An Evolutionary Algorithm for Large Scale Set Covering Problems with Application to Airline Crew Scheduling

Elena Marchiori[1] and Adri Steenbeek[2]

[1] Free University
Faculty of Sciences, Department of Mathematics and Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
elena@cs.vu.nl
[2] CWI,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
adri@cwi.nl

Abstract. The set covering problem is a paradigmatic NP-hard combinatorial optimization problem which is used as model in relevant applications, in particular crew scheduling in airline and mass-transit companies. This paper is concerned with the approximated solution of large scale set covering problems arising from crew scheduling in airline companies. We propose an adaptive heuristic-based evolutionary algorithm whose main ingredient is a mechanism for selecting a small core subproblem which is dynamically updated during the execution. This mechanism allows the algorithm to find covers of good quality in rather short time. Experiments conducted on real-world benchmark instances from crew scheduling in airline companies yield results which are competitive with those obtained by other commercial/academic systems, indicating the effectiveness of our approach for dealing with large scale set covering problems.

1 Introduction

The set covering problem (SCP) is one of the oldest and most studied NP-hard problems (cf. [14]).

Given a m-row, n-column, zero-one matrix (a_{ij}), and an n-dimensional integer vector (w_j), the problem consists of finding a subset of columns covering all the rows and having minimum total weight. A row i is covered by a column j if the entry a_{ij} is equal to 1. This problem can be formulated as a constrained optimization problem as follows:

$$\text{minimize} \quad \sum_{j=1}^{n} w_j x_j$$

$$\text{subject to the constraints} \quad \begin{cases} x_j \in \{0,1\} & j = 1, \ldots, n, \\ \sum_{j=1}^{n} a_{ij} x_j \geq 1 & i = 1, \ldots, m. \end{cases}$$

The variable x_j indicates whether column j belongs to the solution ($x_j = 1$) or not ($x_j = 0$). The m constraint inequalities are used to express the requirement that each row be covered by at least one column. The weight w_j is a positive integer that specifies the cost of column j. When all w_j's are equal to 1, then the SCP is called unicost SCP.

Relevant practical applications of the SCP include crew scheduling [1, 2, 12, 15]: find a set of pairings having minimum-cost which covers a given set of trips, where a pairing is a sequence of trips that can be performed by a single crew. A widely used approach to crew scheduling works as follows. First, a very large number of pairings is generated. Next, a SCP is solved, having as rows the trips to be covered, and as columns the pairings generated. When this approach is used in mass-transit applications, very large scale SCP instances may arise, involving thousands of rows and millions of columns.

The most successful heuristic algorithms for large scale SCP's are based on Lagrangian relaxation [13]. Lagrangian relaxation is used to compute the score of a column according to its likelihood to be selected in an optimal solution. These scores are employed in simple greedy heuristics for computing a solution. A very effective heuristic algorithm for large scale SCPs based on this approach is [7]. We refer the reader to [8] for a recent survey on exact and heuristic algorithms for SCP. All effective heuristics for large scale SCP's act on a subset of the columns, called core, which is selected before the execution of the algorithm. In the static approach the core remains the same during the execution (cf. [6, 11]), while in the dynamic approach it is updated using an adaptive mechanism (e.g. [7, 10, 9]).

In this paper we propose a novel heuristic algorithm for large scale SCPs arising from crew sheduling problems in airline companies. At each iteration a near optimal cover is constructed using the information provided by the previous iterations to guide the search. The final solution is the best cover found in all the iterations.

Given a problem instance, the algorithm extracts an initial core from the set of columns given in the input. Then the algorithm consits of the iterated application of the following three steps: 1) First, an approximated solution to the actual SCP core is constructed by means of a novel greedy heuristic. 2) Next, a local search optimization algorithm is applied to the resulting solution. 3) Finally, some columns that occur in the best solution found in all iterations up to now are selected for forming the initial partial solution for the next iteration.

The size of the core is determined by an adaptive size parameter, while the selection of a column is specified by a suitable merit criterion. During the execution, the score of the columns is modified as well as the size parameter, and the core is dynamically updated.

This algorithm can be viewed as a hybrid $(1 + 1)$ steady-state evolutionary algorithm, where at each iteration a child is generated from the parent using the above described heuristic, and the best between the parent and the child survives.

In order to assess the performance of the algorithm, we conduct extensive experiments on real-world problem instances arising from crew scheduling in airlines, as well as on other benchmark instances from the literature. The results of the experiments are rather satisfactory: our algorithm is able to find covers of very good quality in a short amount of time, yielding results which are competitive with those reported by the best industrial as well as academic methods for solving large set covering problems.

The rest of the paper is organized as follows. In the next subsections we briefly discuss some related work, and set up the notation and terminology used throughout the paper. In Section 2 we introduce the overall method and present in detail the four main modules of the algorithm. In Section 3 we report the results of extensive computational experiments. We conclude with some final remarks on the present investigation and on future work.

1.1 Related Work

An experimental comparison of the most effective exact and heuristic algorithms for the weighted SCP is given in a recent paper by Caprara et al [8].

A rather effective heuristic algorithm based on Lagrangian relaxation is the CFT algorithm [7] by Caprara et al. This algorithm has been tested also on large scale problem instances arising from crew scheduling in railway, yielding rather satisfactory results. In [15] a approximation algorithm for solving large 0-1 integer programming problems is proposed. This algorithm is used in the CARMEN system for airline crew scheduling, a industrial system used by several major airlines.

Research based on evolutionary computation includes the following two papers.

Beasley and Chu in [6] introduce a genetic algorithm for the SCP. The authors employ a representation where a chromosome is a bit string of lenght equal to the number of columns, one bit for each column, representing the set of columns whose bit in the string are equal to 1. The algorithm employs a heuristic repair mechanism for transforming infeasible chromosomes into solutions. Moreover, a core is used for constructing the chromosomes of the initial population.

A genetic algorithm based on a non-binary representation has been proposed by Eremeev in [11]. Here a chromosome is a string of lenght equal to the number of rows, where the i-th entry contains (the index of) a column covering the i-th row. As a consequence, all chromosomes are feasible solutions, thus they do not need to be repaired as in [6]. Moreover, heuristics are used for eliminating redundant columns as well as for defining the crossover operator.

In Section 3 we will compare experimentally the above mentioned algorithms with the algorithm introduced in this paper.

1.2 Notation and Terminology

In order to describe our method, we use the following terminology and notation.

In the sequel, the indexes i, j denote a generic row and column, respectively. A column will also be denoted by c, and a row by r, possibly subscripted. Moreover, S denotes a set of columns.

Let $cov(S)$ be the set of rows that are covered by the columns in S:

$$cov(S) = \{i \mid a_{ij} = 1 \text{ for some } j \in S\}.$$

For simplicity, we write $cov(j)$ instead of $cov(\{j\})$. We say that a column j is *redundant* with respect to S if $cov(S \setminus \{j\}) = cov(S)$.

A *partial cover* (also called partial solution) is a set of columns containing no redundant column.

Let $cov(j, S)$ be the set of rows which are covered by column j, but are not covered by any column in $S \setminus \{j\}$:

$$cov(j, S) = \{i \mid a_{ij} = 1 \text{ and } a_{ij'} = 0 \text{ for all } j' \in S \setminus \{j\}\}.$$

Moreover, let $min\_weight(i)$ be the minimum weight of the columns that cover i:

$$min\_weight(i) = minimum\ \{w_j \mid i \in cov(j)\}.$$

We can now define the function cov_val, called *cover value*, which is used to evaluate a column j with respect to a partial cover S in order to select a column to be added (resp. removed) to (resp. from) S:

$$cov\_val(j, S) = \sum_{i \in cov(j,S)} min\_weight(i).$$

A convenient property of cov_val is that $cov\_val(j, S) = cov\_val(j, S \setminus \{j\})$. This allows one to compute the cover value of a column without taking into account whether it belongs to the partial solution S or not. Moreover, we can characterize the redundancy of a column by means of the condition $cov\_val(j, S) = 0$.

The cover value is used to define the *selection value* $sel\_val(j, S)$ of a column j with respect to the partial cover S:

$$sel\_val(j, S) = \begin{cases} Lim & \text{if } j \text{ redundant wrt } S, \\ w_j / cov\_val(j, S) & \text{otherwise.} \end{cases}$$

The selection value of redundant columns is set to a very big constant Lim. In this way, redundant columns do not have any chance of being selected.

2 The Overall Method

The algorithm we propose consists of an iterated procedure, where each iteration generates an approximated solution using only columns from the actual core.

Roughly, at each iteration a greedy heuristic is used to construct incrementally a cover starting from a partial cover: in the first iteration the partial cover is empty, while in the following iterations the partial cover is a proper subset of the best cover found in all iterations up to now. The cover found after the

application of the greedy heuristic is given as input to an optimization procedure which tries to improve the partial solution. The core is updated from time to time during the execution. The final result is the best cover found in all iterations.

The corresponding algorithm WSCP (Weighted Set Covering Problem) is illustrated below in pseudo-code, where `Sbest` represents the best cover found so far, `S` denotes the actual partial cover, and `value(S)` is the sum of the weights of the columns of `S`, that is, $\sum_{j \in S} w_j$. Therefore the optimal cover is the cover S having minimum `value(S)`.

```
FUNCTION WSCP()
BEGIN
   RECOMPUTE_CORE();
   Sbest <- { 1..ncol };
   S <- { };
   FOR 1 .. param.number_of_iterations DO
        IF ( core_selection() ) RECOMPUTE_CORE(); ENDIF;
        S <- GREEDY(S);
        S <- OPTIMIZE(S);
        IF ( value(S) <= value(Sbest) ) THEN Sbest <- S; ENDIF;
        S <- SELECT_PARTIAL_COVER(Sbest);
   ENDFOR
   RETURN Sbest
END
```

2.1 Greedy Heuristic

Our greedy heuristic GREEDY is described in pseudo-code below. Lines starting with "//" are comments. The algorithm constructs a solution (a cover), starting from a (possibly empty) partial cover S. Columns are added (resp. removed) to (resp. from) S until S covers all the rows.

```
// extend S until it is a  cover:
FUNCTION GREEDY( var S )
BEGIN
   WHILE ( S is not a cover ) DO
      // select and add one column to S
      S <- S + select_add();
      // remove 0 or more columns from S
      WHILE ( remove_is_okay() ) DO
         S <- S - select_rmv();
      ENDWHILE
   ENDWHILE
   // S is a cover, without redundant columns
   return S;
END
```

The function *select_add* selects a column j not in S having minimum selection value $sel\_val(j, S)$.

The test *remove_is_okay* determines whether columns should be removed from S. If S is empty it returns $false$; if S contains at least one redundant column then it returns *true*; otherwise, with probability *param.p_rmv* (typical value 0.3) it returns *true*, otherwise $false$.

Finally, the function *select_rmv* selects a column in S having maximum selection value.

2.2 Local Optimization

The local optimization procedure OPTIMIZE is based on the following idea. Given a cover S, suppose there is a column $j \notin S$ such that $S \cup \{j\}$ contains at least two columns other than j, say $j_1, \ldots, j_l$, with $l \geq 2$ that are redundant, and such that the sum of their weights is greater than the weight of j, that is, $\sum_{k=1,\ldots,l} w_{j_k} > w_j$. Then $(S \setminus \{j_1, \ldots, j_l\}) \cup \{j\}$ is a better cover than S. In this case we call j a *superior column*. The gain of j is defined by

$$gain(j) = \sum_{k=1,\ldots,l} w_{j_k} - w_j.$$

So a best superior column is the one having highest gain. Note that the optimization procedure operates on a cover containing no redundant columns. The optimization algorithm OPTIMIZE in pseudo-code is given below.

```
// S is a cover, without redundant columns
FUNCTION OPTIMIZE( var S )
BEGIN
   Sup <- select_superior();
   WHILE ( Sup not empty) DO
      // select best column from Sup
      best <- select_best();
      Sup <- Sup - best;
      // add superior and remove redundant columns from S
      IF ( best superior )
         S <- S + best;
         S <- S - select_redundant();
      ENDIF
   ENDWHILE
   // S is a cover, without redundant columns
   return S;
END
```

First, the function *select_superior*() is used, which generates the list *Sup* consisting of all the superior columns ordered in decreasing order according to their gain. Next, the list *Sup* is scanned in the WHILE loop. At each iteration, the head of *Sup* is removed and memorized in the variable *best* using

the function *select_best*(). If the selected column is still superior (that is if the test `best superior` is satisfied) then it is added to S, and the set of redundant columns are removed from the resulting partial cover S using the function *select_redundant*().

2.3 Restoring Part of the Actual Best Solution

In the first iteration of WSCP the heuristic GREEDY constructs a cover starting from the empty set; in the following iterations, GREEDY builds a cover starting from a subset of the best cover found so far. For a column j, we keep track of the number $chosen(j)$ of times that j has been part of a best solution. The function SELECT_PARTIAL_COVER considers the set E of so-called *elite columns*, consisting of those columns j of the best solution `Sbest` such that $cov\_val(j, Sbest) > w_j$. Then SELECT_PARTIAL_COVER selects from E the set of columns having low $chosen(j)$ (in our implementation $chosen(j)$ has to be smaller than $\sum_j chosen(j)/(neli * 10)$, where *neli* is the number of elements of E) while the remaining columns of E are selected with a probability that is set to a random value between 0.1 and 0.9.

2.4 Selecting the SCP Core

This is a fundamental step in the design of an algorithm for dealing with large SCP instances. We introduce the following method for constructing an SCP core, which has been implemented in the function RECOMPUTE_CORE.

The SCP cover is constructed from the empty set by incrementally adding columns according to the following criterion. Columns are selected in increasing order according to their selection value. Suppose column j has been selected:

1. if j is an elite column then with probability close to 1 it is added to the actual SCP core;
2. otherwise, j is added if there exists a row i such that j covers i and $w_j < min\_weight(i) \cdot K_0$, with K_0 a given constant real value greater or equal than 1;
3. otherwise, j is added if there exists a row i such that j covers i and i is covered by less than K_1 columns of the actual SCP core, with K_1 a given constant integer value greater or equal than 1.

Note that K_0, K_1 are parameters which are chosen depending on the class of problems one considers. Condition 3 implies that the SCP core contains for each row, at least the first K_1 best columns (according to the ordering induced by the selection value function) that cover that row.

The function *core_selection*() determines when the actual SCP core has to be recomputed. In our implementation, we recompute the SCP core every 100 iterations of WSCP.

During the execution of GREEDY, when ninety per cent of a cover has been constructed, the *min_weight* of those rows that are not yet covered is increased

by a small quantity (in our implementation $min\_weight(i)$ is multiplied by 1.1). This affects the selection value of the columns, hence their order of selection in the construction of the SCP core changes during the execution of the overall algorithm WSCP.

3 Experimental Evaluation

The algorithm WSCP has been tested on large set covering problems arising from crew scheduling applications in various airline companies. Moreover, we have considered the weighted SCP instances from the OR library maintained by J.E. Beasley [1]. These instances are considered standard benchmarks for testing the effectiveness of exact and heuristic algorithms for the SCP. In particular, they have been used in [8] for comparing experimentally various exact and heuristic algorithms for SCP.

WSCP has been implemented in C++. The algorithm was run on a Sun Ultra 10 (UltraSPARC-IIi 300MHz).

The results of the experiments are based on 10 runs on each problem instance of the OR Library, and on 5 runs on the other instances. In each table, the entry labeled **Id** contains the name of the problem instance. The label **BK** denotes the best known solution for that instance; **Bst** denotes the best result found by the algorithm; **Fbst** indicates the frequency of obtaining the best solution in the performed runs; **Apd** denotes the average percentage deviation $\sum_{k=1^{10}}(z_k - z^*)/(10 \cdot z^*) \cdot 100\%$, where z_k is the solution found in the k-th run, and z^* is the optimal or best known solution. **Tbst** denotes the average cpu time for obtaining the best solution Bst, while **Tsol** denotes the average cpu time for finding a solution. Finally, **Ibst** and **Isol** denote the average number of iterations of obtaining the best solution Bst, and a solution, respectively.

3.1 Experiments on Airline Crew Scheduling Problem Instances

We consider three sets of benchmark instances from real-world airline crew scheduling problems. A set of instances from a major airline company, here called AIR instances, the airline scheduling instances from Wedelin [15], and the instances from Balas and Carrera [3]. The characteristics of these problems are reported in Tables 2, 3, and 7, respectively. Observe that in many instances, like, e.g., the Wedelin instances, the weights of the columns are very large numbers, because the weight represents the cost of a pairing and takes into account several factors.

We compare experimentally WSCP with the industrial system used by an airline company on the AIR instances, with the CFT algorithm by Caprara et al [7], and with the Wedelin algorithm [15]. The results of the experiments are given in Tables 2, 4, 5, and 7. Note that the results for the Wedelin and CFT algorithms are taken from the paper [7], where the cpu time is estimated in DECstation

[1] see http://mscmga.ms.ic.ac.uk/jeb/orlib/scpinfo.html

5000/240 CPU seconds. Only the value of the best solution is reported. For the CFT algorithm, the time for finding the best solution is given, while for the Wedelin algorithm, only the overall execution time **Texe** of the algorithm is reported. The authors do not specify the setting of the various parameters in their algorithms, and the total number of trials performed.

Id	Rows	Columns	Density (%)	Weight Range
A01	5265	258303	0.167	1319-35302
A02	3878	19441	0.135	1437-37206
A03	4965	40580	0.092	1337-37148
A04	4916	79481	0.123	1460-37142
A05	4656	72377	0.126	1411-37251
A06	1971	23741	0.135	1437-37037
A07	4203	32363	0.15	1319-36370
A08	4320	45286	0.18	1345-36370
A09	4287	50047	0.19	1361-36370
A10	4369	49525	0.18	1344-36370
A11	150	389388	5.55	1800-18768
A12	682	642613	1.45	1630-19000

Table 1. Characteristics of AIR instances

Id	Industry	WSCP						
	Bst	Bst	Fbst	Apd	Tbst	Tsol	Ibst	Isol
A01	16351667	16351667	1.0	0.0	550.9	550.9	919.8	919.8
A02	12879297	12879297	1.0	0.0	131.0	131.0	822.4	822.4
A03	15663720	**15663688**	1.0	0.0	254.2	254.2	1004.4	1004.4
A04	16110608	16110608	1.0	0.0	363.1	363.1	1085.0	1085.0
A05	16315241	**16315070**	0.3	0.0001	923.5	848.1	3501.6	3203.2
A06	13162511	13162511	1.0	0.0	156.9	156.9	907.6	907.6
A07	13301520	13301520	1.0	0.0	200.9	200.9	945.6	945.6
A08	13510606	**13510584**	1.0	0.0	254.4	254.4	946.2	946.2
A09	13489489	13489489	1.0	0.0	235.4	235.4	944.2	944.2
A10	13571530	13571530	1.0	0.0	237.3	237.3	933.8	933.8
A11	247775	247775	1.0	0.0	224.2	224.2	1087.8	1087.8
A12	732587	732587	0.3	0.11	1064.9	896.8	1460.6	1167.4

Table 2. Results for AIR instances

On three AIR instances WSCP found a solution which is better than the best solution found by the industrial system, while on the other instances WSCP found solutions of equal value as those found by the industrial system.

Id	Rows	Columns	Density (%)	Weight Range
B727scratch	29	157	8.2	1600-11850
ALITALIA	118	1165	3.1	2200-2110900
A320	199	6931	2.3	1600-2111450
A320coc	235	18753	1.9	1900-1812000
SASjump	742	10.370	0.6	4720-55849
SASD9imp2	1366	25032	0.3	3860-35200

Table 3. Characteristics of Wedelin instances

Id	BK	CFT		Wedelin	
		Bst	Tbst	Bst	Texe
B727scratch	94400	94.400	0.3	94400	4.7
ALITALIA	27258300	27258300	6.2	27258300	37.2
A320	1262100	1262100	79.5	1262100	216.9
A320coc	14495500	14495600	577.8	14495500	1023.7
SASjump	7338844	7339537	396.3	7340777	806.8
SASD9imp2	5262190	5263640	2082.1	5262190	1579.7

Table 4. Results of CFT and Wedelin on Wedelin instances

Id	BK	WSCP						
		Bst	Fbst	Apd	Tbst	Tsol	Ibst	Isol
B727scratch	94400	94400	1.0	0.0	0.018	0.018	38.4	38.4
ALITALIA	27258300	27258300	1.0	0.0	0.63	0.63	106.8	106.8
A320	1262100	1262100	1.0	0.0	17.34	17.34	326.2	326.2
A320coc	14495500	14495500	0.2	0.0006	651.08	446.20	3494.5	2402.0
SASjump	7338844	7339541	0.1	0.02	269.3	200.98	4635.0	3454.6
SASD9imp2	5262190	5263590	0.1	0.04	741.9	608.452	4603.0	3671.4

Table 5. Results of WSCP on Wedelin instances

Id	Rows	Columns	Density (%)	Weight Range
AA03	106	8661	4.05	91-3619
AA04	106	8002	4.05	91-3619
AA05	105	7435	4.05	91-3619
AA06	105	6951	4.11	91-3619
AA11	271	4413	2.53	35-2966
AA12	272	4208	2.52	35-2966
AA13	265	4025	2.60	35-2966
AA14	266	3868	2.50	35-2966
AA15	267	3701	2.58	35-2966
AA16	265	3558	2.63	35-2966
AA17	264	3425	2.61	35-2966
AA18	271	3314	2.55	35-2966
AA19	263	3202	2.63	35-2966
AA20	269	3095	2.58	35-2966
BUS1	454	2241	1.89	120-877
BUS2	681	9524	0.51	120-576

Table 6. Characteristics of Balas and Carrera instances

	CFT		WSCP						
Id	Bst	Tbst	Bst	Fbst	Apd	Tbst	Tsol	Ibst	Isol
AA03	33155	61.0	33155	1.0	0.0	1.26	1.266	40.0	40.0
AA04	34573	3.6	34573	1.0	0.0	1.73	1.73	74.6	74.6
AA05	31623	3.1	31623	1.0	0.0	0.48	0.48	9.6	9.6
AA06	37464	5.2	37464	1.0	0.0	2.67	2.67	128.2	128.2
AA11	35384	193.7	35384	1.0	0.0	19.11	19.11	755.4	755.4
AA12	30809	53.8	30809	1.0	0.0	7.88	7.88	350.8	350.8
AA13	33211	8.3	33211	1.0	0.0	2.32	2.32	103.8	103.8
AA14	33219	30.3	33219	1.0	0.0	11.74	11.74	557.8	557.8
AA15	34409	18.8	34409	1.0	0.0	8.92	8.92	485.6	485.6
AA16	32752	33.6	32752	1.0	0.0	4.63	4.63	257.4	257.4
AA17	31612	10.9	31612	1.0	0.0	4.69	4.69	262.2	262.2
AA18	36782	13.5	36782	0.1	0.01	17.1	6.94	1108.0	433.0
AA19	32317	5.9	32317	1.0	0.0	2.73	2.73	175.4	175.4
AA20	34912	13.6	34912	1.0	0.0	4.76	4.76	318.4	318.4
BUS1	27947	5.0	27947	1.0	0.0	8.19	8.19	382.6	382.6
BUS2	67760	19.2	67760	1.0	0.0	37.24	37.24	616.2	616.2

Table 7. Results of CFT and WSCP on Balas and Carrera instances

On the instances from Wedelin the performance of WSCP is comparable to the one of the CFT and Wedelin algorithms.

Finally, on the instances from Balas and Carrera, both WSCP and CFT are always able to find the optimal solution. In the AA instances WSCP is faster that CFT, while in the BUS instances CFT finds the optimum in a shorter time.

The results of the experiments indicate that WSCP is a rather powerful tool for solving large real-life airline crew scheduling problems.

3.2 Experiments on the OR Library SCP Instances

We consider the families A-D from [4], and the NRE-NRH from [5], consisting of randomly generated SCP instances. Each class contains 5 instances. The values of the characteristic parameters of these problem classes, like number of rows and columns, are given in Table 8.

We compare experimentally WSCP with the genetic algorithms by Beasley and Chu [6], and by Eremeev [11], and with the CFT algorithm by Caprara et al [7]. The results of the experiments are summarized in Tables 9, 10, and 11.

The results for the CFT, Beasley Chu, and Eremeev algorithms are from [11]. In particular, the cpu time is estimated in 100MHz Pentium CPU seconds.

All the algorithms are able to solve the instances of the classes A-D. On these instances, WSCP seems to have a more robust behaviour that the two genetic algorithms, finding the optimum in each of the 10 trials. The performance of WSCP on the other problem instances of classes E-H is rather satisfactory, both in terms of quality of the solutions as well as running time. On each instance, WSCP is able to find the optimum or best known solution, while the two genetic algorithms BC and Er do not find the optimum value on instances H1 and H2. Moreover, WSCP finds the solutions for instances in the harder classes G and H in a much shorter time than all the other algorithms.

Id	A	B	C	D	E	F	G	H
Rows	300	300	400	400	500	500	1000	1000
Columns	3000	3000	4000	4000	5000	5000	10000	10000
Density (%)	2	5	2	5	10	20	2	5
Weight Range	1-100	1-100	1-100	1-100	1-100	1-100	1-100	1-100

Table 8. Characteristics of Classes A, B, C, D

4 Conclusion

In this paper we have introduced a novel heuristic method for solving large weighted set covering problems. The results of the experiments indicate that WSCP is able to find covers of satisfactory quality in short running time.

Id	CFT	Beasley Chu			Eremeev				WSCP					
	Tbst	Fbst	Apd	Tsol	Fbst	Apd	Tbst	Tsol	Fbst	Apd	Tbst	Tsol	Ibst	Isol
A	47.15	0.86	0.20	65.98	0.44	0.35	82.00	71.8	1.00	0.00	0.98	0.98	108.0	108.0
B	3.34	1.00	0.00	68.63	1.00	0.00	20.80	20.80	1.00	0.00	0.30	0.30	7.8	7.8
C	29.23	0.68	0.41	87.93	0.74	0.26	53.50	52.40	1.00	0.00	0.76	0.76	72.6	72.6
D	7.64	0.96	0.06	101.70	0.94	0.08	26.62	23.33	1.00	0.00	0.40	0.40	26.0	26.0

Table 9. Results for Classes A, B, C, D

Id	BK	CFT		Beasley Chu				Eremeev				
		Bst	Tbst	Bst	Fbst	Apd	Tsol	Bst	Fbst	Apd	Tbst	Tsol
E1	29	29	11.5	29	1.0	0.0	16.9	29	1.0	0.0	1.0	1.0
E2	30	30	180.5	30	0.4	2.0	266.9	30	1.0	0.0	94.8	94.8
E3	27	27	41.7	27	0.3	2.6	85.1	27	1.0	0.0	23.1	23.1
E4	28	28	11.6	28	1.0	0.0	238.5	28	1.0	0.0	11.0	11.0
E5	28	28	16.2	28	1.0	0.0	15.5	28	1.0	0.0	2.1	2.1
F1	14	14	14.7	14	1.0	0.0	33.8	14	1.0	0.0	7.9	7.9
F2	15	15	13.8	15	1.0	0.0	34.5	15	1.0	0.0	1.3	1.3
F3	14	14	110.0	14	1.0	0.0	117.9	14	1.0	0.0	55.4	55.4
F4	14	14	13.7	14	1.0	0.0	92.6	14	1.0	0.0	20.4	20.4
F5	13	13	89.0	13	0.3	5.4	67.1	13	0.3	5.4	497.4	151.2
G1	176	176	65.0	176	0.2	1.0	451.3	176	0.7	0.3	115.0	96.0
G2	154	154	346.6	155	0.5	1.5	159.3	154	0.5	0.65	318.3	226.6
G3	166	166	432.7	166	0.1	1.1	312.1	166	0.1	0.8	627.6	319.1
G4	168	168	105.0	168	0.2	1.4	665.4	168	0.4	0.7	160.0	172.5
G5	168	168	105.0	168	0.2	0.8	242.6	168	0.7	0.05	161.2	170.4
H1	63	63	642.1	64	1.0	1.6	743.0	64	1.0	1.6	90.5	90.5
H2	63	63	392.5	64	1.0	1.6	234.3	64	1.0	1.6	34.7	34.7
H3	59	59	690.4	59	0.9	0.2	796.6	59	1.0	0.0	493.2	493.2
H4	58	58	105.1	58	0.4	91.6	62.9	58	1.0	0.0	218.2	218.2
H5	55	55	68.8	55	0.9	0.2	198.6	55	1.0	0.0	25.2	25.2

Table 10. Results of CFT, Beasley and Chu, and Eremeev on Classes E, F, G, H

Id	BK	WSCP						
		Bst	Fbst	Apd	Tbst	Tsol	Ibst	Isol
E1	29	29	1.0	0.0	1.8	1.8	2	2
E2	30	30	1.0	0.0	2.7	2.7	62.9	62.9
E3	27	27	1.0	0.0	2.3	2.3	48.3	48.3
E4	28	28	1.0	0.0	2.1	2.1	31.1	31.1
E5	28	28	1.0	0.0	1.8	1.8	5.0	5.0
F1	14	14	1.0	0.0	3.6	3.6	19.6	19.6
F2	15	15	1.0	0.0	3.6	3.6	9.0	9.0
F3	14	14	1.0	0.0	6.2	6.2	153.2	153.2
F4	14	14	1.0	0.0	3.6	3.6	13.1	13.1
F5	13	13	1.0	0.0	34.1	34.1	2061.1	2061.5
G1	176	176	1.0	0.0	2.2	2.2	29.7	29.7
G2	154	154	0.5	1.1	8.9	4.0	315	107.9
G3	166	166	0.2	0.6	30.2	14.9	1433.5	640.3
G4	168	168	0.4	0.8	18.4	25.0	812.3	1114.7
G5	168	168	0.9	0.6	5.9	5.7	207.7	197.4
H1	63	63	0.2	1.1	9.5	11.7	161.5	269.2
H2	63	63	1.0	0.0	50.4	50.4	1872	1872
H3	59	59	0.3	1.1	25.2	21.6	778.3	678.9
H4	58	58	0.8	0.3	28.1	23.9	1016.3	834.1
H5	55	55	1.0	0.0	5.5	5.5	64.1	64.1

Table 11. Results of WSCP on Classes E, F, G, H

In all the experiments we have worked with a core which is a proper subset of the set of all columns. The size of the core depends on the problem instance. However, in general a small fraction (which varies from 10 per cent to 50 per cent) of the set of columns is used as core. Using small cores helps the efficiency of the algorithm. Moreover, extensive experiments with different core sizes have revealed a somehow counter intuitive phenomenon: in many instances, the quality of the results become worse by using a larger core, even if the same number of iterations is used. This seems to indicate that the merit criterion used in WSCP is not the best possible, because it can make the wrong decision when all the columns are present in the core. We are actually investigating the use of alternative merit criteria and their relationship with the selection of the core.

Future work concerns the investigation of how to tune automatically the parameters K_0, K_1 for determining the core problem, and how the value of the probability of removing a column can be adaptively changed during the execution.

Acknowledgements

We would like to thank Thomas Baeck and Martin Schuetz for interesting discussions on the subject of this paper.

References

1. E. Andersson, E. Housos, Kohl, and D. Wedelin. Crew pairing optimization. In *Operation Research in the Airline Industry*. Kluwer Scientific Publishers, 1997.
2. J.P. Arabeyre, J. Fearnley, F.C. Steiger, and W. Teather. The airline crew scheduling problem: A survey. *Transportation Science*, (3):140–163, 1969.
3. E. Balas and M.C. Carrera. A dynamic subgradient-based branch-and-bound procedure for set covering problem. *Operations Research*, 44:875–890, 1996.
4. J.E. Beasley. An algorithm for set covering problem. *European Journal of Operational Research*, 31:85–93, 1987.
5. J.E. Beasley. A lagrangian heuristic for set covering problems. *Naval Research Logistics*, 37:151–164, 1990.
6. J.E. Beasley and P.C. Chu. A genetic algorithm for the set covering problem. *European Journal of Operational Research*, 94:392–404, 1996.
7. A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. In W.H. Cunningham, T.S. McCormick, and M. Queyranne, editors, *Proc. of the Fifth IPCO Integer Programming and Combinatorial Optimization Conference*. Springer-Verlag, 1996.
8. A. Caprara, M. Fischetti, and P. Toth. Algorithms for the set covering problem. Technical report, DEIS Operation Research Technical Report, Italy, 03 1998.
9. S. Ceria, P. Nobili, and A. Sassano. A Lagrangian-based heuristic for large-scale set covering problems. *Mathematical Programming*, 1995. to appear.
10. H.D. Chu, E. Gelman, and E.L. Johson. Solving large scale crew scheduling problems. *European Journal of Operational Research*, 97:260–268, 1997.
11. A.V. Eremeev. A genetic algorithm with a non-binary represenation for the set covering problem. In *Proc. of OR'98*, pages 175–181. Springer-Verlag, 1998.
12. M.M. Etschmaier and D.F. Mathaisel. Airline scheduling: An overview. *Transportation Science*, (19):127–138, 1985.
13. M.L. Fisher. An application oriented guide to Lagrangian relaxation. *Interfaces*, 15(2):10–21, 1985.
14. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1979.
15. D. Wedelin. An algorithm for large scale 0-1 integer programming with application to airline crew scheduling. *Annals of Operational Research*, 57:283–301, 1995.