn.

6 Set Partitioning Problems

Set partitioning problems (also known as grouping problems [15]) were first introduced in Section 1, and two examples – graph coloring and bin packing – have been referred to briefly as examples in Section 4. Recall that the generic version involves partitioning n objects into m sets, without violating problem specific constraints, so that each object is assigned to exactly one set and the objective function is optimized. The precise nature of the constraints and objective function will vary depending on the variant concerned. In the following Subsections we shall look at some examples of set partitioning problems. To start with we will cover the three main problems addressed in the current Chapter: graph coloring, bin packing and examination timetabling. The Section will then conclude with a brief overview of some other important set partitioning problems.

6.1 The Graph Coloring Problem

As mentioned in Section 4, the graph coloring problem (GCP) involves finding a minimum set of colors for the vertices of a given graph, so that no two adjacent vertices have the same color. The optimum set of colors for a particular graph coloring instance is often referred to in the literature as its *chromatic number*. A legal coloring for a graph with 12 nodes is illustrated in Figure 4. Thus, we aim to partition a set of vertices into the minimum number of color classes, so that each vertex belongs to exactly one color class. The restriction that imposes different colors on adjacent vertices is an example of a *hard constraint*, because no coloring where this condition is violated is allowed.

In many ways the GCP is the archetypal set partitioning problem, because it has probably attracted more interest than any other problem of its type. Indeed, the field is highly competitive and in 1993 the problem was the subject of a Discrete Mathematics and Theoretical Computer Science (DIMACS) implementation challenge [24]. This involved pitting the best algorithms of the day against each other on a collection of large and specially devised difficult benchmark instances. The GCP provides a useful test bed for techniques applicable more widely to real world problems such as timetabling [4], frequency assignment [42], and many others.

6.2 The Bin Packing Problem

Bin packing problems are concerned with packing a given number of items, having different sizes, into a minimum number of equal-sized bins. In this

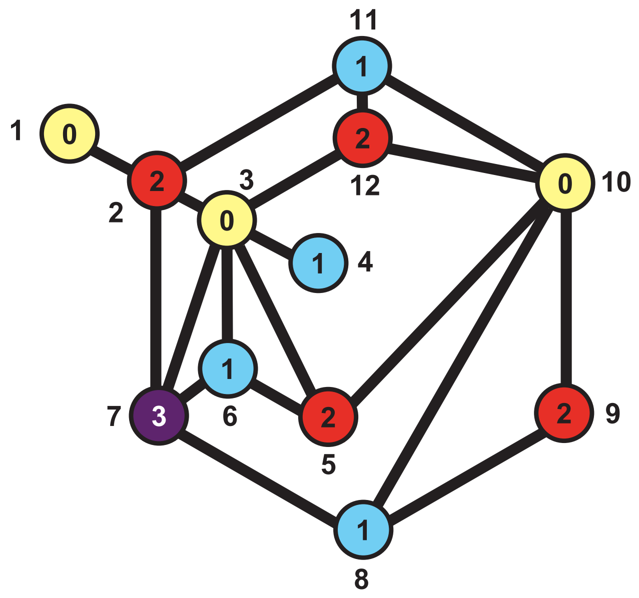


Fig. 4. A legal coloring for a graph with 12 nodes; color labels appear inside the vertices and vertex labels outside

Chapter we shall consider only the simplest of these problems, known as the one-dimensional bin packing problem [30]. In this version of the problem we are concerned only with weights of the items, and not with their areas, volumes or shapes. The goal is to partition a set of items, of differing weights, into a minimum number of bins with equal weight capacity. Like graph coloring, the bin packing problem imposes a hard constraint: the total weight of items occupying any bin must not exceed its weight capacity.

6.3 The Examination Timetabling Problem

The examination timetabling problem involves scheduling a set of examinations into a number of time slots in such a way that the schedule obeys any given constraints and also gives due consideration to any other issues conducive to producing a 'good' timetable. Many different variants exist for this important real-world problem, and the choice of practical solution method will depend on the types of constraints involved and also on the objectives that need to be optimized (see [4] and [40] for more details). In its most basic version, the examination timetabling problem is identical to the graph coloring problem, with the colors representing time slots and vertices representing examinations. In this model, an undirected edge between vertices indicates that at least one student is taking both exams. The goal of this basic version is to schedule all the examinations in the minimum number of time slots, so that there are no clashes – that is, no student is scheduled to take more than one exam in any time slot.

In practice available resources are finite and additional hard constraints are usually imposed, over and above the need to schedule examinations with no clashes. A university has an upper limit on the number of candidates it can seat in a time slot, for instance. It is instructive to note that the seating capacity limitation is identical to the bin packing constraint: items of various sizes being replaced with examinations with various numbers of candidates. Thus, a feasible solution to the timetabling problem that seats all students and avoids all clashes requires the simultaneous solution to the underlying graph coloring and bin packing problems. Other common constraints include:

- Candidates taking a particular exam must not be split across rooms.
- Exam A must be scheduled before exam B.
- Exam A must be scheduled at the same time as exam B (because they contain similar material).
- Exam A must take place in a particular room (because special resources are needed).

In addition to the various hard constraints imposed by different institutions, universities have different views as to what constitutes a 'good' timetable, as opposed to simply a feasible one. Most commonly these desirable but not essential properties (sometimes called *soft constraints*) include some measure of a 'fair spread' of examinations for the students taking them. For example, scheduling students to take two examinations in consecutive time slots is usually avoided if possible. Indeed, some institutions will go much further than this to ensure that as many students as possible have good revision gaps between their examinations. Other desirable properties may consider efficiency or convenience related to the staff involved in marking the papers. For example, examinations with large numbers of candidates may be scheduled early to give more time for marking the scripts. The present study is confined to two hard constraints:

1. avoiding clashes, and
2. keeping within the total seating capacity.

Thus, the version of the examination timetabling problem addressed here is a simple combination of graph coloring and bin packing, our other two test problems.

6.4 Other Set Partitioning Problems

There are many set partitioning problems with great practical application. Here are a few examples:

- Equal piles and assembly line balancing.
- Frequency assignment problem.
- Vehicle scheduling problem.

The equal piles problem was first studied by Jones and Beltramo [25] and involves partitioning N numbers into K subsets, such that the sums of the subsets are as near equal as possible. When applied to assembly lines [38], the equal piles problem seeks to assign assembly tasks to a fixed number of workstations in such a way that the workload on each workstation is nearly equal.

The frequency assignment problem involves the assignment of radio frequencies to reduce interference between transmitters. It can be derived from the graph coloring problem, but the level of conflict (that is, interference) between the nodes is related to their distance apart rather than a simple adjacency conflict [23, 42]. Vehicle scheduling involves assigning customers to vehicles for the pickup and/or delivery of goods [36].

7 Motivation for the Present Study

Set partitioning problems are very challenging for genetic algorithms, and designing effective crossover operators is notoriously difficult. We have already explored some of these issues in Section 4. There are two main challenges:

1. solution infeasibility, and
2. representational redundancy.

Solution infeasibility occurs when candidate set partitions violate problem constraints. As an example, consider the twelve node graph coloring problem illustrated in Figure 2(a) and a direct representation of a legal coloring as shown in Figure 2(b). It is easy to imagine a simple one point (or multi-point) crossover producing an illegal coloring, with some adjacent nodes having the same color. Infeasible solutions to the bin packing problem are easily generated in a similar way, producing overfull bins. *Representational redundancy* can also be a serious problem in set partitioning problems in which the class labels are interchangeable. The arbitrary allocation of color labels for the GCP and bin labels for the BPP establish these two problems in this category. Representational redundancy can artificially inflate the size of the search space and also reduce the effectiveness of crossover operators. Given these difficulties, it is probably not surprising that, despite the predominance of population-based methods, crossover plays a very minor role in many state-of-the-art approaches to solving set partitioning problems. Standard crossover operators are just not very effective at propagating meaningful properties about set membership from parents to offspring.

Of special note, however, are two evolutionary techniques that have recently appeared in the literature. These, unlike their predecessors, include crossover operators that appear to contribute significantly to the overall success of the algorithms. Furthermore, both of these techniques have produced world-beating results for hard literature benchmarks in their respective fields of application. The first of these, known as the *grouping genetic algorithm (GGA)* was developed by Falkenauer [16] for the bin packing problem. The second algorithm, the *hybrid coloring algorithm (HCA)*, by Galinier and Hao [18], was written for the graph coloring problem.

A common feature shared by the two novel crossover operators used in these methods, is the focus on propagating complete partitions, or sets, from parents to offspring. Essentially, both algorithms rely on a direct encoding scheme to assign set membership to items in the manner of Figure 2(b). However with the GGA this direct representation is augmented with a grouping part, which encodes each group (that is, set) with a single gene, and it is to the grouping part only that the genetic operators are applied. In the GGA, the standard part of each chromosome merely serves to identify which items belong to which group. On the other hand, the *greedy partition crossover (GPX)* used in the HCA works on the direct encoding explicitly. The parents take it in turns to contribute a complete color class to the offspring. GPX will always select the largest remaining color class from the chosen parent (hence the term 'greedy' in GPX), and following its transfer to the offspring, all copied vertices will be removed from both of the parents to avoid any duplication of vertices in the offspring.

A disadvantage shared by both the above techniques is their need to repair the infeasible solutions that are inevitably produced by the genetic operators. In addition, successful implementations of these methods also make extensive use of local search to further improve the quality of the solutions. The GGA, for example, uses a powerful backtracking technique adapted from [30] to unpack items from some bins and attempt to repack them more favorably in others. The HCA algorithm of [18] relies on small populations of just five or ten individuals and typically applies several thousand iterations of a tabu search algorithm to each new offspring produced by the evolutionary algorithm (one could speculate on the relative contribution evolutionary part to the methods as a whole). Nevertheless, the researchers in each case present convincing evidence to support the inclusion of their evolutionary components.

Interestingly, the GGA has been adapted by various authors for several other set partitioning problems, including the graph coloring problem [13, 14], the equal piles problem [15], and the course timetabling problem [29]. For each application the choice of problem specific repair and local search heuristics has probably had a major influence on its level of success. To the best of this author's knowledge the HCA algorithm has not yet been adapted for other applications.

Having now introduced the reader to arguably the best known and most successful evolutionary approaches to set partitioning for which crossover plays a significant role, I will now move on to outline the motivation for the present work. It is clear that a major weakness is shared by all evolutionary techniques that rely on direct encoding for set partitioning problems – this being their need to repair infeasible solutions. In addition, we have also noted an extensive use of problem specific local search heuristics in the algorithms we have reviewed above. These are not only time consuming, but they also call into question the relative contribution of the evolutionary algorithm. Furthermore, the repair and local search heuristics used by these methods are not very portable from one set partitioning problem to another, and success seems to be quite variable, depending heavily on the quality of supporting heuristics.

The main aim of the present Chapter is to present a new, and reasonably generic, order based framework suitable for application, with minimum adaptation, to a wide range of set partitioning problems. The clear advantage of using an order based approach is that every permutation is decoded as a feasible solution, meaning no costly repair mechanisms are required following a crossover event, however heavily constrained the problem. On the other hand, it is conceivable that techniques that employ direct encoding will find it increasingly difficult to repair the result of a crossover, the more heavily (or multiply) constrained a problem becomes. Historically the downside of order based genetic algorithms is that, to the best of this author's knowledge, nobody has so far been able to come up with a really effective crossover capable of transmitting useful features from parents to offspring for set partitioning problems. It is hoped that the new crossover operators presented in the present work do something to redress the balance and make order based approaches more competitive. Indeed, the new operators share a key property inspired by the GGA and GPX crossovers: they tend to propagate whole partitions or sets from parents to their offspring.

Perhaps the most innovative feature of the new order based approach is the inclusion of some simple grouping and reordering heuristics to preprocess the chromosomes prior to crossover. The idea is to encourage the transmission of whole set partitions, when a suitably designed crossover is used, in a way that is normally not possible with order based crossovers. We shall see in the next Section that the grouping and reordering heuristics of [9], used in the present study to preprocess the chromosomes prior to crossover, can be applied readily to a range of different set partitioning problems and, unlike many of the repair mechanisms used by direct encoding methods are not restricted to one type. Furthermore, no lengthy local search procedures are required and only a very few iterations of Culberson and Luo's heuristics are needed for preprocessing, Thus, although the exact choice of objective or fitness function will very likely depend on the specific set partitioning application, it is envisaged that the complicated problem specific heuristics and costly backtracking, typical of many other approaches, can largely be avoided. The next Section introduces the heuristics of [9], and explains their power and versatility. It is the opinion of the present author that these very elegant techniques have been rather neglected by researchers.

8 Culberson and Luo's Grouping and Reordering Heuristics

Culberson and Luo's (CL) heuristics were originally devised to solve the graph coloring problem and belong to a family of methods that use simple rules to produce orderings of vertices (or items). Once created, the orderings are presented to a greedy decoder for transformation into legal colorings (or set partitions). Successful ordering heuristics are distinguished by the production of high quality solutions. The simplest and fastest ordering heuristics, unlike the CL heuristics, generate a solution in one go. For the graph coloring problem the best known one-shot techniques determine the orderings by placing the most heavily constrained vertices (namely, those with many edges connecting them to other vertices) before those that are less constrained. While most of these techniques can be described as static, because the orderings remain unchanged during the greedy color assignment process [31, 43], a somewhat more sophisticated technique, known as `DSatur` [2] operates dynamically. Starting with a list ordered in non-ascending sequence of vertex degree, `DSatur` assigns the first vertex to the smallest color label, then it reorders the remainder of the list to favor vertices adjacent to the newly assigned vertex. The algorithm continues in this way, sequentially assigning the lowest available color label to the next vertex on the list, then reordering the remainder, until every vertex has received a color. Thus, `DSatur` assigns colors to unassigned vertices, giving priority to the vertices with the most neighbors already colored, using vertex degree to break the ties.

One-shot ordering heuristics have also been developed for other set partitioning problems, and they operate in a similar fashion to the graph coloring heuristics discussed above. Ordering heuristics for the frequency assignment problem (the assignment of radio frequencies to reduce interference between transmitters), for example, are almost identical to those used for graph coloring (see [23] for a survey). This is probably not surprising, given that frequency assignment is a derivative of the graph coloring problem. Simple versions of the examination timetabling problem also make use of graph coloring heuristics [4]. A popular method for ordering items for the bin packing problem is to place them in non-ascending sequence of their weights. A simple greedy algorithm can then be used to assign the items to the first available bin, bins being identified by consecutive integer labels. This scheme for bin packing is know as the *first fit decreasing weight(FFD)* algorithm [7].

Despite their attractiveness in terms of speed and simplicity, however, one-shot ordering heuristics do not always perform very well in practice, although there are exceptions. FFD can solve many large benchmark bin packing instances to optimality, for example, and DSatur works well on certain graph coloring instances. In other cases though, such algorithms are only useful to supply upper bounds or provide a starting point for a more sophisticated method.

In a different category are the ordering heuristics of Culberson and Luo (CL) [9]. These methods group and rearrange whole color classes (or sets of items), rather than sequencing the vertices individually. Unlike the one-shot methods discussed above, the CL heuristics can be applied repeatedly, leading to the gradual improvement of a solution. Of particular significance is a rare property of the CL heuristics which ensures that it is impossible to get a *worse* coloring by applying any of their reordering techniques to the GCP, and it is possible that a *better* coloring (using fewer colors) may result (see [9] for details). They apply a random mix of various reordering heuristics and call the composite algorithm *iterated greedy (IG).*

Two main stages of IG can be identified:

1. grouping, and
2. reordering.

Figure 5 illustrates some key operations from IG applied to a small graph with 12 vertices and 14 edges. Figure 5(b) gives a typical random permutation of the vertices from Figure 5(a) and also the resulting greedy coloring. Figure 5(c) shows the grouping operation used to sort the list in non-descending sequence of color label, and 5(d) gives the arrangement following the application of one of the CL reordering heuristics called *largest first.* The largest first heuristic rearranges the color classes in non-ascending sequence of their size. Note that the positions of color classes 1 and 2 have been reversed in Figure 5(d). This follows advice in [9] to interchange positions of equal sized color classes. In Figure 5(f) vertices are randomly 'shuffled' within (but not between) the color classes. (Note: shuffle, although mentioned, does not appear to have been extensively used by [9] in the IG algorithm. However it is included here because of its value in the present study). Finally, the greedy algorithm is applied to the new arrangement – Figure 5(f) – and the result is shown in Figure 5(g). Note that vertices 4 and 1 are reassigned lower color labels, leading to a reduction in the size of color class 2. Thus, given an initial permutation of vertices, the IG algorithm can be defined by the following repeating sequence:

1. greedy assignment,
2. grouping of color classes,
3. reordering of complete color classes,
4. shuffle within each color class (optional).

Various properties of the color classes were assessed by [9] as criteria for reordering:

1. *Reverse*: reverse the order of the color classes
2. *Random*: place the groups in random order
3. *Largest first*: place the groups in order of decreasing size (Figure 5(d))
4. *Smallest first*: place the groups in order of increasing size

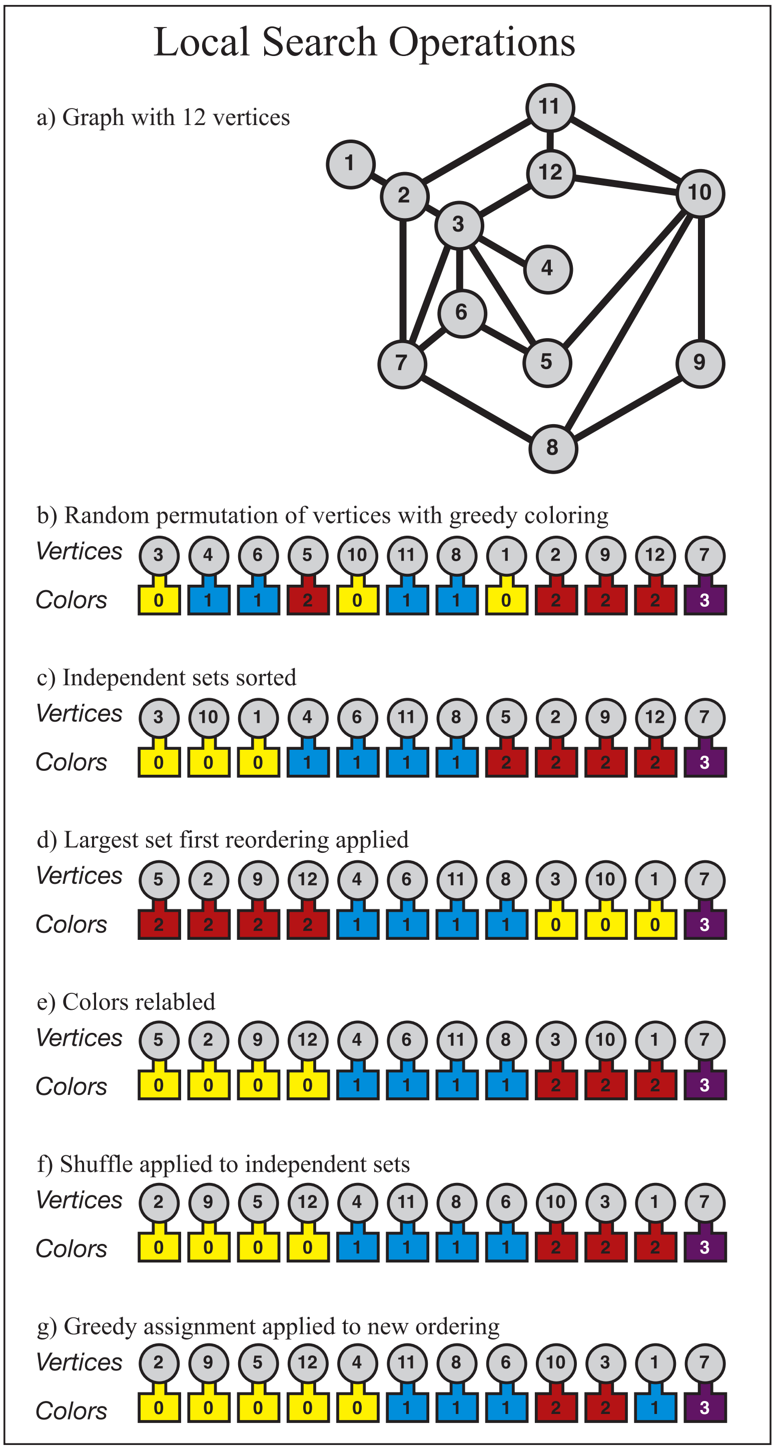

Local Search Operations
a) Graph with 12 vertices
1 2 3 4 5 6 7 8 9 10 11 12
b) Random permutation of vertices with greedy coloring
Vertices 3 4 6 5 10 11 8 1 2 9 12 7
Colors 0 1 1 2 0 1 1 0 2 2 2 3
c) Independent sets sorted
Vertices 3 10 1 4 6 11 8 5 2 9 12 7
Colors 0 0 0 1 1 1 1 2 2 2 2 3
d) Largest set first reordering applied
Vertices 5 2 9 12 4 6 11 8 3 10 1 7
Colors 2 2 2 2 1 1 1 1 0 0 0 3
e) Colors relabled
Vertices 5 2 9 12 4 6 11 8 3 10 1 7
Colors 0 0 0 0 1 1 1 1 2 2 2 3
f) Shuffle applied to independent sets
Vertices 2 9 5 12 4 11 8 6 10 3 1 7
Colors 0 0 0 0 1 1 1 1 2 2 2 3
g) Greedy assignment applied to new ordering
Vertices 2 9 5 12 4 11 8 6 10 3 1 7
Colors 0 0 0 0 0 1 1 1 2 2 1 3

Fig. 5. Various operations by [9] used in the local search procedure

5. *Increasing total degree*: place the groups in increasing order by the total degree of the group
6. *Decreasing total degree*: place the groups in decreasing order by the total degree of the group

The favored combination turned out to be largest first, reverse and random, used in the ratio 50:50:30. Although [9] applied their IG algorithm to the GCP, many of the reordering heuristics are equally applicable to other set partitioning problems. Reverse, random, largest first and smallest first, for instance, can be used for the bin packing problem. Note however, that heuristics based on vertex/item degrees have no meaning in this context, so the increasing and decreasing total degree heuristics are not appropriate for bin packing. Nevertheless, an alternative measure can be used: the total weight of items in each bin. In this way the two CL heuristics that sequence on the total degrees for each group can be replaced with heuristics that do their ordering on the basis of the total weights of items in each bin. CL heuristics can also be applied to a simple version of the examination timetabling problem where the only considerations are to avoid clashes and/or seating capacity violations. As mentioned in Section 6.3, clash avoidance and seating capacity compliance simply represent underlying graph coloring and bin packing problems, respectively.

The crucial feature that determines the applicability of the CL reordering heuristics to a particular set partitioning problem is whether a solution is changed if the groups are simply re-labelled. For convenience, groups are usually identified by integer labels, to represent their color class, bin ID or time slot, and so forth. With graph coloring and bin packing it does not matter which integer label is assigned to which group – it will not change the total number of colors or bins required. On the other hand, if revision gaps are required in examination timetables, the sequence of integer labels will be relevant, and will correspond to a sequence of time slots; by contrast, time slots can be shuffled into any sequence to avoid clashes and comply with seating arrangements. Interestingly, the reverse heuristic maintains the relative positions of time slots. The frequency assignment problem has similar limitations, and the applicability of reverse has been proven in [42].

We will now look at some new crossover variations that attempt to preserve color classes, when used in conjunction with the grouping and sorting heuristics described above.

9 Modifications to a Standard Order Based GA for Set Partitioning

Genetic algorithms require crossover techniques that preserve *building blocks* [19] appropriate to the problem at hand. A building block can be viewed as a group of elements on a chromosome that 'work together' to produce or influence some identifiable feature in the solution. For example, the 'sets' in set partitioning problems come into this category, and thus it makes sense to use a crossover that preserves them. In the present work CL grouping and reordering heuristics are used to preprocess the order based chromosomes to make it easier to preserve the set groupings. Two new crossover operators, POP and MIS (first described in [34]), seem to be particulary effective in maintaining this group integrity when used in conjunction with CL heuristics. Indeed, results for these operators are impressive when compared to those obtained using other order based crossovers for the graph coloring problem [34]. Further evidence in support of these operators is presented later in the Chapter. Note that no special modifications were necessary for order based mutations, and that inversion and insertion mutations proved the most useful in the present study.

The new crossover operators are compared with three selected order based operators of historical importance: cycle crossover (CX) [35], uniform order based crossover (UOBX) [11], and merging crossover (MOX) [1]. CX, OX and MOX have already been described in Section 4. UOBX was developed from OX by [11] with the GCP in mind and is good at preserving relative positions and orderings. CX is good at preserving absolute positions of vertices, and every vertex in the offspring list will occur in exactly the same position in one or other of its parents. CX has proven effective for the frequency assignment problem [42]. As mentioned previously, MOX is good at preserving relative positions. I will now outline the two new crossover operators, POP and MIS.

Permutation Order Based Crossover (POP)

Permutation order based (POP) crossover uses ideas from the well known order crossover (OX) [35], described earlier, but at the same time it tries to emulate the basic one point crossover of the 'standard' bit string GA, which simply selects two parents and a cut point. The first portion of parent 1 up to the cut point becomes the first portion of offspring 2. However, the remainder of offspring 2 is obtained by copying the elements absent from the first portion of the offspring in the same sequence as they occur in parent 2 (see Figure 6). The same idea was used in [8], although the crossover was not given a specific name. However, the present implementation relies on the CL heuristics for preprocessing the chromosomes, without which it did not work very well, as demonstrated in [34]. We will identify two variants of POP: POP1 and POP2. These differ slightly in the way the cut point is selected: for POP1 it is chosen at random and can appear anywhere in the list, but for POP2 the cut point is restricted to a boundary between two set groupings. Of course application of POP2 is dependent on having previously sorted the color classes.

Merging Independent Sets Crossover (MIS)

Merging independent sets (MIS) is a new crossover, adapted from MOX. It requires that the color sets are first grouped together in both of the parents,

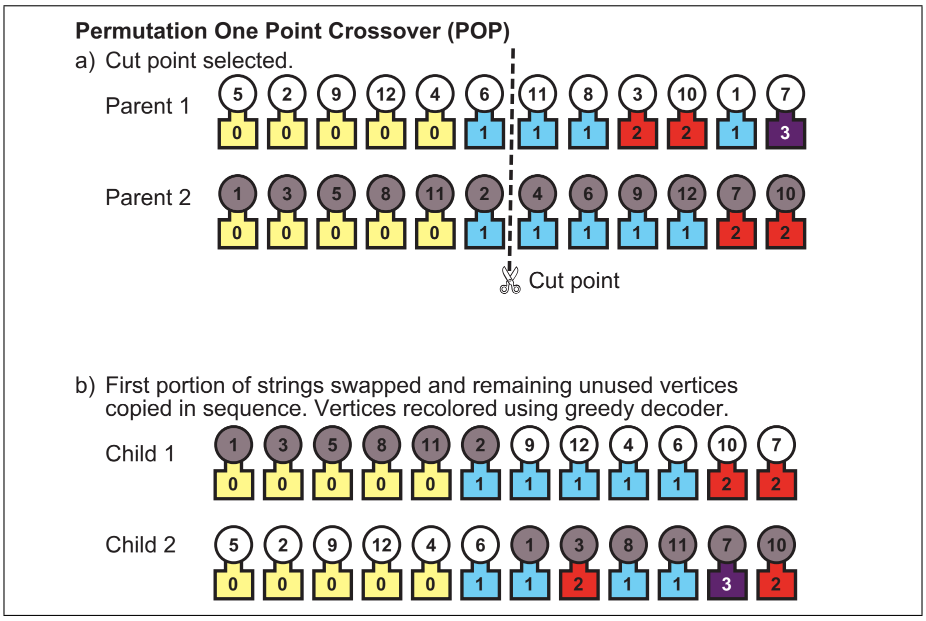

Permutation One Point Crossover (POP)
a) Cut point selected.
Parent 1
Parent 2
Cut point
b) First portion of strings swapped and remaining unused vertices copied in sequence. Vertices recolored using greedy decoder.
Child 1
Child 2

Fig. 6. POP Crossover

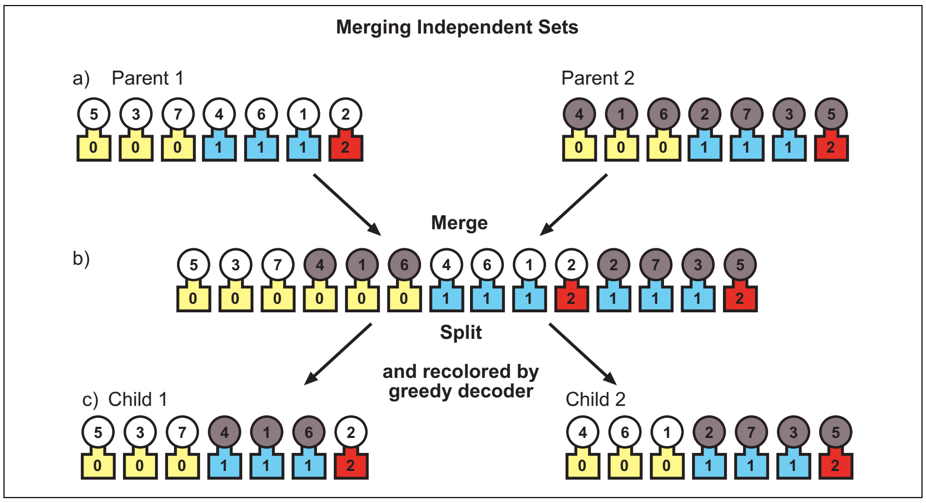


Fig. 7. Merge independent sets crossover, MIS

as illustrated in Figure 7(a). MIS then proceeds in the same way as MOX, but whole color sets are copied from the parents to the merged list in one go (Figure 7(b)), rather than individual vertices. The merged list is split in exactly the same way as for MOX, with the first occurrence of each vertex appearing in the first offspring and the second occurrence in the second offspring, reapplying the greedy decoder to the new offspring (Figures 7(c)). The idea of MIS is to better preserve the parents' color classes than MOX.

9.1 Performance Measures/Fitness Values

The objective function (namely, the value we are trying to optimize) is not always the best measure of progress for an optimization algorithm to use. For example, a common objective function for set partitioning problems is to count the number of classes – for instance, the number of colors and bins respectively – for graph coloring and bin packing. Unfortunately, as previously mentioned, representational redundancy can mean that for a given number of classes, a very large number of different assignments (of vertices to colors, or items to bins) is possible. For this reason, we will use progress measures that attempt to distinguish between 'good assignments' and 'bad assignments', for a given class count. We will start by looking at a performance measure that is generally applicable to most set partitioning problems, and then we will examine some more specific measure that can be used for particular problems.

General Set Partitioning

The performance measure in Eqn.(2) was introduced by [9] for the GCP. However, it is equally applicable to other set partitioning problems.

$$P_1 = \sum_1^n c_i + nc \tag{2}$$

Interpreting this formula for the GCP, Eqn.(2) shows the *coloring sum* (that is, $\sum_1^n c_i$, where c_i is the color assigned to vertex i) added to the term nc, where n is the number of vertices and c the number of colors. For bin packing, $\sum_1^n c_i$ represents the 'bin sum', with the bins numbered consecutively, $\{1, 2, 3, ..., c\}$, and item i assigned to bin number c_i. The other term in Eqn.(2), nc, is simply the number of items multiplied by the number of bins. A disadvantage of this performance measure is that it is sensitive to color (or bin) class labelling, and color classes or bin assignments need be sequenced so that the smallest integer label is assigned to the largest class, and the second smallest integer to the second largest class, and so on, for the measure P_1 to work effectively. This performance measure was used in an earlier study by the present author [34].

Graph Coloring

The following equation was devised by Erben [14].

$$P_2 = \frac{1}{c}\sum_1^c D_j^2 \tag{3}$$

In Eqn.(3) $D_j = \sum_{i \in S_j} d_i$ represents the 'total degree' for group j with d_i denoting the vertex degree of the i_{th} node. Unlike P_1, P_2 is insensitive to class labelling, and in Eqn.(3) color classes having a high total degree are favored.

Bin Packing

The final formula we will consider is due to Falkenauer [16].

$$P_3 = \frac{\sum_1^c (W_j/C)^2}{c} \tag{4}$$

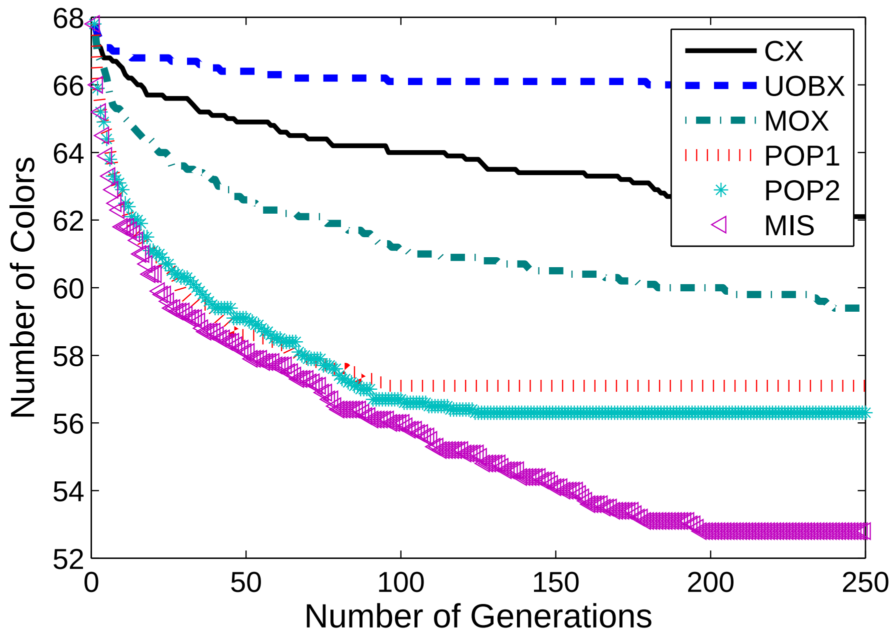
Eqn.(4) has a similar structure to Eqn.(3), but the important class measurement is total weight of items in bin j, $\sum_{i \in S_j} w_i$, where w_i is the weight of item i in bin j. The bin capacity is denoted by C.

9.2 Comparing Order based Crossovers

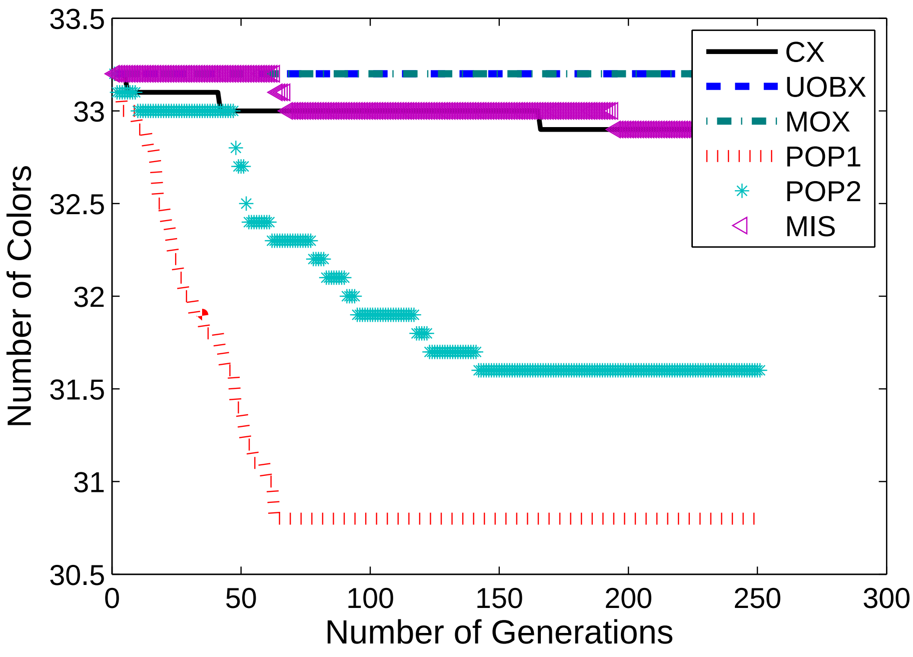
Experiments were conducted to assess the viability of the various crossover operators for the GCP and the BPP. The simple steady-state GA, outlined in Algorithm 2, was used as a framework for this, omitting mutation. Two DIMACS (Discrete Mathematics and Theoretical Computer Science Implementation Challenge) benchmarks – `DSJC500.5` and `le450_25c` – were used to demonstrate the performance of the crossovers on the GCP, and `N4C3w2_A` and `N4C3W4_A` from Scholl and Klein were used for the BPP (see the resources Section for the data sets). The fitness function devised by Erben (Eqn.(3)) was used for the GCP and Falkenauer's fitness function was chosen for the BPP. A population size of 300 was used for each experiment, and the GA run for 250 generations. A single iteration of local search steps, similar to those illustrated in Figure 5, immediately followed each application of crossover. Recall that local search is based on the CL heuristics and consists of a 'grouping' and a reordering phase. However, the 'largest first' heuristic was changed to reflect features of the different fitness functions that were used for the GCP and the BPP. In the case of the GCP, the classes were sequenced according to the sum of vertex degrees in each color class (that is, decreasing total degree), rather than just counting the number of vertices belonging to each group. For the BPP the classes (which correspond to the contents of the various bins) were sequenced in non-ascending order of bin weights.

Results for all the experiments are displayed graphically as 'best-so-far' curves averaged over 10 replicate runs (see Figures 8 and 9). Clearly the best results are obtained with MIS on the GCP and POP1 on the BPP. Analysis of variance tests show highly significant differences in the performance of the various crossover operators at the 0.01% level.

The reader is referred to [34] for a more rigorous set of comparisons for the GCP. Results presented in this earlier paper also demonstrate the important contribution of the grouping (or sorting) and reordering heuristics.

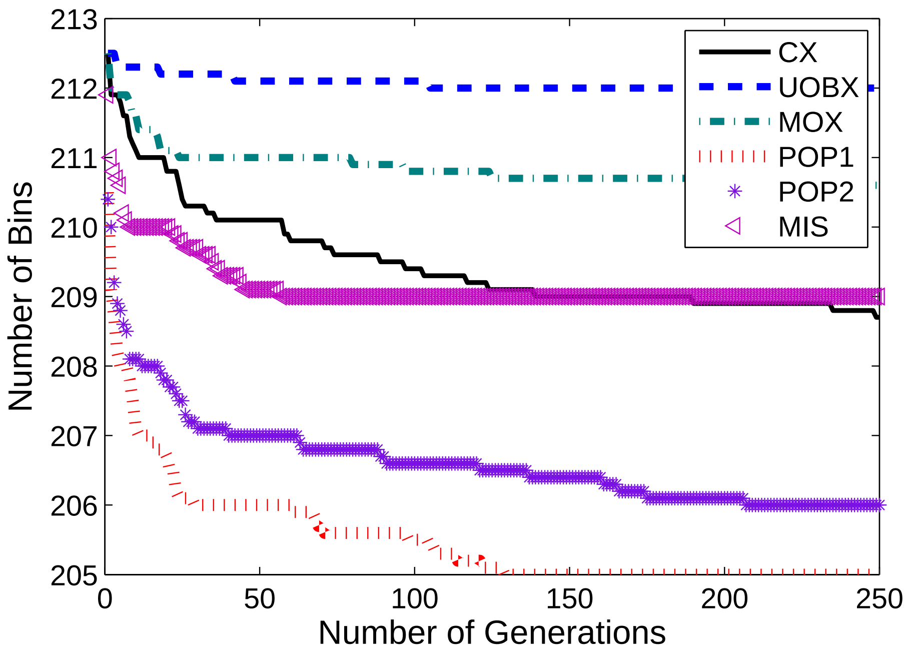


(a) DSJC500.5

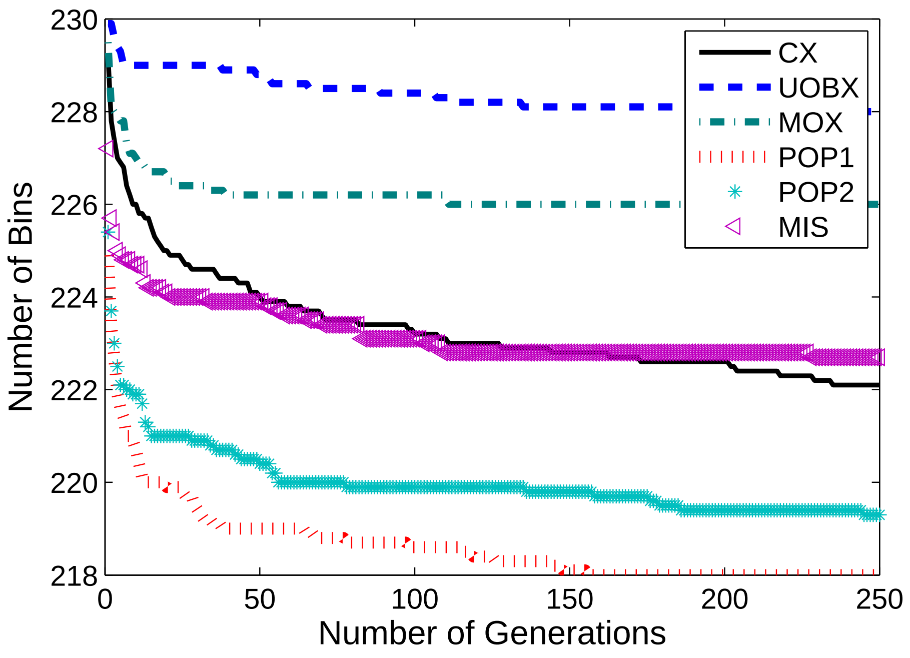


(b) le450_25c

Fig. 8. Comparing order based crossovers with sorting of independent sets and largest total degree first



(a) N4C3W2_A



(b) N4C3W4_A

Fig. 9. Comparing order based crossovers with sorting of independent sets and largest total weight first

9.3 The Genetic Simulated Annealing Algorithm

Having assessed the relative performance of the various crossover operators, the next stage is to apply a suitably adapted GA to literature benchmark instances for various set partitioning problems, to see how the approach will compete with other published algorithms. In order to obtain really good solutions, it is necessary to balance population diversity with GA convergence, and consider larger populations and/or longer runs than were needed for comparing the crossovers. Clearly, the addition of a mutation operator will probably help maintain diversity within the population, giving a better opportunity to explore the search space and helping to avoid premature convergence. Mutation was deliberately left out of the previous experiments when we were assessing the crossovers.

Several of the mutation operators discussed in Section 4 were tried in some pilot studies: insertion and swap mutation, as well as scramble sublist. Of these, insertion produced the best results with the successful crossovers, producing good solutions quickly. Unfortunately these results were not quite world class, and despite the useful contribution of the mutation operator, the population tended to lose its variability towards the end of the run. In an attempt to improve matters, different ways of injecting extra variability into the population were explored.

Periodic restarts was the first of these ideas to be tried. This involved temporarily halting the GA, once it stagnated. Various 'super-mutation' operations were then selected stochastically and applied to the individual permutations in the population, in an attempt to inject some new variability. Operations that were tried include Davis' scramble subset and Holland's inversion (see Section 4) as well as a simple 'delete-and-append' operation, which deletes part of a string and then appends this deleted section to the end of the string. Once this super-mutation stage had been completed, the GA was restarted. Although this approach proved to be very successful on some instances, it required unacceptably long run times on others. A more efficient approach was subsequently found which involves making a small change to the replacement criterion in Algorithm 5: instead of simply replacing the weaker parent by its offspring when offspring is better than its parent, a simulated annealing cooling schedule was introduced, which allows a parent to be replaced occasionally by a poorer offspring. The main features of a simulated annealing cooling schedule are outlined below.

What is Simulated Annealing?

In physics the term 'annealing' refers to the very slow cooling of a gas into a crystalline solid of minimum energy configuration. Simulated annealing algorithms (SAs) [27] attempt to emulate this physical phenomenon. The process usually begins with the generation of a random solution, and this will act as the initial focus of the search. The SA will then make a very small change to a copy of this solution, generating a *neighborhood solution*, in an attempt to produce an improvement. If a better solution is found, then the improved solution will replace the original as the focus. However, it is well known amongst researchers that a simple hill climbing search such as this (accepting only 'forward' moves and never 'backward' ones) can easily become trapped in a local optimum. The main strength of an SA algorithm is its potential to escape from such traps. This is achieved by the occasional acceptance of a neighborhood solution that is somewhat worse than the current focus. The rate at which this is allowed to occur is carefully controlled using an *acceptance probability*, and this is used to determine whether or not a newly generated neighborhood solution will replace the current focus solution. In general, the poorer the new solution, the less likely it is to be accepted. However, the analogy with the physical situation requires that inferior solutions should be less likely to be accepted as the search progresses. Initially the algorithm will probably accept almost anything, but towards the end of the search, the algorithm will behave more like hill climbing, accepting inferior solutions only on very rare occasions. Values for the acceptance probability – *prob* – for a minimization problem, are evaluated using Eqns.(5) and (6). Δ represents the difference between the objective functions (or costs) of the new solution, $C(S')$, and the focus solutions, $C(S)$. Note that the value of *prob* depends on the value of Δ and also on T, the current temperature, which is determined by the cooling schedule.

$$\Delta = C(S') - C(S) \tag{5}$$

$$prob = min(1, e^{-\Delta/T}) \tag{6}$$

The new solution is accepted with probability 1 if $\Delta \leq 0$ (in other words, if the neighborhood solution is better than S) and with probability $e^{-\Delta/T}$ if $\Delta > 0$ (that is, if the neighborhood solution is worse than S). Throughout the execution of an SA algorithm, the temperature T is progressively lowered.

The Genetic Simulated Annealing (GSA) Implementation

For the present GSA implementation for set partitioning problems, the precise annealing schedule is determined from user-specified values for the number of cooling steps and the initial and final solution acceptance probabilities. We use n cooling steps to correspond to the number of generations, so that the temperature is decreased between each generation. Thus, knowing n and the initial and final acceptance probabilities, P_0 and P_n, as well as an additional parameter M that signifies an initial number of random trials, the starting temperature T_0, the final temperature T_n, and the cooling factor α can be calculated, as indicated below.

$$\Delta_i = Perform(offspring) - Perform(weaker) \tag{7}$$

$$\Delta_{ave} = \frac{\sum_{i=1}^{i=M} \mid \Delta_i \mid}{M} \tag{8}$$

$$T_0 = -\frac{\Delta_{ave}}{\log P_0} \tag{9}$$

$$T_n = -\frac{\Delta_{ave}}{\log P_n} \tag{10}$$

$$\alpha = \exp^{\frac{\log T_n - \log T_0}{n}} \tag{11}$$

Please note that Δ_{ave} (Eqn.(8)) is obtained by applying the genetic operators and also local search, if appropriate, to M randomly selected pairs of individuals from the initial population. In this way the performance measures of M offspring are compared with those of their weaker parents to obtain an estimate of Δ_{ave}. This estimate is then used to determine the starting temperature, the final temperature and the cooling schedule. The offspring generated during this initialization phase are discarded.

Algorithm 3 provides an outline of the GSA. Although the exact implementation details for the GSA, such as choice of crossover and mutation operators, and type of local search, vary according to the nature of the problem, the basic framework remains the same. Worthy of note is the simple adjustment made to the calculation of Δ_i, necessary because all the fitness functions (in other words, objective functions) used for the problems in the present study involve maximization, yet simulated annealing requires that the objective functions are minimized. We simply set $\Delta_i = Perform(weaker) - Perform(offspring)$ instead of $Perform(offspring) - Perform(weaker)$.

10 Results on Literature Benchmarks

The versatility of the new techniques will now be demonstrated on some literature benchmarks for graph coloring, bin packing and timetabling. Except where otherwise stated, the following parameter settings were used for the GSA: $M = 100$ (the number of preliminary trials to help establish the starting temperature), $P_0 = 0.999$ and $P_n = 0.0001$ (the starting and ending probabilities, respectively, of accepting an inferior offspring with an average magnitude of deviation in value from its weaker parent). Population sizes of 200 were used and ten replicate runs carried out on each benchmark instance, with average and best values quoted for the solutions in the results tables. Values of n, the number of generations, vary with different problem instances, as do precise details of the genetic operators and CL heuristics used.

Algorithm 3 Genetic Simulated Annealing (GSA)

```
Generate $N$ random strings {$N$ is the population size}
Evaluate the performance measure for each string and store it
Apply local search to the offspring {optional}
Initialize data, {obtaining $T_0$, $T_n$ and $\alpha$}
$S = S_0$
$T = T_0$
for $n$ generations do
  for all strings in the population do
    Each string, in turn, becomes the first parent
    Select a second parent at random
    Apply crossover to produce a single offspring
    Apply mutation to the offspring
    Apply local search to the offspring {optional}
    Evaluate the performance measure for the offspring,
    $\Delta = Perform(weaker) - Perform(off)$
    $P_t = min(1, \exp^{-\Delta/T})$
    if $random(0,1) \leq P_t$ then
      offspring replaces weaker parent
    else
      the offspring dies
  $T = \alpha \times T$
```

10.1 Graph Coloring

The previously mentioned DIMACS benchmarks [24] provide most of the test instances for the present study, and are dealt with first. Further experiments are then reported on two special types of graphs, based around cliques. All test instances were chosen because they have been reported as 'difficult' by previous researchers.

The DIMACS Benchmarks

Seven benchmark instances were taken from the DIMACS challenge benchmark set, [24]. `D250.5, D500.5` and `D1000.5` are *random graphs* with edge density 0.5 and unknown chromatic number. `Le450_15c` and `le450_25c` are *Leighton graphs* with 450 vertices, and `flat300_28` and `flat1000_76` are *flat graphs* with 300 and 1000 vertices, respectively. The flat and Leighton graphs are structured with known chromatic numbers of 15, 25, 28 and 76, as indicated.

MIS crossover was selected because it worked well in the GSA for most of the instances. However, POP1 proved better for `le450_25` so this crossover was used for this instance only. 'Inversion' acted as the mutation operator, which involved inverting the substring between two randomly selected cut points. Erben's fitness function (Eqn.(3)) provided the performance measure, and three iterations of the local search, based on Figure 4, seemed to be sufficient for the GCP. Increasing the number of iterations slowed the algorithm down considerably without improving the results. The local search was modified a little from Figure 4 however (as was the case when comparing the crossovers), with the 'decreasing total degree' heuristic replacing the 'largest first'. This was done to make the reordering criterion tie in better with Erben's fitness function: both encourage the formation of classes with high values for total vertex degree. Results for the seven DIMACS benchmarks are presented in Table 1.

Table 1. Results for Genetic Simulated Annealing on graph coloring Instances

Instance	Order Based GSA				Mut GSA		It Greed		DSat	Best
	# Gens	Time	Mean	Min	Mean	Min	Mean	Min		known
DSJC250.5	2000	314	29.1	**29**	31.6	31	30.0	30	37	*28*
DSJC500.5	3000	1895	49.9	**49**	56.4	56	53.8	53	65	*48*
DSJC1000.5	5000	11706	87.2	**87**	101.0	100	98.0	97	115	*83*
le450_15c	500	190	***15***	***15***	24.9	24	24.0	24	23	*15*
le450_25c	2000	854	29.3	**29**	29.9	**29**	29.0	**29**	**29**	*26*
flat300_28	1000	205	32.6	**32**	36.0	36	34.3	34	42	*31*
flat1000_76	5000	11711	86.3	**85**	100.0	99	97.4	96	114	*83*

In Table 1 results for the GSA (columns 4 and 5) are compared with those obtained running a mutation only version (columns 6 and 7), which incorporates all the same parameters and features of the GSA but does not have the crossover. The iterated greedy algorithm was also tried, and the results for this can be found in columns 8 and 9. The penultimate column contains the `Desatur` result for each instance and the best known results are listed, for comparison purposes, in the final column. The best known results were obtained by [18] using their hybrid evolutionary algorithm for graph coloring (HEA). Ten replicate runs were carried out for the GSA, mutation only GSA and also iterated greedy on each benchmark instance.

In more detail, column 2 gives the number of generations for the GSA (and also the mutation only version), and column 3 the average run time in seconds. The average and best results for the GSA and the mutation only version are presented in columns 4, 5, 6 and 7. The results for iterated greedy in columns 8 and 9 are produced by running this algorithm for the same length of time as the GSA, namely 314 seconds for `DSJC250.5`, 1895 seconds for `DSJC500.5`, and so forth. Bold font is used to highlight where the best results have been obtained for the current set of experiments, and italic font indicates the best known results.

Clearly, the GSA outperforms `Desatur` and iterated greedy on most instances, and the version with crossover works much better than the one without. However, the GSA results do not quite match the results obtained by the HEA algorithm, which is clearly state-of-the-art. Nevertheless, the benchmarks are tough and the results are generally very good, when compared to many other published results for these benchmarks. The HEA algorithm uses several thousand tabu search iterations following the creation of each new offspring in the population, thus it is a very different type of algorithm from the current order based GSA. The new order based approach introduced in the present Chapter is presented largely for its generic qualities, and its potential for a wide range of set partitioning problems.

Some Further Experiments

In addition to the DIMACs benchmarks, further experiments were undertaken on two special types of graphs first presented by [39] and used by [14] to test his version of the grouping genetic algorithm. These instances are all arranged around *cliques* – that is, complete subgraphs (with each vertex connected to every other vertex by an edge) present within each instance. The two types are called the *pyramidal chain* and the *sequences of cliques*. In all cases the chromatic number c is known beforehand. A simple order based GA (in other words, without the GSA cooling schedule) easily solved all instances tried: one pyramidal chain instance with $c = 6$, 20 cliques, 60 nodes and 200 edges; seven instances of sequences of cliques with $c = 6$, 20 cliques and 120 nodes. All the instances that could be found were kindly supplied by Erben, using email attachment. The pyramidal chain example needed about 1,300 evaluation steps of the order based GA, a similar number to that was reported by [14] for the grouping genetic algorithm. For the 7 sequence of cliques examples, however, the order based approach needed a maximum of 10,000 evaluations, which is less than the 150,000 reported in [14].

10.2 Bin Packing

Two sources of data provide the benchmark instances for the bin packing tests. Once again, an order based GSA is used with inversion providing the mutation operator. POP1 is chosen as the crossover operator for bin packing, because it produced better results than MIS in some preliminary tests. The fitness function adopted for bin packing is the one devised by Falkenauer (see Eqn.(4)) which favors bins that are as full as possible. A single iteration of local search, following each crossover, seems to be sufficient for bin packing. This time the 'largest first' reordering heuristic in Figure 4 is replaced by a reordering scheme based on largest total bin weight (or fullest bin).

Many authors have noted the efficiency of the simple bin packing heuristic algorithm, first fit decreasing weight, FFD, (briefly discussed earlier in Section 8). For large numbers of items the worst case solution is $\frac{11}{9} \times OPT$, [7], where 'OPT' refers to the optimum solution. However, best case and average case behavior tend to be very much better than this, with FFD easily solving many problems to optimality. Schwerin and Wäscher [41] coined the phrases

FFD-easy, *FFD-hard* and *extremely FFD-hard* to help classify problems with different properties according to the proportion solved by FFD:

- FFD-easy: 80 - 100 % solved
- FFD-hard: 20 - 80 % solved
- extremely FFD-hard: 0 - 20 % solved

The Data Sets of Scholl and Klein (SK)

Scholl and Klein provide three data sets (see the resources section at the end of the Chapter) with a total of 1,010 bin packing instances on their web site, with optimum solutions (that is, minimum number of bins) given for each. All these instances have been generated in groups of either 10 or 20, so that instances within groups have the same properties regarding bin capacities, numbers of items and ranges of weights for the items. The majority of these instances are easily solved, however. Indeed, the present author found optimum solutions to 781 of the 1,010 instances using FFD. All the test data selected for this Chapter belong to classes where FFD has solved zero instances. The first two instances, `N4C3W2_A` and `N4C3W4_A`, are taken from SK's first data set. `N4C3W2_A` has N = 500 items, bin capacity C = 150, and item weights varying uniformly between 20 and 100. `N4C3W4_A` has the same values for N and C, but the item weights are between 30 and 100. The next six instances are all taken from the second SK data set. All these instances have N = 500 items and bin capacity = 1,000. The average weight per item varies according to W, with W1 = 1000/3, W2 = 1000/5, W3 = 1000/7, and W4 = 1000/9. Thus, for `N4W1B1R0` we would expect to find a maximum of 3 items in each bin, and for `N4W2B1R0` about 5 items per bin, and so on. The value of B indicates the maximum deviation of single weight values from the average weight. B1 = 20%, B2 = 50%, and B3 = 90%. For the remaining instances (HARD0 - HARD9 from data set 3) the parameters are: N = 200 items, capacity C = 100,000, and weights range from 20,000 to 35,000. The number of items per bin lies between 3 and 5.

Results presented in Table 2 show that the both the GSA and mutation only GSA are able to solve most of the instances to optimality, and get very close for the others. For the SK data, crossover does not appear to make a significant contribution however, although the GSA clearly produces better solutions than FFD and iterated greedy. Bold and italic font is used as previously, to highlight the best results for the current experiments and the best known results, respectively.

Falkenauer's Data Sets

Falkenauer's data sets are also included here to make comparisons possible with state-of-the-art algorithms, such as the MTP algorithm by Martello and Toth [30] and the hybrid grouping genetic algorithm (HGGA) of Falkenauer himself [16]. Falkenauer generated two type of data:

Table 2. GSA results on Bin Packing Instances

Instance	Order Based GSA				Mut GSA		It Greed		FFD	Optimum
	# Gens	Time	Mean	Min	Mean	Min	Mean	Min		
N4C3W2_A	2000	324	**204**	**204**	**204**	**204**	204.6	**204**	206	*203*
N4C3W4_A	2000	340	**217**	**217**	**217**	**217**	219	219	220	*216*
N4W1B1R0	1000	134	***167***	***167***	***167***	***167***	184	184	184	*167*
N4W2B1R0	1000	89	**102**	**102**	**102**	**102**	107.3	105	109	*101*
N4W3B1R0	1000	73	***71***	***71***	***71***	***71***	73	73	74	*71*
N4W3B2R0	1000	73	***71***	***71***	***71***	***71***	***71***	***71***	72	*71*
N4W4B1R0	1000	63	***56***	***56***	***56***	***56***	56.8	***56***	58	*56*
HARD0	1000	30	***56***	***56***	***56***	***56***	59	59	59	*56*
HARD1	1000	30	***57***	***57***	***57***	***57***	58.7	***57***	60	*57*
HARD2	1000	30	**57**	**57**	**57**	**57**	59	59	60	*56*
HARD3	1000	30	**56**	**56**	**56**	**56**	57.7	57	59	*55*
HARD4	1000	30	***57***	***57***	***57***	***57***	58.9	58	60	*57*
HARD5	1000	30	***56***	***56***	***56***	***56***	57.8	57	59	*56*
HARD6	1000	30	***57***	***57***	***57***	***57***	58.6	57	60	*57*
HARD7	1000	30	***55***	***55***	***55***	***55***	58	58	59	*55*
HARD8	1000	30	***57***	***57***	***57***	***57***	58.8	58	60	*57*
HARD9	1000	30	***56***	***56***	***56***	***56***	58.7	58	60	*56*

1. uniform item size distribution, and
2. triplets.

Both types were produced along similar lines to the hard instances in the SK data sets. For the first set of data, items are uniformly distributed between 20 and 100, with bin capacity 150. Falkenauer generated instances with varying numbers of items (120, 250, 500 and 1,000), producing 20 examples of each, making 80 instances of this type in total.

With the second set of data, item weights were drawn from the range 0.25 to 0.5 to be packed in bins of capacity 1. Given the improbability of finding four items of weight exactly 0.25, it follows that a well-filled bin will normally contain one big item (larger than a third of the bin capacity) and two small ones (each smaller than a third of the bin capacity), which is why the instances are referred to as 'triplets'. What makes this class difficult is that putting two big items or three small items into a bin is possible but inevitably leads to wasted space, because the remaining space is less than 0.25, and thus cannot be filled. Falkenauer generated instances with known optima, based on a bin capacity of 1,000, as follows. An item was first generated with a size drawn uniformly from the range [380, 490], leaving space s of between 510 and 620 in the bin. The size of the second item was drawn uniformly from [250, s/2]. The weight of the third item was then chosen to completely fill the bin. This process was repeated until the required number of items had been generated. The number of bins needed was subsequently recorded. Triplets were generated with 60, 120, 249 and 501 items – 20 instances of each. Optimum solutions are 20, 40, 83 and 167 bins, respectively.

Table 3 compares the results obtained by running the GSA with those published in [16]. Results for FFD and iterated greedy are also included. Each algorithm was run only once on each instance, and population sizes for the GSA were set at 100, the same as was used for HGGA. The GSA was run for the same number of generations as the HGGA, 2,000 for the first two data sets, 5,000 for the next two, 1,000 for the first two triplet groups, and 2,000 for the last two.

Once again, the GSA clearly outperforms FFD and iterated greedy (run for the same length of time as the GSA), and it would appear that POP1 crossover makes a useful contribution because the GSA with crossover does slightly better than the GSA without it. The GSA clearly performs better than MTP in all columns. However, apart from the uniform instances with 120 items, the HGGA performs slightly better than the GSA. It is worth noting, however, that the HGGA employs specialized backtracking in its local search, that will unpack up to 3 items per bin and try to repack. On the other hand the order based GSA does not use backtracking and runs very fast – requiring one or two seconds for the smaller problems and up to a maximum of about 22 minutes for some of the uniform problems with 1,000 items. Furthermore, the representation, operators and CL heuristics used in the GSA are more generic, and can equally be applied to other set partitioning problems, as previously mentioned.

Table 3. Results for Falkenauer's data sets

Type	# items	FFD	MTP	HGGA	GSA	Mut GSA	IG
Uniform	120	49.75	49.15	49.15	*49.1*	49.45	49.4
Uniform	250	103.1	102.15	*101.7*	101.9	102.5	102.3
Uniform	500	203.9	203.4	*201.2*	201.5	202.5	202.65
Uniform	1000	405.4	404.45	*400.55*	401.3	402.7	403.7
Triplets	60	23.2	21.55	*20.1*	21	21	22.25
Triplets	120	45.8	44.1	*40*	41	41	44.95
Triplets	249	95	90.45	*83*	84	84.15	93.7
Triplets	501	190.15	181.85	*167*	168	169.1	188.6

10.3 Timetabling

Recall the version of the timetabling problem addressed here combines bin packing with graph coloring. The maximum number of seats per time slot corresponds to the BPP constraint, and the avoidance of clashes to the GCP constraint. Given a set of students to be examined for different courses, we wish to schedule the examinations so that all clashes are avoided and the seating capacity is not exceeded in any time period. A selection of real world instances from Carter's benchmarks, [6] (see resources section), was thought to provide a suitable challenge for the new order based approach. Only those instances for which maximum seating capacity has been specified by Carter have been chosen, and the main characteristics of these six problems are summarized in Table 4. The first five columns of this table are self explanatory, and column 6 lists the best known solutions to the underlying graph coloring instances. The `uta-s-92` best GCP is taken from [5], `pur-s-92` from [3], and the other four graph coloring solutions from [6]. Column 7 presents solutions to the underlying bin packing instances, as calculated with a simple FFD algorithm by the present author. Interestingly, every one of the BPP solutions obtained by FFD match the so called 'ideal solutions', found simply by counting the total number of student-examination events and filling up the seats in consecutive time slots, ignoring all other information, until all the events are used up. Thus, all the solutions in column 7 are optimal for the underlying bin packing problem. Assuming that the graph coloring solutions in column 6 are also optimal, we can say that the larger solutions of GCP and BPP gives a lower bound for the corresponding timetabling problem (indicated with a '*', in Table 4).

Table 4. Characteristics of Timetabling Problems

Instance	# exams	# students	# edges	s̃eats	GCP slots	BPP slots
car-f-92	543	18419	20305	2000	28*	28*
car-s-91	682	16926	29814	1550	28	37*
kfu-s-93	461	5349	5893	1955	19*	13
pur-s-93	2419	30032	86261	5000	30*	25
tre-s-92	261	4362	6131	655	20	23*
uta-s-92	622	21266	24249	2800	30*	22

Results for the GSA on Carter's instances are presented in Table 5. The table also shows results for a mutation-only version of the GSA and a pure iterated greedy algorithm. As before, the same run time was used for the GSA and iterated greedy algorithm. We set the population size to 200 for both versions of the GSA, and each experiment was run for 2000 generations. The form of local search used for the previous experiments in graph coloring and bin packing was altered slightly for the timetabling problem. Recall that we used one to three iterations of a local search based on reordering the classes according to some form of 'largest first' criterion – either 'decreasing total degree' (for the GCP) or the 'fullest bin first' (for the BPP). Preliminary experiments with the timetabling instances showed that better results could be obtained if 5 iterations of iterated greedy were used, instead of the usual local search, with largest:reverse:random set at 50:50:30. The 'fullest bin first' approach, as used for the BPP replaced the 'largest first' reordering heuristic in the iterated greedy routine, for the GSA and the iterated greedy proper. Here the fullest bin corresponds to the time slot with the largest number of students taking examinations. Additionally, Falkenauer's bin packing fitness function of (see Eqn.(4)) was used for the timetabling instances, which favored full time slots. POP1 crossover was used, together with insertion mutation.

Table 5. Results for Timetabling Problems

Instance	Order Based GSA				Mut GSA		It Greed	
	# Gens	Time	Mean	Min	Mean	Min	Mean	Min
car-f-92	2000	1513	30.5	30	30.7	30	30.9	30
car-s-91	2000	2254	38.0	38	38.0	38	38.0	38
kfu-s-93	2000	1130	*19.0*	*19*	*19.0*	*19*	*19.0*	*19*
pur-s-93	2000	22527	33.6	33	33.3	33	33.7	33
tre-s-92	2000	376	23.8	*23*	23.4	*23*	23.8	*23*
uta-s-92	2000	2277	30.8	*30*	30.8	*30*	30.8	*30*

It is clear examining Table 5 that the results are very similar for both versions of the GSA and also iterated greedy. The values highlighted in italics denote optimum solutions. It would appear that these particular instances may be easy for all three algorithms.

11 Summary

This Chapter has introduced a new and generic order based framework suitable for application, with minimum adaptation, to a wide range of set partitioning problems. The approach contrasts with other state-of-the-art techniques that rely mostly on direct representations. A clear advantage of using an order based approach is that every permutation is decoded as a feasible solution, meaning no costly repair mechanisms are required, following a crossover event, however heavily constrained the problem. Perhaps the most innovative feature of the new order based approach is the inclusion of some simple grouping and reordering heuristics to preprocess the chromosomes and make them more amenable to crossover. The idea is to encourage the transmission of whole set partitions, from parent to offspring, in a way that is not usually possible with an order based approach. Results presented herein indicate that the new memetic algorithm is highly competitive, yet no lengthy problem specific local search procedure is required. Only a very few iterations of Culberson and Luo's heuristics are required for preprocessing the chromosomes prior to crossover. Thus, although the exact choice of objective or fitness function will vary according to the specific set partitioning application, problem specific heuristics and costly backtracking – so common in other approaches – can largely be avoided.

A detailed examination of the results reveals that different components of the GSA framework are more or less effective, depending on the nature of the test instances. The six timetabling instances, for example, seem to be rather easy for the iterated greedy algorithm to solve, making it difficult to assess the potential of the GSA – it is possible that more challenging instances are needed here. On the other hand, the GSA with crossover proved very effective on the DIMACS graph coloring instances and also on the bin packing instances supplied by Falkenauer. Yet the need for crossover was not particulary well established for the bin packing data sets of Scholl and Klein.

Future work will concentrate on improving results for set partitioning benchmarks, and undertaking further investigations into the relative contributions of genetic operators versus simple reordering heuristics. The challenge will be to make improvements to the approach, while keeping the techniques as generic as possible, avoiding time-consuming backtracking and repair wherever possible. The present author also plans to extend the new approach to more realistic timetabling problems, incorporating additional hard constraints as well as a range of soft constraints. Order based approaches have a distinct advantage over techniques that use a direct representation when dealing with multiply constrained problems: a suitable greedy decoder can ensure that only feasible solutions are generated. On the other hand, a directly encoded method may struggle to find *any* feasible solution in similar circumstances. Multi-objective optimization is of particular interest when several soft constraints conflict. For example, a favorable spread of examinations to allow students revision gaps may conflict with the interests of the teaching staff who may prefer examinations with many students to be held early on, to give sufficient time for marking. Extending the order based techniques to other set partitioning problems is another interesting priority for the future.

A Resources

A.1 Key Books

De Jong, K.A. (2006) Evolutionary Computation: a Unified Approach (Complex Adaptive Systems). MIT Press, Cambridge MA.

Eiben, A.E. and Smith, J.E.(2003) Introduction to Evolutionary Computing. Springer-Verlag, New York.

Goldberg, D.E. (1989) Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Boston, MA.

Holland, J.H. (1992) Adaptation in Natural and Artificial Systems. MIT Press, Cambridge MA.

Mitchell, M. (1998) an Introduction to Genetic Algorithms. MIT Press, Cambridge MA.

Zbigniew, M. (1996) Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, New York.

A.2 Key International Conferences

EvoCOP 2006: 6th European Conference on Evolutionary Computation in Combinatorial Optimization.
http://evonet.lri.fr/eurogp2006/?page=evocop

GECCO 2006: Genetic and Evolutionary Computation Conference.
http://www.sigevo.org/gecco-2006/

IEEE WCCI 2006: World Congress on Computational Intelligence.
http://www.wcci2006.org/

PATAT 2006: The 6th International Conference for the Practice and Theory of Automated Timetabling.
http://www.asap.cs.nott.ac.uk/patat/patat06/patat06.shtml

PPSN IX 2006: Parallel Problem Solving from Nature.
http://ppsn2006.raunvis.hi.is/

A.3 Interest Groups/Web sites

SIGEVO: ACM Special Interest Group on Genetic and Evolutionary Computation.
http://www.sigevo.org/

IEEE CIS: IEEE Computational Intelligence Society.
http://ieee-cis.org/pubs/tec/

A.4 (Open Source) Software

GAGS: A genetic algorithm C++ class library
http://kal-el.ugr.es/GAGS/

GAlib: A C++ Library of Genetic Algorithm Components
http://lancet.mit.edu/ga/

GAJIT: A Simple Java Genetic Algorithms Package
http://www.micropraxis.com/gajit/index.html

GA Playground (Genetic Algorithms Toolkit): Java genetic algorithms toolkit.
http://www.aridolan.com/ga/gaa/gaa.html

JAGA: Java API for genetic algorithms
http://www.jaga.org/

Java genetic algorithms package
http://jgap.sourceforge.net/

GATbx: Genetic Algorithm Toolbox for use with MATLAB.
http://www.shef.ac.uk/acse/research/ecrg/gat.html

GAOT: The Genetic Algorithm Optimization Toolbox for use with MATLAB.
http://www.ise.ncsu.edu/mirage/GAToolBox/gaot/

A.5 Data Sets used in the Chapter

1. Timetabling: http://www.cs.nott.ac.uk/˜rxq/data.htm
2. Bin Packing: Scholl and Klein: http://www.wiwi.uni-jena.de/Entscheidung/binpp/index.htm
3. Bin Packing: Falkenauer: http://people.brunel.ac.uk/m̃astjjb/jeb/orlib/binpackinfo.html
4. DIMACS Challenge GCP: ftp://dimacs.rutgers.edu/pub/challenge/graph/

References

[1] Anderson, P. G. and Ashlock, D. (2004) *Advances in ordered greed.* Available from http://www.cs.rit.edu/˜pga/annie_2004.pdf (last accessed 28 February 2007).

[2] Brélaz, D. (1979) New methods to color the vertices of graphs. *Communications of the ACM*, 24(4):251–256.

[3] Burke, E. and Newell, J. (1999) A multi-stage evolutionary algorithm for the timetabling problem. *IEEE Transactions on Evolutionary Computation*, 3(1):63–74.

[4] Burke, E. and Petrovic, S. (2002) Recent research directions in automated timetabling. *European Journal of Operational Research*, 140(2):266–280.

[5] Caramia, M., Dell'Olmo, P., and Italiano, G. (2001) New algorithms for examination timetabling. In *WAE '00: Proceedings of the 4th International Workshop on Algorithm Engineering September 05 - 08, 2000*, volume 1982 of *Lecture Notes in Computer Science*, pages 230–242, London, UK. Springer-Verlag.

[6] Carter, M. W., Laporte, G., and Lee, S. Y. (1996) Examination timetabling: algorithms, strategies and applications. *European Journal of Operational Research*, 47:373–383.

[7] Coffman, E. G., Garey, M. R., and Johnson, D. S. (1984) Approximation algorithms for bin packing - an updated survey. In Ausiello, G., Lucertini, M., and Serafini, P., editors, *Algorithm Design for Computer System Design*, pages 49–106. Springer-Verlag, Berlin.

[8] Croitoru, C., Luchian, H., Gheorghieş, O., and Apetrei, A. (2002) A new genetic graph coloring heuristic. In *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pages 63–74, Ithaca, New York, USA.

[9] Culberson, J. and Luo, F. (1996) Exploring the k-colorable landscape with iterated greedy. In [24], pages 499–520.

[10] Davis, L. (1985) Applying adaptive algorithms to epistatic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 162–164, Los Angeles, CA.

[11] Davis, L. (1991) Order-based genetic algorithms and the graph coloring problem. In *Handbook of genetic algorithms*, chapter 6, pages 72–90. Van Nostrand Reinhold, New York.

[12] Dorigo, M., Maniezzo, V., and Colorni, A. (1996) The ant system: Optimization by a colony of cooperating agents. *IEEE Trans. System Man Cybernetics Part B*, (26):29–41.

[13] Eiben, A., der Hauw, J. V., and Hemert, J. V. (1998) Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4:25–46.

[14] Erben, W. (2001) A grouping genetic algorithm for graph colouring and exam timetabling. In *Practice and Theory of Automated Timetabling, PATAT 2000*, volume 2079 of *Lecture Notes in Computer Science*, pages 132–156, Berlin/Heidelberg. Springer.

[15] Falkenauer, E. (1995) Solving equal piles with the grouping genetic algorithm. In Eshelman, L. J., editor, *Proceedings of the 6th International Conference on Genetic Algorithms, Pittsburgh, PA, USA, July 15-19, 1995*, pages 492–497, San Francisco, CA, USA. Morgan Kaufmann.

[16] Falkerauer, E. (1996) A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30.

[17] Fogel, L., Owens, A., and Walsh, M. (1966) *Artificial intelligence through simulated evolution.* John Wiley, New York.

[18] Galinier, P. and Hao, J. K. (1999) Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397.

[19] Goldberg, D. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Boston, MA.

[20] Goldberg, D. E. and Lingle, R. (1985) Alleles, loci and the traveling salesman problem. In *Proceedings of an International Conference Genetic Algorithms and their Applications*, pages 154–159, Mahwah, NJ, USA. Lawrence Erlbaum Associates, Inc.

[21] Greenwood, G. W. and Tyrrell, A. M. (2006) *Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems.* Wiley-IEEE Press, Chichester.

[22] Holland, J. H. (1975) *Adaptation in natural and artificial systems.* The University of Michigan Press, Ann Arbor.

[23] Hurley, S., Smith, D., and Thiel, S. (1997) Fasoft: A system for discrete channel frequency assignment. *Radio Science*, 32(5):1921–1940.

[24] Johnson, D. S. and Trick, M. A., editors (1996) volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society.

[25] Jones, D. R. and Beltramo, M. A. (1991) Solving partitioning problems with genetic algorithms. In Belew, R. K. and Booker, L. B., editors, *Proceedings of the 4th International Conference on Genetic Algorithms, San Diego, CA, USA, July 1991*, pages 442–449, San Francisco, CA, USA. Morgan Kaufmann.

[26] Kennedy, J. and Eberhart, R. (1995) Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948, Piscataway, NJ.

[27] Kirkpatrick, S., Gelatt, C., and Vecchi, M. (1983) Optimization by simulated annealing. *Science*, 220(4598):671–680.

[28] Langton, C., editor (1995) *Artificial life: An overview.* MIT Press, Cambridge, MA.

[29] Lewis, R. and Paechter, B. (2005) Application of the grouping genetic algorithm to university course timetabling. In Raidl, G. R. and Gottlieb, J., editors, *Evolutionary Computation in Combinatorial Optimization, 5th European Conference, EvoCOP 2005, Lausanne, Switzerland, March 30 - April 1, 2005, Proceedings*, volume 3448 of *Lecture Notes in Computer Science*, pages 144–153, Berlin/Heidelberg. Springer.

[30] Martello, S. and Toth, P. (1990) *Knapsack problems: algorithms and computer implementations.* Wiley, Chichester.

[31] Matula, D., Marble, G., and Isaacson, J. (1972) Graph coloring algorithms. In Read, R., editor, *Graph theory and computing*, pages 104–122. Academic Press, New York.

[32] Michalewicz, Z. (1996) *Genetic Algorithms + Data Structures = Evolutionary Programs, Third, revised and extended edition.* Springer, London, UK.

[33] Mitchell, M. (1996) *An introduction to genetic algorithms.* MIT Press, Cambridge, MA.

[34] Mumford, C. L. (2006) New order-based crossovers for the graph coloring problem. In Runarsson, T. P., Beyer, H.-G., Burke, E. K., Guervós, J. J. M., Whitley, L. D., and Yao, X., editors, *Parallel Problem Solving from Nature - PPSN IX, 9th International Conference, Reykjavik, Iceland, September 9-13, 2006, Procedings*, volume 4193 of *Lecture Notes in Computer Science*, pages 880–889, Berlin/Heidelberg. Springer.

[35] Oliver, I. M., Smith, D. J., and Holland, J. R. C. (1987) A study of permutation crossover operators on the traveling salesman problem. In Grefenstette, J. J., editor, *Proceedings of the 2nd International Conference on Genetic Algorithms, Cambridge, MA, USA, July 1987*, pages 224–230, Mahwah, NJ, USA. Lawrence Erlbaum Associates.

[36] Pankratz1, G. (2005) A grouping genetic algorithm for the pickup and delivery problem with time windows. *OR Spectrum*, 27(1):21–41.

[37] Rechenberg, I. (1965) Cybernetic solution path of an experimental problem. Technical Report Translation number 1122, Ministry of Aviation, Royal aircraft Establishment, Farnborough, Hants, UK.

[38] Rekiek, B., Lit, P. D., Pellichero, F., Falkenauer, E., and Delchambre, A. (1999) Applying the equal piles problem to balance assembly lines. In *Proceedings of the 1999 IEEE International Symposium on Assembly and Task Planning (ISATP '99)*, pages 399–404, Portugal.

[39] Ross, P., Hart, E., and Corne, D. (1998) Some observations about ga-based exam timetabling. In *PATAT '97: Selected papers from the Second International Conference on Practice and Theory of Automated Timetabling II*, pages 115–129, London, UK. Springer-Verlag.

[40] Schaerf, A. (1999) A survey of automated timetabling. *Artificial Intelligence Review*, 13:87–127.

[41] Schwerin, P. and Wäscher, G. (1997) The bin-packing problem: a problem generator and some numerical experiments with ffd packing and mtp. *International Transactions in Operational Research*, 4(5-6):377–389.

[42] Valenzuela, C. (2001) A study of permutation operators for minimum span frequency assignment using an order based representation. *Journal of Heuristics*, 7(1):5–22. C.L. Valenzuela is now known as C. L. Mumford.

[43] Welsh, D. and Powell, M. (1967) An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10:85–86.

Index