A Group Encoding Technique for Set Partitioning Problems

Congjun Yang, Dipankar Dasgupta, Yuehua Cao
Department of Mathematical Sciences
The University of Memphis, Memphis TN 38152
E-mails: yangc,dasgupta,ycao@msci.memphis.edu

Abstract

In this paper, we describe an efficient GA representation scheme for solving the set-partitioning problem (SPP). The SPP is an important class of combinatorial optimization problem that may be found in many industrial/business applications. The paper introduces a group representation scheme in a genetic search where columns of the partition matrix (problem space) with the same characteristics are grouped together. We gave some theoretical analysis to argue that the proposed grouping representation reduces the search space substantially. We used a steady-state GA in conjunction with a repair operator to search for the optimal solution in the representation space. The performance is tested on a set of scheduling problems – airline flight crew scheduling. The results of these experiments are reported with some concluding remarks.

1 INTRODUCTION

The *set partitioning problem* (SPP) is the problem of selecting a set of mutually disjoint subsets of the universal set such that the union of the chosen subsets is exactly the universal set, and the total cost associated with selected subsets is minimal. More precisely, consider a set $I = \{1, \cdots, m\}$ and a set $P = \{P_1, \cdots, P_n\}$, where $P_j \subseteq I$, $j \in J = \{1, ..., n\}$. A subset $J^* \subseteq J$ is a partition of I if

$$\bigcup_{j \in J^*} P_j = I \text{ and } P_j \cap P_k = \emptyset, \quad \forall j, k \in J^*, \ j \neq k.$$

A cost c_j is associated with each set P_j, $j \in J$. The SPP is to find a partition of minimum cost. It is also a zero-one *integer linear programming problem* (ILP). Its ILP formulation is as follows

$$\text{Minimize} \qquad \sum_{j=1}^{n} c_j x_j \,, \tag{1.1}$$

$$\text{Subject to} \qquad \sum_{j=1}^{n} a_{ij} x_j = 1, \quad i = 1, \cdots\cdots, m, \tag{1.2}$$

$$x_j \in \{0, 1\} \quad j = 1, \cdots\cdots, n, \tag{1.3}$$

where $a_{ij} = 0$ or 1. The SPP is generally represented by a unique partition matrix: $(a_{ij})_{m \times n}$, where the rows of the partition matrix are represented by the set $I = \{1, ..., m\}$ and the columns (or variables) are represented by the set $J = \{1, ..., n\}$.

The set partition problem is a difficult combinatorial optimization problem, and has been studied extensively over the years because of its many real-world applications (Hoffman and Padberg (1993), Levine (1994)). One of the well-known applications of the SPP is airline crew scheduling. In this formulation, each row ($i = 1, ..., m$) represents a flight leg (a takeoff and landing) that must be scheduled. The columns ($j = 1, ..., n$) represent feasible round-trip rotations for a crew. Associated with each rotation there is a cost, c_j. The partition matrix $(a_{ij})_{m \times n}$ is given by the following

$$a_{ij} = \begin{cases} 1, & \text{if flight } i \text{ is covered by rotation } j; \\ 0, & \text{otherwise.} \end{cases} \tag{1.4}$$

Airline crew scheduling is a very visible and economically significant problem. The objective of the crew scheduling problem is to find the best schedule, i.e., a collection of crew rotations such that each flight is covered by exactly one rotation and the cost is minimum.

In this paper, we introduce a grouping representation scheme for genetic encoding in order to reduce the search space. The rest of the paper is organized as follows. Section 2 gives a brief survey of some related works of SPP. In section 3, we describe the group encoding scheme, and the genetic operators used. We then report results of some experiments on airline crew scheduling in section 4. Finally, section 5 provides some concluding remarks.

2 RELATED WORK

Because of the widespread application of SPP, a number of algorithms have been developed in the last three decades. They can be classified into two categories: exact algorithms that attempt to solve the SPP to optimality, and heuristic approximate algorithms that try to find a reasonably good solution quickly. The implicit enumeration method (Garfinkel and Nemhauser(1972), Marsten (1974)) is the first exact algorithm for the SPP. Then, a method based on cutting planes was proposed by Balas and Padberg (1976). The most successful method in the first category is the work of Hoffman and Padberg (1993). They used the approach of branch-and-cut, a branch-and-bound like scheme. Another important approach is is to solve the linear programming (LP) relaxation of the SPP (Gershkoff (1989), Bixby (1990)). In this approach, the integrality constraint on x_i is relaxed and the corresponding LP problem is solved. At the end the fractional values are resolved.

There are a few other heuristic algorithms for the SPP exist in the literature. Ryan and Falkner (1988) provided a method of obtaining a good feasible solution by imposing additional structure derived from the real data sets. Levine (1994) experimented with a parallel genetic algorithm for the SPP, and used penalty terms in the fitness measure. His algorithm is based on an island model having multiple independent sub-populations where highly fit solutions occasionally migrate between the islands. Test results on forty real-world SPP showed that his algorithm was able to find optimal solutions for some problems but had difficulty in finding feasible solutions for problems having many rows and columns (increased problem size).

Noticing the weakness of the penalty methods of Levine(1994), Chu (1997) used separate *fitness* and *unfitness* scores in stead of combining the objective and penalty terms into a single fitness measure. Accordingly, he classifies the population into four subgroups of distinct characteristics. It was this mutual exclusive population classification that made possible the effective *ranking replacement* method, which took into account both the fitness and unfitness scores when selecting a member for replacement. Considering that some rows might be under-covered or over-covered as a result of crossover and mutation, Chu designed a heuristic improvement operator that includes two basic procedures: DROP and ADD. The aim of these operators was to improve the solution by moving to a near feasible, or possibly feasible, solution. He also used a *matching selection* method to select parents such that combining them would result in an improvement in feasibility without undermining solution quality. According to the matching selection, the first parent P_1 was selected using a binary tournament based on fitness and the second parent P_2 was then selected that gave a maximum compatibility score. Chu's experiments on 55 real world problems gave a better result than that of Levine(1994). Despite some very encouraging results, it was still not competitive with the existing exact solvers, such as CLPEX, in both speed and quality for the problems tested.

3 A GROUP REPRESENTATION SCHEME FOR THE SPP

To apply a genetic search, the encoding of the problem is the first issue we need to address. The representation scheme used by Levine and Chu was straightforward. The SPP was coded as a string of length n over the binary alphabet {0, 1}, representing the underlying 0 – 1 integer variables. In this representation, a bit in the string is associated with each column. The j^{th} bit is 1 if the j^{th} column is included in the solution and 0 otherwise. With an $m \times n$ input matrix, the number of different ways to choose i columns from n is C_n^i. Hence the exact size of the search space *(SS)* for an n column matrix can be expressed as:

$$SS = \sum_{i=1}^{n} C_n^i = 2^n - 1 \tag{3.1}$$

In order to reduce the search space, some standard preprocessing procedures have been developed by Hoffman and Padberg (1993) and Levine (1994). Though they work effectively, the reduction is still insignificant compared to the original problem size. In fact, many treasured properties remain unexplored in the input matrix. These properties, when properly utilized, may reduce the problem size to a much manageable level and produce better results.

$$\begin{array}{cc}
\text{Cost:} & 3\;5\;4\;2\;6\;4\;8\;3\;5\;8\;8\;3\;5\;6\;3 \\
\text{Flight Leg} & \begin{bmatrix} 1\;1\;0\;1\;1\;0\;0\;0\;1\;0\;1\;0\;0\;0\;0 \\ 1\;1\;1\;0\;0\;0\;0\;1\;1\;1\;0\;0\;1\;1\;1 \\ 0\;0\;0\;0\;1\;1\;0\;0\;1\;0\;0\;0\;0\;0\;0 \\ 0\;0\;1\;1\;0\;1\;1\;0\;1\;0\;1\;1\;0\;0\;0 \\ 1\;0\;0\;0\;0\;0\;0\;1\;0\;1\;0\;0\;0\;0\;1 \end{bmatrix} \\
 & \text{Crew Round-trip Rotation}
\end{array}$$

Figure 1. An instance of SPP given by the input matrix and the associated cost for each column.

Figure 1 shows a typical instance of the SPP. In the input matrix of a SPP, each row presents a constraint that has to be satisfied, i.e., it has to be covered by exactly one column. The difficulty to satisfy a certain constraint depends on the number of 1's in that row. With many 1's in a row, there are certainly many different ways to choose columns to cover that row. In other words, the fewer 1's there are in a certain row, the more difficult the constraint is to be satisfied. For instance, if there are only two columns covering a certain row, then a solution can not be a feasible solution when none of these two columns is selected. Only when one of the two columns is selected, is it possible for a solution to be feasible. With this observation, we can say that the number of 1's in a row characterizes that row. More precisely, we give the following definition.

DEFINITION 1: The characteristic of a row $i \in I = \{1, 2, \cdots, m\}$, denoted as *ch_row*($i$), is the sum of 1's in the row, that is,

$$ch\_row(i) = \sum_{j \in J} a_{ij} \tag{3.2}$$

Note that the characteristic of a row is basically due to the corresponding columns that have a "1" in that row. So, in any column j, if a_{ij} = 1, a_{ij} will contribute to the characteristic of the i^{th} row. In order to define the characteristic of a column, we introduce the characteristic for each entry a_{ij} next.

DEFINITION 2: The characteristic of an entry a_{ij}, denoted as $ch(a_{ij})$, is defined to be the characteristic of the corresponding row if a_{ij} = 1 and ∞ otherwise. More precisely, we have

$$ch(a_{ij}) = \begin{cases} \infty & if \quad a_{ij} = 0 \\ ch\_row(i) & if \quad a_{ij} = 1 \end{cases} \tag{3.3}$$

Hence, in any column, all 1's in the column have their characteristics less than infinity. With the above definitions, we can give the definition of the characteristic of a column.

DEFINITION 3: The characteristic of a column $j \in J = \{1, 2, \cdots, n\}$, denoted as *ch_col*($j$), is a pair of numbers <U_j, D_j>. Here, $U_j = \min_{i \in I}\{ch(a_{ij})\}$ and $D_j = i \in I$ such that $U_j = ch(a_{ij})$. That is, U_j is the minimum characteristic of all entries in column j and D_j is the index of the corresponding row that has the minimum characteristic. U_j is called the *uniqueness* of the column and D_j is called the *defining row* for the uniqueness of the column.

Note that the characteristics of any two columns are the same if and only if both the uniqueness and the defining row are the same. The following fact is an immediate result of the above definitions.

CLAIM 1: Two columns have the same uniqueness if the defining rows of them are the same. Thus, the characteristics of two columns are the same if and only if the defining rows are the same.

$$\begin{array}{lccc}
 & \text{Cost: } 3\ 5\ 4\ 2\ 6\ 4\ 8\ 3\ 5\ 8\ 8\ 3\ 5\ 6\ 3 & \text{\# of 1's} & \text{rank} \\
A = & \left[\begin{array}{ccccccccccccccc}
1&1&0&1&1&0&0&0&1&0&1&0&0&0&0\\
1&1&1&0&0&0&0&1&1&1&0&0&1&1&1\\
0&0&0&0&1&1&0&0&1&0&0&0&0&0&0\\
0&0&1&1&0&1&1&0&1&0&1&1&0&0&0\\
1&0&0&0&0&0&0&1&0&1&0&0&0&0&1
\end{array}\right] & \begin{array}{c}6\\9\\3\\7\\4\end{array} & \begin{array}{c}3\\5\\1\\4\\2\end{array}
\end{array}$$

(a). Shows the input matrix with the associated cost of each column, and the characteristics and ranks of each row

$$\begin{array}{lcc}
 & \text{Cost: } 3\ 5\ 4\ 2\ 6\ 4\ 8\ 3\ 5\ 8\ 8\ 3\ 5\ 6\ 3 & \text{rank} \\
A = & \left[\begin{array}{ccccccccccccccc}
1&1&0&1&1&0&0&0&1&0&1&0&0&0&0\\
1&1&1&0&0&0&0&1&1&1&0&0&1&1&1\\
0&0&0&0&1&1&0&0&1&0&0&0&0&0&0\\
0&0&1&1&0&1&1&0&1&0&1&1&0&0&0\\
1&0&0&0&0&0&0&1&0&1&0&0&0&0&1
\end{array}\right] & \begin{array}{c}3\\5\\1\\4\\2\end{array} \\
\text{Group Index:} & 2\ 3\ 4\ 3\ 1\ 1\ 4\ 2\ 1\ 2\ 3\ 4\ 5\ 5\ 2 &
\end{array}$$

(b). The input matrix with group index for each column

$$\begin{array}{lc}
 & \left[\begin{array}{ccccccccccccccc}
1&0&1&1&0&0&0&1&1&1&0&0&0&0&0\\
0&0&1&1&1&1&1&1&0&0&1&0&0&1&1\\
1&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&0&0&0&0&1&1&1&1&1&0&0\\
0&0&0&1&1&1&1&0&0&0&0&0&0&0&0
\end{array}\right] \\
\text{Group Index:} & 1\ 1\ 1\ 2\ 2\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 4\ 5\ 5
\end{array}$$

(c). The input matrix after grouping and sorting

Figure 2. An illustration of Group Encoding Scheme, processing steps are shown in figures (a), (b), and (c).

Now, with the above definitions, we can introduce our representation scheme, the *grouping representation*. For each column, we first calculate the characteristic of it. Then, all columns are divided into groups according to their characteristics. Two columns are put in the same group if and only if they have the same characteristics. In other words, all columns in the same group have the same characteristics and hence have the same defining rows and uniqueness values. Any two columns from different groups have different characteristics. The way in which we do grouping makes it possible for us to define the characteristics for the groups. In fact, the characteristic of a group can be defined to be the common characteristic of the columns in that group. From claim 1, we can immediately get the following.

CLAIM 2: Different groups have different characteristics and hence have different defining rows. As a result, the number of groups is not more than the number of rows in the partition matrix.

Now, let us use the previous example to illustrate the grouping strategy (see *figure 2*). An instance of the SPP is gicen by the partition matrix and the associated cost of each column shown in *figure 2(a)*. We count the number of 1's in each row, which is the characteristic of the row, and rank all the rows. The row with the smallest characteristic is ranked 1, a row with the second smallest characteristic is ranked 2, and so on.

We do not store the characteristics of the columns explicitly. Instead, we use the rank. Next, we will see that the rank of each row will actually give the index of the group a column belongs to if the row is the defining row of that column.

The next step is to calculate the characteristic of each column and hence find the index of the group it belongs to. Let us begin with the first column. It has a 1 on the 1^{st}, the 2^{nd}, and the 5^{th} row. The 5^{th} row has the smallest characteristic 4, so the 5^{th} row is the defining row of the 1^{st} column. Since the rank of the 5^{th} row is 2, the 1^{st} column belongs to group 2, i.e., the group index of the first column is 2. Similarly, for each column, we can calculate the index of the group it belongs to, as shown in *Figure 2(b)*. After sorting all columns according to the group indices, we get a matrix as shown in *Figure 2 (c)* with all columns belonging to the same group being put together.

If the input is matrix $(a_{ij})_{m\times n}$, the number of groups is not necessarily *m*. In general, each row defines the uniqueness for some columns. But, the columns having a "1" in this row may also have a "1" in some other rows with smaller characteristics. If all the columns presenting a "1" in a row also present a "1" in some other rows of smaller characteristics, then no columns are defined by this row. When this happens, the number of groups is less than the number of rows (*m*) in the partitioning matrix. Because of these properties, the search space of our GA with grouping representation is substantially reduced. Once we finish grouping, we sort the groups in ascending order according to their uniqueness and breaking ties using the *D* value, the defining row(given in definition 3). Thus, the first group has the smallest uniqueness, the second group has the second smallest uniqueness, and so on. The following facts are obvious.

CLAIM 3: If S is a feasible solution of the SPP, then at most one column from each group can be selected.

CLAIM 4: With the groups in sorted order, if row *i* is the defining row of group *k*, then row *i* may be covered by some columns from group *s* with $s < k$ but can not be covered by any columns from group t with $t > k$.

Since all columns in the same group cover the same defining row, selecting two columns from the same group will make the defining row over-covered. Hence, Claim 3 is correct. The correctness of Claim 4 is also very obvious. It is essentially due to the definition of the characteristic of a column. A column may cover some rows other than the defining row. We observe that if a column from group *t* covers the defining row of group *k* then we have $t < k$, otherwise, assume $t > k$. Then the column would have been put into group *k* according to the definition, a contradiction. The correctness of the above claims is established.

Assume the number of groups is *g* and the number of columns in group *k* is G_k. The grouping representation of the problem is a integer string *X* of length *g*. For each $k < g$, $X[k]$ is a integer in $[0, G_k]$ representing the column been selected. $X[k]=0$ means no column is selected in this group while $X[k] = i$ for any $0 < i<G_k$ means that the i^{th} column in group *k* is selected. Figure 3 shows a chromosomal representation of the grouping scheme.

From Claim 4, we see that, in any feasible solution *X*, exactly one column is selected from group 0. If the selected column in group 0 does not cover the defining row of group 1, and then one column is selected from group 1, otherwise, no column is selected from group 1. For group 2, if the defining row is not covered by the columns selected from group 0 and group 1, one column is selected from group 2, otherwise, no column is selected from it. We repeat the above procedure for the remaining groups until all groups are considered. This procedure gives us a systematic way of generating near feasible solution efficiently.

Groups:	0	1	2		g-2	g-1
X[j]	X[0]	X[1]	X[2]	………	X[g-2]	X[g-1]

Figure 3: Chromosomal representation of grouping scheme of a SPP solution.

Comparing the group representation to the straightforward representation scheme used by Levine and Chu, we see that the solution string here is much shorter. Moreover, the major advantage of grouping representation can be summarized as follows.

PROPOSITION: With the grouping representation, the size of the search space is $SS' = \prod_{k=0}^{g-1} G_k$, where g is the number of groups and G_k the number of columns in group k. Moreover, compared to the search space SS under the straightforward encoding scheme, we have

$$\frac{SS}{SS'} \geq \left(\frac{2^e}{e}\right)^{n/e} \tag{3.4}$$

where n is the number of columns. The equality holds when $n/m = e$.

Proof : Assume there are n columns. With the grouping representation, n columns are grouped into g different groups. By Claim 3, at most one column can be selected from each group. In each group, there are G_k columns and hence we have G_k different ways to select a column. Hence, the total number of possible solutions is $SS' = \prod_{k=0}^{g-1} G_k$. To prove the second half of the proposition, observe that $\sum_{k=0}^{g-1} G_k = n$. It is well known that when the sum of g numbers is fixed, the product of the g numbers achieves maximum if and only if the g numbers are all the same. Hence we have $SS' \leq (n/g)^g$. Now, let $f(x) = (n/x)^x$. Then, taking the derivative of the function we get $f'(x) = (n/x)^x (\ln n - \ln x - 1)$. Clearly, $x = n/e$ is the only critical point of the function. Furthermore, $f'(x) < 0$ if $x > n/e$ and $f'(x) > 0$ if $x < n/e$. Thus, function $f(x)$ achieves the maximum when $x = n/e$. So, we get $SS' \leq e^{n/e}$. Therefore, we have $SS/SS' \geq (2^e/e)^{n/e}$.

From the above we see that the search space with the grouping representation is reduced by a factor of at least $(2^e/e)^{n/e}$. This is a very significant reduction of the search space. From the proof, we see that *SS'* achieves maximum when there are *n/e* groups, i.e., about 3 columns in each group. This is rarely the case. In most real world problems, the ratio of the number of columns to the number of rows is much bigger than 3. Thus, the factor of reduction is usually much bigger than we give in (3.4).

3.1 FITNESS FUNCTION

The fitness function should take into account the cost of the columns selected that is needed to be minimized. Assume there are g groups, $X[1..g]$ is a solution string, and $c[i,j]$ is the cost associated with the j^{th} column in the i^{th} group. Then the objective function can be defined to be as follows

$$O(X) = \sum_{i=1}^{g} c[i, X[i]] \tag{3.5}$$

Because the SPP is a highly constrained problem, many of the solution strings may be infeasible. Note that a row is *over-covered* if it is covered by more than 1 column and *under-covered* if it is not covered at all. Like most other constrained problems, the evaluation of solutions is another main concern if infeasible solutions are allowed to exist in the population. Usually, a penalty function is introduced. For the SPP, many different penalty functions and penalty terms are suggested, see Levine (1994) for details. In our implementation, we define the penalty function of a solution *X* as follows

$$P(X) = \sum_{i \in I} | w_i - 1 | \tag{3.6}$$

where w_i is the number of columns in solution *X* covering row i. The SPP is to find a solution *X* such that *O*(*X*) is minimized while *P(X) = 0*. A widely used approach is to combine the objective function with a penalty term. On the other hand, it is well known that constrained single objective problems like SPP share some common properties with *unconstrained multiobjective problems*. In such problems, there are multiple objective functions. Each of them has to be minimized (or maximized). Usually, the goal is to find *Pareto-optimal* solutions. We used a similar approach.

The particular structure of the grouping representation scheme summons a set of specific properties. To best utilize the properties that the encoding scheme provides, we need to carefully design the basic genetic operators, i.e., selection, mutation, and crossover. For this purpose, we used multi-point crossover and multi-point mutation operators, i.e., the number of points where crossover or mutation happens dynamically changes. In the following, let g be the length of the chromosome. Then, at the beginning, it is a p-point crossover (mutation). After a certain number of iterations, say, 100 iterations, if there is no feasible solutions generated, then p is increased by a certain amount such that $p < g$. The increase can be a fixed percentage. In addition, we also use a repair operator to improve the existing individuals in the population.

3.2 REPAIR OPERATOR

After performing mutation and crossover, we used a repair operator to systematically improve "impaired" individuals in the population. According to Claim 4, no columns from a group other than the first one cover the defining row of the first group. So, any solution must select one column from the first group. If no column or more than one column is selected from the first group, we must change it and select exactly one column from the first group. Note that the column selected from the first group may also cover some other rows. Any group whose defining row is covered by this column will be blocked, i.e., no column can be selected from that group. Accordingly, we mark those affected groups as blocked and no column can be selected from those groups in the future. Next, we consider the second group. If the second group is blocked, i.e., its defining row is covered by the column selected in the first group, then no column can be selected, otherwise one column can be selected from the second group. In the original solution string, if no column was selected, then randomly select a column. After we select a column, we need to check whether the selected column covers any rows covered by the column from the previous group. If this is the case, we will select another column from the group by iterating through it. If no such column exists in this group, no column will be selected. Repeating this procedure with the remaining groups, we can get a near feasible solution from a highly infeasible solution. After applying the repair operator to any solution, we may either get a feasible solution or a slightly infeasible solution with some rows uncovered. No rows will be overcovered.

Based on the above analysis, we develop a repair algorithm as shown in Figure 4. In this algorithm, the '*for*' loop iterates through all groups. For each group, it first checks if the defining row was covered by any of the previous groups or not. If the defining row was covered already, block this group and continue the loop with next group. Next, select a column if no column is selected. At last, we need to ensure that no rows are over-covered. If the column selected in the current group over-covers any row, we try to select a different column such that no rows are over-covered. To accomplish each of the above steps, we need to know which rows are covered by the previous groups.

ALGORITHM 1: Repair (x)

/* x is a solution string. $x[i]$ is the index of the selected column in group i. r is the number of rows of the partition matrix. $A[j]$ is the j^{th} column of the partition matrix. $C[r]$ is a vector recording the covered rows */

```
 1: C[r] = 0;                    //initialization
 2: FOR (k=0; k<g; k++)//g is the number of groups
 3:     IF(defining row of group k has been covered)
 4:        x[k] = 0; //no row can be selected from this group
 5:        CONTINUE;
 6:     ENDIF
 7:     IF(no column is selected from group k)
 8:        Select a column;
 9:     ENDIF
10:     IF(column A[x[k]] overlaps C[r])
11:        iterate through group k and try to find a column
12         so that it does not  overlap C[r].
13:     ENDIF
14:     /* record the rows been covered by the columns
15:     considered so far.*/
16:     C = C + A[x[k]];
17:ENDFOR
```

Figure 4. Algorithm of the Repair operator

For this purpose, we use a binary vector $C[r]$ to keep track of the rows been covered up to the current iteration. A '1' in any row indicates that the row is covered and a '0' means the row uncovered. To check if a column causes any row been over-covered, we just check if the column overlaps vector $C[r]$ or not, i.e., both have a '1' in the same row. $C[r]$ is updated at the end of each iteration to take the newly selected column into account.

The repair algorithm improves any solution string to a feasible or near feasible solution. Line 7 can slightly be modified by randomly sampling a column from the group a few times to introduce a certain degree of variability. If we can successfully find a column, then stop, otherwise, begin iterate through the group systematically. Doing so, it is possible to avoid bias towards any particular rows or columns without sacrificing the efficiency. The Repair operator plays the same roles as the ADD and DROP operators introduced by Chu(1997). However, our implementation is much more efficient because of the grouping representation scheme, which can reduce the search space.

4 AIRLINE CREW SCHEDULING EXPERIMENTS

To test the group encoding scheme in a genetic algorithm, we selected a subset of ten problems from a set of real world airline crew scheduling problems. These problems were used by Hoffman and Padberg (1988) to test their branch-and-cut algorithm and used by Levine (1994) and Chu (1997) to test their GA solutions respectively. These airline crew scheduling problems are given by the partitioning (input) matrix, which are constructed by equation (1.4). In the partitioning matrix, each row represents a flight leg that must be flown and each column represents a feasible round-trip rotation for a crew. Most of the problems we considered are taken from NorthWest Airlines (NW) database (also found in Chu (1997)).

We used the standard preprocessing procedures as given in Chu (1997). These procedures delete certain rows and columns of the partitioning matrix by applying some logical rules. Next, we reorganize the partitioning matrix and gather the grouping information, i.e., calculate the characteristics of each column and sort all columns according to their characteristics. Then, 100 individuals are randomly generated as the initial population. After generating each solution string, the repair operator is applied so that the initial population becomes more feasible. We use a steady state GA where in each iteration, a new individual is generated by selecting two parents from the population and applying the genetic operators to them. After each 100 iterations, if no feasible solution is generated then the mutation and crossover rates are increased by 10 percent.

The results of our experiments are given in *Table 1*. In the first column of the table, the first two letters show the name of the Airline Company from where the dataset is collected. The following two digits indicate the index of the problem. The second and the third columns are the number of rows and the number of columns of the partition matrix (indicating the problem size). The fourth column lists the results of the Integer Programming Optimal (IP Optimal), see Hoffman and Padberg (1988) for details. The last two columns are the results from our GA experiments. An entry of 'o' in this column indicates the optimal solution was found (same as the IP Optimal). A numerical entry is the cost of the best feasible solution found in 100,000 iterations. The last column shows the average number of iterations required for each test case.

Table 1. Report of the experimental results. (median over 10 runs). The crossover and mutation rates are varied between 50% - 100% and 1% - 10%, respectively.

Problem	Row	Col	IP Optimal Solution	Results of GA	# of iterations
NW41	17	197	11307	o	41
NW26	23	771	6796	o	5743
NW10	24	853	68271	o	10736
NW34	20	899	10488	o	508
NW43	18	1072	8904	8974	8162
NW30	26	2653	3942	o	25341
NW31	26	2662	8038	o	54365
NW19	40	2879	10898	o	8596
US02	100	13635	5965	o	45932
NW18	124	10757	340160	357646	75864

From the results listed in *Table 1* we see that, for most of the problems, our group-encoded GA could find the optimal solutions. For a few problems of relatively large in size, the GA finds a reasonable near-optimal solution. Moreover, the quality of the solutions is comparable to that of any other algorithms (such as existing GA approaches or exact solvers).

5 CONCLUDING REMARKS

The set partitioning problem (SPP) has been studied extensively in the last decades because of its many applications. One such important application is the airline crew scheduling. However, the size of the problem has grown significantly in size and complexity over the years which poses a big challenge to SPP investigators. The increase in the number of rows (in input matrix) introduces more constraints to the problem and makes it more difficult to find good feasible solutions. Two different approaches have been taken to tackle the difficulties of this problem: exact algorithms (such as CPLEX and SPP_OPT) and genetic algorithms (Levine (1994) and Chu (1997)). In these approaches, the difficulties increase exponentially when the size of the problem increases. In particular, more rows mean more constrains and more columns mean a larger search space. For example, of the six AA (American Airline) problems tested by Chu (1997) failed to find any feasible solution in three of these test cases. As Chu concluded, the difficulty of the AA problem is largely due to the relative large number of rows (constraints) compared to other problems. The instances of AA problems were also found to be more difficult using the CPLEX solver in terms of the number of nodes searched and the running time. Moreover, the size of the search space directly depends on the number of columns.

Based on the above observations, we designed a genetic encoding scheme to tackle the difficulties associated with the large problem size. In particular, we described an indirect representation scheme to reduce the search space substantially, and devised efficient genetic operators accordingly to search for near optimal solutions. We also gave an analytic proof on the factor of reduction in the search space with the group encoding technique. The GA based on the grouping representation appears to be efficient. However, results from our experiments show that large problems still remain relatively difficult to find the optimal solution. With the grouping representation, our main concerns are the genetic operators. The problem of devising a good set of genetic operators coupled with the grouping representation is task in the future.

References

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

E. Balas (1965). An additive algorithm for solving linear programs with zero-one variables. *Operational Research Quarterly*, 13:517 – 546.

E. Balas and C. Martin (1980). Pivot and complement – a heuristic for 0 – 1 programming. *Management Science*, 26(1):86 – 96.

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

R. E. Bixby (1990). Using the CPLEX Callable Library. Manual distributed by Cplex Optimization Inc., 7710-T Cherry Park, Houston, TX.

P.C. Chu (1997). A genetic algorithm approach for combinatorial optimization problems. *PhD thesis*, University of London.

P.C. Chu and J.E. Beasley (1996). Constraint Handling in Genetic Algorithms: the Set Partitioning Problem. *European Journal of Operational Research*, Vol 95 (2): 393 – 404

M. Fischer and P. Kedia (1990). Optimal Solution of Set Covering/Partitioning Problems Using Dual Heuristics. *Management Science*, 36(6):674 – 688.

Garfinkel and Nemhauser(1972) Integer Programming. John Wiley & Sons Inc., New York.

R. Gerbracht (1978). A New Algorithm for Very Large Crew Pairing Problems. *18th AGIFORS Symposium*, Vancouver, British Columbia, Canada.

I. Gershkoff (1989) Optimizing flight crew schedules. *Interfaces*, 19(4): 29 – 43.

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

K. L. Hoffman and M. Padberg (1985). LP-based Combinatorial Problem Solving. *Annals of Operations Research* 4:145 – 194.

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

D. Levine (1994). A parallel genetic algorithm for the set partitioning problem. PhD thesis, Department of Computer Science, Illinois Institute of Technology.

R. E. Marsten (1974). An Algorithm for Large Set Partitioning Problems. *Management Science*, 20:774 – 787.

Z. Michalewicz, D. Dasgupta, R. G. Riche and M. Schoenauer (1996). Evolutionary algorithms for constrained Engineering problems. *Computers ind. Engng* Vol30, N4 :851 – 870.

G. Nemhauser and L. Wolsey (1988). *Integer and Combinatorial Optimization*. John Wiley & Sons, New York.

C. R. Reeves (1993). *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific.

J. Rubin (1973). A Technique for the Solution of Massive Set Covering Problems, with Applications to Airline Crew Scheduling. *Transportation Science*. 7:34 – 48.

D.M. Ryan and J.C. Falker (1988). On the integer properties of scheduling set partitioning models. *European Journal of Operational Research*. 35:422 – 456.

P.D. Surry, N.J. Radcliffe and I.D. Boyd (1995). A multi-objective approach to constrained optimisation of gas supply networks: the COMOGA method. In T.C. Fogarty, editor, Evolutionary Computing: AISB Workshop, 166--180. Springer-Verlag, Lecture Notes in Computer Science 993.

L. Tasi (1995). The modified differencing method for the set partitioning problem with cardinality constraints. Discrete Applied Mathematics, 63:175 – 180.