Operational Research (2021) 21:2391–2408
https://doi.org/10.1007/s12351-019-00514-z

ORIGINAL PAPER

A GRASP-based scheme for the set covering problem

Victor Reyes[1] · **Ignacio Araya**[1]

Received: 15 March 2018 / Revised: 2 April 2019 / Accepted: 11 August 2019 /
Published online: 21 August 2019
© Springer-Verlag GmbH Germany, part of Springer Nature 2019

Abstract
In this work we present a greedy randomized adaptive search procedure (GRASP)-based strategy for the set covering problem. The goal of this problem is to find a subset of columns from a zero-one matrix in order to *cover* all the rows with the minimal possible cost. The GRASP is a technique that through a sequential and finite number of steps constructs a solution using a set of simple randomized rules. Additionally, we also propose an iterated local search and reward/penalty procedures in order to improve the solutions found by the GRASP. Our approach has been tested using the well-known 65 non-unicost SCP benchmark instances from OR-library showing promising results.

Keywords GRASP · Set covering problem · Local search · Metaheuristics

1 Introduction

The non-unicost set covering problem (SCP) is a combinatorial problem that can be described as the problem of finding a subset of columns x from a *m-row*, *n-column* zero-one matrix A such that they cover all the rows of A at minimal cost. The SCP can be formulated as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq 1, \ i \in \{1, 2, \dots, m\}, \\ & x_j \in \{0, 1\}. \end{aligned}$$

✉ Victor Reyes
victor.reyes.r@mail.pucv.cl

Ignacio Araya
ignacio.araya@pucv.cl

[1] Pontificia Universidad Católica de Valparaíso, Valparaiso, Chile

where c_j is a n-size vector that represents the cost of each column in A. The SCP is an NP-hard problem (Garey and Johnson 1979) that can be used to model many different problems such as crew scheduling (Bartholdi 1981), data bases (Munagala et al. 2005), vehicle routing (Bramel and Simchi-Levi 1997), among others. Several kind of algorithms have been developed for solving SCP instances. Preliminary works used exact algorithms, such as branch and bound and branch and cut strategies (Balas and Carrera 1996; Beasley 1987), however large SCP instances were intractable for them. Greedy algorithms (Chvatal 1979) are a good approach for large SCP instances, but they rarely generate good solutions because of its myopic and deterministic nature. Nevertheless, a greedy algorithm proposed in Lan and DePuy (2006) addresses these problems by incorporating randomization and memory to the process. Metaheuristics, such as genetic algorithms (Beasley and Chu 1996; Solar et al. 2002), simulated annealing (Brusco et al. 1999), ant colonies (Crawford et al. 2014), have been the most common approach to solve SCP instances. Also, modern metaheuristics (García et al. 2017, 2019; Crawford et al. 2016, 2015; Lu and Vasko 2015) have been used in the recent years.

In this work we present a greedy randomized adaptive search procedure (GRASP) (Resende and Ribeiro 2010) strategy for solving the SCP. GRASP is a random iterative optimization procedure that works in two phases: a constructive and a local search phase. At the constructive phase, the procedure uses a randomized greedy heuristic in order to obtain an initial solution. For instance, unlike deterministic greedy heuristics, at each step the procedure selects one of the most promising columns from a set with size defined by the user. This can be seen as an exploration phase. On the other hand the local search phase allows the algorithm to exploit the neighborhood of the initial solution hoping to find much higher quality solutions. For instance, a swap method can be performed between an instantiated and an uninstantiated column.

Besides the SCP, GRASP-based techniques have been used extensively to solve similar operational research problems, such as the unicost SCP (Bautista and Pereira 2007), the maximum covering problem (Resende 1998), the max−min diversity problem (Resende et al. 2010), the set-k covering (Pessoa et al. 2013), among others. Despite the difference between these problems, the GRASP phases are quite similar. At the constructive phase most of the approaches construct a limited list of promising columns, assigning a probability to each of them according to a certain rule, varying from the number of uncovered rows to more sophisticated functions. On the other hand, at the local search phase, most techniques are based on 0–1 swap movements between columns. Additionally, penalizing procedures for the columns are very common in order to retrieve information at the local search phase.

Meta-RaPS is one of the most well-known GRASP-based solvers used to solve SCP instances (Lan et al. 2007). At the beginning, Meta-RaPS constructs a solution by introducing randomness using two rule parameters: %priority and %restriction. The first one determines the percentage of time that the best feasible element will be chosen. The remaining time, the element added to the solution will be randomly chosen from a candidate list. The second rule is used to determine the level of acceptance and thus the size of the candidate list. A set of rules are used to *evaluate* each column. After a feasible solution is constructed, a local search method is a applied

in order to improve the current solution. A number of columns are removed from the solution while the other are fixed by using a user-defined parameter. Finally, it uses adaptive memory mechanisms, that is, save information from the construction phase in order to obtain better solutions. This last procedure can be seen as elite methods used in genetic algorithms for the columns.

Unlike other techniques, our approach first selects the uncovered row with the smallest number of columns that can cover this row. Then we use the same set of rules as Meta-RaPS for evaluating these columns, instantiating one of them according to a probability. The process is repeated until a solution is fully generated. In the local search phase we perform an iterated local search (ILS) to the found solution. The process is based in a *divide and conquer* scheme, i.e, we divide the solution in two random halves, we reset one of them (perturbation) and restart the construction from this partial solution. Additionally, we use a reward/penalty procedure to each column after the second phase is applied. This last procedure is used to improve the rate of convergence of our approach. We think that the main differences of our approach, related to other GRASP-based approaches, are:

1. It does not require an additional user-defined parameter for restricting the size of the list of promising columns.
2. It does not require any parameter besides the reward/penalty procedure.
3. Compared to modern metaheuristics our approach does not demands high tuning effort of its parameters for an optimal performance.

Although our approach is simpler and more general than other state-of-the-art algorithms, it still offers competitive and promising results in well-known set of SCP instances from OR-library and in a new set of large instance SCPs proposed in Umetani (2017).

The rest of the paper is organized as follows. In Sect. 2 we describe our approach in detail. In Sect. 3 we show the experimental results using our approach and in Sect. 4 we present the conclusions and our future work.

2 A GRASP-based technique for the SCP

In this section we describe in detail our approach for solving and finding quality solutions for the SCP. The algorithm works in two phases. The first phase consists in constructing iterative a solution using a set of simple rules for evaluating and instantiating the columns of the input matrix A. After the solution is constructed we proceed to repair it by using an iterated local search procedure. In this phase we use a divide and conquer scheme for dividing the solution, resetting a part of it and starting the search again from this point. The whole procedure is repeated until reaching a time limit.

Additionally, we use a reward/penalty mechanism for penalizing the columns of the matrix in order to accelerate the convergence of the whole strategy.

2.1 Constructing a random solution

In this section we describe how our algorithm constructs a solution. First, we construct a map structure that maps each row to the set of columns that cover it. We sort this mapping from the lowest to the highest number of columns. For example, if we use the following A input matrix and the vector cost c:

$$A = \begin{bmatrix} 1&0&0&1&0&0&1 \\ 0&1&1&0&0&0&0 \\ 0&1&0&1&1&0&0 \\ 1&0&1&0&1&0&1 \\ 0&0&0&1&1&0&1 \end{bmatrix}; \quad c = \begin{bmatrix} 1&2&3&4&5&6&2 \end{bmatrix} \tag{1}$$

We obtain the following map:

$$\begin{array}{rl} r_2 \rightarrow & 2,3 \\ r_1 \rightarrow & 1,4,7 \\ r_3 \rightarrow & 2,4,5 \\ r_5 \rightarrow & 4,5,7 \\ r_4 \rightarrow & 1,3,5,7 \end{array} \tag{2}$$

The rows are then covered in the order disposed by the structure. For each uncovered row, we instantiate one column to cover it. For rows that may be covered by only one column, we simply instantiate the corresponding column (see Sect. 2.4). For each row with two or more columns, we evaluate its columns j by using a function $f(j)$ selected randomly from a set of six evaluation functions proposed in Lan and DePuy (2006): c_j/p_j, $c_j/\log(1+p_j)$, $c_j/\sqrt{p_j}$, c_j/p_j^2, $\sqrt{c_j}/p_j$, $c_j/p_j \log p_j$; where c_j is the cost of the column j and p_j represents the number of rows that may be covered by the column j (not counting the already covered rows). The idea of such amount of functions is to maintain diversity among the solutions constructed in this phase. Once we evaluate the columns, we propose two criteria to select the column to instantiate:

- Instantiate the column j with probability proportional to $-f(j)$
- Instantiate the column j minimizing $f(j)$

Note that, according to the evaluation functions and the selection criteria, columns that covers more rows with a low cost are more likely to be instantiated.

The first criterion is applied with a probability of $\frac{1}{\#cols}$ and the second one with a probability of $1 - \frac{1}{\#cols}$, where $\#cols$ is the number of columns that can cover the current row. After instantiating a column, all the rows covered by this column are also removed from the map (2).

Going back to our example, we have to select one of the columns (2 or 3) in order to cover the row r_2 [see (2)]. Suppose that the function $f(j) = c_j/p_j$ is selected, thus evaluating the columns we obtain $f(2) = 1$ and $f(3) = 1.5$. The first

selection criterion would select the column 2 with a probability of 0.6 and the column 3 with a probability 0.4. On the other hand, the second criterion would select the second column.

Algorithm 1 shows the procedure related to this phase, where:

- The method create-pairs generates the sorted map (2) by using the input matrix A.
- The while loop is performed until the map is empty, i.e., all the rows are covered.
- The procedure random-function selects one of the six random evaluation functions.

```
1  procedure random-greedy (A,c) out: x
2      x ← (0, 0, ...);
3      map ← create-map(A);
4      while map is not empty do
5          cols ← first(map);
6          f ←random-function();
7          if rand(0, 1) ≤ 1/size(cols) then
8              j ← select j ∈ cols with probability
                 proportional to −f(j);
9          else
10             j ← select j ∈ cols minimizing f(j);
11         end
12         remove each element in map containing j;
13         x_j ← 1;
14     end
15     return x;
16 end.
```

Algorithm 1: The construction process

2.2 Iterated local search (ILS)

Although using the procedure random-greedy leads to quality solutions, there is still a considerable gap between the solution fitness and optimum (see Fig. 1). Thus, we propose a method for improving the generated solutions.

Algorithm 2 shows the procedure. The idea is to take the solution and *reset* a random part of it, i.e. setting some variables x_j to 0, and solving the problem again from this state. To do that, first we group randomly the variables into two lists: L_1 and L_2 (line 6). These lists are also put into a queue of lists Q. We pop a list L from Q and each x_j in this list is set to 0. Then, we reuse the procedure random-greedy to complete the solution. We apply the procedure with some minor modifications:

- The columns in the map are restricted to the corresponding variables in L.
- In each iteration we simply select the column minimizing the value c_j/p_j, i.e., lines 6–11 are replaced by line 10 with $f(j) = c_j/p_j$.

If the new generated solution x' is better than the previous one, then we replace the previous solution and the variables in L are grouped into two new lists. These new lists are pushed into Q. If the new solution x' does not show any improvements to the solution, it is simply discarded. Then the process is repeated until Q is empty. If any of the lists of the current queue Q produces an improvement of the solution, then the whole process is repeated.

```
1  procedure ILS (A,x); out: x
2      Q ← {}; improvement← true;
3      while improvement do
4          improvement← false;
5          L ← {1, 2, ..., #cols(A)};
6          {L1, L2} ← grouping(L);
7          push(Q, L1);
8          push(Q, L2);
9          while Q is not empty do
10             L ←pop(Q);
11             for each i ∈ L, x_j = 0;
12             x' ← random-greedy*(A, c);
13             if cost(x') < cost(x) then
14                 x ← x';
15                 {L1, L2} ← grouping(L);
16                 push(Q, L1);
17                 push(Q, L2);
18                 improvement← true;
19             end
20         end
21     end
22 end.
```

Algorithm 2: The iterative local search process

2.3 The reward/penalty procedure

As is explained above, the construction and ILS procedures are repeated until a time limit is reached. Then, the best solution found so far is returned. One issue related to this strategy is that no information about the good or bad decisions is extracted or used during the search.

Thus, in this section we propose a simple reward—penalty procedure for extracting and using information during the search. This mechanism hopefully will allow us to improve the process for generating quality solutions.

Our reward/penalty procedure simply assigns a *penalty* to each column of the matrix. At the beginning of the search, the penalty of each column is initialized to 1. If the method ILS improves the solution quality, then the penalties are updated in the following way:

- If the column j is instantiated in the last reported solution, then its penalty p_j decreases in $\alpha > 0$.
- If the column j is not instantiated in the last reported solution, then its penalty p_j increases in $\beta > 0$

Finally, instead of using the value of $f(j)$, we use $p_j * f(j)$ for selecting the next column in the methods *random-greedy* and *random-greedy**. Note that high values of p_j decreases the likelihood of selecting the column j and vice versa. In order to avoid local optima we have limited the value of each penalty to the interval $[min, max]$.

2.4 Pre-processing

In order to reduce the size of the different SCP instances, we used two well known pre-processing procedures before the actual search: column domination and column inclusion (Fisher and Kedia 1990).

2.4.1 Column domination

If the set of rows covered by a column j is also covered by another column j' with a lower cost, then we said that j is dominated by j'. Dominated columns are removed from A.

2.4.2 Column inclusion

If a row is only covered by one column, then this column is included in every solution.

3 Experiments

Our proposal has been implemented in C++,[1] on an 2.4GHz CPU Intel Core i7-4700MQ with 8GB RAM computer using Ubuntu 16.04 LTS x86_64. In order to test the proposal, we used the 65 non-unicost SCP instances from OR-library[2] which are described in Table 1. Optimal solutions are known for all of these instances. The results of the experiments are evaluated using: the mean value, the standard deviation and the relative percentage deviation (RPD). The RPD quantifies the deviation of the objective value x from the best known value x^*. This value is computed as follows:

$$RPD = \left(\frac{x - x^*}{x^*}\right) \times 100$$

Strategies were run 30 times on each instance. On each table we show the best known solution (column Best-sol), the mean cost of the solutions found by our approach (column mean), the standard deviation of the costs (column σ), the minimum cost found considering all the runs (column min) and the RPD related to the

[1] https://github.com/vareyesr/GRASP-SCP.

[2] http://people.brunel.ac.uk/~mastjjb/jeb/orlib/scpinfo.html.

Table 1 Detail of the test instances

Instance set	No. of instances	# Rows	# Columns	Cost range
4	10	200	1000	[1, 100]
5	10	200	2000	[1, 100]
6	5	200	1000	[1, 100]
A	5	300	3000	[1, 100]
B	5	300	3000	[1, 100]
C	5	400	4000	[1, 100]
D	5	400	4000	[1, 100]
E	5	500	5000	[1, 100]
F	5	500	5000	[1, 100]
G	5	1000	10,000	[1, 100]
H	5	1000	10,000	[1, 100]

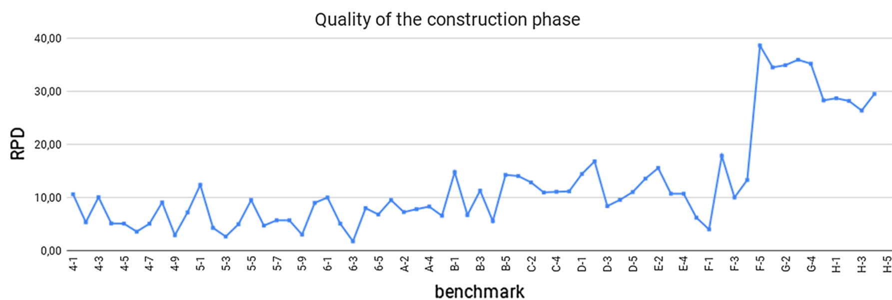

Fig. 1 Quality of solutions generated by the random greedy

minimum cost. The algorithm stops when 20 seconds pass since the last best solution was found.

3.1 Measuring the solution quality from the construction phase

As a first experiment, we would like to measure the solution quality (cost) generated by our random greedy approach.

In Fig. 1 we show the RPD corresponding to the random greedy for each of the 65 instances. As it can be seen the greedy by itself can not reach to any optimum solution in these instances. However, we expect that the ILS algorithm can improve the generated solutions, converging to the optimum in a reasonable time.

3.2 Measuring the impact of the ILS algorithm

In this section we show the improving provided by the ILS process.

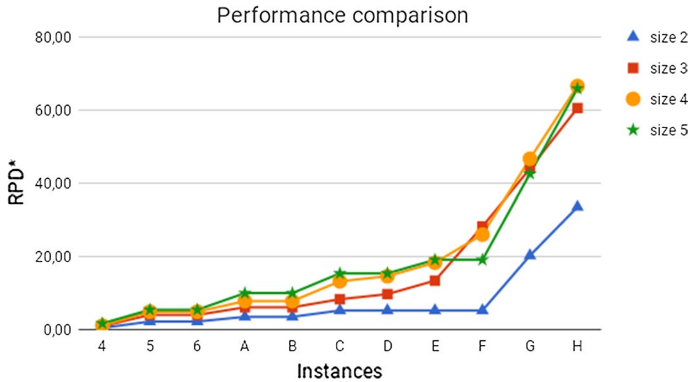

Fig. 2 Comparison of different sizes of repairing lists. The y-axis shows the *accumulative* RPD value while the x-axis indicates to the instance set

As a first experiment we show the importance of using a grouping list of size 2, as is shown in the algorithm 2. In Fig. 2 we show the comparison between using 2,3,4 and 5 grouping lists.

From this result, we can see that increasing the number of initial partitions has a negative outcome on the quality of the solution provided from our approach. This can be explained because as we increment the number of initial partitions the ILS process has less impact as a more local search is performed, i.e., it is more difficult to leave local optima. Additionally, we also report that the convergence is in average a 25% slower w.r.t. a grouping of size 2.

Table 2 reports the results using the construction and ILS process explained in Sect. 2, using a grouping list of size 2. Note that the first number or letter of each instance corresponds to the set. The best results were obtained when three of the six evaluation functions were used (i.e., c_j/p_j, $c_j/\log(1+p_j)$ and $c_j/\sqrt{p_j}$),

In 47/65 instances the RPD is equal to 0. Also for 19 of these instances we report a σ equal to 0, which means that we reach the optimal value in each of the 30 runs. On the other hand, for the rest of the instances, we report results that are very close to the optimal costs. Finally, note that the standard deviation is small for all the instances, highlighting the stability of the approach.

In Fig. 3 we show a comparison between using and not using the ILS algorithm. Note that using this algorithm outperforms significantly the previous results.

3.3 Including the reward/penalty procedure

In this section we include the reward/penalty procedure. At the beginning of the search we set the penalty p_j to 1 for each column j. Additionally, p_j can only take values between $min = 0.5$ and $max = 1.5$. If some p_j is greater than 1.5 (resp. lower than 0.5) we set this penalty to 1.5 (resp 0.5). Several configuration were tested, but the best results were obtained using the values of $\alpha = 0.05$ and $\beta = 0.01$

Table 2 Results using the ILS procedure

Instance	Best-sol	Mean	σ	Min	RPD (%)	Instance	Best-sol	Mean	σ	Min	RPD (%)
4-1	429	429.53	0.19	429	**0**	B-4	79	79	0	79	**0**
4-2	512	513.07	0.37	512	**0**	B-5	72	72	0	72	**0**
4-3	516	516.1	0.11	516	**0**	C-1	227	232.2	0.56	230	1.32
4-4	494	494.17	0.14	494	**0**	C-2	219	222.5	0.51	219	**0**
4-5	512	512.93	0.38	512	**0**	C-3	243	245.57	0.40	244	0.41
4-6	560	560	0	560	**0**	C-4	219	220.1	0.30	219	**0**
4-7	430	432	0	432	0.47	C-5	215	217.53	0.38	215	**0**
4-8	492	492.1	0.11	492	**0**	D-1	60	60.77	0.16	60	**0**
4-9	641	644.67	0.49	641	**0**	D-2	66	66.43	0.19	66	**0**
4-10	514	514	0	514	**0**	D-3	72	72.53	0.25	72	**0**
5-1	253	255.3	0.34	253	**0**	D-4	62	62	0	62	**0**
5-2	302	305.2	0.35	303	0.33	D-5	61	61	0	61	**0**
5-3	226	226.73	0.37	226	**0**	E-1	29	29	0	29	**0**
5-4	242	242.4	0.19	242	**0**	E-2	30	30.7	0.22	30	**0**
5-5	211	212	0	212	0.47	E-3	27	27.9	0.11	27	**0**
5-6	213	215	0	215	0.94	E-4	28	28.27	0.17	28	**0**
5-7	293	293	0	293	**0**	E-5	28	28	0	28	**0**
5-8	288	288	0	288	**0**	F-1	14	14.03	0.07	14	**0**
5-9	279	279	0	279	**0**	F-2	15	15	0	15	**0**
5-10	265	265	0	265	**0**	F-3	14	14.83	0.14	14	**0**
6-1	138	138.13	0.19	138	**0**	F-4	14	14.13	0.13	14	**0**
6-2	146	146	0	146	**0**	F-5	13	13.97	0.07	13	**0**
6-3	145	145	0	145	**0**	G-1	176	185.17	0.87	182	3.41
6-4	131	131	0	131	**0**	G-2	154	161.6	0.60	159	3.25
6-5	161	161	0	161	**0**	G-3	166	175.3	0.59	172	3.61
A-1	253	254.6	0.30	253	**0**	G-4	168	177.83	0.77	173	2.98
A-2	252	254.17	0.46	252	**0**	G-5	168	176.23	0.71	171	1.79
A-3	232	235	0.33	234	0.86	H-1	63	67.57	0.38	65	3.17
A-4	234	234.7	0.17	234	**0**	H-2	63	66.57	0.34	65	3.17
A-5	236	237.23	0.16	237	0.42	H-3	59	62.47	0.35	61	3.39
B-1	69	69	0	69	**0**	H-4	58	61.5	0.31	59	1.72
B-2	76	76	0	76	**0**	H-5	55	57.83	0.35	56	1.82
B-3	80	80	0	80	**0**						

We highlight in bold the instances where the RPD is equal to 0

This procedure shows improvements in both, the quality of the solutions and the time of convergence.

The results can be seen in the Table 3. In 51/65 instances we get RPD equal to 0 (i.e., two more than before). We also obtain a great improvement in terms of the standard deviation. From the 51 instances obtaining a RPD equal to 0, in 28 we report $\sigma = 0$ and we can observe an important reduction in the rest of them.

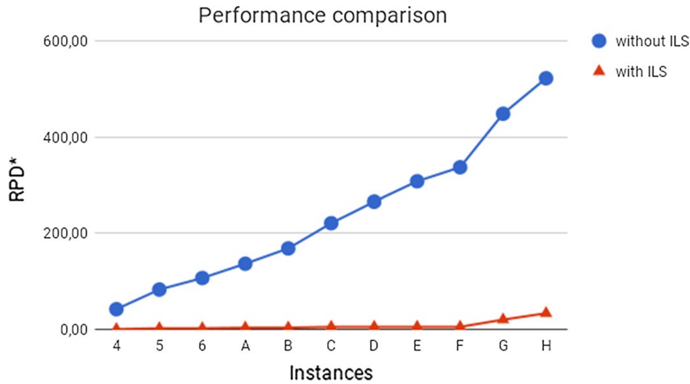

Fig. 3 Comparison of our approach using and not using the ILS process. The y-axis shows the *accumulative* RPD value while the x-axis indicates to the instance set

In Fig. 4 we include a comparison between our complete approach and the technique without the reward/penalty procedure. Note that the most important reductions occur in the last 10 instances, the hardest ones.

3.4 Comparing to modern metaheuristics

In this section we include a comparison between modern metaheuristics used in the last few years to deal with the SCP. The metaheuristics used for this comparison are: binary cuckoo search (BCS), binary black hole (BBH), the binary shuffled frog leaping algorithm (BSFLA) (Soto et al. 2017; Crawford et al. 2015), teaching-learning based optimization (TLBO) (Lu and Vasko 2015) and the K-means transition algorithm for the BBH (KMTA-BBH) (García et al. 2019). Note that, unlike our approach, these metaheuristics use sophisticated mechanisms and demands high tuning effort of its parameters for an optimal performance.

Table 4 reports the comparison between the previous techniques using the set problems E, F, G and H of the OR-library. Compared to these modern metaheuristics our approach behaves well, obtaining in 16/20 instances the best results (only being surpassed by KMTA-BBH with 19/20).

In Fig. 5 we include the comparison between our complete approach (GRASP) and some of the previous metaheuristics using the accumulative RPD. From the figure our approach outperforms the strategies BBH and BSFLA, and it performs similar to BCS. Our approach has a good performance in the instances B,C,D,E, however as it can be seen, in the last two group of instances BCS and GRASP have comparable performances.

In order to confirm the effectiveness of our approach, we use the ranked based statistical analysis. It consist in assigning ranks to each algorithm based on the performance. The performance of an algorithm a_1 is considered better than a_2; if a_1 obtains an average minimum objective function than a_2 in a shortest computational time. A rank 1 is assigned to the best performing algorithm, a rank 2 is assigned to the second best perform algorithm and so on. The average ranks for

Table 3 Results using the ILS process

Instance	Best-sol	Mean	σ	Min	RPD (%)	Instance	Best-sol	Mean	σ	Min	RPD (%)
4-1	429	429.87	0.13	429	**0**	B-4	79	79	0	79	**0**
4-2	512	512	0	512	**0**	B-5	72	72	0	72	**0**
4-3	516.2	516.2	0.25	516	**0**	C-1	227	229.83	0.44	228	0.44
4-4	494	494	0	494	**0**	C-2	219	221.8	0.25	221	0.91
4-5	512	512.43	0.31	512	**0**	C-3	243	243.83	0.24	243	**0**
4-6	560	560	0	560	**0**	C-4	219	219.47	0.21	219	**0**
4-7	430	432	0	432	0.47	C-5	215	216.4	0.25	215	**0**
4-8	492	492	0	492	**0**	D-1	60	60.73	0.17	60	**0**
4-9	641	643.2	0.37	641	**0**	D-2	66	66.1	0.11	66	**0**
4-10	514	514	0	514	**0**	D-3	72	72	0	72	**0**
5-1	253	255.23	0.35	253	**0**	D-4	62	62	0	62	**0**
5-2	302	304.73	0.43	303	0.33	D-5	61	61	0	61	**0**
5-3	226	226.57	0.32	226	**0**	E-1	29	29	0	29	**0**
5-4	242	242.43	0.19	242	**0**	E-2	30	30.5	0.19	30	**0**
5-5	211	212	0	212	0.47	E-3	27	27.83	0.14	27	**0**
5-6	213	215	0	215	0.94	E-4	28	28.1	0.11	28	**0**
5-7	293	293	0	293	**0**	E-5	28	28	0	28	**0**
5-8	288	288	0	288	**0**	F-1	14	14.2	0.15	14	**0**
5-9	279	279	0	279	**0**	F-2	15	15	0	15	**0**
5-10	265	265	0	265	**0**	F-3	14	14.77	0.16	14	**0**
6-1	138	138	0	138	**0**	F-4	14	14.2	0.15	14	**0**
6-2	146	146	0	146	**0**	F-5	13	13.77	0.16	13	**0**
6-3	145	145	0	145	**0**	G-1	176	177.2	0.28	176	0
6-4	131	131	0	131	**0**	G-2	154	157.47	0.27	156	1.30
6-5	161	161	0	161	**0**	G-3	166	170.97	0.39	169	1.81
A-1	253	255.17	0.28	254	**0**	G-4	168	174.27	0.38	172	2.38
A-2	252	252.6	0.27	252	**0**	G-5	168	172.5	0.4	170	1.19
A-3	232	234.67	0.30	233	0.43	H-1	63	65.37	0.27	64	1.59
A-4	234	234.03	0.07	234	**0**	H-2	63	65	0.14	64	1.59
A-5	236	237.17	0.20	236	**0**	H-3	59	60.97	0.07	60	1.69
B-1	69	69	0	69	**0**	H-4	58	59.57	0.25	58	**0**
B-2	76	76	0	76	**0**	H-5	55	55.83	0.22	55	**0**
B-3	80	80	0	80	**0**						

We highlighted in bold the instances where the RPD is equal to 0

the algorithms considered are: GRASP= 1.45, BCS = 1.75, BBH = 3.05, BSFLA = 3.75. According to this ranking, GRASP is the best performing algorithm among these techniques, followed by BCS, then BBH and BSFLA achieving the worst results. To analyze the statistical significance of differences between the evaluated ranks, we make use of the Nemenyi post-hoc test (Nemenyi 1963).

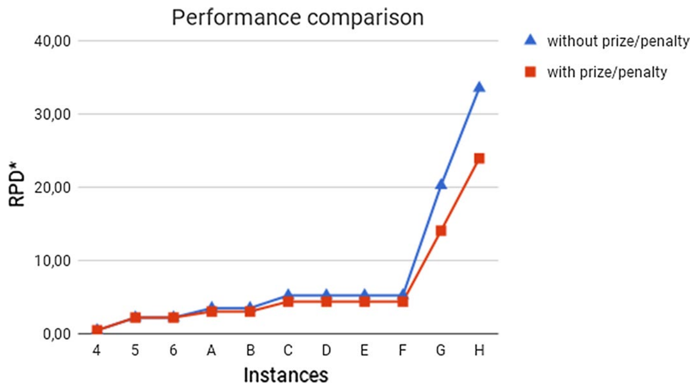

Fig. 4 Comparison between our approach using the ILS process and including the reward/penalty procedure. The y-axis shows the *accumulative* RPD value while the x-axis indicates to the instance set

Table 4 Comparison between our approach and modern metaheuristics in a subset of the OR-library

Instance	Best-sol	GRASP	BCS	BBH	BSFLA	TLBO	KMTA-BBH
E-1	29	**29**	**29**	**29**	**29**	**29**	**29**
E-2	30	**30**	31	31	31	**30**	**30**
E-3	27	**27**	28	28	28	28	**27**
E-4	28	**28**	30	29	29	**28**	**28**
E-5	28	**28**	**28**	**28**	**28**	**28**	**28**
F-1	14	**14**	**14**	**14**	15	**14**	**14**
F-2	15	**15**	**15**	**15**	**15**	**15**	**15**
F-3	14	**14**	15	16	16	**14**	**14**
F-4	14	**14**	15	15	15	15	**14**
F-5	13	**13**	14	14	15	**13**	**13**
G-1	176	**176**	**176**	179	182	179	**176**
G-2	154	156	156	158	161	156	**155**
G-3	166	169	169	169	173	168	**166**
G-4	168	172	**170**	**170**	173	172	**170**
G-5	168	170	170	**168**	174	**168**	**168**
H-1	63	**64**	**64**	66	68	**64**	**64**
H-2	63	**64**	**64**	67	66	**64**	**64**
H-3	59	**60**	61	65	62	61	**60**
H-4	58	**58**	59	63	63	59	59
H-5	55	**55**	56	62	59	56	**55**

For each instance, we highlighted in bold the best solution

This test is useful to determinate if the performance of two algorithms significantly differs or not. It considers the performance of two algorithms significantly different if their corresponding average ranks differ by at least a specific threshold critical difference. Considering a significance level of 1% level of significance the critical difference

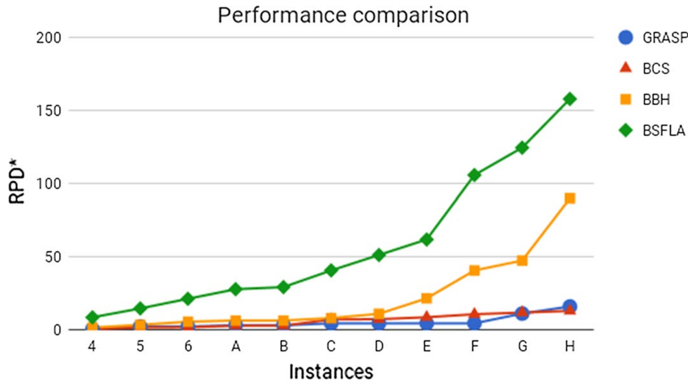

Fig. 5 Comparison between our approach and some modern metaheuristics

Table 5 Pairwise difference between the average ranks of the algorithms

Algorithm (rank)	GRASP (1.45)	BCS (1.75)	BBH (3.05)	BSFLA (3.75)
GRASP	–	0.30	1.60	2.30
BCS	–	–	1.30	2.00
BBH	–	–	–	0.70
BSFLA	–	–	–	–

Critical difference = 1.05 for a significance level of 1% for the Nemenyi post-hoc test

value is 1.05. In Table 5 shows the difference between the average ranks of the algorithms. From this table we infer that our GRASP approach and BCS have comparable performance, and that BBH and BSFLA are outperformed by our approach.

3.5 CPU times

In this final section we report the mean CPU times in seconds spent by our approach, BCS and Meta-RaPS for all the benchmark instances used in this work. The results are shown in Table 6. Our approach has good results compared to the other strategies, obtaining the best solution from Table 3 in less than 1 s in 39 instances. However in the hard instances (G and H) the algorithm requires a good amount of time to obtain good results. Instances G-1, G-2 and G-3 are the ones with the worst results. In 37/65 instances our approach converge faster.

3.6 Testing with new large-scale SCP instances

For today standards, the previous instances might not be large enough to prove the efficiency of our approach. In Umetani (2017) the author uses a SCP-generator to create a new set of larger SCP instances and propose a data mining approach for reducing the search space of local search algorithms. By generating a graph of the

Table 6 Cpu times in seconds for our grasp-based approach, binary cuckoo search and Meta-RaPS

Instance	GRASP	BCS	Meta-RaPS	Instance	GRASP	BCS	Meta-RaPS
4-1	**0.04**	1.92	1.36	B-4	**1.06**	3.39	2.25
4-2	0.29	1.92	**0.24**	B-5	0.43	3.45	**0**
4-3	0.44	2.15	**0.29**	C-1	1.99	3.27	**0.43**
4-4	**0.22**	2.07	0.39	C-2	4.69	**3.39**	12.8
4-5	**0.08**	2.14	0.90	C-3	4.37	**3.39**	26.2
4-6	**0.07**	2.09	0.10	C-4	**2.97**	3.35	24.2
4-7	0.16	1.90	**0.04**	C-5	**1.33**	3.49	1.79
4-8	**0.07**	1.95	1.46	D-1	**0.74**	4.68	3.13
4-9	**0.47**	1.95	3.47	D-2	**4.12**	4.70	13.5
4-10	0.10	2.16	**0.08**	D-3	**0.77**	4.69	1.31
5-1	**0.51**	2.01	1.55	D-4	5.43	5.69	**0.20**
5-2	**0.22**	2.08	0.59	D-5	1.44	5.74	**0.29**
5-3	**0.22**	2.19	1.14	E-1	**0.13**	5.82	0.73
5-4	0.47	2.21	**0.32**	E-2	**1.30**	4.81	46.1
5-5	**0.05**	2.02	0.33	E-3	**4.58**	4.91	5.95
5-6	**0.05**	2.30	0.14	E-4	**0.89**	4.84	39.6
5-7	**0.15**	2.14	1.03	E-5	**0.30**	4.76	0.81
5-8	**0.04**	2.13	0.08	F-1	**0.20**	8.36	4.29
5-9	0.10	2.05	**0.04**	F-2	**0.08**	9.86	3.80
5-10	0.06	2.14	**0.03**	F-3	**1.29**	9.82	1.84
6-1	**0.09**	2.81	0.25	F-4	**0.92**	9.23	5.44
6-2	0.04	2.84	**0.02**	F-5	**0.04**	8.82	33.2
6-3	**0.19**	2.85	0.02	G-1	80.3	**10.9**	298
6-4	**0.04**	2.90	0.34	G-2	59.3	**9.51**	222
6-5	**0.12**	2.93	1.02	G-3	55.8	**9.82**	21.5
A-1	2.04	**2.89**	6.22	G-4	**8.29**	9.33	194
A-2	**0.23**	2.71	0.28	G-5	66.6	**9.22**	47.5
A-3	4.66	**2.82**	16.9	H-1	43.4	**13.8**	3917
A-4	2.22	2.99	**0.04**	H-2	43.5	**14**	238
A-5	**0.68**	2.92	9.37	H-3	33.4	**14**	783
B-1	**0.11**	3.25	0.14	H-4	47.7	**16.1**	1358
B-2	**0.40**	3.38	0.53	H-5	36.8	19.3	**5.62**
B-3	1.42	3.37	**0.62**				

For each instance, we highlighted in bold the best CPU time

problem, the authors can identify promising pairs (resp. quartets) of variables in order to apply a 2-flip (resp. 4-flip) operation between them. Note that this approach is only useful in large instance problems, as the graph-related procedure is highly CPU-time demanding. Some of the benchmarks used in this work are described in Table 7. As it can be seen, these instances are considerably larger than the OR-library ones. To our knowledge, these new instances have not been tested by other

Table 7 Detail of the test instances

Instance set	No. of instances	# Rows	# Columns	Cost range
I	5	1000	50,000	[1, 100]
J	5	1000	100,000	[1, 100]

Table 8 Results of our approach using larger SCPs instances

Instance	Best-sol	Mean	σ	Min	RPD (%)	Time (s)
I-1	153	161	2.46	157	2.61	1089
I-2	158	165	2.00	163	3.16	918
I-3	153	159	2.63	155	1.31	853
I-4	165	172	2.86	168	1.82	646
I-5	161	169	2.25	165	2.48	640
J-1	128	137	2.98	132	3.13	1122
J-2	130	140	1.45	135	3.85	1722
J-3	128	135	1.58	133	4	1735
J-4	128	136.5	1.41	131	2	1640
J-5	131	140	3.91	134	2	1676

metaheuristics. The results reported by our approach can be seen in Table 8, which are compared to the optimum values reported in Umetani (2017). Additionally the CPU-time spent by the authors correspond to 1200 s (resp. 1800 s) for the I class instances (resp. J).

Despite not find any optimal solution, our approach behaves well in larger instances where both σ and the RPD have low values.

4 Conclusions

In this paper we present a GRASP scheme for solving the SCP. Our approach works in three phases: construction solutions, applying an ILS process and penalizing columns. In the first phase we basically select and instantiate columns using a set of evaluation functions based on a previous research. Unlike other GRASP algorithms, we consider *all* the columns satisfying certain criteria, thus no additional parameter is involved here. In the second phase we perform an iterative local search procedure to the solution, resetting a part of it and restarting the search from that point. Finally, we penalize columns by analyzing the solution reported by the ILS algorithm. We have tested our approach using a well known set of benchmark instances, obtaining promising results.

Without taking into account the reward/penalty procedure and the time limit, our GRASP algorithm is parameter free. Compared to more sophisticated

metaheuristics, we obtain comparable solutions in only a fraction of the time of the other approaches.

As a future work we plan to implement a more sophisticated mechanism for controlling the penalty of the columns in order to improve the results in larger instances and avoid local optima.

Acknowledgements This work is supported by the Fondecyt Project 1160224.

References

Balas E, Carrera MC (1996) A dynamic subgradient-based branch-and-bound procedure for set covering. Oper Res 44(6):875–890

Bartholdi JJ III (1981) A guaranteed-accuracy round-off algorithm for cyclic scheduling and set covering. Oper Res 29(3):501–510

Bautista J, Pereira J (2007) A grasp algorithm to solve the unicost set covering problem. Comput Oper Res 34(10):3162–3173

Beasley JE (1987) An algorithm for set covering problem. Eur J Oper Res 31(1):85–93

Beasley JE, Chu PC (1996) A genetic algorithm for the set covering problem. Eur J Oper Res 94(2):392–404

Bramel J, Simchi-Levi D (1997) On the effectiveness of set covering formulations for the vehicle routing problem with time windows. Oper Res 45(2):295–301

Brusco M, Jacobs L, Thompson G (1999) A morphing procedure to supplement a simulated annealing heuristic for cost- and coverage-correlated set-covering problems. Ann Oper Res 86:611–627

Chvatal V (1979) A greedy heuristic for the set-covering problem. Math Oper Res 4(3):233–235

Crawford B, Soto R, Cuesta R, Paredes F (2014) Application of the artificial bee colony algorithm for solving the set covering problem. Sci World J 2014:1–8

Crawford B, Soto R, Peña C, Palma W, Johnson F, Paredes F (2015) Solving the set covering problem with a shuffled frog leaping algorithm. In: Asian conference on intelligent information and database systems. Springer, pp 41–50

Crawford B, Soto R, Riquelme-Leiva M, Peña C, Torres-Rojas C, Johnson F, Paredes F (2015) Modified binary firefly algorithms with different transfer functions for solving set covering problems. In: Software engineering in intelligent systems. Springer, pp 307–315

Crawford B, Soto R, Córdova J, Olguín E (2016) A nature inspired intelligent water drop algorithm and its application for solving the set covering problem. In: Artificial intelligence perspectives in intelligent systems. Springer, pp 437–447

Fisher ML, Kedia P (1990) Optimal solution of set covering/partitioning problems using dual heuristics. Manage Sci 36(6):674–688

García J, Crawford B, Soto R, García P (2017) A multi dynamic binary black hole algorithm applied to set covering problem. In: International conference on harmony search algorithm. Springer, pp 42–51

García J, Crawford B, Soto R, Astorga G (2019) A clustering algorithm applied to the binarization of swarm intelligence continuous metaheuristics. Swarm Evol Comput 44:646–664

Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, San Francisco

Lan G, DePuy GW (2006) On the effectiveness of incorporating randomness and memory into a multi-start metaheuristic with application to the set covering problem. Comput Ind Eng 51(3):362–374

Lan G, DePuy GW, Whitehouse GE (2007) An effective and simple heuristic for the set covering problem. Eur J Oper Res 176(3):1387–1403

Lu Y, Vasko FJ (2015) An or practitioner's solution approach for the set covering problem. Int J Appl Metaheuristic Comput (IJAMC) 6(4):1–13

Munagala K, Babu S, Motwani R, Widom J, Thomas E (2005) The pipelined set cover problem. In: ICDT, vol 5. Springer, pp 83–98

Nemenyi P (1963) Distribution-free multiple comparisons. Unpublished Ph.D. dissertation, Princeton University, New Jersey, 73 pp

Pessoa LS, Resende MG, Ribeiro CC (2013) A hybrid lagrangean heuristic with grasp and path-relinking for set k-covering. Comput Oper Res 40(12):3132–3146

Resende MG (1998) Computing approximate solutions of the maximum covering problem with grasp. J Heuristics 4(2):161–177

Resende MG, Ribeiro CC (2010) Greedy randomized adaptive search procedures: advances, hybridizations, and applications. In: Handbook of metaheuristics. Springer, pp 283–319

Resende MG, Martí R, Gallego M, Duarte A (2010) Grasp and path relinking for the max–min diversity problem. Comput Oper Res 37(3):498–508

Solar M, Parada V, Urrutia R (2002) A parallel genetic algorithm to solve the set-covering problem. Comput Oper Res 29(9):1221–1235

Soto R, Crawford B, Olivares R, Barraza J, Figueroa I, Johnson F, Paredes F, Olguín E (2017) Solving the non-unicost set covering problem by using cuckoo search and black hole optimization. Nat Comput 16(2):213–229

Umetani S (2017) Exploiting variable associations to configure efficient local search algorithms in large-scale binary integer programs. Eur J Oper Res 263(1):72–81

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.