ELSEVIER

European Journal of Operational Research 94 (1996) 392–404

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

Theory and Methodology

A genetic algorithm for the set covering problem

J.E. Beasley *, P.C. Chu

The Management School, Imperial College, London SW7 2AZ, UK

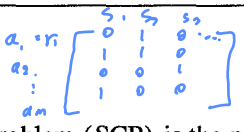
Received July 1994; revised June 1995

Abstract

In this paper we present a genetic algorithm-based heuristic for non-unicost set covering problems. We propose several modifications to the basic genetic procedures including a new fitness-based crossover operator (fusion), a variable mutation rate and a heuristic feasibility operator tailored specifically for the set covering problem. The performance of our algorithm was evaluated on a large set of randomly generated problems. Computational results showed that the genetic algorithm-based heuristic is capable of producing high-quality solutions.

Keywords: Genetic algorithms; Set covering; Combinatorial optimisation

1. Introduction



The set covering problem (SCP) is the problem of covering the rows of an m-row, n-column, zero–one matrix (a_{ij}) by a subset of the columns at minimal cost. Defining $x_j = 1$ if column j (with cost $c_j > 0$) is in the solution and $x_j = 0$ otherwise, the SCP is

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

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

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

Eq. (2) ensures that each row is covered by at least one column and (3) is the integrality constraint. If all the cost coefficients c_j are equal, the problem is called the unicost SCP. The SCP has been proven to be NP-complete [10].

A number of optimal and heuristic solution algorithms have been presented in the literature in recent years. Fisher and Kedia [9] presented an optimal solution algorithm based on a dual heuristic and applied it to SCP's with up to 200 rows and 2000 columns. Beasley [7] combined a Lagrangian heuristic, feasible solution exclusion constraints, Gomory f-cuts and an improved branching strategy to enhance his previous algorithm [5] and solved problems with up to 400 rows and 4000 columns. Of recent interest is the work of Harche and Thompson [13] who developed an exact algorithm, called column subtraction, which is capable of solving large sparse instances of set covering problems. These optimal solution algorithms are based on tree-search procedures.

Among the heuristic methods, Beasley [6] presented a Lagrangian heuristic algorithm and reported that his heuristic gave better-quality results than a number of other heuristics [2,19], for problems in-

* Corresponding author. E-mail: j.beasley@ic.ac.uk
http://mscmga.ms.ic.ac.uk/jeb/jeb.html

SSDI 0377-2217(95)00159-X

تعداد زیر مجموعه‌ها

$U = \{0, 1, \dots, M\}$

$S =$ مجموعه زیرمجموعه

$A \subseteq S \rightarrow A \subseteq U$

M تعداد اعضای مجموعه

$A = \{0, 1, 3, 4\}$

$A = (1, 1, 0, 1, 0, 1, 0, 0, 0, \dots)$

$1 \times M$

n تعداد زیرمجموعه

M

$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 8 & 7 & 6 & 5 \end{bmatrix}$

$5 + 5 \times 0 + 1 \times 1 + \dots$

$1 \quad 2 \quad 4$

$0 \times n + 2 \times 1 + 4 \times 1$

volving up to 1000 rows and 10000 columns. Recently, Jacobs and Brusco [15] developed a heuristic based on simulated annealing and reported considerable success on problems with up to 1000 rows and 10000 columns. Sen [18] investigated the performances of a simulated annealing algorithm and a simple genetic algorithm on the minimal cost set covering problem, but few computational results were given. A comparative study of several different approximation algorithms for the SCP was conducted in Ref. [12].

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 [16]. The theoretical foundations of GAs were originally developed by Holland [14]. The idea of GAs is based on the evolutionary process of biological organisms in nature. During the course of the 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 (i.e. *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 Refs. [3,4,11,16].

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 is considered. Without loss of generality, we shall assume that the columns in the SCP are ordered in *increasing* order of cost and columns of equal cost are ordered in *decreasing* order of the number of rows that they cover. Our modified GA is as follows.

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 SCP since it represents the underlying 0–1 integer variables. We used an n-bit binary string as the chromosome structure where n is the number of columns in the SCP. A value of 1 for the ith bit implies that column i is in the solution. The binary representation of an individual's chromosome (solution) for the SCP is illustrated in Fig. 1. The fitness of an individual is directly related to its objective function value. With the binary representation, the fitness f_i of an individual i is calculated simply by

$$f_i = \sum_{j=1}^{n} c_j s_{ij}\,, \tag{4}$$

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

Fig. 1. Binary representation of an individual's chromosome.

row(gene)	1	2	3	4	5	···	$m-1$	m
string	10	7	10	213	5	···	49	7

Fig. 2. Non-binary representation of an individual's chromosome.

where s_{ij} is the value of the jth bit (column) in the string corresponding to the ith individual and c_j is the cost of bit (column) j.

An important issue concerning the use of the binary representation is that when applying genetic operators to the binary strings, the resulting solutions are no longer guaranteed to be feasible. There are two ways of dealing with infeasible solutions.

One way is to apply a penalty function to penalise the fitnesses of infeasible solutions without distorting the fitness landscape [17]. The other way is to design heuristic operators which transform infeasible solutions into feasible solutions. We chose the later approach of using heuristic operators because a "good" penalty function is often difficult to determine.

The problem of maintaining feasibility may also be resolved by using a non-binary representation. One possible representation is to have the chromosome size equal to the number of rows in the SCP. In this representation, the location of each gene corresponds to a *row* in the SCP and the encoded value of each gene is a *column* that covers that row (see Fig. 2). Since the same column may be represented in more than one gene location, a modified decoding method for fitness evaluation is used by extracting only the unique set of columns which the chromosome represents (i.e. repeated columns are only counted once).

With this representation, feasibility can generally be maintained throughout the crossover and mutation procedures. But the evaluation of the fitness may become ambiguous because the *same* solution can be represented in *different* forms and each form may give a different fitness depending on how the string is represented. Limited computational experience led us to believe that the performance of the GA using this non-binary representation was inferior to that of the GA using the binary representation.

3.2. Parent selection techniques

Parent selection is the task of assigning reproductive opportunities to each individual in the population. There are a number of widely used methods including proportionate selection and tournament selection.

The proportionate selection method calculates the probabilities of individuals being selected as proportional to their fitnesses and selects individuals for mating based on this probability distribution. The tournament selection method works by forming two pools of individuals, each consisting of T individuals drawn from the population randomly. Two individuals with the best fitness, each taken from one of the two tournament pools, are chosen for mating. Using a larger value for T has the effect of increasing selection pressure on the more fit individuals.

We chose the *binary* tournament selection (i.e. $T = 2$) as the method for parent selection because it can be implemented very efficiently (without having to calculate an individual's selection probability as required by proportionate selection). Our study showed that the performance of binary tournament selection in terms of solution quality is comparable to that of proportionate selection.

3.3. Crossover operators

In a traditional GA, simple crossover operators such as one-point or two-point crossover are often used. These crossover operators work by randomly generating one or more *crossover point(s)* and then swapping segments of the two parent strings to produce two child strings. The one-point crossover is formally defined as follows.

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. Generate

a crossover point k, $1 \leqslant k < n$. Then the child strings C_1 and C_2 are

$$C_1 := P_1[1], \ldots, P_1[k], P_2[k+1], \ldots, P_2[n],$$
$$C_2 := P_2[1], \ldots, P_2[k], P_1[k+1], \ldots, P_1[n].$$

The two-point crossover operator is similar to the one-point crossover operator except that it generates two crossover points instead of one. With the one-point and two-point crossover operators, the *location* of the crossover point is clearly important in two respects. Firstly, as the GA converges, it is expected that most genes on the *right* portion of the strings P_1 and P_2 will have a value of zero. This is because SCP solutions will consist of mostly low cost columns at this stage (remember that columns are ordered in increasing cost order). Therefore, if the crossover point(s) is (are) chosen anywhere *within* that region, the resulting child solutions will be *identical* to their parent solutions. Secondly, when the GA converges, the solution *structures* will vary little from one to another. Hence, the ability of the one-point and two-point crossover operators to generate new structures from the parent structures may be limited.

In order to ensure that the child is not identical to one or other of the parents we prefer to work with a restricted one-point crossover operator. This simply consists of generating a crossover point k where

$$\min[i, P_1[i] \neq P_2[i]] \leqslant k < \max[i, P_1[i] \neq P_2[i]].$$

The restricted two-point crossover operator can be defined in a similar way.

Another commonly used crossover technique is the uniform crossover operator. Each gene in the child solution is created by copying the corresponding gene from one or other parent, chosen according to a binary random number generator $[0, 1]$. If the random number is a 0, the gene is copied from the first parent, if it is a 1, the gene is copied from the second parent.

Here we propose a *generalised fitness-based* crossover operator called the *fusion* operator which takes into account both the structure and the relative fitnesses of the parent solutions. The fusion operator produces just a single child (unlike the one-point and two-point crossover operators where two children are produced) and is as follows.

Let f_{P_1} and f_{P_2} be the fitnesses of the parent strings P_1 and P_2 respectively, and let C be the child string. Assuming a minimisation problem, then for all $i = 1, \ldots, n$:

(i) if $P_1[i] = P_2[i]$, then $C[i] := P_1[i] = P_2[i]$,
(ii) if $P_1[i] \neq P_2[i]$, then
  (a) $C[i] := P_1[i]$ with probability $p = f_{P_2}/(f_{P_1} + f_{P_2})$,
  (b) $C[i] := P_2[i]$ with probability $1 - p$.

For example, if $f_{P_1} = 4$, $f_{P_2} = 6$ and $P_1[i] \neq P_2[i]$, the probability that $C[i] := P_1[i]$ is $6/(4 + 6) = 0.6$ and the probability that $C[i] := P_2[i]$ is $1 - 0.6 = 0.4$.

The rationale behind this fitness-based crossover operator can be understood in the following way. Since the overall fitness of an individual is determined by the values of the genes as in Eq. (4), then when combining two parent strings, the choice of whose gene values are passed to the child should be made based on the relative fitnesses of the two parents. It is assumed that the inheritance of a particular gene from a *more* fit parent is likely to contribute *more* to the child's overall fitness than that from a less fit parent. Such choices need to be made only for those genes with different values in both parents. Gene values which are identical in both parents are copied to the child. The advantage of this fitness-based fusion operator is that it is more capable of generating new solutions when the two parent solutions are similar in structure than the one-point or two-point crossover operators. This is because it focuses on the *differences* in the two combining structures. Note that the fusion operator can be applied in any binary-coded GA, not just in a GA for the SCP.

3.4. Variable mutation rate

Mutation is applied to each child after crossover. It works by *inverting* each bit in the solution with some small probability. 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.

Bäck [1] suggested a mutation rate of $1/n$ as a lower bound on the optimal mutation rate, where n is

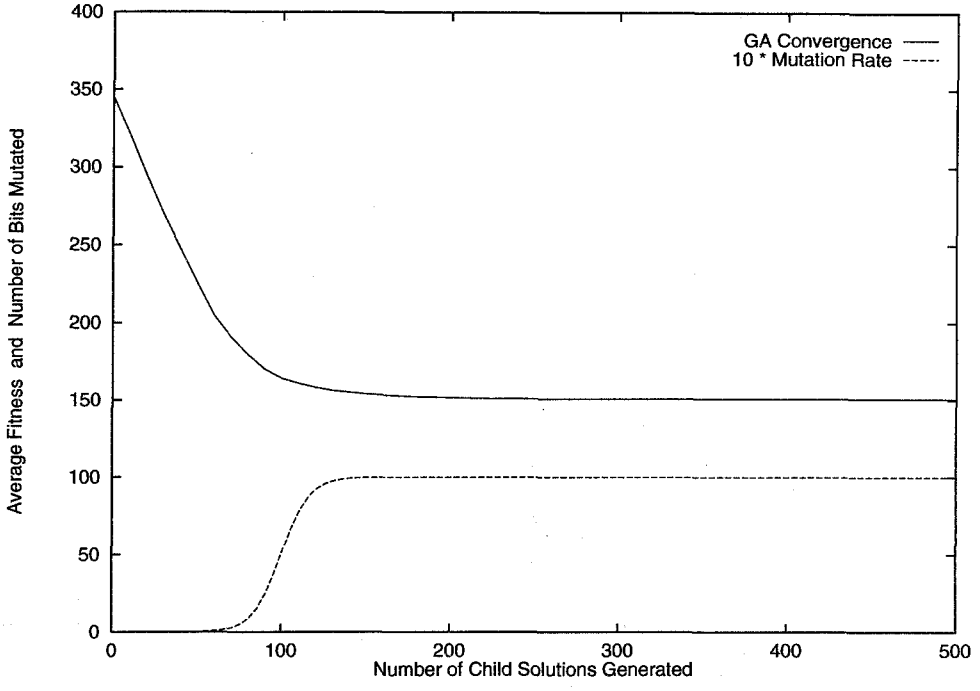


Fig. 3. GA convergence and variable mutation rate.

the length of the chromosome. This lower bound is equivalent to mutating one randomly chosen bit per string. Our study showed that in order for the GA to be effective, a higher mutation rate is necessary when the GA has converged. We also found that it is beneficial to utilise a variable mutation rate rather than a fixed one and this variable rate should depend on the rate at which the GA converges. In other words, at an initial stage of the GA the crossover operator is mainly responsible for the search and so the mutation rate is set at a low value to allow minimal disruption. As the GA progresses, the crossover operator becomes less productive and so the mutation rate increases. When the GA finally converges, the mutation rate will also become stable at some constant rate. The rate at which the GA converges depends on the population replacement method.

The relationship we developed between the convergence of the GA and the variable mutation rate is illustrated in Fig. 3 (note that the mutation rate has been scaled up by a constant factor for display purposes). The convergence of the GA in Fig. 3 is based on the steady-state replacement model. The mutation schedule in Fig. 3 can be generalised as

$$\text{Number of bits mutated} = \left\lceil \frac{m_f}{1+\exp(-4m_g(t-m_c)/m_f)} \right\rceil . \quad (5)$$

where t is the number of child solutions that have been generated, m_f specifies the final stable mutation rate, m_c specifies the number of child solutions generated at which a mutation rate of $m_f/2$ is reached and m_g specifies the gradient at $t = m_c$. Since the number of bits mutated can only take integer values, the return value in Eq. (5) is rounded up to its nearest integer value. Note here that if m_g is set appropriately then this function is effectively a step function at $t = m_c$.

The value of m_f is specified by the user and the values of m_c and m_g are determined based on the rate at which the GA converges for a particular problem. These coefficients will be discussed below in greater detail.

3.5. Heuristic feasibility operator

As mentioned above, the solutions generated by the fusion and mutation operators may violate the problem constraints (i.e. some rows may not be covered).

To make all solutions feasible additional operators are needed. Here we propose a heuristic operator that not only maintains the feasibility of the solutions being generated but also provides an additional local optimisation step in an attempt to make the GA more effective.

The steps required to make each solution feasible involve the identification of all uncovered rows and the addition of columns such that all rows are covered. The search for these missing columns is based on the ratio

$$\frac{\text{cost of a column}}{\text{number of } \textit{uncovered} \text{ rows which it covers}}.$$

Once columns are added and a solution becomes feasible, a local optimisation step is applied to remove any redundant columns in the solution. A redundant column is one such that by removing it from the solution, the solution still remains feasible. The algorithm is as follows.

Let
I = the set of all rows,
J = the set of all columns,
α_i = the set of columns that cover row i, $i \in I$,
β_j = the set of rows covered by column j, $j \in J$,
S = the set of columns in a solution,
U = the set of uncovered rows,
w_i = the number of columns that cover row i, $i \in I$ in S.

(i) Initialise $w_i := |S \cap \alpha_i|, \forall i \in I$.
(ii) Initialise $U := \{i \mid w_i = 0, \forall i \in I\}$.
(iii) For each row i in U (in increasing order of i):
  (a) find the first column j (in increasing order of j) in α_i that minimises $c_j / |U \cap \beta_j|$,
  (b) add j to S and set $w_i := w_i + 1, \forall i \in \beta_j$. Set $U := U - \beta_j$.
(iv) For each column j in S (in decreasing order of j), if $w_i \geqslant 2, \forall i \in \beta_j$, set $S := S - j$ and set $w_i := w_i - 1, \forall i \in \beta_j$.
(v) S is now a feasible solution for the SCP that contains no redundant columns.

Steps (i) and (ii) identify the uncovered rows. Steps (iii) and (iv) are "greedy" heuristics in the sense that in step (iii) columns with low cost ratios are being considered first and in step (iv) columns with high costs are dropped first whenever possible.

3.6. Population replacement model

Once a new feasible child solution has been generated, the child will replace a randomly chosen member (usually one with an *above-average* fitness value) in the population. Note that an above-average fitness means *less* fit. This type of replacement method is called incremental replacement or *steady-state* replacement. Another commonly used method is *generational* replacement, where a new population of children is generated and the *whole* parent population is replaced.

The advantages of the steady-state replacement method are that the best solutions are always kept in the population and the newly generated solution is immediately available for selection and reproduction. Hence, a GA which uses the steady-state replacement method tends to converge faster than one which uses the generational replacement method. Limited computational experience showed that our GA using the steady-state replacement method produced better results than when using the generational replacement method.

When using the steady-state replacement method, 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 N solution structures in the population. Allowing duplicate solutions to exist in the population may be undesirable because a population could come to consist of N identical solutions.

3.7. Overview

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

(i) Generate an initial population of N random solutions. Set $t := 0$.
(ii) Select two solutions P_1 and P_2 from the population using binary tournament selection.
(iii) Combine P_1 and P_2 to form a new solution C using the fusion crossover operator.
(iv) Mutate k randomly selected columns in C where k is determined by the variable mutation schedule.
(v) Make C feasible and remove redundant columns in C by applying the heuristic opera-

tor.

(vi) If C is identical to any one of the solutions in the population, go to step (ii); otherwise, set $t := t + 1$ and go to step (vii).

(vii) Replace a randomly chosen solution with an above-average fitness in the population by C (steady-state replacement method).

(viii) Repeat steps (ii)–(vii) until $t = M$ (i.e. M non-duplicate) solutions have been generated. The best solution found is the one with the smallest fitness in the population.

4. Parameter consideration

Problem-dependent parameters such as the population size, the choice of the initial population and the variable mutation rate are considered in this section.

4.1. Population size and initial population

In principle, the size of the population and the initial population are chosen such that the solution domain associated with the population is adequately covered. The size of the population in turn depends on the criteria for selecting the initial solutions. To illustrate this point, we assume that each of the initial solutions S_p is generated randomly using the following method (using the same notation as in Section 3.5).

(i) Initialise $S_p := \emptyset$. Initialise $w_i := 0, \forall i \in I$.

(ii) For each row i in I:
  (a) randomly select a column j in α_i,
  (b) add j to S_p and set $w_i := w_i + 1, \forall i \in \beta_j$.

(iii) Let $T := S_p$.

(iv) Randomly select a column j, $j \in T$ and set $T := T - j$. If $w_i \geqslant 2, \forall i \in \beta_j$, set $S_p := S_p - j$ and set $w_i := w_i - 1, \forall i \in \beta_j$.

(v) Repeat step (iv) until $T = \emptyset$.

Step (ii) generates a random *feasible* solution. Step (iv) eliminates redundant columns and is similar to that used in the heuristic feasibility operator except that the redundant columns are dropped in a *random* manner rather than by cost.

Now consider a randomly generated SCP with density ϕ (the density of a SCP is the fraction of ones in the a_{ij} matrix). The average number of columns in each solution S_p is $1/\phi$ columns. An initial population of size N will consist on average of N/ϕ columns. In order for the initial population to cover the *entire* solution domain (i.e. having $\bigcup_{p=1}^{N} S_p = J$), we must have $N/\phi \geqslant \mu n$ where $\mu \geqslant 1$ and is a factor representing the *average* number of appearances of a column in the initial population. The size of the population is then

$$N = \mu \phi n . \tag{6}$$

Eq. (6) shows that the required size of the population is proportional to the number of columns and the density of a SCP. For a large SCP, the size of the population may become very large. For example, if we let $\mu = 20$ to ensure adequate coverage, a SCP with 10 000 columns and 5% density will require a population size $N = (20)(10000)(0.05) = 10000$. This size is clearly too large for the GA to work efficiently.

In order to make population size less dependent on problem size, we need to modify the initial solution selection rule above. One simple way is to generate an initial population that covers only *part* of the whole solution domain. To do this, we modify step (iia) above to

(iia) randomly select a column j in α_{ik}, $\alpha_{ik} \subset \alpha_i$,

where α_{ik} is defined as the set of the k *least*-cost columns in α_i. Since the columns in α_i are in increasing order of cost, α_{ik} is simply the set of the first k columns in α_i and $\alpha_{ik} = \alpha_i$ when $|\alpha_i| < k$. The initial population will now need to cover $\alpha_{1k} \cup \alpha_{2k} \cup \cdots \cup \alpha_{mk}$ which is a subset of J. In general, the value of k should be chosen such that there is a high probability that the set of columns in the optimal solution S_{opt} is a subset of $\bigcup_{i=1}^{m} \alpha_{ik}$. For the non-unicost SCP, S_{opt} generally consists of columns which have low cost. In our study, we chose $k = 5$ for all the test problems we considered.

To estimate the effect that the modified initial population selection rule above has on the required population size with respect to different problem sizes and densities, we obtained some empirical data based on our test problems. Table 1 shows the average number of columns in the new restricted solution domain ($|\bigcup_{i=1}^{m} \alpha_{ik}|$) and the values of μ with respect to different problem sizes and densities when $N = 100$ and $k = 5$. The sizes and densities of these test problems are given in Table 2. The data in Table 1 shows that by modifying the initial population selection criterion,

Table 1
Average μ with respect to different problem sizes and densities

Problem set	Average $\left\lvert \bigcup_{i=1}^{m} \alpha_{i5} \right\rvert$	μ
4	361	18.7
5	357	18.5
6	173	22.4
A	391	19.8
B	185	24.1
C	418	20.8
D	195	25.1
E	105	29.5
F	57	31.1
G	489	25.0
H	225	28.4

Table 2
Test problem details

Problem set	Rows (m)	Columns (n)	Density (%)	Number of problems
4	200	1000	2	10
5	200	2000	2	10
6	200	1000	5	5
A	300	3000	2	5
B	300	3000	5	5
C	400	4000	2	5
D	400	4000	5	5
E	500	5000	10	5
F	500	5000	20	5
G	1000	10000	2	5
H	1000	10000	5	5

only a small population size ($N = 100$) is necessary to provide adequate coverage of the initial restricted solution domain *regardless* of the size and density of the problem.

4.2. Mutation schedule

The variable mutation rate described in Section 3.4 involved three constant coefficients. These were:

(i) m_f, which specifies the final stable mutation rate;
(ii) m_c, which specifies the number of child solutions (t) that are generated before the mutation schedule reaches half of the stable mutation rate;
(iii) m_g, which specifies the gradient of the mutation function at $t = m_c$.

As discussed in Section 3.4, the mutation operator becomes the main force for searching when the GA begins to converge. However, because mutation is only an intermediate step in the reproduction process, the bits which have been mutated may still be altered later by the heuristic feasibility operator. Therefore, m_f does not explicitly determine the magnitude of the search by the mutation operator as it would do without the presence of the heuristic feasibility operator. Our study indicated that the final converged solutions are generally *not* very sensitive to the values of m_f. Choosing a value between 5 and 10 for m_f is usually adequate to prevent the GA from being trapped in a local minimum. The values of m_c and m_g are determined according to the manner in which the GA converges. In principle, the mutation schedule should "match" the convergence of the GA in the manner shown in Fig. 3. The GA convergence curve can be estimated by running the GA once without the mutation operator. The desirable mutation curve is then approximated (through visual inspection) by manipulating the mutation function in Eq. (5) using m_c and m_g.

Another important parameter concerning mutation is in which *set* of bits (columns) in a string should the mutation take place such that the GA is effective. It is obvious that mutating high-cost columns is not as productive as mutating low-cost columns since the chance of high-cost columns constituting part of the optimal solution is rather small. Therefore, it is reasonable, as well as computationally more efficient, to allow the mutation to occur only in a set of "elite" bits (columns) which have a relatively better chance to be in the optimal solution. In our GA, we defined the elite column set based on the same criterion as that used for generating the initial population, that is

$$S_{\text{elite}} = \bigcup_{i=1}^{m} \alpha_{i5}\,, \tag{7}$$

where α_{i5} is the set of the 5 least-cost columns in α_i.

5. Computational results

The algorithm presented in this paper was coded in C and tested on a Silicon Graphics Indigo workstation (R4000, 100MHz). Our computational study was conducted on 11 test problem sets (a total of 65 SCP's) of various sizes and densities. These test problem sets were obtained electronically from OR-Library

Table 3
Mutation coefficients

Problem set	m_f	m_c	m_g
4	10	200	1.3
5	10	200	0.6
6	10	200	2.0
A	10	200	2.0
B	10	300	1.0
C	10	200	2.0
D	10	200	2.0
E	10	350	1.1
F	10	400	1.0
G	10	250	1.3
H	10	400	0.8

(e-mail the message *scpinfo* to *o.rlibrary@ic.ac.uk*). The details of these test problems are given in Table 2.

Problem sets 4–6 and A–D are ones for which optimal solution values are known. Problem sets E–H are large SCP's for which optimal solution values are not known. Note here that we did not incorporate into our algorithm any of the problem reduction tests [5,6] available for SCP's.

In our computational study, 10 trials of the GA heuristic (each with a different random seed) were made for each of the test problems. Each trial terminated when $M = 100\,000$ non-duplicate solutions had been generated. The population size N was set to 100 for all the problems and the coefficients of the mutation function (m_f, m_c and m_g) for each of the problems sets are listed in Table 3. The coefficients in Table 3 are determined with relation to the GA convergence by using the first problem in each of the problem sets. Since the convergence behaviours are similar within each of the problem sets, we used only the first problem in each set to generalise the convergence behaviour for that set. Table 3 shows that the final mutation rate m_f is set to 10 for all the problems and the coefficients m_c and m_g are set based on the GA convergence curves given in Fig. 4. Since the m_c's and m_g's are relatively similar for all the problems, we made a further generalisation and used $m_c = 200$ and $m_g = 2.0$ for all the test problems. The GA convergence curves in Fig. 4 also show that the GA generally converges very rapidly (when roughly 200 non-duplicate child solutions had been generated).

The computational results are shown in Table 4. In Table 4 we give, for each problem:

(i) the optimal solution value (problem sets 4–6 and A–D) from Ref. [7] or the previous best-known solution value (problem sets E–H) from Ref. [15];
(ii) the best solution value in each of the 10 trials, the average percentage deviation from the optimal solution value (problem sets 4–6 and A–D) or the average percentage deviation from the previous best-known solution value (problem sets E–H);
(iii) the average solution time and the average execution time.

The average percentage deviation (σ) is calculated by $\sum_{i=1}^{10}(S_{Ti} - S_o)/(10S_o) \times 100\%$ where S_{Ti} is the ith trial minimum (best) solution value and S_o is the optimal solution value or the previous best-known solution value. The solution time is measured in CPU seconds and is the time that the GA takes to first reach the final best solution. The execution time is the time (CPU seconds) that the GA takes to generate 100 000 non-duplicate child solutions. For a comparative performance of different computer systems, see Ref. [8].

Examining Table 4, we observe that:

(i) For problem sets 4–6 and A–D, the GA-based heuristic found optimal solutions in at least one of the 10 trials for all but one of the 45 test problems. The average percentage deviations from the optimal values are between zero and 1.4%.
(ii) For problem sets E–H, the GA-based heuristic produced *better* (or equal) solutions than the previous best-known solutions in at least one of the 10 trials for all 20 test problems.
(iii) In 7 out of the 20 problems in problem sets E–H, the GA-based heuristic generated *better* solution values than the previous best-known solutions. The negative average percentage deviation indicates the average percentage improvement from the previous best-known solution.
(iv) The average solution times are reasonably small for all the problems (under 800 CPU seconds). The average solution times are also much less than the average execution times. This suggests that the GA terminating condition of generating 100 000 solutions could be reduced without affecting solution quality.

We also compared the performance of different crossover operators, namely, the one-point, the two-point, the uniform and the fusion crossover operators

Table 4
Computational results

Problem	Optimal/PBS[a]	Best solution in each of the 10 trials										Average % σ	Average solution time	Average execution time
4.1	429	o[b]	o	o	432	o	430	430	430	o	430	0.16	104.2	279.4
4.2	512	o	o	o	o	o	o	o	o	o	o	0	3.2	276.6
4.3	516	o	o	o	o	o	o	o	o	o	o	0	5.8	244.2
4.4	494	o	o	o	o	o	502	o	o	o	o	0.16	50.2	238.7
4.5	512	o	o	o	o	o	o	o	o	o	o	0	15.6	273.9
4.6	560	o	o	o	o	o	o	o	o	o	o	0	5.7	219.0
4.7	430	o	o	432	o	o	o	o	o	o	o	0.05	49.0	259.8
4.8	492	493	o	o	o	o	o	o	o	o	o	0.02	73.0	216.3
4.9	641	o	645	o	o	o	o	o	650	645	645	0.33	48.1	235.0
4.10	514	o	o	o	o	o	o	o	o	o	o	0	4.8	277.6
5.1	253	o	o	o	o	o	o	o	o	o	o	0	14.9	382.5
5.2	302	305	o	305	o	305	o	o	305	305	o	0.50	90.4	380.5
5.3	226	228	228	228	228	228	228	228	228	228	228	0.88	3.9	465.1
5.4	242	o	o	o	o	243	o	o	o	o	o	0.04	3.6	376.0
5.5	211	o	o	o	o	o	o	o	o	o	o	0	5.3	458.2
5.6	213	o	o	o	o	o	o	o	o	o	o	0	10.6	573.8
5.7	293	o	o	o	o	o	o	o	o	o	o	0	68.9	410.4
5.8	288	289	289	289	o	289	289	289	289	289	o	0.28	8.5	436.9
5.9	279	o	o	o	o	o	o	o	o	o	o	0	4.8	406.2
5.10	265	o	o	o	o	o	o	o	o	o	o	0	6.8	516.8
6.1	138	o	o	o	o	o	o	o	o	o	o	0	16.3	276.6
6.2	146	o	147	o	o	o	o	o	o	o	147	0.14	74.4	256.7
6.3	145	o	o	o	o	o	o	o	o	o	o	0	4.2	240.0
6.4	131	o	o	o	o	o	o	o	o	o	o	0	1.7	256.5
6.5	161	o	o	o	o	o	o	164	o	o	o	0.19	4.3	246.6
A.1	253	o	254	o	o	o	o	o	o	o	254	0.79	78.6	599.6
A.2	252	o	o	o	o	o	o	o	o	o	o	0	115.9	577.4
A.3	232	o	o	o	o	233	233	o	233	233	233	0.22	44.9	622.7
A.4	234	o	o	o	o	o	o	o	o	o	o	0	16.1	686.8
A.5	236	o	o	o	o	o	o	o	o	o	o	0	8.4	663.9
B.1	69	o	o	o	o	o	o	o	o	o	o	0	7.1	788.0
B.2	76	o	o	o	o	o	o	o	o	o	o	0	4.1	817.7
B.3	80	o	o	o	o	o	o	o	o	o	o	0	250.8	668.5
B.4	79	o	o	o	o	o	o	o	o	o	o	0	10.6	752.7
B.5	72	o	o	o	o	o	o	o	o	o	o	0	1.9	726.3
C.1	227	o	o	229	o	o	o	o	o	o	o	0.09	66.4	902.1
C.2	219	o	o	221	221	221	221	o	o	o	221	0.46	14.4	956.0
C.3	243	o	248	245	247	251	248	248	o	244	247	1.40	191.3	786.8
C.4	219	o	o	o	o	o	o	o	o	220	o	0.05	51.1	803.0
C.5	215	o	o	216	o	o	o	o	o	o	o	0.05	28.5	1116.5
D.1	60	o	o	o	o	o	o	o	o	o	o	0	4.9	1240.6
D.2	66	o	o	o	o	o	o	o	o	o	o	0	70.2	992.6
D.3	72	o	o	73	o	o	o	o	73	o	o	0.28	277.5	1091.3
D.4	62	o	o	o	o	o	o	o	o	o	o	0	26.0	1019.9
D.5	61	o	o	o	o	o	o	o	o	o	o	0	28.2	1146.1
E.1	29	o	o	o	o	o	o	o	o	o	o	0	13.5	2411.3
E.2	30	31	31	o	31	o	o	31	31	o	31	2.00	213.5	2481.2
E.3	27	28	28	28	28	o	28	28	o	28	o	2.60	68.1	2488.3
E.4	28	o	o	o	o	o	o	o	o	o	o	0	190.8	2474.8
E.5	28	o	o	o	o	o	o	o	o	o	o	0	12.4	2497.9

Table 4 — continued

Problem	Optimal/PBS[a]	Best solution in each of the 10 trials										Average % σ	Average solution time	Average execution time
F.1	14	o	o	o	o	o	o	o	o	o	o	0	27.0	4020.8
F.2	15	o	o	o	o	o	o	o	o	o	o	0	27.6	4428.4
F.3	14	o	o	o	o	o	o	o	o	o	o	0	94.3	4254.5
F.4	14	o	o	o	o	o	o	o	o	o	o	0	74.1	4568.2
F.5	14	o	*13*	o	*13*	o	o	o	o	*13*	o	−2.14	53.7	4608.0
G.1	179	*178*	*178*	*178*	o	*176*	*178*	*178*	*178*	*178*	*176*	−0.73	361.0	2577.6
G.2	158	*155*	*155*	*157*	o	*155*	o	*157*	o	*155*	*155*	−1.08	127.4	2902.4
G.3	169	o	*166*	*168*	*168*	*168*	*168*	o	*167*	*168*	*168*	−0.65	249.7	2521.2
G.4	172	*171*	*170*	*170*	*168*	o	*168*	*170*	*171*	*171*	o	−0.99	532.3	2803.1
G.5	168	o	o	169	170	174	169	169	169	169	169	0.83	194.1	2957.1
H.1	64	o	o	o	o	o	o	o	o	o	o	0	594.4	4306.7
H.2	64	o	o	o	o	o	o	o	o	o	o	0	187.4	4600.0
H.3	60	*59*	o	*59*	*59*	*59*	*59*	*59*	*59*	*59*	*59*	−1.50	637.3	4562.9
H.4	59	o	60	*58*	*58*	*58*	o	60	o	60	*58*	−0.17	770.3	4427.9
H.5	55	o	o	o	56	o	o	o	o	o	o	0.18	158.9	4488.1

[a] Previous best-known solution value.
[b] Optimal solution value or previous best-known solution value.

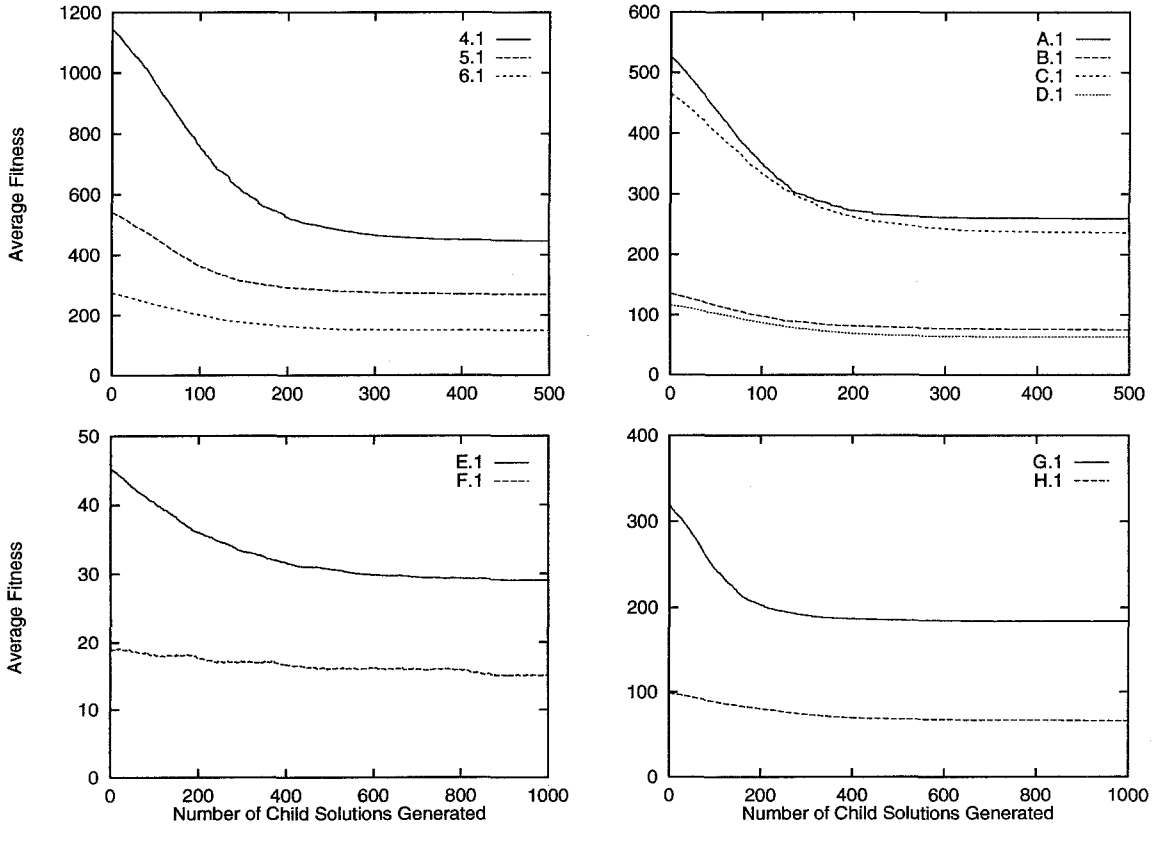


Fig. 4. GA convergences.

Table 5
Average redundancy rates

Problem set	One-point	Two-point	Uniform	Fusion
4	59.4	57.9	44.4	45.3
5	60.4	57.8	46.5	47.0
6	37.5	34.4	26.7	27.3
A	56.1	53.8	42.9	45.5
B	33.6	33.3	24.4	25.9
C	61.2	58.0	47.0	47.5
D	44.7	43.4	31.6	32.0
E	18.7	18.4	14.7	15.2
F	4.2	4.1	2.8	3.3
G	64.1	64.0	49.6	49.5
H	38.3	37.7	26.6	27.3

in terms of solution quality and computational cost using all the problem sets. We found that the four different crossover techniques are comparable in terms of solution quality and solution time. The differences in the number of optimal/current best-known solutions found between the crossover operators were slight, although the fusion operator performed best, failing to find the optimal/current best-known solution in only one of the 65 test problems. Overall we can conclude that our GA-based heuristic performs well irrespective of the choice of crossover operator.

To measure the efficiency of the GA using each of the crossover operators, we recorded the redundancy rate for each of the four crossover operators as shown in Table 5. The redundancy rate is defined as

$$\frac{\#\text{ duplicate children}}{\#\text{ duplicate children} + \#\text{ non-duplicate children}} \times 100\%$$

Recall here that a duplicate child is discarded in the steady-state replacement model. Table 5 shows that the one-point and two-point crossover operators had higher redundancy rates than the uniform and fusion crossover operators. This indicates that the one-point and two-point crossover operators are less successful at generating new solution structures from the mating parents than the uniform and fusion crossover operators.

6. Conclusion

We have developed a heuristic for the non-unicost set covering problem based on genetic algorithms. We designed a new crossover-fusion operator, a variable mutation rate and a heuristic feasibility operator to improve the performance of our GA. Computational results indicated that our GA-based heuristic is able to generate optimal solutions for small-size problems as well as to generate high-quality solutions for large-size problems.

References

[1] T. Bäck, "Optimal mutation rates in genetic search", in: S. Forrest, ed., *Proc. Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 1993, 2–9.

[2] E. Balas and A. Ho, "Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study", *Mathematical Programming Study* 12 (1980) 37–60.

[3] D. Beasley, D.R. Bull, and R.R. Martin, An overview of genetic algorithms: Part I. Fundamentals", *University Computing* 15 (1993) 58–69.

[4] D. Beasley, D.R. Bull, and R.R. Martin, "An overview of genetic algorithms: Part II. Research topics", *University Computing* 15 (1993) 170–181.

[5] J.E. Beasley, "An algorithm for set covering problems", *European Journal of Operational Research* 31 (1987) 85–93.

[6] J.E. Beasley, "A Lagrangian heuristic for set-covering problems", *Naval Research Logistics* 37 (1990) 151–164.

[7] J.E. Beasley and K. Jornsten, "Enhancing an algorithm for set covering problems", *European Journal of Operational Research* 58 (1992) 293–300.

[8] J.J. Dongarra, "Performance of various computers using standard linear equations software", *Computer Architecture News* 20 (1992) 22–44.

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

[10] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[11] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, New York, 1989.

[12] T. Grossman and A. Wool, "Computational experience with approximation algorithms for the set covering problem", Working paper, Theoretical Division and CNLS, Los Alamos National Laboratory, Los Alamos, NM, February 1995.

[13] 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 (1994) 689–705.

[14] J.H. Holland, *Adaption in Natural and Artificial Systems*, MIT Press, Cambridge, MA, 1975.

[15] L.W. Jacobs and M.J. Brusco, "A simulated annealing-based heuristic for the set-covering problem", Working paper, Operations Management and Information Systems Department, Northern Illinois University, Dekalb, IL, March 1993.

[16] C.R. Reeves, *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific, 1993.

[17] J. Richardson, M. Palmer, G. Liepins, and M. Hilliard, "Some guidelines for genetic algorithms with penalty functions", in: J. Schaffer, ed., *Proc. Third International Conference on Genetic Algorithms*, Morgan Kaufmann, 1989, 191–197.

[18] S. Sen, "Minimal cost set covering using probabilistic methods", *Proc. 1993 ACM/SIGAPP Symposium on Applied Computing*, 1993, 157–164.

[19] F.J. Vasko and G.R. Wilson, "An efficient heuristic for large set covering problems", *Naval Research Logistics Quarterly* 31 (1984) 163–171.