A Genetic Algorithm for the Set Partitioning Problem

P.C. Chu and J.E. Beasley

The Management School
Imperial College
London SW7 2AZ, England

p.chu@ic.ac.uk and j.beasley@ic.ac.uk
http://mscmga.ms.ic.ac.uk/pchu/pchu.html
http://mscmga.ms.ic.ac.uk/jeb/jeb.html

April 1995

Abstract

In this paper we present a genetic algorithm-based heuristic for solving the set partitioning problem. The set partitioning problem is an important combinatorial optimisation problem used by many airlines as a mathematical model for flight crew scheduling. We develop a steady-state genetic algorithm in conjunction with a specialised heuristic feasibility operator for solving the set partitioning problem. Some basic genetic algorithm components, such as fitness definition, parent selection and population replacement are modified. The performance of our algorithm is evaluated on a large set of real-world set partitioning problems provided by the airline industry. Computational results show that the genetic algorithm-based heuristic is capable of producing high-quality solutions. In addition a number of the ideas presented (separate fitness, unfitness scores and subgroup population replacement) are applicable to any genetic algorithm for constrained problems.

Keywords: combinatorial optimisation; crew scheduling; genetic algorithms; set partitioning.

1 Introduction

1.1 The set partitioning problem

The set partitioning problem (SPP) is the zero-one integer programming problem of the form:

$$\text{Minimise} \quad \sum_{j=1}^{n} c_j x_j \tag{1}$$

$$\text{Subject to} \quad \sum_{j=1}^{n} a_{ij} x_j = 1, \qquad i = 1, \ldots, m \tag{2}$$

$$x_j \in \{0, 1\}, \qquad j = 1, \ldots, n \tag{3}$$

where $a_{ij} = 0$ or 1.

The best-known application of the SPP is airline crew scheduling [1, 2, 4, 9, 12, 15]. In this formulation each row ($i = 1, \ldots, m$) represents a flight leg (a takeoff and landing) that must be flown. The columns ($j = 1, \ldots, n$) represent feasible round-trip rotations for a crew (i.e. a sequence of flight legs for a crew that begin and end at individual base locations and that conform to all applicable work rules). Associated with each rotation is a cost c_j. The matrix a_{ij} is constructed as:

$$a_{ij} = \begin{cases} 1 & \text{if flight leg } i \text{ is covered by rotation } j, \\ 0 & \text{otherwise.} \end{cases}$$

Then the objective of the crew scheduling problem is to find the "best" collection of rotations such that each flight is covered by exactly one rotation. For some practical crew scheduling problems, because of flight assignments, union rules and other factors some additional constraints are imposed. These constraints are called “base constraints” and have the following form:

$$d^l \leq \sum_{j \in B} d_j x_j \leq d^u \tag{4}$$

where $d_j > 0$ and $B \subseteq \{1, 2, \ldots, n\}$. In this paper, we will only deal with the pure set partitioning problem without base constraints.

1.2 Literature survey

Because of the widespread use of the SPP, a number of algorithms have been developed. These can be classified into two categories: exact algorithms which attempt to solve the SPP to optimality, and heuristic algorithms which try to find ”good” solutions quickly.

The starting point for most exact solution algorithms is to solve the linear programming (LP) relaxation of the SPP (i.e. replace equation (3) by $0 \leq x_j \leq 1,\ j = 1, \ldots, n$). A number of authors [9, 15] have noted that for many “small” SPP problems the solution to the LP relaxation is either all integer, in which case it is also the optimal integer solution, or has only a few fractional values and can be easily resolved. One traditional exact method is the use of cutting planes in conjunction with the simplex method. Balas and Padberg [3] noted that cutting plane algorithms were moderately successful even while using general purpose cuts and without taking advantage of any special knowledge of the SPP polytope.

Another exact method is the use of the tree search (branch and bound). Various bounding strategies have been used, including LP and Lagrangian relaxation. Fisher and Kedia [7] used continuous analogs of the greedy and 3-opt methods to provide improved lower bounds. Of recent interest is the work of Harche and Thompson [11] who developed an exact algorithm based on a new method, called column subtraction (or row sum) method, which is capable of solving large sparse instances of set covering, packing and partitioning problems. The most successful optimal solution algorithm so far appears to be the work of Hoffman and Padberg [12]. They presented an exact algorithm based on branch and cut (which involves solving the LP relaxation of the problem and incorporating cuts derived from polyhedral considerations) and reported optimal solutions for a large set of real-world set partitioning problems.

There have been relatively few heuristic solution algorithms for the SPP in the literature. This is because the SPP is a highly constrained problem and thus finding a feasible solution to a SPP is itself a difficult problem. Ryan and Falkner [19] provided an effective method of obtaining a good feasible solution by imposing additional structure, derived from the real-world problem but not already implicit in the mathematical model. With the rising interest in evolutionary algorithms and their application to combinatorial problems, Levine [14] experimented with a parallel genetic algorithm and applied it to the SPP. Although his algorithm was capable of finding optimal solutions for some problems having up to a few thousand columns, it had difficulty finding feasible solutions for problems having many rows.

Of particular interest to us is the design of a genetic algorithm for highly constrained problems. The SPP, with all constraints being equalities, is a good example of such a problem. This paper introduces a genetic algorithm-based heuristic for solving the set partitioning problem. The paper is organised as follows. In section 2 the basic steps of a simple genetic algorithm are described. In section 3 the genetic algorithm-based heuristic for the SPP is presented. In sections 4 and 5 the preprocessing of the test problems and the computational results are given respectively. Finally in section 6 some conclusions are drawn.

2 Genetic algorithms

A genetic algorithm (GA) can be understood as an “intelligent” probabilistic search algorithm which can be applied to a variety of combinatorial optimisation problems [17]. The theoretical foundations of GAs were originally developed by Holland [13]. The idea of GAs is based on the evolutionary process of biological organisms in nature. During the course of evolution, natural populations evolve according to the principles of natural selection and “survival of the fittest”. Individuals which are more successful in adapting to their environment will have a better chance of surviving and reproducing, whilst individuals which are less fit will be eliminated. This means that the *genes* from the highly fit individuals will spread to an increasing number of individuals in each successive generation. The combination of good characteristics from highly adapted ancestors may produce even more fit offspring. In this way, species evolve to become more and more well adapted to their environment.

A GA simulates these processes by taking an initial population of individuals and applying genetic operators in each reproduction. In optimisation terms, each individual in the population is encoded into a string or *chromosome* which represents a possible *solution* to a given problem. The fitness of an individual is evaluated with respect to a given objective function. Highly fit individuals or *solutions* are given opportunities to reproduce by exchanging pieces of their genetic information, in a *crossover* procedure, with other highly fit individuals. This produces new "offspring" solutions (children), which share some characteristics taken from both parents. Mutation is often applied after crossover by altering some genes in the strings. The offspring can either replace the whole population (generational approach) or replace less fit individuals (steady-state approach). This evaluation-selection-reproduction cycle is repeated until a satisfactory solution is found. The basic steps of a simple GA are shown below. A more comprehensive overview of GAs can be found in [5, 6, 17].

```
Generate an initial population;
Evaluate fitness of individuals in the population;
repeat
    Select parents from the population;
    Recombine (mate) parents to produce children;
    Evaluate fitness of the children;
    Replace some or all of the population by the children;
until a satisfactory solution has been found;
```

3 The GA-based heuristic

We modified the basic GA described in the previous section in a way such that problem-specific knowledge of the SPP is considered. In the following discussion we will use the terms "individuals", "solutions" and "strings" interchangeably.

3.1 Representation and fitness function

The first step in designing a genetic algorithm for a particular problem is to devise a suitable representation scheme. The usual 0-1 binary representation is an obvious choice for the SPP since it represents the underlying 0-1 integer variables. We used a n-bit binary string as the chromosome structure where n is the number of columns in the SPP. A value of 1 for the i-th bit implies that column i is in the solution. The binary representation of an individual's chromosome (solution) for the SPP is illustrated in Figure 1.

column(gene)	1	2	3	4	5	$\cdots$	$n-1$	n
bit string	1	0	1	1	0	$\cdots$	1	0

Figure 1: Binary representation of an individual's chromosome

The fitness of an individual is calculated according to a given fitness function. Fitness measures "how good" an individual is with regard to solving the SPP and is used during the selection and replacement phases to determine which individuals are selected for reproduction and replacement. It must take into account not just the costs of the columns included in the solution (i.e. the SPP objective function value), but also the degree of (in)feasibility of a solution. Note that traditional operational research algorithms restrict their search to feasible solutions, and so no additional term is included in the SPP objective function to handle constraint violations. In the GA approach, however, the GA operators may produce infeasible solutions. In fact, since finding a feasible solution to the SPP is difficult, it may be that the majority of the solutions generated by the GA are infeasible.

There are two approaches to defining the fitness of an individual. The most common approach is the use of a penalty function [14, 16, 18]. Penalty methods allow constraints to be violated. Depending on the magnitude of the violation, however, a penalty that is proportional to the size of the infeasibility is incurred that degrades the objective function. If the penalty is large enough, highly infeasible individuals will rarely be selected for reproduction, and the GA will concentrate on feasible or near-feasible solutions. A generic fitness function $f(x)$ with penalty is of the form

$$f(x) = c(x) + p(x) \tag{5}$$

where $c(x)$ is the objective function and $p(x)$ is a penalty function. A typical penalty function may be defined as:

$$p(x) = \sum_{i=1}^{m} \lambda_i \Phi_i(x) \tag{6}$$

where λ_i is a scalar weight that penalises the violation of constraint i (thus providing a scaling between objective function costs and constraint violation units) and $\Phi_i(x)$ is a function of the violation (or the amount of infeasibility) in constraint i.

Choosing the optimal value for λ_i is a difficult problem because a good choice of λ_i should reflect not just the "costs" associated with making constraint i feasible, but also the impact on other constraints' (in)feasibilities. If the overall penalty is too harsh (i.e. λ_i too large), infeasible individuals that may carry useful information will be ignored. On the other hand, if the penalty is not strong enough, the GA may search only among infeasible individuals. In addition, the optimal values for λ_i are likely to be data dependent. These limitations were discussed and reported in [14].

Our alternative approach for defining fitness involves separating the single fitness measure into two, one is called *fitness* and the other is called *unfitness*. Each individual can then be represented by a pair of values $f(x)$ and $\sum_{i=1}^{m} \Phi_i(x)$, obviating the need for any λ_i. For the SPP we chose to set the fitness f_p of an individual p equal to its objective function value as calculated by

$$f_p = \sum_{j=1}^{n} c_j s_{pj} \tag{7}$$

where s_{pj} is the value of the j-th bit (column) in the string corresponding to the p-th individual and c_j is the cost of bit (column) j. Note that since the SPP is a minimisation problem, the *lower* the fitness score the *more* fit the solution is. The unfitness u_p of an individual p measures the amount of infeasibility (in relative terms) and we chose to define it as

$$u_p = \sum_{i=1}^{m} |w_i - 1| \tag{8}$$

where $w_i = \sum_{j=1}^{n} a_{ij} s_{pj}$ is the number of columns that cover row i. The absolute value in equation (8) implies that an individual is feasible if $u_p = 0$ and infeasible if $u_p > 0$. Defining the unfitness to be at a minimum of zero if and only if the solution is feasible seems a natural approach.

There are several advantages to using separate fitness and unfitness scores instead of a single penalty-adjusted fitness score. Firstly, it eliminates the need for λ_i and the penalty term in equation (5). Secondly, it transforms the "fitness landscape" from a one-dimensional line (fitness only) into a two-dimensional plane (fitness and unfitness axes), thus allowing the point which a solution represents to be identified more precisely. Finally, it leads to a more elaborate population replacement scheme which will be discussed later.

We should stress here that although these ideas are being presented in the context of the SPP they are applicable to any genetic algorithm for constrained problems.

3.2 Crossover and mutation operators

The crossover operator takes bits from each parent string and "combines" them to create a child string. The idea is that by creating new strings from substrings of fit parent strings, new and promising areas of the search space will be explored. Many crossover techniques exist in the literature. In the past several years, however, GA researchers [20, 21] have given evidence to support the claim that uniform crossover has a better re-combination potential than do other crossover operators, such as the classical one-point and two-point crossover operators. The uniform crossover operator works by generating a random *crossover mask* B (using Bernoulli distribution) which can be represented as a binary string:

$$B\colon\ b_1 b_2 b_3 \cdots b_{n-1} b_n$$

where n is the length of the chromosome. Let P_1 and P_2 be the parent strings $P_1[1], \ldots, P_1[n]$ and $P_2[1], \ldots, P_2[n]$ respectively. Then the child solution is created by letting:

$$C[i] := \begin{cases} P_1[i] & \text{if } b_i = 0 \\ P_2[i] & \text{if } b_i = 1 \end{cases}$$

for all $i = 1, \ldots, n$. A new crossover mask is generated for each mating.

Mutation is applied to each child after crossover. It works by *inverting* M randomly chosen bits in a string where M is experimentally determined. Mutation is generally seen as a background operator which provides a small amount of random search. It also helps to guard against loss of valuable genetic information by reintroducing information lost due to premature convergence and thereby expanding the search space.

There are two types of mutation used in our GA. One is called *static* mutation, which has a constant mutation rate, M_s; the other is called *dynamic* mutation, which mutates only certain bits at each iteration according to given rules. As mentioned earlier, the SPP is a highly constrained problem and a feasible solution is often difficult to construct heuristically. Our initial experiments showed that for some difficult problems, the GA may fail to evolve a feasible solution after a large number of iterations. This problem arises because an attempt to satisfy a particular constraint may introduce infeasibilities into other currently feasible constraints. As the population begins to converge, the GA will gradually evolve a "dominating" set of columns which favour certain constraints at the expense of others. This problem also arises because some constraints are easier to satisfy than others. Dynamic mutation helps the GA avoid such situations and is as follows:

1. At each GA iteration and for each constraint (row) i, compute the total number of solutions, θ_i, in the population that satisfy constraint i.

$$\begin{aligned} \theta_i &= \sum_{p \in \mathcal{P}} r_{pi} \\ r_{pi} &= \begin{cases} 1 & \text{if } w_i = 1 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{9}$$

   where $\mathcal{P}$ is the population set.

2. For each row i, if $\theta_i < \epsilon|\mathcal{P}|$ $(0 < \epsilon < 1)$, then set M_d randomly chosen bits (columns) which cover row i to one in the child string.

Step 2 assumes that if constraint i is satisfied in less than a fraction of the total population (i.e. $\epsilon|\mathcal{P}|$, ϵ small), it may be an indication that the GA is bringing the population towards infeasibility because the individuals which satisfy constraint i are "dying out". In order to re-introduce solutions which satisfy constraint i into the population, columns which cover row i are added into the child string to "compete" with other columns. This is done by setting M_d bits (those that cover row i) in the child string to one, where M_d is arbitrarily chosen.

Limited computational experience showed that a low static mutation rate $(2 < M_s < 5)$ was preferable to a higher mutation rate $(M_s > 5)$ and that within the low range the quality of the solution was not particularly sensitive to the rate. We set $M_s = 3$ in our GA. We also set the dynamic mutation parameters ϵ and M_d to be 0.5 and 5 respectively. These mean that when less than 50% of the population fail to satisfy constraint i, dynamic mutation will take place in the child string with a mutation rate $M_d = 5$. These values have experimentally been shown to be quite satisfactory.

Notice that the crossover and mutation operators described above will not necessarily produce a feasible solution. In reality, it is likely that the child solution will become infeasible after applying the crossover and mutation operators. Thus the crossover and mutation operations should only be treated as an intermediate step that produce a partial solution which will then be reconstructed using a heuristic operator in an attempt to obtain a feasible solution.

3.3 Heuristic feasibility operator

Because the SPP is highly constrained, the solutions generated by the crossover and mutation operators are often infeasible (i.e. some rows may be under-covered and some rows may be over-covered). The objective of the heuristic operator is then to make the solutions less infeasible (if not feasible).

The heuristic operator has two steps. The first step is to identify all over-covered rows and randomly remove columns until all rows are covered by at *most* one column. The second step is to identify all under-covered rows and add columns such that as many under-covered rows as possible can be covered without causing any other rows to be over-covered. Let

I = the set of all rows
J = the set of all columns
α_i = the set of columns that cover row i, i.e. $\{j \mid a_{ij} = 1, \forall j \in J\}$
β_j = the set of rows covered by column j, i.e. $\{i \mid a_{ij} = 1, \forall i \in I\}$
S = the set of columns in a solution
U = the set of uncovered rows
w_i = the number of columns in S that cover row i

The heuristic feasibility operator is as follows:

1. Initialise $w_i := |\alpha_i \cap S|$, $\forall i \in I$. Let $T := S$.
2. Randomly select a column j, $j \in T$ and set $T := T - j$. If $w_i \geq 2$, for any $i \in \beta_j$, set $S := S - j$ and set $w_i := w_i - 1$, $\forall i \in \beta_j$.
3. Repeat step 2 until $T = \emptyset$.
4. Initialise $U := \{i \mid w_i = 0, \forall i \in I\}$. Let $V := U$.
5. Randomly select a row $i \in V$ and set $V := V - i$. Find the column $j \in \alpha_i$ that
   (a) satisfies $\beta_j \subseteq U$, and
   (b) minimises $c_j / |\beta_j|$.
6. If no such j exists, go to step 7; else add j to S, and set $w_i := 1$, $\forall i \in \beta_j$. Set $U := U - \beta_j$ and $V := V - \beta_j$.
7. Repeat step 5 until $V = \emptyset$.

Steps 1–3 attempt to make over-covered rows ($i \in I$, $w_i \geq 2$) feasible by setting all but one of the columns $j \in (\alpha_i \cap S)$ to zero. In steps 4–7, columns are added back to the solution such that as many under-covered rows as possible are satisfied without introducing over-coverage into other rows. Note that the heuristic operator allows under-covered rows, but not over-covered rows, in the solution and that it does not guarantee to produce a feasible solution.

From the computational standpoint, steps 1–3 take $O(mn)$ operations. Steps 4–7 would take $O(m^2n)$ operations since it requires a search through each under-covered row i ($O(m)$) and for each row i, it searches all the columns that covers row i ($O(n)$) whilst examining the condition in 5(a) is $O(m)$. Taking the larger term of the sum $O(mn) + O(m^2n)$, the heuristic operator has a complexity of $O(m^2n)$.

3.4 Parent selection method

Parent selection is the task of assigning reproductive opportunities to each individual in the population based on their relative fitnesses. One commonly used method is binary tournament selection [10]. In a binary tournament selection, two individuals are chosen randomly from the population. The more fit individual is then allocated a reproductive trial. In order to produce a child, two binary tournaments are held, each of which produces one parent string. These two parent strings are then combined to produce a child.

The tournament selection criterion may be based on either the fitness or the unfitness of the individuals as defined previously. The difficulty with these criteria is that the individuals which have lower fitness scores (more fit) generally have higher unfitness scores (more infeasible) and vice versa at the early stages of the GA. Thus if we favour selection on more fit individuals, we are likely to select parents which are mostly infeasible. This will not help the GA in finding feasible solutions. On the other hand, if we favour selection on less infeasible individuals, we might select individuals which are less fit. Whilst feasibility may improve, the solution quality will suffer. To avoid this, we developed a maximum compatibility selection (MCS) method for selecting parents that attempts to improve solution quality as well as feasibility.

The MCS method is designed specifically for the SPP and it takes into account the row coverage of the candidate parents. In a maximum compatibility selection, one parent P_i is first selected using a

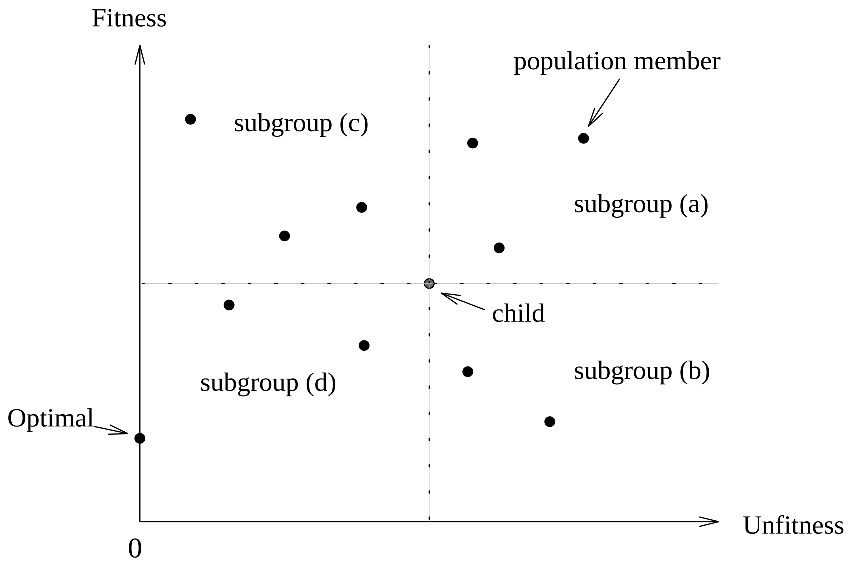

Figure 2: Population subgroups and the fitness-unfitness landscape

binary tournament based on *fitness*. The other parent P_j $(j \neq i)$ is then selected to give a maximum *compatibility* score as measured by

$$|R_{P_i} \cup R_{P_j}| - |R_{P_i} \cap R_{P_j}|$$

where R_{P_i} and R_{P_j} are the set of rows covered by P_i and P_j respectively. If j is not unique, then the tie-breaking rule is to select the member of the population with the lowest fitness score.

The logic here is that we would like the two parents together to cover as many rows as possible (i.e. a high $|R_{P_i} \cup R_{P_j}|$) with as few rows in common as possible (i.e. a low $|R_{P_i} \cap R_{P_j}|$). Note that if the first parent P_i is a feasible solution ($u_{P_i} = 0$), then the second parent P_j is selected using the tournament selection method based on fitness instead of the MCS method because a feasible child can be easily constructed from a feasible parent.

3.5 Population replacement scheme

Once a child solution has been produced through the GA operators, the child will replace a "less fit" member of the population. The average fitness and/or unfitness of the population will improve if the child solution has lower fitness and unfitness scores than those of the solution being replaced.

We designed a subgroup ordering replacement scheme (SOR) which divides the population into four mutually exclusive subgroups with respect to the child. These four subgroups can be illustrated as in Figure 2. Figure 2 shows the fitness-unfitness landscape of the population where the x and y axes represent unfitness and fitness respectively. Each point in the plot represents an individual (solution) in the population and is positioned according to the individual's fitness and unfitness scores. These points are separated into four disjoint subgroups with respect to the fitness and unfitness of the child. Note that the higher the fitness and unfitness a solution has, the worse the solution is.

Consider the subgroups in order (a), (b), (c) and (d). A child will replace a selected member of the *first* non-empty subgroup in this order. In the SOR scheme, a child solution will first attempt to replace a solution in subgroup (a), which has higher fitness and unfitness scores than the child's, thus improving both the average fitness and unfitness of the whole population. If subgroup (a) is empty, then subgroup (b) is considered next. The reason for considering subgroup (b) before subgroup (c) is that for problems for which feasible solutions are harder to find, the priority is to first search for a feasible solution before trying to improve the fitness (or the quality) of the solution. Therefore by considering subgroups (a) and (b) first, the average unfitness of the population will decrease (i.e. the points will shift towards the fitness axis).

For problems for which feasible solutions are easier to find, the population will soon consist of mostly feasible solutions because all feasible solutions found will be kept in the population unless a feasible child with a lower fitness is found. Any infeasible child that enters the population will be replaced by the next feasible child found. This means that the majority of the population will lie on the fitness axis (unfitness equal to zero). Then the effort is to concentrate on improving the fitness (quality) of the solutions. In any subgroup the member selected for replacement by the child is the member with the worst unfitness (ties broken by worst fitness).

The SOR scheme, when combined with the MCS method, becomes effective in improving the feasibility and quality of the solutions. To see this, we refer back to Figure 2. Figure 2 shows that in order for the GA to evolve good feasible solutions, the initial solutions (denoted by the points on the plot) should move towards the optimal solution (which lies on the fitness axis) as the GA progresses. The MCS method helps this move by selecting parent solutions which are more likely to create a child solution with low fitness and unfitness. The new improved child solutions will eliminate other solutions in the population with high unfitness (subgroups (a) and (b)) via the SOR scheme. The overall effect is to shift the population appropriately.

Figure 3 shows the population at different stages of the GA for test problem `AA02` (see later) using the SOR scheme. The intersection between the dotted line and the fitness axis is where the optimal solution lies. In the initial population (at iteration 0) the points are scattered. As the GA progresses, the points begin to converge and move towards the fitness axis. In the last plot (at iteration 50,000) the points are clustered near the optimal solution.

Note that when replacing a solution, care must be taken to prevent a duplicate solution from entering the population. A *duplicate* child is one such that its solution structure is identical to any one of the solution structures in the population. Allowing duplicate solutions to exist in the population may be undesirable because a population could come to consist of all identical solutions, thus severely limiting the GA's ability to generate new solutions.

3.6 Population size and initial population

In principle, the size of the population and the initial population are chosen such that a highly diverse solution space is sampled. Limited computational experience indicated that the quality of the solution is not very sensitive to the size of the population and so a population size $N = 100$ was used. The initial population is generated randomly. Each of the initial solutions S_p is generated using the following method.

1. Set $S_p := \emptyset$. Set $U := I$.
2. Randomly select a row $i \in U$:
   (a) randomly select a column $j \in \alpha_i$ such that $\beta_j \cap (I - U) = \emptyset$,
   (b) if no such j exists, set $U := U - i$; else add j to S_p and set $U := U - i$, $\forall i \in \beta_j$.
3. Repeat step 2 until $U = \emptyset$.

3.7 Overview

To summarise our GA for the SPP, the following steps are used.

1. Generate an initial population of N random solutions. Set iteration counter $t := 0$.
2. Select two solutions P_i and P_j from the population using the MCS method.
3. Combine P_i and P_j to form a new solution C using the uniform crossover operator.
4. Mutate M_s randomly selected bits in C and perform dynamic mutation if required.
5. Apply the heuristic feasibility operator to C in an attempt to make C more feasible.
6. If C is identical to any one of the solutions in the population, go to step 2; otherwise, set $t := t + 1$ and go to step 7.
7. Replace a solution in the population with C using the SOR scheme.
8. Repeat steps 2-7 until $t = M$ non-duplicate solutions have been generated. The best feasible solution found is the one with the smallest fitness and zero unfitness in the population.

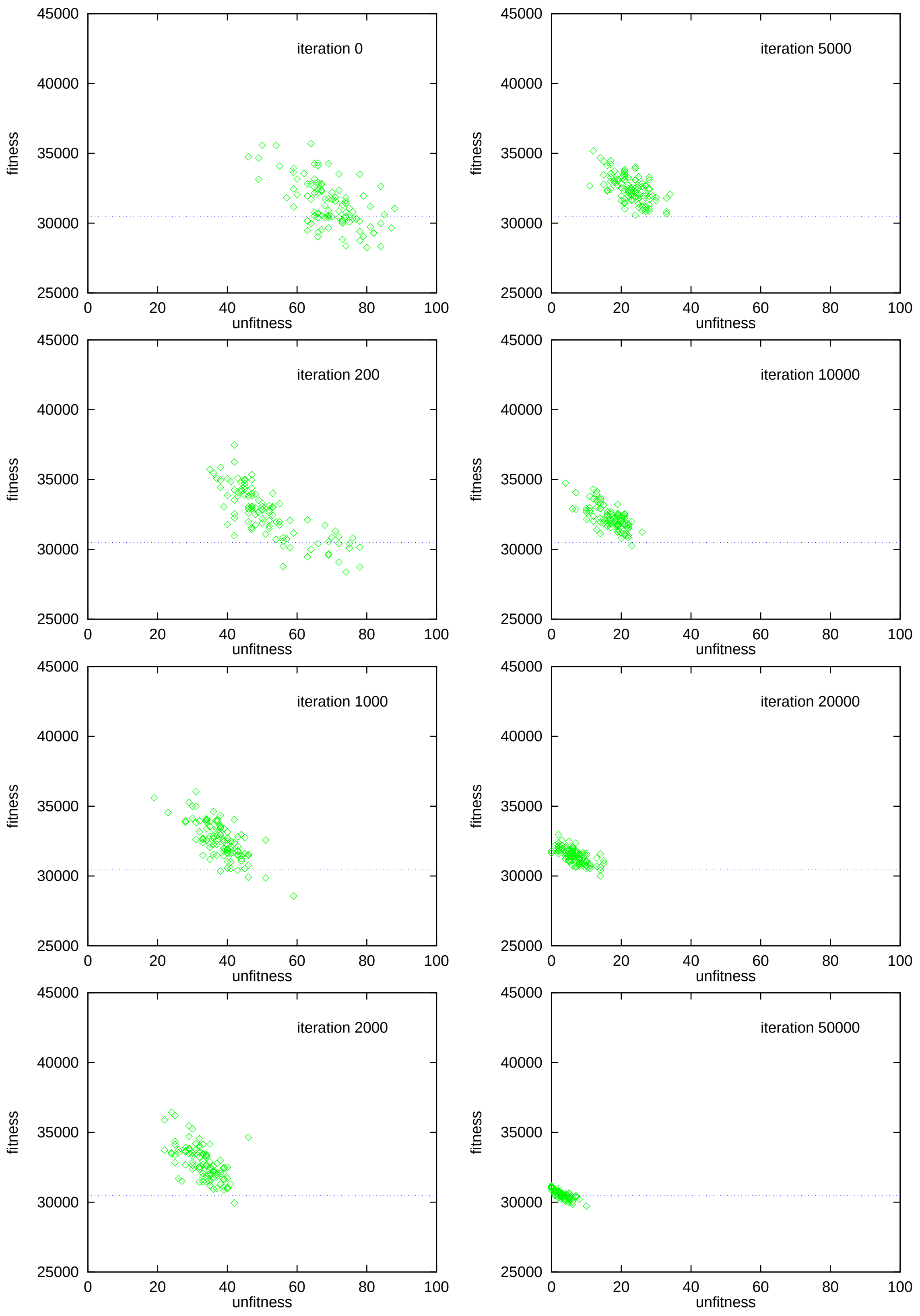

Figure 3: Population convergence using the SOR method

4 Preprocessing

The fifty-five real-world set-partitioning problems of various sizes used by Hoffman and Padberg in [12] are used in our study. For details of how to obtain these test problems, email the message *sppinfo* to *o.rlibrary@ic.ac.uk*. There are a number of preprocessing procedures [8, 12] that can be used to reduce the size of these problems. These are:

Reduction 1: If $\beta_j = \beta_k$ for any pair of $(j,k) \in J$, $j \neq k$ and if $c_j \leq c_k$, then delete column k (i.e. column k is a duplicate column).

Reduction 2: If $|\alpha_i| = 1$, $i \in I$, then the column j in α_i must be in the optimal solution ($x_j = 1$). Delete j and all rows $i \in \beta_j$.

Reduction 3: If $\alpha_i \subseteq \alpha_k$ for any pair of $(i,k) \in I$, $i \neq k$, then delete all columns $j \in (\alpha_k - \alpha_i)$ and row k. This reduction follows because any column that covers row i also covers row k.

Reduction 4: If $|\alpha_i - (\alpha_i \cap \alpha_k)| = |\alpha_k - (\alpha_i \cap \alpha_k)| = 1$ for any pair of $(i,k) \in I, i \neq k$, then let column $j = \alpha_i - (\alpha_i \cap \alpha_k)$ and column $p = \alpha_k - (\alpha_i \cap \alpha_k)$.

1. If $\beta_j \cap \beta_p = \emptyset$, then columns j and p are merged into a single column having a cost $c_j + c_p$. Delete row k.
2. If $\beta_j \cap \beta_p \neq \emptyset$, then delete columns j and p. Delete row k.

Reduction 5: For each $j \in J$, suppose $x_j = 1$, then $x_k = 0$, $\forall k \in T = (\{\cup_{i \in \beta_j} \alpha_i\} - j)$. If $U = \{i \mid \alpha_i \subseteq T,\ i \in I - \beta_j\} \neq \emptyset$, then the problem is infeasible (since row $i \in U$ cannot be covered). Therefore, we can deduce that x_j cannot have the value one. So x_j must be zero and hence column j can be deleted from the problem since no feasible solution can contain column j.

Collectively we refer to these procedures as PREP. Reduction 5 (a new reduction procedure) is particularly helpful to the GA since it can reduce the chance that the GA may converge into infeasible solutions. These reduction procedures are applied to the original problem repeatedly until no further reduction can be achieved. The original problem sizes, the reduced problem sizes given by Hoffman and Padberg and the reduced problem sizes using PREP are given in Table 1. These problem sizes are measured by number of rows; number of columns; and density (percentage of ones in the a_{ij} matrix). Table 1 shows that PREP reduced the problems more than the reduction procedures used by Hoffman and Padberg for most of the problems. Note also here that substantial problem reductions can occur, much larger reductions then would be expected for randomly generated problems.

5 Computational results

The algorithm presented was coded in C and tested on a Silicon Graphics Indigo workstation (R4000, 100MHz). In our computational study, 10 trials of the GA heuristic (each with a different random seed) were generated for each of the 55 test problems. Each trial terminated when $M = 100,000$ non-duplicate children had been generated. The population size N was set to 100 for all problems. The static mutation rate of $M_s = 3$ and the dynamic mutation rate of $M_d = 5$ were used. The maximum compatibility selection method and the subgroup ordering replacement scheme were used. To compare the results against the traditional operational research method, we used the `CPLEX` Mixed Integer Solver, Version 3.0 (with all the default settings) to solve the problems to optimality. Computational results are shown in Table 2 and Table 3.

In Table 2 we give, for each problem:

- The optimal solution value from Hoffman and Padberg [12]
- The best solution value for each of the 10 trials

and in Table 3 we give, for each problem:

- The average percentage deviation (σ) from optimal ($\sigma = \sum_{i=1}^{10} \frac{(S_{T_i} - S_o)}{10 S_o} \times 100\%$ where S_{T_i} is the best solution value of the i-th trial and S_o is the optimal solution value)
- The average number of feasible (non-duplicate) children generated

	Original			Hoffman & Padberg			PREP		
Prob	Rows	Cols	Dens	Rows	Cols	Dens	Rows	Cols	Dens
NW01	135	51975	5.86	135	50069	5.87	135	49903	5.87
NW02	145	87879	5.66	145	85258	5.68	145	85256	5.68
NW03	59	43749	14.13	59	38964	14.12	53	38956	15.45
NW04	36	87482	20.22	36	46190	20.20	35	46189	20.49
NW05	71	288507	10.06	71	202603	10.07	62	202594	11.35
NW06	50	6774	18.17	50	5977	18.27	38	5956	19.81
NW07	36	5172	22.12	36	3108	21.86	34	3105	22.71
NW08	24	434	22.39	34	356	22.36	21	352	25.55
NW09	40	3103	16.20	40	2305	16.05	38	2301	16.64
NW10	24	853	21.18	24	659	21.25	21	655	23.34
NW11	39	8820	16.64	39	6488	16.81	34	6482	18.75
NW12	27	626	20.00	27	454	19.06	25	451	14.66
NW13	51	16043	12.78	51	10905	12.57	50	10903	12.68
NW14	73	123409	10.04	73	95178	10.11	70	95172	10.44
NW15	31	467	19.55	29	463	21.00	29	405	20.56
NW16	139	148633	7.27	139	138951	7.23	135	138947	7.39
NW17	61	118607	13.96	61	78186	13.96	54	78179	15.34
NW18	124	10757	6.82	124	8460	6.83	110	8439	6.96
NW19	40	2879	21.88	40	2145	21.59	32	2134	21.92
NW20	22	685	24.70	22	566	25.00	22	536	25.00
NW21	25	577	24.89	25	426	24.33	25	421	24.32
NW22	23	619	23.87	23	531	24.10	23	520	24.09
NW23	19	711	24.80	18	473	24.87	18	423	24.38
NW24	19	1366	33.20	19	925	33.19	19	926	33.22
NW25	20	1217	30.16	20	844	30.15	20	844	30.15
NW26	23	771	23.77	18	473	23.12	21	464	25.02
NW27	22	1355	31.55	22	926	30.76	22	817	30.22
NW28	18	1210	39.27	18	825	38.43	18	580	35.95
NW29	18	2540	31.04	18	2034	30.99	18	2034	30.99
NW30	26	2653	29.63	26	1884	29.80	26	1877	29.78
NW31	26	2662	28.86	26	1823	29.21	26	1728	28.86
NW32	19	294	24.30	18	251	25.81	18	251	25.81
NW33	23	3068	30.76	23	2415	30.75	23	2308	30.47
NW34	20	899	28.06	20	750	28.16	20	718	27.70
NW35	23	1709	26.70	23	1403	27.02	23	1132	26.32
NW36	20	1783	36.90	20	1408	36.14	20	1204	35.17
NW37	19	770	25.83	19	639	25.89	19	639	25.89
NW38	23	1220	32.33	23	911	31.44	21	690	33.45
NW39	25	677	26.55	25	567	26.25	25	565	26.28
NW40	19	404	26.95	19	336	26.83	19	336	26.86
NW41	17	197	22.10	17	177	22.27	17	177	22.33
NW42	23	1079	26.33	23	895	26.05	23	795	25.92
NW43	18	1072	25.18	17	982	26.43	17	982	26.43
AA01	823	8904	1.00	607	7532	1.00	605	7399	1.03
AA02	531	5198	1.32	360	3846	1.54	360	3837	1.54
AA03	825	8627	1.00	537	6694	1.32	536	6657	1.13
AA04	426	7195	1.70	342	6122	1.80	342	6118	1.80
AA05	801	8308	0.99	521	6235	1.12	520	6206	1.12
AA06	646	7292	1.10	488	5862	1.21	485	5807	1.22
US01	145	1053137	9.10	90	370642	9.80	86	351018	10.47
US02	100	13635	14.13	45	9022	16.57	45	4617	14.82
US03	77	85552	18.40	53	27084	21.42	50	20171	20.10
US04	163	28016	6.52	112	6564	7.48	91	3732	8.45
KL01	55	7479	13.67	50	5957	13.47	47	5915	13.44
KL02	71	36699	8.16	69	16542	8.34	69	16542	8.34

Table 1: Test problem details and reduction comparisons

Prob	Optimal	Best solution in each of the 10 trials									
NW01	114852	–	–	–	–	–	–	–	–	–	–
NW02	105444	109368	109125	109560	109713	108816	109443	110184	110148	109677	110151
NW03	24492	25095	24510	o	24495	24561	o	26448	24501	25086	25785
NW04	16862	o	16876	16986	16970	16970	o	16970	o	17004	o
NW05	132878	138150	138890	138878	138330	138636	138240	138888	137602	134170	135780
NW06	7810	o	o	o	o	o	o	o	o	o	o
NW07	5476	o	o	o	o	o	o	o	o	o	o
NW08	35894	o	o	o	o	o	o	o	o	o	o
NW09	67760	o	o	o	o	o	o	o	o	o	o
NW10	68271	o	o	o	o	o	o	o	o	o	o
NW11	116256	o	o	o	o	o	117585	o	o	o	o
NW12	14118	o	o	o	o	o	o	o	o	o	o
NW13	50146	o	50650	50152	o	50152	o	50332	50164	50152	50158
NW14	61844	62532	62356	62724	62304	62696	62388	62918	62932	62262	62532
NW15	67743	o	o	o	o	o	o	o	o	o	o
NW16	1181590	o	o	o	o	o	o	o	o	o	o
NW17	11115	o	11133	o	11196	11133	o	o	o	o	11133
NW18	340160	363820	345762	359160	345130	365398	358484	357646	359148	358550	385596
NW19	10898	o	o	o	o	o	o	o	o	o	o
NW20	16812	o	o	o	o	o	o	o	o	o	o
NW21	7408	o	o	o	o	o	o	o	o	o	o
NW22	6984	o	o	o	o	o	o	o	o	o	o
NW23	12534	o	o	o	o	o	o	o	o	o	o
NW24	6314	o	o	o	o	o	o	o	o	o	o
NW25	5960	o	o	o	o	o	o	o	o	o	o
NW26	6796	o	o	o	o	o	o	o	o	o	o
NW27	9933	o	o	o	o	o	o	o	o	o	o
NW28	8298	o	o	o	o	o	o	o	o	o	o
NW29	4274	o	o	o	o	o	o	o	o	o	o
NW30	3942	o	o	o	o	o	o	o	o	o	o
NW31	8038	o	o	o	o	o	o	o	o	o	o
NW32	14877	o	o	o	o	o	o	o	o	o	o
NW33	6678	o	6724	o	o	o	o	o	o	o	o
NW34	10488	o	o	o	o	o	o	o	o	o	o
NW35	7216	o	o	o	o	o	o	o	o	o	o
NW36	7314	o	o	o	o	o	o	o	7322	o	o
NW37	10068	o	o	o	o	o	o	o	o	o	o
NW38	5558	o	o	o	o	o	o	o	o	o	o
NW39	10080	o	o	o	o	o	o	o	o	o	o
NW40	10809	o	o	o	o	o	o	o	o	o	o
NW41	11307	o	o	o	o	o	o	o	o	o	o
NW42	7656	o	o	o	o	o	o	o	o	o	o
NW43	8904	o	o	o	o	o	o	o	o	o	o
AA01	56138	–	–	–	–	–	–	–	–	–	–
AA02	30494	30601	30704	30500	30639	31161	31616	30726	32165	31056	30601
AA03	49649	–	–	–	–	–	–	–	–	–	–
AA04	26402	30115	28261	29779	28397	28585	29440	28442	28986	29791	29400
AA05	53839	–	–	–	–	–	–	–	–	–	–
AA06	27040	27932	28060	28118	28048	28022	28004	28608	28500	28355	27883
US01	10022	12627	12317	11614	12237	10557	11640	11144	11875	11496	10921
US02	5965	o	o	o	o	o	o	o	o	o	o
US03	5338	o	o	o	o	o	o	o	o	o	o
US04	17854	o	o	o	o	o	o	o	o	o	o
KL01	1086	1088	o	o	o	1088	o	o	1090	o	o
KL02	219	o	220	o	220	226	o	220	220	o	220

–	No feasible solution is found
o	Optimal solution value

Table 2: Computational results

					CPLEX MIP Solver		
Prob	Avg. % σ	Avg. No. Feasible Soln's	Avg. Sol'n Time	Avg. Exe. Time	Best Sol'n	No. of Nodes	Sol'n Time
NW01	N/A	0	N/A	10435.9	o [a]	0	127.4
NW02	3.96	24703	15132.0	19392.4	o	0	143.6
NW03	1.17	100000	2782.0	6398.9	o	2	27.1
NW04	0.36	52446	3458.6	8700.2	o	997	850.2
NW05	3.67	100000	24914.5	42391.9	– [b]	–	–
NW06	0	100000	326.4	1030.6	o	16	6.6
NW07	0	100000	47.6	442.6	o	0	1.2
NW08	0	100000	0.7	99.9	o	0	0.1
NW09	0	100000	14.2	369.0	o	0	0.7
NW10	0	100000	1.8	132.9	o	0	0.2
NW11	0.11	100000	69.8	905.3	o	1	2.5
NW12	0	100000	3.5	74.6	o	0	0.1
NW13	0.15	100000	389.4	1187.9	o	9	5.9
NW14	1.16	100000	9782.0	20072.6	o	0	102.3
NW15	0	1802	1.4	101.1	o	0	0.1
NW16	0	100000	11810.5	116675.7	o	0	340.2
NW17	0.12	100000	5650.7	13999.1	o	23	150.8
NW18	5.80	89336	1540.8	2033.4	o	3	11.1
NW19	0	100000	69.7	404.5	o	0	0.6
NW20	0	85814	3.2	119.7	o	5	0.2
NW21	0	95380	1.1	80.0	o	1	0.2
NW22	0	49026	0.5	90.2	o	3	0.2
NW23	0	19431	1.5	105.3	o	6	0.2
NW24	0	99438	4.9	144.1	o	5	0.3
NW25	0	100000	3.6	113.9	o	4	0.3
NW26	0	52931	4.4	87.4	o	2	0.1
NW27	0	81581	2.6	135.6	o	2	0.3
NW28	0	74176	3.0	121.2	o	4	0.3
NW29	0	54395	48.8	243.0	o	8	0.7
NW30	0	72783	28.1	260.1	o	4	0.8
NW31	0	73493	42.2	261.1	o	4	0.7
NW32	0	100000	1.9	56.3	o	18	0.2
NW33	0.07	63661	5.5	315.5	o	2	0.7
NW34	0	69252	2.2	141.6	o	1	0.2
NW35	0	36175	4.5	178.3	o	2	0.4
NW36	0.01	11016	46.2	212.3	o	32	1.8
NW37	0	99094	2.3	108.0	o	2	0.2
NW38	0	33348	6.7	132.4	o	2	0.3
NW39	0	84923	1.1	106.7	o	4	0.2
NW40	0	100000	1.4	68.8	o	5	0.1
NW41	0	100000	0.9	44.7	o	2	0.1
NW42	0	15498	16.3	163.6	o	6	0.4
NW43	0	61399	6.5	123.2	o	1	0.2
AA01	N/A	0	N/A	3101.3	o	2373	10109.3
AA02	1.58	5105	1145.4	1567.4	o	0	63.43
AA03	N/A	0	N/A	2810.3	o	12	341.2
AA04	10.29	14854	1731.7	1890.4	26521	3645	6923.2
AA05	N/A	0	N/A	2670.1	o	136	276.3
AA06	4.12	15651	2114.8	2439.4	o	40	209.5
US01	16.17	12814	43683.5	74728.4	–	–	–
US02	0	44522	76.8	667.0	o	0	6.5
US03	0	27197	1350.2	3859.5	o	0	27.5
US04	0	20528	135.1	789.7	o	8	13.6
KL01	0.07	32601	159.6	909.9	o	66	10.7
KL02	0.55	20288	485.7	2099.7	o	95	66.5

[a]Optimal solution value
[b]See text for discussion

Table 3: Performance measures

- The average solution time (in CPU seconds) which is the time that the GA takes to first reach the final best solution
- The average execution time (in CPU seconds) which is the time that the GA takes to generate 100,000 non-duplicate child solutions
- The best solution found using `CPLEX`
- The number of nodes searched by `CPLEX` in its branch-and-bound search tree
- The total time (in CPU seconds) for `CPLEX`

Examining Tables 2 and 3 we observe that:

1. The GA is able to find at least one feasible solution for all but 4 of the problems. In fact, in 43 out of the 55 problems the GA is able to find the optimal solution in at least one trial.
2. The GA is able to find the optimal solution consistently (i.e. in every trial) for 34 problems.
3. The GA is capable of generating a large percentage of child solutions that are feasible for many problems. This indicates that the heuristic feasibility operator is effective in constructing feasible solutions. Limited computational experience showed that without the heuristic feasibility operator, the GA is much less effective in generating feasible solutions.
4. The problems (particularly the `AA` problems) with low density are generally more difficult for the GA in terms of its ability to generate feasible solutions and its final solution quality. The `AA` problems contain considerably more rows (constraints) than the other problems. The `AA` problems are also found to be more difficult using the `CPLEX` solver in terms of their solution time.
5. The `CPLEX` Mixed Integer Solver is able to find optimal solutions for all the problems except `NW05`, `US01` and `AA04`. For `CPLEX` many of the smaller problems are fairly easy to solve, with the integer optimal solution being found after only a small branch-and-bound tree search. What makes these problems easy for `CPLEX` is that the number of integer values in the LP relaxation is relatively high and the difference between the lower bound and the optimal value is relatively small. `CPLEX` was unable to obtain any solution for `NW05` and `US01` because their memory requirements exceeded the memory capacity (48 MB) of our machine. For `AA04`, a feasible solution was found before the memory was exhausted.
6. Overall we must conclude that, for the problems considered in this paper, the GA is not computationally competitive with a modern "black-box" Mixed Integer Solver such as `CPLEX`.
7. We would envisage that our GA would become more effective than `CPLEX` either when the problem is very big or when there is a considerable gap between the LP relaxation solution value and the optimal integer solution value.

Other limited computational experience showed that different GA parameter settings may produce different results because these test problems are real-world problems and the problem structures can be quite different from one to the other (e.g. compare the `AA` problems with the `NW` problems). Therefore, it is possible for our GA to produce better results for some problems by using a particular parameter setting which takes advantage of that particular problem structure. Rather than using different parameter settings for different problems, we in reporting the computational results shown in Tables 2 and 3, decided on one parameter setting which was applied consistently across the entire set of 55 problems.

6 Conclusions

We have developed a heuristic for the set partitioning problem based on a genetic algorithm. Our main interest was to investigate the use of GAs for solving highly constrained problems, in which case we chose the SPP as the input to our GA. We designed a fitness-unfitness pair evaluation function, a heuristic feasibility operator, a new parent selection method and a new population replacement scheme to improve the performance of our GA. The heuristic feasibility operator is specialised for the SPP and is shown to be very effective in constructing feasible solutions. Of the fifty-five real-world problems we tested, only four problems failed to obtain a feasible solution in one of ten random trials. Computational results also indicated that our GA-based heuristic is capable of generating optimal or near-optimal solutions for many problems.

In addition a number of the ideas presented in this paper, specifically separate fitness and unfitness scores and subgroup population replacement, are applicable to any genetic algorithm for constrained problems.

References

[1] J. Arabeyre, J. Fearnley, F. Steiger, and W. Teather. The airline crew scheduling problem: a survey. *Transportation Science*, 3(2):140–163, 1969.

[2] E. Baker and M. Fisher. Computational results for very large air crew scheduling problems. *OMEGA*, 9(6):613–618, 1981.

[3] E. Balas and M. Padberg. Set partitioning: a survey. *SIAM Review*, 18(4):710–760, 1976.

[4] J. Barutt and T. Hull. Airline crew scheduling: supercomputers and algorithms. *SIAM News*, 23(6), 1990.

[5] D. Beasley, D.R. Bull, and R.R. Martin. An overview of genetic algorithms: Part I, fundamentals. *University Computing*, 15:58–69, 1993.

[6] D. Beasley, D.R. Bull, and R.R. Martin. An overview of genetic algorithms: Part II, research topics. *University Computing*, 15:170–181, 1993.

[7] M. Fisher and P. Kedia. Optimal solution of set covering / partitioning problems using dual heuristics. *Management Science*, 36(6):674–688, 1990.

[8] R. Garfinkel and G. Nemhauser. *Integer Programming*, chapter 8, pages 302–304. John Wiley & Sons Inc., New York, 1972.

[9] I. Gershkoff. Optimizing flight crew schedules. *Interfaces*, 19:29–43, 1989.

[10] D.E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G.J.E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[11] F. Harche and G.L. Thompson. The column subtraction algorithm: an exact method for solving weighted set covering, packing and partitioning problems. *Computers & Operations Research*, 21(6):689–705, 1994.

[12] K. Hoffman and M. Padberg. Solving airline crew-scheduling problems by branch-and-cut. *Management Science*, 39(6):657–682, 1993.

[13] J.H. Holland. *Adaption in Natural and Artificial Systems.* MIT press, 1992.

[14] D. Levine. *A parallel genetic algorithm for the set partitioning problem.* PhD thesis, Illinois Institute of Technology, Department of Computer Science, May 1994.

[15] R. Marsten and F. Shepardson. Exact solutions of crew scheduling problems using the set partitioning model: recent successful applications. *Networks*, 11, 1981.

[16] D. Powell and M. Skolnick. Using genetic algorithms in engineering design optimisation with non-linear constraints. In S. Forrest, editor, *Proceeding of the Fifth International Conference on Genetic Algorithms*, pages 424–431. Morgan Kaufmann, 1993.

[17] C.R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*, chapter 4. Blackwell Scientific, 1993.

[18] J. Richardson, M. Palmer, G. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 191–197. Morgan Kaufmann, 1989.

[19] D.M. Ryan and J.C. Falkner. On the integer properties of scheduling set partitioning models. *European Jounal of Operational Research*, 35:422–456, 1988.

[20] W. Spears and K. DeJong. On the virtues of parametized uniform crossover. In R. Belew and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236. Morgan Kaufmann, 1991.

[21] G. Syswerda. Uniform crossover in genetic algorithms. In J. Schafffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1989.