Applied Soft Computing xxx (2016) xxx–xxx

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

A genetic algorithm for the minimum generating set problem

Q1 Manuel Lozano[a], Manuel Laguna[b], Rafael Martí[c], Francisco J. Rodríguez[d,*], Carlos García-Martínez[e]

[a] Department of Computer Science and Artificial Intelligence, University of Granada, Granada, Spain
[b] Leeds School of Business, University of Colorado, Boulder, USA
[c] Department of Statistics and Operations Research, University of Valencia, Valencia, Spain
[d] Department of Computer Science, University of Extremadura, Mérida, Spain
[e] Department of Computing and Numerical Analysis, University of Córdoba, Córdoba, Spain

ARTICLE INFO

Article history:
Received 28 September 2015
Received in revised form 6 July 2016
Accepted 11 July 2016
Available online xxx

Keywords:
Minimum generating set problem
Genetic algorithms
Multiple knapsack problem
Real-parameter crossover operator

ABSTRACT

Given a set of positive integers S, the minimum generating set problem consists in finding a set of positive integers T with a minimum cardinality such that every element of S can be expressed as the sum of a subset of elements in T. It constitutes a natural problem in combinatorial number theory and is related to some real-world problems, such as planning radiation therapies.

We present a new formulation to this problem (based on the terminology for the multiple knapsack problem) that is used to design an evolutionary approach whose performance is driven by three search strategies; a novel random greedy heuristic scheme that is employed to construct initial solutions, a specialized crossover operator inspired by real-parameter crossovers and a restart mechanism that is incorporated to avoid premature convergence. Computational results for problem instances involving up to 100,000 elements show that our innovative genetic algorithm is a very attractive alternative to the existing approaches.

1. Introduction

Q2 The minimum generating set (MGS) problem, a natural problem in combinatorial number theory [1], is defined as follows: given a set of positive integers, $S = \{s_1, \ldots, s_n\}$, the problem consists of finding a minimum cardinality set of distinct integers $T = \{t_1, \ldots, t_m\}$, called *generating set*, such that every element of S is equal to the sum of a subset of T. The MGS problem has been shown to be NP-hard [1], and is related, among other problems, to planning radiation therapies [2–4]: the elements of S represent radiation dosages required at various points, while an element of T represents a dose delivered simultaneously to multiple points. Then, the objective is to find the set of doses that properly combined (subsets of T), produces the initial requirements (S). Other variants, namely the cases in which the elements of T can be negative or fractional, were considered elsewhere [5,6].

The greedy algorithm presented by Collins et al. [1] is the unique proposed approach for the MGS problem so far. Its idea is to represent the largest set of integers s_i by means of the combination of other integers s_j, previously accepted solution components t_k, and a new candidate solution component d. The process is repeated until all the integers $s_i \in S$ have a representation based on solution components. Fagnot et al. [7] gave some elementary properties of the minimum 2-generating set, a natural restriction of the MGS problem where each element of S must be represented by the sum of at most two elements from T, and proved its hardness. However, surprisingly, not a single metaheuristic approach has been applied so far (to our knowledge) to tackle the problem from a practical point of view. This fact was our main motivation for the development of a genetic algorithm (GA) that aims at optimizing the MGS problem. GA is a well known metaheuristic that has proved to be very effective in solving hard optimization problems [8,9].

In GA, a population of candidate solutions, called chromosomes, evolves over successive generations using three genetic operators: selection, crossover, and mutation. First of all, based on some criteria, every chromosome is assigned a fitness value, and then the selection mechanism is invoked to choose relatively fit chromosomes to be part of the reproduction process. Then, new chromosomes are created through the crossover and mutation operators. The crossover generates new individuals by recombining the characteristics of existing ones, whereas the mutation operator is used to maintain population diversity with the goal of avoiding premature convergence.

* Corresponding author.

http://dx.doi.org/10.1016/j.asoc.2016.07.020
1568-4946/

The proposal presented in this paper to successfully address the MGS problem rests on four pillars:

- We redefine the MGS problem using the terminology employed in the well-known multiple knapsack (MK) problem, which has been extensively studied within this class of algorithms.
- We devise a randomized greedy procedure specifically designed for generating feasible solutions for the given MGS case in reasonable computing times. Highly constrained combinatorial optimization problems such as the MGS problem have proved to be a challenge for metaheuristic solvers [10]. This is a situation in which it is difficult to define an efficient neighborhood, thus no local search is available [11]. Therefore, the incorporation of specialized constructive greedy heuristics is often necessary in order to produce practical implementations [10].
- On the basis of the new formulation, we propose a GA approach to deal with the MGS problem that comprises an initial population generation method (based on the proposed randomized greedy algorithm) with the goal of acquiring a population of diversified, yet adequate quality solutions, and a restart mechanism, substituting the usual GA mutation, to regenerate population diversity when chromosomes become very similar.
- In addition, the proposed GA incorporates an innovative specialized recombination operator, inspired by real-parameter crossovers [12], which maintains the feasibility and legality of the offspring as solutions to the problem.

The rest of this paper is organized as follows. Section 2 presents the MK-based interpretation of the MGS problem, their similarities and differences. Section 3 introduces the new randomized greedy heuristic for the MGS problem, which constitutes one of the essential components of the proposed GA. Section 4 describes the evolutionary approach for the MGS problem. Section 5 provides an analysis of the GA performance and draws comparisons with the existing literature. Finally, Section 6 contains a summary of results and conclusions.

2. The MGS problem as searching objects for knapsacks

In the MK problem, we are given a set of objects O and a set of knapsacks K. Each object $o_j \in O$ has a profit, $p(o_j)$, and a weight, $w(o_j)$, and each knapsack $k_i \in K$ has a capacity, $C(k_i)$. The objective in the MK problem is to allocate each object to at most one knapsack in such a way that the total weight of the objects in each knapsack does not exceed its capacity and the total profit of all the objects included in the knapsacks is maximized. Its mathematical formulation, shown in Fig. 1(left) [13,14], is based on a set of binary variables $x_{o_j k_i}$, where $x_{o_j k_i} = 1$ indicates that object o_j is included in knapsack k_i, and $x_{o_j k_i} = 0$, otherwise. GAs and other metaheuristics applications to the MK problem and its variants [15–17,14] usually encode solutions as integer arrays whose lengths are equal to the number of objects, and the respective o_jth element indicates the knapsack k_i where it is included into, or an invalid value if it is not assigned to any knapsack.

In the MGS problem, the elements $s_i \in S$ may be recognized as knapsacks with capacities equal to their s_i values ($C(s_i) = s_i, \forall s_i \in S$). Then, the elements in a candidate generating set, $t_j \in T$, are objects that may be inserted in the knapsacks, with weights equal to their values ($w(o_j) = t_j, \quad \forall t_j \in T$). In this fashion, the objective of the MGS problem may be reformulated as constructing the smaller set of objects T, such that every knapsack is completely filled by including replicas of different objects from T. Noticing that no integer value j greater than the maximal element in S, S_{max}, may belong to a generating set T, we can reformulate the problem of constructing the set of objects T as the one of *selecting* those from the set $\{1, \ldots, S_{max}\}$ that will belong to T.

The MK mathematical formulation can be adapted for the MGS problem as shown in Fig. 1(right). There $x_{js_i} = 1$ indicates that integer j contributes to fill knapsack s_i ($x_{js_i} = 0$, otherwise); and consequently, x_j must be equal to 1, which expresses that integer j belongs to the generating set T.

Whereas this knapsack-based interpretation provides a mathematical adaptation and a pictorial analogy for the MGS problem, which is finding objects that properly combined fill all the knapsacks, their differences should be clearly remarked:

- The capacity constraints (2) become equality constraints, i.e., the sum of the weights of the objects in a knapsack must be equal to its capacity.
- There are not profits, so they disappear from the objective (1). Additionally, the objective is transformed into a minimization problem, to reduce the number of created objects.
- Objects may be placed in more than one knapsack, so the constraint (3) is not present.
- Knapsacks cannot carry two or more objects with the same weight, so integers j apply only once in the summation in the constraint (2).
- Objects must be created for solving the problem, whereas they are initially given in the MK problem.

  This becomes a hard restriction, since the solver has to consider combinations of every possible object. This can be addressed by searching in the space of combinations of elements in the set $\{1, \ldots, S_{max}\}$, either exploiting the mathematical formulation (Fig. 1, right) or applying a metaheuristic with integer arrays for the possible S_{max} objects. However, this becomes impractical for large S_{max} values. For example, we could not obtain any valid solution with CPLEX V12.1 and the model in Fig. 1(right) for a random instance with $|S| = 20$ and $S_{max} = 4096$ after one hour.

Regarding our proposed GA, since the direct adaptation of GAs for the MK problem to the MGS one is not viable, we will propose a randomized greedy heuristic that evaluates sets with a restricted number of samples from $\{1, \ldots, S_{max}\}$ (Section 3). Our GA will use it at different stages, namely initialization, restart, and crossover. To address extremely hard problems, a common strategy concerns to include heuristic subordinate procedures into the stages of metaheuristics [18–20].

Finally, note that given a solution for this reformulated knapsack problem, i.e. the set of objects ($O = \{o_j\}$), we may directly obtain a generating set T for S by building a set with the weight values of the objects ($T = \{t_j = w(o_j), o_j \in O\}$).

3. Randomized greedy heuristic

In this section, we propose a randomized greedy heuristic for the MGS problem, which is called RG-MGS. The design of RG-MGS (Fig. 2) is specified under the new formulation for the MGS problem presented in this paper. Therefore, one of its inputs is the set of knapsacks K associated with S, and the output is the set of created objects, O.

RG-MGS starts with all the knapsacks empty and constructs one object at a time, which is added to the current partial solution, O, until all the knapsacks are completed. Specifically, the algorithm manages the free spaces in the knapsacks, $F = \{f_1, \ldots, f_n\}$ (f_i stores the free space in knapsack k_i), and creates an object with a weight value belonging to the set $\{1, \ldots, F_{max}\}$ (the weight of the biggest possible object is equal to the greatest free space in any knapsack, F_{max}) with the aim of minimizing the global free space in the knapsacks after the insertion of the new object. To do this, RG-MGS uses

Please cite this article in press as: M. Lozano, et al., A genetic algorithm for the minimum generating set problem, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.07.020

$$\max \sum_{k_i \in K} \sum_{o_j \in O} p(o_j) x_{o_j k_i} \quad (1)$$

subject to:

$$\sum_{o_j \in O} w(o_j) x_{o_j k_i} \leq C(k_i), \ \forall k_i \in K \quad (2)$$

$$\sum_{k_i \in K} x_{o_j k_i} \leq 1, \ \forall o_j \in O \quad (3)$$

$$x_{o_j k_i} \in \{0,1\}, \ \forall o_j \in O, k_i \in K \quad (4)$$

$$\min \sum_{j=1}^{S_{max}} x_j \quad (1)$$

subject to:

$$\sum_{j=1}^{S_{max}} j x_{j s_i} = C(s_i), \ \forall s_i \in S \quad (2)$$

$$x_{j s_i} \in \{0,1\}, \ \forall j \in \{1, \ldots, S_{max}\}, s_i \in S \quad (3)$$

$$x_j = \begin{cases} 1, & \sum_{s_i \in S} x_{j s_i} \geq 1 \quad (j \in T) \\ 0, & \text{otherwise} \quad (j \notin T) \end{cases} \quad (4)$$

Fig. 1. Mathematical model of the multiple knapsack problem (left), and the minimum generating set problem (right).

```
   Input: K, n_sample
   Output: O
1  O ← ∅;
2  F ← {f_1, ··· , f_n} where f_i ← C(k_i), k_i ∈ K, for i = 1, ··· , n;
3  while F_max ≠ 0 do
4  |  R ← Sample n_sample random numbers from {1, ··· , F_max};
5  |  ω' ← argmax_{ω∈R} ψ(ω, F);
6  |  o ← Create-Object (ω');
7  |  O ← O ∪ {o};
8  |  κ(o) ← ∅;
9  |  for i = 1, ··· , n do
10 |  |  if w(o) ≤ f_i then
11 |  |  |  f_i ← f_i − w(o);
12 |  |  |  κ(o) ← κ(o) ∪ {k_i};
13 |  |  end
14 |  end
15 end
16 O ← Eliminate-Duplicates (O, κ);
```

Fig. 2. Pseudocode algorithm for RG-MGS.

```
   Input: O, κ
   Output: O'
1  O' ← O ;
2  while ∃ i, j ∈ {1, ··· , |O'|} | i ≠ j and w(o_i) = w(o_j) do
3  |  if κ(o_i) ∩ κ(o_j) ≠ ∅ then
4  |  |  o_k ← Create-Object (2 · w(o_i));
5  |  |  O' ← O' ∪ {o_k};
6  |  |  κ(o_k) ← κ(o_i) ∩ κ(o_j);
7  |  end
8  |  κ(o_i) ← (κ(o_i) ∪ κ(o_j)) \ (κ(o_i) ∩ κ(o_j));
9  |  if κ(o_i) = ∅ then
10 |  |  O' ← O' − {o_i};
11 |  end
12 |  O' ← O' − {o_j};
13 end
```

Fig. 3. Pseudocode algorithm for `Eliminate-Duplicates` procedure.

the *global contribution* ($\psi(\omega, F)$) of an object with weight ω to fill up the knapsacks:

$$\psi(\omega, F) = \sum_{i=1}^{n} \delta(\omega, f_i), \quad \text{where } \delta(\omega, f_i) = \begin{cases} \omega, & \text{if } \omega \leq f_i \\ 0, & \text{otherwise} \end{cases}$$

Since the calculation of the global contribution for all the possible weight values in the set {1, ..., F_{max}} may become a time-consuming strategy, RG-MGS employs an alternative random sampling technique, which consists of choosing, at random, n_{sample} (a control parameter of RG-MGS) weight values in this set (Step 4). Later, it selects the one for which the global contribution value across these sampled weights is maximal (Step 5), and builds an object with exactly that weight (Steps 6 and 7). Subsequently, RS-MGS introduces the new object into every knapsack with sufficient free space and updates their free spaces (Steps 8–13). $\kappa(o)$ denotes the set of knapsacks containing object o. The algorithm finishes when the objects assigned to the knapsacks make exactly their respective capacities.

During the run of RG-MGS, it may be possible to create objects with the same weights, and then, the final associated generating set would become infeasible. Therefore, RG-MGS ends its execution by invoking a repair procedure, called `Eliminate-Duplicates`, which resolves this problematic situation (Fig. 3). The main ideas of `Eliminate-Duplicates` are the following. Whenever there exist two different objects, o_i and o_j, with $w(o_i) = w(o_j)$, this procedure pays attention to the set of knapsacks containing both objects ($\kappa(o_i) \cap \kappa(o_j)$). If this set is not empty, a new object with twice the weight of o_i is created (Step 4) and it replaces both o_i and o_j in these knapsacks. Moreover, `Eliminate-Duplicates` replaces o_j by o_i in those knapsacks that uniquely contained o_j. Finally, it removes object o_j from the solution (Step 12). If there was already an object with twice the weight of o_i, the procedure will address the new conflict in a subsequent iteration. To sum up, the proposed procedure replaces two objects with equal weights with one of the same weight and another with double the weight.

In another metaheuristic based on constructions, namely GRASP, it is customary to construct a solution by selecting iteratively elements at random from a restricted candidate list of elements with good evaluations. Recent designs, however, have proved that this classic design can be improved in some problems by first applying a random sampling and then performing a greedy selection from the sampled elements (see for example [21]). Our proposal here is in line with these recent GRASP designs.

4. Steady-state genetic algorithm for the MGS problem

GAs [8,9] are adaptive methods based on the genetic process of biological organisms. They are widely used in many combinatorial and real-parameter optimization problems. GAs start with an initial set of random solutions (or seeded candidates with some good heuristic method) forming a population of so-called chromosomes of size N_p. Chromosomes evolve from population to population through successive iterations, called generations, keeping N_p fixed throughout the iterations. Each chromosome has a fitness value associated with it.

In each generation, chromosomes are evaluated by the fitness function (related to the objective of the problem) to assess their competence to survive in the next generation. The fittest chromosomes are selected to form a new population, which subsequently undergoes genetic operations: mutation of a single chromosome and crossover between two ones (parents get offspring). Genetic operators are applied to the selected solutions to produce new ones with inherited characteristics of their parents and the associated fitness function evaluates the extent to which they achieve the goal of the optimization problem.

Generational GAs abandon the current population once the whole offspring population had been created, which becomes the

Please cite this article in press as: M. Lozano, et al., A genetic algorithm for the minimum generating set problem, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.07.020

```
   Input: K, N_p, n_sample, c_sample
   Output: C_b
   // Initialization
1  for i = 1, ..., N_p do
2  |  C_i ← RG-MGS (K, n_sample);
3  end
   // Main GA cycle
4  while not termination-condition do
   |  // Selection
5  |  P_1 ← Binary-Tournament (Pop);
6  |  P_2 ← Binary-Tournament (Pop);
   |  // Crossover
7  |  C_cx ← Crossover-Operator
   |  (P_1, P_2, K, c_sample);
8  |  C_b ← Best-Found-Solution ();
   |  // Replacement
9  |  C_w ← argmax_{C∈Pop} fitness(C);
10 |  if fitness(C_cx) < fitness(C_w) then
11 |  |  C_w ← C_cx ;
12 |  end
   |  // Restart
13 |  if Fitness values of individuals in Pop are equal then
14 |  |  C_1 ← Sample a random
   |  |  individual from Pop;
15 |  |  for i = 2, ..., N_p do
16 |  |  |  C_i ← RG-MGS (K, n_sample);
17 |  |  end
18 |  end
19 end
```

Fig. 4. Pseudocode algorithm for GA-MGS.

new current population. On the contrary, steady-state GAs [22–24] typically generate one single new solution and insert it into the population at any one time. A replacement/deletion strategy defines which member of the current population is forced to perish for the new offspring to compete in the next iteration.

In this section, we introduce the use of an advanced steady-state GA to solve the MGS problem. Next, Section 4.1 summarizes the overall evolutionary process flow in the proposed GA and Section 4.2 explains the details of the specific crossover operator designed to generate feasible solutions from two parent chromosomes.

4.1. Overall GA algorithm

An outline of the proposed GA, called GA-MGS, is given in Fig. 4. It is a steady-state GA, whose chromosomes represent sets of objects that fill the different knapsacks, and their fitness values are equal to the number of objects used. The chromosome encodes for each object its weight and the set of knapsacks where it is placed.

GA-MGS operates in two phases: first, the initialization, during which the population is filled with N_p solutions generated by the RG-MGS procedure (Section 3), and next, the population is subject to an evolutionary loop that adopts the following operations:

1. Select two parents from the population using the *binary tournament* selection mechanism (Steps 5 and 6). This selection technique is widely used in GAs due to its simplicity and ability to escape from local optima. It selects the fittest chromosome between two that are randomly picked out from the population.
2. Create an offspring applying the crossover operator (Step 7; Section 4.2). This is a method for sharing information between individuals that combines the features of two parents to create potentially better offspring. The underlying idea is that the exchanging of genetic material among good individuals is expected to generate good or even better individuals. Therefore, this operator exploits the available information from the population.
3. Select an individual from the population and decide if this individual will be replaced by the offspring (Steps 9–12). For this decision, we consider the *replace worst* strategy, which replaces the worst individual in the population only if the new individual is better. This mechanism induces a high selective pressure even when the parents are selected randomly [25].
4. A *restart* process is fired (Steps 13–18) when the population has converged (the fitness function values of all the individuals in the population are equal). In this situation, the population is reinitialized with $N_p - 1$ new solutions generated by the RG-MGS method and a randomly chosen one from the current population. This source of diversity allows the high selective pressure associated with the replacement and parent selection mechanisms to be counteracted with the aim of avoiding premature convergence.

These steps are repeated until some termination condition (e.g. maximum number of iterations, or maximum computation time allowed; Step 4) has been met. The best chromosome, C_b, generated during the iterative process is kept as the overall result.

4.2. Crossover operators to induce objects preferences

The crossover operator has always been regarded as one of the main components that guides the search process of GAs [9,26], because it gathers up, combines, and exploits the available information on previous samples to influence future search directions [27–29]. Real-coded genetic algorithms [30,31,12] are a prominent research field where a substantial effort has been put into the development of sophisticated real-coded crossover operators. Herrera et al. [27] presented a taxonomy to classify the crossover operators for real-coded GAs. *Neighborhood-based* crossover operators are a class of the taxonomy that has been found to be effective in many cases. They determine the genes of the offspring by sampling probability distributions associated with the values of the genes of the parents, which often define the central position and extent of the distribution.

Inspired by several neighborhood crossover operators proposed in the literature, we present in this section three crossover operators (CX-BLX-α, CX-PBX-α, and CX-MP-α) aimed at exploiting the information from two or more parents (their objects o_i) to induce some preferences when creating the objects for the offspring. Section 4.2.1 describes the general and common structure they share to generate objects for the offspring and fill up its knapsacks. Section 4.2.2 details two prominent types of probability distribution usually considered in neighborhood-based crossover operators, and thus, used in this work. Section 4.2.3 assembles previous concepts into three concrete crossover operators, CX-BLX-α, CX-PBX-α, and CX-MP-α.

4.2.1. General structure of the crossover operators for the MGS problem

In contrast to the uniform distribution sampled by RG-MGS, which covers all the possible weights for new objects, the idea here is to exploit the information in the population to induce some preferences when sampling new weight values. Specifically, the motivation is to promote a fruitful synergy between the selection pressure of the parent selection and the replacement strategy, which maintains good solutions in the population, and the crossover operator, which generates new solutions, to concentrate the sampling in promising limited intervals.

The general structure of the proposed crossover operators is similar to that of RG-MGS but using the objects in the parents

Please cite this article in press as: M. Lozano, et al., A genetic algorithm for the minimum generating set problem, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.07.020

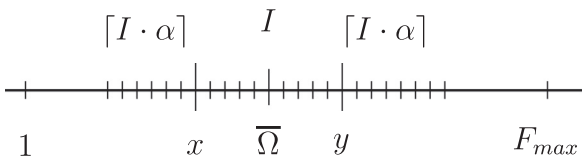

Fig. 5. Possible values sampled from $Pr_{mean}(\omega|\Omega, F)$.

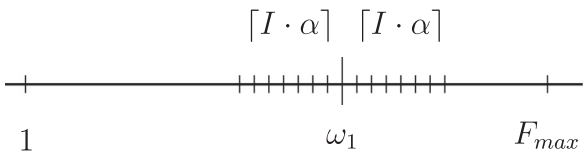

Fig. 6. Possible values sampled from $Pr_{parent}(\omega|\Omega, F)$.

to modify the originally-uniform probability distribution; starting with all the knapsacks of the offspring empty and iterating the following steps until they are filled up.

1. Construct a set of objects with one object per parent, $O_{parents}$. For this step, we select the largest object from the first parent, not previously selected, and the most similar to this one from the second parent, not previously selected, too. In the case of more than two parents, parents are revised iteratively and, each one contributes with the object most similar to those already selected. The idea here is to favor the creation of objects with high global contributions $\psi(\omega_i, F)$ (Section 3), by using the largest objects first, and the smaller ones later to fill the gaps. Additionally, similarity is justified to concentrate the associated probability distribution, from which new objects are sampled, on these objects.
2. Define a probability distribution $Pr(\omega|\Omega, F)$ according to the weights of the selected objects ($\Omega = \{w(o), \forall o \in O_{parents}\}$), and the free spaces in the knapsacks of the offspring (F). Here, we follow the idea from two significant families of neighborhood-based crossover operators, *mean-centric* and *parent-centric* ones. The former centers the probability distribution on an aggregated measure of the values of the parents, whereas the latter centers it on the value of one of the parents. Particularly, we design appropriate distributions for the MGS problem, inspired by BLX-α [31] (mean-centric) and PBX-α [32] (parent-centric) (Section 4.2.2).
3. Sample c_{sample} integers from $Pr(\omega|\Omega, F)$.
4. Evaluate their global contributions $\psi(\omega_i, F)$.
5. Create an object with weight equals to the integer with $\psi(\omega_i, F)$ maximum and insert it in every knapsack with sufficient free space.

If this process left some knapsacks incomplete, because all the objects from the parents have been selected once, they would be filled up with RG-MGS. Finally, the `Eliminate-Duplicates` procedure (Section 3) is invoked to repair the possible existence of more than one object with the same weight.

4.2.2. Mean-centric and parent-centric distributions

Given a set of weights, from objects selected from the parents that undergo crossover, Ω, we define the following probability distributions from which new weights are sampled. We shall indicate that they come (when $|\Omega| = 2$ and $\omega \in \mathbb{R}$) from the definition of the BLX-α [31] and PBX-α [32] crossover operators, respectively:

- $Pr_{mean}(\omega|\Omega, F)$ is uniform in the following integer range, which is centered on the mean of the set of weights ($\overline{\Omega}$), and is equal to 0 out of it (Fig. 5):

$$V_I = \{\max\{1, \overline{\Omega} - \lceil I \cdot (1+\alpha) \rceil\}, \ldots, \min\{F_{\max}, \overline{\Omega} + \lceil I \cdot (1+\alpha) \rceil\}\},$$

where $I = y - x$, $x = \min(\Omega)$, $y = \max(\Omega)$, and α is a control parameter.

- $Pr_{parent}(\omega|\Omega, F)$ is uniform in the following integer range, which is centered on the first weight of the set ($\omega_1 \in \Omega$), and is equal to 0 out of it (Fig. 6):

$$V_{II} = \{\max\{1, \omega_1 - \lceil I \cdot \alpha \rceil\}, \ldots, \min\{F_{\max}, \omega_1 + \lceil I \cdot \alpha \rceil\}\},$$

where $I = y - x$, $x = \min(\Omega)$, $y = \max(\Omega)$, and α is a control parameter.

4.2.3. Crossover operators

Gathering up previous ideas, we present the following crossover operators:

- CX-BLX-α, named after BLX-α [31], takes two parent solutions and produces an offspring applying the steps commented in Section 4.2.1 and using the $Pr_{mean}(\omega|\Omega, F)$ probability distribution. Thus, this is a mean-centric crossover operator.
- CX-PBX-α, named after PBX-α [32], is similar to CX-BLX-α but uses the $Pr_{parent}(\omega|\Omega, F)$ probability distribution. Therefore, this is a parent-centric crossover operator.
- CX-MP-α is motivated by the superior performance shown by some *multi-parent* real-parameter crossover operators that combine the features of more than two parents [29] (examples are UNDX [33], SPX [34], and PCX [30]). This operator takes n_p parents, which is a parameter, and generates an offspring applying the steps in Section 4.2.1 using the $Pr_{mean}(\omega|\Omega, F)$ probability distribution. Thus, this is a mean-centric crossover operator. The complimentary multi-parent crossover operator using $Pr_{parent}(\omega|\Omega, F)$ is not analyzed in this work because the first empirical results, shown in Section 5.2, conclude that mean-centric crossover operators produce better solutions than parent-centric ones for the MGS problem.

The designed crossover operators exploit a subordinate heuristic procedure, which samples a set of weights and returns the one with the maximal global contribution, to generate new solutions for this complex problem. In particular, this helps the GA with dealing with a very large set of possible candidate objects to be created ($\{1, \ldots, S_{max}\}$; as mentioned in Section 2. This strategy resembles the indirect encoding methodology of several metaheuristics that incorporate a subordinate greedy decoder. In these cases, the individuals in the population do not directly represent candidate solutions, but the way a greedy algorithm should construct feasible ones [18–20].

5. Computational experiments

This section describes the computational experiments that we conducted to assess the performance of the evolutionary approach introduced in the previous section to face the MGS problem. Firstly, we detail the experimental setup (Section 5.1) then, we analyze the results obtained from different experimental studies carried out with this algorithm. Our aim is: (1) to analyze the influence of the parameters and settings associated with GA-MGS (Section 5.2), and (2) to compare the results of the proposal with those of a greedy

Please cite this article in press as: M. Lozano, et al., A genetic algorithm for the minimum generating set problem, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.07.020

heuristic approach for the MGS problem from the literature (Section 5.3).

5.1. Experimental setup

All algorithms have been implemented in C and the source code has been compiled with gcc 4.8.2. The experiments were conducted on a computer with a 3.2 GHz Intel® Core™ i7 processor with 24 GB of RAM running Fedora™ Linux V20. Importantly, the establishment of common execution conditions for all algorithms is essential for an adequate replication of previous computational experiments, which is a necessary condition for obtaining meaningful findings [35]. In this regard, we have chosen execution time as comparison metric, instead of number of evaluations, since our proposal employs a repair procedure prior to the evaluation of each solution, which undoubtedly adds an additional computational cost that it is difficult to quantify to perform a fair comparison with other algorithms if not using time as comparison metric. In addition of using for all experiments the same conditions such as programming language or operating system, we have also tried to minimize the effect of any OS background task on the algorithm's performance. To achieve this, we limit the number of simultaneous algorithm executions to five in the same computer, being 3 of the 8 running threads in the Intel Core i7 processor available to operating systems tasks. Moreover, it is important to note that the findings of the experimental comparison are obtained from a large number of executions on different instances, limiting the incidence of a one-time OS task on those findings.

We considered two types of benchmark instances for our experiments, which are described below; they were automatically generated and parameterized by the number of values to be represented, $n = |S|$.

- **Instances Type U.** They are unconstrained instances that consist of n integers randomly sampled from the positive range of possible values of a signed integer variable of the C programming language (gcc 4.8.2, {1, . . ., 2, 147, 483, 647}). An upper bound for the value of the optimal solution for this type of instances is $log_2(S_{max}) = 31$ [1].
- **Instances Type L.** They are limited-representation instances that are additionally parameterized by the maximal value in the set S, S_{max}, and a desired optimal objective value, $m = |T|$. The procedure that generates the problem instance firstly samples an initial candidate solution with m random integers in $\{1, \ldots, T_{max}\}$. Then, the numbers in S are computed as sums of random subsets of T, whose sum is less or equal to S_{max}.

  The parameter m limits, in contrast to unconstrained instances, the complexity of the representations of the elements in S. For instance, m equals to ten impedes the existence of any number in S, whose representation, in the optimal solution, required the sum of more than ten numbers. In addition, m is a known upper bound for the value of the optimal solution. For these instances, T_{max} has to be computed properly. On the one hand, T_{max} has to be inferior or equal to S_{max}, and on the other, values of T_{max} close to S_{max}, favors the apparition of numbers in S with simple representations (sum of few elements from T). Thus, we have computed T_{max} as $2 \cdot S_{max}/m$, what makes that the expected complexity of the numbers in S be at least $m/2$.

When comparing algorithms, we compute the overall best solution value for each problem instance, *BestValue*, obtained by the execution of the algorithms under consideration (in each comparison analysis). Afterwards, for each algorithm, we calculate the relative deviation between the best solution value found by the method and *BestValue*. Then, we report the average of this relative deviation in percentage (*%D*) across all the instances considered in each particular experiment and the percentage of instances (*%B*) for which the value of the best solution obtained by a given method matches *BestValue*. We will also show the average *rankings* (*Av Rank*) achieved by these algorithms, computed by the Friedman test. This measure is obtained by computing, for each instance, the ranking r_a of the observed results for algorithm a assigning to the best of them the ranking 1, and to the worst the ranking $|A|$ (where A is the set of algorithms). Then, an average measure is obtained from the rankings of this algorithm for all test problems. For example, if a certain algorithm achieves rankings 1, 3, 1, 4, and 2, on five instances, the average ranking is (1 + 3 +1 + 4 +2)/5 =2.20. Note that the lower the ranking, the better the algorithm.

```
Input: K, n_sample
Output: O
O ← RG-MGS(K, n_sample);
while stopping criterion not satisfied do
    O' ← RG-MGS(K, n_sample);
    if fitness(O') < fitness(O) then
        O ← O';
    end
end
```

Fig. 7. Pseudocode algorithm for MS-RG-MGS.

Table 1
Results for the multi-start RG-MGS algorithm.

n_{sample}	*Av Rank*	*%D*	*%B*
1	5.470	19.0	0
2	2.614	4.5	20.4
5	1.322	0.5	86
10	2.914	8.1	10
25	4.584	16.4	4
50	5.256	20.5	5.2
100	5.840	23.3	2.4

5.2. Components and parameter tuning

In this section, we investigate the effect of the different parameters and strategies applied in GA-MGS and their interactions. For these experiments, we consider a set of 250 instances *Type U* that range from $n = 50$ to $n = 1000$. All the algorithms were stopped after a time limit of $\frac{n}{10}$ s to have a fair comparison. Additionally, each algorithm was executed once for each problem instance.

Given the relevance of the RG-MGS procedure as generator of good starting solutions for our GA, the first preliminary experiment is devoted to adjusting the n_{sample} parameter of this procedure. To do this, we have implemented a multi-start metaheuristic that invokes repeatedly RG-MGS until a termination condition (e.g. maximum computation time allowed) has been met. The best solution generated during the iterative process is kept as the overall result. The complete pseudocode of this procedure, which will be termed MS-RG-MGS, may be found in Fig. 7. MS-RG-MGS is an instance of the so-called *randomized greedy multi-start* metaheuristic, which was described by Martí et al. [36] as a type of multistart method that fires a greedy procedure integrating random diversification. Early instances can be found under the name of *semi-greedy* heuristic [37].

We ran 7 different MS-RG-MGS variants with $n_{sample} = \{1, 2, 5, 10, 25, 50, 100\}$. Table 1 shows the results: averaged ranking, relative deviation from the best result, and percentage of successful runs.

The results in Table 1 show that the choice of the value for n_{sample} has an important influence on the RG-MGS behavior. According to the three performance measures, the variant with $n_{sample} = 5$

Table 2
CX-BLX-α vs. CX-PBX-α.

Cross. Op.	c_{sample}	*Av Rank*	*%D*	*%B*
	1	10.424	8.4	0
	2	6.638	3.6	14
CX-BLX-α	5	4.192	1.8	48
$Pr_{mean}(\omega\|\Omega, F)$	10	3.716	1.3	56.4
	20	4.764	2.2	44.4
	30	6.732	3.9	21.2
	1	11.972	16.2	0
	2	9.262	6.3	0.8
CX-PBX-α	5	5.428	2.7	22
$Pr_{parent}(\omega\|\Omega, F)$	10	4.646	2.1	38.4
	20	4.742	2.2	38.8
	30	5.484	2.8	28.8

obtains, clearly, the best results (therefore, we choose this n_{sample} value for all remaining experiments involving GA-MGS). Moreover, the worst performance is achieved when: (1) there is not a competition among different candidate objects to enter into the knapsacks ($n_{sample} = 1$) or (2) too much time is wasted in this competitive process ($n_{sample} = 100$).

We now undertake to analyze the effects of the type of probability distribution used by the crossover operator on the GA-MGS performance. Specifically, we have implemented 12 GA-MGS variants that apply CX-BLX-α ($\alpha = 0.25$; $Pr_{mean}(\omega|\Omega, F)$) and CX-PBX-$\alpha$ ($\alpha = 1$; $Pr_{parent}(\omega|\Omega, F)$) with different values for c_{sample}({1, 2, 5, 10, 20, 30}). Though the values for α are different for each crossover operator, they produce interval amplitudes for the probability distributions, more similar than using the same value. The population size for these GAs is set to 50 and $n_{sample} = 5$ for the RG-MGS procedure, which is employed by the GAs in the initialization phase and by the restart mechanism. Table 2 shows the associated results, reporting again the same statistics. We can draw the following conclusions from this table:

- In general, GA-MGS variants with CX-BLX-α are better than the corresponding ones with CX-PBX-α. In addition, the two best ranked algorithms are based on CX-BLX-α (those with c_{sample} equal to 10 and 5, respectively). As was pointed out in Section 4.2, this operator samples weight values from a mean-centric probability distribution. This fact indicates that the sampling of the weights for objects of the offspring becomes more profitable when the two parents participate equitably in the crossover operation than when one of the parents has a predominant rule in this operation.
- For both crossover operators, the best outcomes (rankings) are obtained when $c_{sample} = 10$ and the worst ones with too low c_{sample} values. Thus, the evaluation of many different objects as candidate components for the knapsacks aroused as a suitable strategy to create promising offspring.

To complement these experiments, we now compare the GA-MGS that has given the best results in the previous experiments, the one invoking CX-BLX-α with $c_{sample} = 10$, with an alternative variant that applies a more simple crossover procedure, SCX. This operator is based on the idea of the classical standard crossover operators of provoking that genes of the parents are inherited by the offspring [23]. SCX follows the general structure of our crossover operators (Section 4.2.1) to pick iteratively objects from two parents and selects, at random, the ones that are directly used to create the offspring. In the same way as our operators, it additionally includes a mechanism to ensure the feasibility of the generated solutions. The comparison results are listed in Table 3.

A visual inspection of Table 3 allows one to remark that CX-BLX-α was able to provide considerably better *%D* and *%B* performance than SCX. This experiment indicates that the scheme of the proposed crossover operators may be beneficial and the improvement is significant, which gives a sense on the highly complex nature of their design. They allow our GA model to have a promising ability to solve the MGS problem that is difficult to achieve from the use of a simple crossover operator.

Table 3
SCX vs. CX-BLX-α.

Cross. Op.	*Av Rank*	*%D*	*%B*
SCX	1.922	4.6	13.6
CX-BLX-α	1.078	0.05	98

Table 4
Results with the multi-parent crossover operator.

Cross. Op.	c_{sample}	*Av Rank*	*%D*	*%B*
	1	13.600	6.4	0
	2	10.260	4.0	5.2
CX-MP $n_p = 3$	5	7.736	2.8	22.8
	10	6.746	2.3	30.8
	20	6.472	2.1	34.8
	30	7.346	2.6	25.6
	1	13.238	6.2	0.4
	2	8.008	2.9	16.8
CX-MP $n_p = 4$	5	5.858	1.8	44.4
	10	5.300	1.6	55.2
	20	8.740	4.1	32.4
	30	12.660	6.9	10
	1	14.636	9.6	0.4
	2	7.382	2.8	26.4
CX-MP $n_p = 5$	5	6.210	2.3	53.6
	10	7.714	3.5	47.6
	20	13.276	8.4	15.6
	30	15.818	11.7	3.6

Table 5
CX-MP vs. CX-BLX-α and CX-PBX-α

Cross. Op.	c_{sample}	*Av Rank*	*%D*	*%B*
CX-MP $n_p = 4$	10	1.242	0.2	94.8
CX-BLX-α	10	2.240	2.8	24
CX-PBX-α	10	2.518	3.6	13.6

Finally, we analyze the performance enrichment that might be produced by the combination of the information of more than two parents. Therefore, we investigate the performance of the GA-MGS algorithm when it applies a multi-parent extension of the CX-BLX-α operator (CX-MP-α; with $\alpha = 0.25$), in which the number of participating parents is directly specified as a crossover parameter, n_p (Section 4.2.3).

To do this, we have built different GA-MGS instances that invoke CX-MP with different values for n_p ({3, 4, 5}) and for c_{sample} ({1,2,5,10,20,30}). In these GAs, parents are selected at random (instead of by tournament selection) to avoid a harmful premature convergence caused for the application of a mean-center crossover with many parents and a fitness biases selection operator. Table 4 summarizes the results of these algorithms and Table 5 compares the best GA-MGS instances that apply CX-BLX-α, CX-PBX-α, and CX-MP-α obtained so far.

We may observe that the two best ranked algorithms in Table 4 combines the information of four parents ($n_p = 4$; and $c_{sample} = 10$ and $c_{sample} = 5$, respectively), and the third one considers five ($c_{sample} = 5$). Then, Table 5 reveals the clear advantage of the CX-MP-α operator on CX-BLX-α and CX-PBX-α. Particularly, these two-parent operators obtain a low performance in terms of *%D*, and *%B*. Therefore, we may conclude that the presented mean-centric CX-MP crossover operator is worth using within our GA template.

Please cite this article in press as: M. Lozano, et al., A genetic algorithm for the minimum generating set problem, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.07.020

5.3. GA-MGS vs. state-of-the-art algorithm for the MGS problem

In this section, we undertake a comparative analysis between GA-MGS ($N_p = 50$, $n_{sample} = 5$, and CX-MP-α with $n_p = 4$ and $c_{sample} = 10$) and the current state-of-the-art algorithm for the MGS problem, the greedy algorithm (named G-MGS) proposed by Collins et al. [1]. This method, which does not require any parameter to be set, starts with an empty solution T and its strategy is to include, at each step, the value d that let us to represent the largest number of elements $s \in S$ which have not been represented yet. To do this, the method computes the differences d from different combinations of elements $s \in S$ and $t \in T$, so the representation of some elements (R_{s_j}) are expressed in terms of the representation of others ($R_{s_j} = R_{s_i} \bigcup \{t\} \bigcup \{d\}$), and chooses the one that participates on the representation of more elements that still have not got any. The considered combinations, avoiding those that produce repetitions of T elements on the proposed representations, are 1) $s_j = s_i + d$, 2) $s_j = s_i - t + t' + d$, 3) $s_j = s_i - t_1 - t_2 - t_3 + d$, 4) $s_j = s_i - t + t'_1 + t'_2 + d$ (where $t, t_i \in R_{s_i}$ and $t', t'_i \in T - R_{s_i}$). To facilitate the comprehension of the method, we show an example in Table 6 with $S = \{4, 7, 11, 13, 17\}$.

At the beginning, T is empty and every element of S is associated to an empty representation R_{s_i}. Then, it computes the elements d that are candidates to enter in T. These elements are obtained according to the aforementioned production rules. First each element s_j produces a candidate $d = s_j$ (note that this production rule is a case of rule 1) $s_j = s_i + d$ with $s_i = 0$; as commented in [1]). Then, more candidates are computed according to rule 1 with $s_i \neq 0$. No more candidates appear with the other rules at this interation because there are not t values, so the method proceeds to count the number of representations each candidate participates in. Since $d = 4$ appears in more representations than the others, it is included in T and the process avances to the following iteration. In the second iteration, the element 4 has got a complete representation $R_4 = \{4\}$, the representations of 7 and 13 are still empty and those of 11 and 17 are expressed in terms of R_7 and R_{13}, respectively. Therefore, the method consider new candidates that may contribute to the representations of 7 and 13. The rules produced in the previous iteration for these two elements are imported. However some of these rules are now invalid because it would produce repetitions in the representations of some elements. For instance, expressing R_7 in terms of $R_4 \bigcup \{3\}$ is not valid, because that would create a repetition in $R_{11} = \{4\} \bigcup R_7 = \{4, 4, 3\}$. Below, there are new production rules according to $s_j = t' + d$, which is a case of rule 2) $s_j = s_i - t + t' + d$ with $s_i = 0$ and $t = 0$, and $s_j = s_i + t' + d$ (case of the same rule with $t = 0$). None of these instances of these rules are valid for being in conflict with the representation of other elements, and no other candidates can be computed according to the other production rules. In this case, all the candidates appear with the same frequency and one is chosen randomly. Finally, assume we inserted $d = 6$ into T and representations are updated accordingly (R_{17}). The third iteration applies the same procedure as previous one and the only candidate is now $d = 7$, which, after being inserted into T, completes the solution (T={4,6,7}) and the representations ($R_4 = \{4\}$, $R_7 = \{7\}$, $R_{11} = \{4, 7\}$, $R_{13} = \{6, 7\}$, and $R_{17} = \{4, 6, 7\}$).

Additionally, we include in this study the MS-RG-MGS algorithm (Fig. 7) and an economized version of G-MGS (G^e-MGS; limited and faster), which only considers combinations 1) $s_j = s_i + d$ and 2) $s_j = s_i - t + t' + d$. All the algorithms were implemented and run under the same computational conditions (machine, programming language, and compiler) to enable a fair comparison between them.

The analysis has been divided into results on small instances (Section 5.3.1), large instances (Section 5.3.2), and very large instances (Section 5.3.3).

5.3.1. Results on small instances

The first comparison between these algorithms is carried out by considering different sets with 17 small *Type L* instances of sizes $n = 20, 25, \ldots, 100$. Each set is distinguished by the values of $|T|$ ({10, 20}) and for S_{max} ({65,536, 1,048,576, 16,777,216}). The algorithms were run once on each instance. The cutoff time for each execution of GA-MGS and MS-RG-MGS was set to $n/10$ s. Table 7 reports the average objective function value (Av) and the $\%B$ measure achieved by all the algorithms, and also, the average CPU time in seconds required by G-MGS and G^e-MGS to construct their solution ($\bar{t}$) ($\bar{t} = 6$ for all the instance sets for the case of MS-RG-MGS and GA-MGS, which is the result from the averaged summation $1/17 \cdot \sum_n n/10$).

Examining the data in Table 7, we may conclude the following:

- Attending to the values of Av and $\bar{t}$ reported by G-MGS and G^e-MGS, instances with $|T| = 20$ seem to be the hardest ones, because the results are much larger than the known upper bounds of the optimal solutions ($min(|T|, log_2(S_{max}))$), and G-MGS needs 441.4 s to construct a single solution. This can be explained by the fact that there are much more possible combinations of 20 elements than 10. In any case, the results of all the algorithms when $|T| = 10$ are clearly distant from the upper bounds for the optimal solution, which exhibits the hardness of the MGS problem, even with small instances with low *complexity* (all the values of S could had been generated with only ten t_i values).
- The search difficulties for G-MGS and G^e-MGS increase as S_{max} becomes greater for $|T| = 20$ instances. However, in this case, GA-MGS (and MS-RG-MGS) performs significantly better than these greedy algorithms, returning values very close to the upper bound $|T|$ or even better ($|T| = 20$ and $S_{max} = 65, 536$ or 1,048,576), and consuming much less time (6 s as computed above).
- It is interesting to observe that the greedy algorithm G-MGS obtains relatively good results when $|T| = 10$, regardless the value of S_{max} (they are the best results for $|T| = 10$ and $S_{max} = 1{,}048{,}576$ or 16,777,216). This suggests that the heuristic combinations of values used by this method (Section 5.3) allow it to mildly approximate the real complexity of the problem when that is less or around 16, i.e., when all the elements in S can be generated with only 16 t_i values (when $|T| = 20$ and $S_{max} = 65, 536$, the upper bound is still $16 = log_2(65, 536)$). However, it finds very difficult to deal with instances of higher complexity (those where more than 16 t_i values are needed to generate all the elements in S), i.e., its heuristics are unable to approximate the real complexity of the problem. Consequently, we observe that G^e-MGS, which considers less heuristic combinations of values, obtains worse results and find more difficulties earlier with lower complexities (though its running times are considerable smaller).
- Regarding MS-RG-MGS and GA-MGS, we observe that their results seem to be determined only by the value of S_{max}, not $|T|$. Even when the problem instances were generated by excessive T sets, because $log_2(S_{max}) < |T|$, they find that smaller T sets are sufficient. On the contrary, when $|T| < log_2(S_{max})$, the large space of values ($\{1, \ldots, S_{max}\}$) becomes a hard challenge that hinders them to find the real simpler complexity of the problem ($|T|$). This is clearly due to the fact that operations in these two algorithms (initialization, crossover, etc.) depend on the range of the values, and they do not look for value combinations as G-MGS and G^e-MGS do.

We may then summarize previous results in the following conclusion, G-MGS performs considerably well for small problem instances when the complexity of the problem is inferior or around 16, otherwise, GA-MGS is a safer alternative.

Please cite this article in press as: M. Lozano, et al., A genetic algorithm for the minimum generating set problem, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.07.020

Table 6
G-MGS, example of execution for $S = \{4, 7, 11, 13, 17\}$.

First iteration			Second iteration				Third iteration		
T		$\{\}$	T	$\{4\}$			T		$\{4,6\}$
	4	$R_4 = ?$		4	$R_4 = \{4\}$			4	$R_4 = \{4\}$
	7	$R_7 = ?$		7	$R_7 = ?$			7	$R_7 = ?$
S	11	$R_{11} = ?$	S	11	$R_{11} = \{4\} \cup R_7$		S	11	$R_{11} = \{4\} \cup R_7$
	13	$R_{13} = ?$		13	$R_{13} = ?$			13	$R_{13} = \{6\} \cup R_7$
	17	$R_{17} = ?$		17	$R_{17} = \{4\} \cup R_{13}$			17	$R_{17} = \{4, 7\} \cup R_7$
	$s_j = d$			$s_j = d$				$s_j = d$	
	4	$R_4 = \{4\}$		7	$R_7 = \{7\}$			7	$R_7 = \{7\}$
	7	$R_7 = \{7\}$		13	$R_{13} = \{13\}$			$s_j = s_i + d$	
	11	$R_{11} = \{11\}$		$s_j = s_i + d$					No new rules for R_7
	13	$R_{13} = \{13\}$		3	$R_7 = R_4 \cup \{3\}$	Conflict R_{11}		$s_j = t' + d$	
	17	$R_{17} = \{17\}$		9	$R_{13} = R_4 \cup \{9\}$	Conflict R_{17}		1	$R_7 = \{6\} \cup \{1\}$ Conflict R_{13}
	$s_j = s_i + d$			6	$R_{13} = R_7 \cup \{6\}$			$s_j = s_i + t' + d$	
	3	$R_7 = R_4 \cup \{3\}$		2	$R_{13} = R_{11} \cup \{2\}$	Conflict R_{17}			No new rules for R_7
	7	$R_{11} = R_4 \cup \{7\}$		$s_j = t' + d$				$s_j = s_i - t + t' + d$	
	9	$R_{13} = R_4 \cup \{9\}$		3	$R_7 = \{4\} \cup \{3\}$	Conflict R_{11}			No new rules for R_7
	13	$R_{17} = R_4 \cup \{13\}$		9	$R_{13} = \{4\} \cup \{9\}$	Conflict R_{17}		Count	
	4	$R_{11} = R_7 \cup \{4\}$		$s_j = s_i + t' + d$				7	$\lvert\{R_7\}\rvert = 1$
	6	$R_{13} = R_7 \cup \{6\}$		5	$R_{13} = R_4 \cup \{4\} \cup \{5\}$	$\{4\}$ repeated			
d	10	$R_{17} = R_7 \cup \{10\}$	d	2	$R_{13} = R_7 \cup \{4\} \cup \{2\}$	Conflict R_{17}	d		
	2	$R_{13} = R_{11} \cup \{2\}$		Count					
	6	$R_{17} = R_{11} \cup \{6\}$		6	$\lvert\{R_{13}\}\rvert = 1$				
	4	$R_{17} = R_{13} \cup \{4\}$		7	$\lvert\{R_7\}\rvert = 1$				
	Count			13	$\lvert\{R_{13}\}\rvert = 1$				
	4	$\lvert\{R_4, R_{11}, R_{17}\}\rvert = 3$							
	7	$\lvert\{R_7, R_{11}\}\rvert = 2$							
	13	$\lvert\{R_{13}, R_{17}\}\rvert = 2$							
	6	$\lvert\{R_{13}, R_{17}\}\rvert = 2$							
	11	$\lvert\{R_{11}\}\rvert = 1$							
	3	$\lvert\{R_7\}\rvert = 1$							
	9	$\lvert\{R_{13}\}\rvert = 1$							
	10	$\lvert\{R_{17}\}\rvert = 1$							
	2	$\lvert\{R_{13}\}\rvert = 1$							

Table 7
Results on small *Type L* instances.

Instances			G-MGS			G^e-MGS			MS-RG-MGS		GA-MGS	
$\lvert T\rvert$	S_{max}	$log_2(S_{max})$	Av	$\%B$	$\bar{t}$	Av	$\%B$	$\bar{t}$	Av	$\%B$	Av	$\%B$
	65,536	16	16.4	29.4	2.2	17.1	17.6	1	16.5	11.8	15.5	70.6
10	1,048,576	20	15.8	82.4	2.2	17.2	29.4	1.2	20.9	5.9	19.5	5.9
	16,777,216	24	15.4	88.2	2.1	17.4	23.5	1.2	25.8	5.9	23.4	5.9
	65,536	16	21.5	11.8	19.7	25.7	5.9	8.2	16.6	29.4	15.8	100
20	1,048,576	20	29.2	5.9	87.7	34.8	5.9	26.3	21.0	11.8	19.6	100
	16,777,216	24	36.2	5.9	441.4	43.1	5.9	33.4	25.9	5.9	23.5	100

5.3.2. Results on large instances

The aim of the next experiment is to compare the algorithms on larger instances. Specifically, we have generated 6 sets of *Type L* instances and one of *Type U* instances with 110 cases with sizes that range from $n = 50$ to $n = 10,000$. The former resulted from the combination of $|T| = \{15, 23\}$ and $S_{max} = \{65,536, 1,048,576, 16,777,216\}$.

Due to the computational requirements of G-MGS and G^e-MGS shown in the previous experiment, we have imposed a maximum running time of one hour for these experiments, and presented the results separately. Table 8 shows the size of the largest instances for which the greedy approaches were able to provide a solution within one hour, along with the *Av* measure for the solved instances (those for which they provided a solution). On the other hand, Table 9 reports the *Av* and *%B* measures of GA-MGS and MS-RG-MGS on the 7 sets of large instances.

Looking at Tables 8 and 9, we may point out the following facts:

- G-MGS and G^e-MGS generated solutions for very few instances in one hour (Table 8). Even more, they were only able to provide a solution for the smallest *Type U* instances ($n = 50$). On the contrary, our algorithms generated solutions for all the instances consuming at most 1000 s ($\frac{n}{10}$, with $n = 10,000$).
- The averaged results (*Av*) of GA-MGS and MS-RG-MGS are better than those of the greedy approaches, even though they are computed over much larger instances.
- Our GA manages to generate much better results than MS-RG-MGS (Table 9). Differences are larger as S_{max} increases (notice that *Type U* instances have the largest S_{max} value). This performance

Table 8
Results of G-MGS and G^e-MGS on large instances.

Instances		G-MGS		G^e-MGS	
$\lvert T\rvert$	S_{max}	$max\ n$	Av	$max\ n$	Av
	65,536	250	28	350	31.16
15	1,048,576	100	30	150	38.6
	16,777,216	100	36	100	50.5
	65,536	200	27.5	250	33.4
23	1,048,576	100	31.5	150	44
	16,777,216	100	41	100	51.5
Type U Inst.		50	49	50	51

Please cite this article in press as: M. Lozano, et al., A genetic algorithm for the minimum generating set problem, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.07.020

Table 9
Results of MS-RG-MGS and GA-MGS on large instances.

Instances		MS-RG-MGS		GA-MGS	
$\|T\|$	S_{max}	*Av*	*%B*	*Av*	*%B*
15	65,536	18.1	14.54	17.2	94.54
	1,048,576	23.5	19.09	21.8	91.81
	16,777,216	30.6	0	25.9	100
23	65,536	18.0	9.09	17.1	98.18
	1,048,576	23.4	5.45	21.3	95.45
	16,777,216	30.9	0	25.9	100
Type U Inst.		44.8	0	36.8	100

Table 10
GA-MGS vs. MS-RG-MGS on very large *Type U* instances.

Inst. Size	GA-MGS	MS-RG-MGS
10,000	33	44
20,000	34	46
30,000	36	45
40,000	35	45
50,000	38	46
60,000	38	47
70,000	39	49
80,000	41	48
90,000	41	48
100,000	39	50

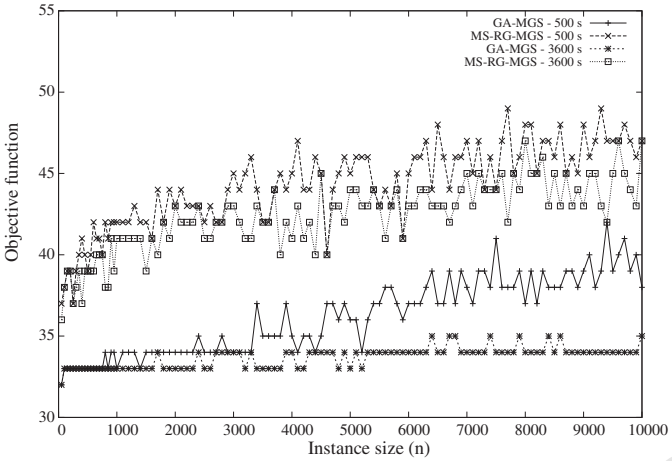

Fig. 8. GA-MGS and MS-RG-MGS on large *Type U* instances.

difference, also present in the previous experiment with small instances, proves that the operators included in our GA, particularly the CX-MP-α crossover operator and the fact of maintaining good solutions in the population, are effectively working better than the random search developed by MS-RG-MGS, and lead the algorithm to better solutions.

To complement the above experimental study, we now analyze the detailed results obtained by GA-MGS and MS-RG-MGS on the complex large *Type U* instances after 500 and 3600 s. Fig. 8 plots the objective function value for the best solutions found by the algorithms with each time limit according to the size of the problem instance. These results allow us to make the following observations:

- GA-MGS with 3600 s clearly obtains always the best results. Interestingly, the results are very similar regardless the size of the problem instance. Remember that the upper bound for this problem instances is $log_2(S_{max}) = log_2(2, 147, 483, 647) = 31$. Therefore, GA-MGS seems to be able to approach the optimal solution closely when it is given sufficient running time.
- The solutions obtained by the GA after 500 s are already clearly better in comparison with those obtained by MS-RG-MGS after 3600 s.
- Given the performance difference gained by GA-MGS between 500 and 3600 s as n increases, we may conclude that the restart mechanism is allowing it to escape from the stagnation in local optima, letting the rest of the operators to progress further on the quest for better solutions. Particularly, this performance improvement is much more reduced in MS-RG-MGS. In this case, since MS-RG-MGS is a memoryless technique and just generates almost random solutions at each iteration, its little performance improvement is only due to randomness and more running time.

Therefore, we may conclude that GA-MGS is really effective on large instances, for which the greedy approaches are not able to produce solutions within reasonable running time.

5.3.3. Results on very large instances

A final computational experiment has been conducted for testing the performance of our GA on very large test cases of size up to 100,000 elements. Specifically, we have generated 10 *Type U* instances with sizes $n = 10,000, 20,000, \ldots, 100,000$. Table 10 provides, for each instance, the value of the best solution found during a 5 hour run of GA-MGS and MS-RG-MGS. As Table 10 shows, the GA again is definitely superior to the multi-start algorithm. Though the problem instances are more difficult than those considered so far, GA-MGS was able to find good solutions at a relatively moderate computation time.

6. Conclusions

In this paper, a new formulation of the MGS problem was developed to conceive a GA that revolves around RG-MGS, a new randomized greedy method being able to construct, in an intelligent, diversified feasible solutions for this highly constrained problem. This heuristic procedure is employed as initialization mechanism for the GA and it is the basic pillar on which the design of the crossover operators is supported. A design that was completed precisely through the knowledge acquired in the research field of GAs for continuous optimization problems [30,38,39,29]. The GA also integrates a restart operator to ensure a reliable evolution toward promising areas throughout the entire search. The proposal has proven to be a very high performing algorithm for the MGS problem, showing it to be very competitive with respect to state-of-the-art algorithms. Specifically, the empirical study reveals a clear superiority when tackling hard and large instances.

The ability of the proposed GA to yield superior outcomes along with the simplicity and flexibility of this approach, allows us to conclude that this metaheuristic arises as a tool of choice to face this problem. Moreover, it invites further consideration to explore other forms of evolutionary algorithms, such as the memetic algorithms [40,39], which apply a local search method to members of the GA population after crossover and mutation operations, with the aim of exploiting the best search regions identified by the global sampling done by the GA.

Acknowledgement

This work was supported by the Research Projects TIN2012-37930-C02-01 and TIN2012-35632-C02.

References

[1] M.J. Collins, D. Kempe, J. Saia, M. Young, Nonnegative integral subset representations of integer sets, Inf. Process. Lett. 101 (3) (2007) 129–133.

[2] N. Bansal, D. Coppersmith, B. Schieber, Minimizing setup and beam-on times in radiation therapy, in: J. Díaz, K. Jansen, J.D. Rolim, U. Zwick (Eds.), Approx., Randomization, and Combin. Optim. Algorithms and Technol., Vol. 4110 of Lecture Notes in Computer Science, 2006, pp. 27–38.
[3] D. Chen, X. Hu, S. Luan, S. Naqvi, C. Wang, C. Yu, Generalized geometric approaches for leaf sequencing problems in radiation therapy, Int. J. Comput. Geomet. Appl. 16 (2–3) (2006) 175–204.
[4] T. Kalinowski, The algorithmic complexity of the minimization of the number of segments in multileaf collimator field segmentation, Univ. Fachbereich Mathematik, 2004 (Master's thesis).
[5] M. Develin, On optimal subset representations of integer sets, J. Number Theory 89 (2) (2001) 212–221.
[6] D.P. Moulton, Representing powers of numbers as subset sums of small sets, J. Number Theory 89 (2) (2001) 193–211.
[7] I. Fagnot, G. Fertin, S. Vialette, On finding small 2-generating sets, in: H.Q. Ngo (Ed.), Comput. and Combin., Vol. 5609 of Lecture Notes in Computer Science, 2009, pp. 378–387.
[8] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Longman Publishing Co, 1989.
[9] J. Holland, Adaptation in Natural and Artificial Systems, The University of Michigan Press, 1975.
[10] M. Randall, A. Lewis, Modifications and additions to ant colony optimisation to solve the set partitioning problem, in: 6th IEEE International Conference on e-Science Workshops, 2010, pp. 110–116.
[11] V. Maniezzo, M. Milandri, An ant-based framework for very strongly constrained problems, in: in: Ant Algorithms 2002, Vol. 2463 of Lecture Notes in Computer Science, 2002, pp. 222–227.
[12] F. Herrera, M. Lozano, J. Verdegay, Tackling real-coded genetic algorithms: operators and tools for behavioral analysis, Artif. Intell. Rev. 12 (4) (1998) 265–319.
[13] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, 2004.
[14] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, Inc, 1990.
[15] C. García-Martínez, F. Rodriguez, M. Lozano, Tabu-enhanced iterated greedy algorithm: a case study in the quadratic multiple knapsack problem, Eur. J. Oper. Res. 232 (3) (2014) 454–463.
[16] C. García-Martínez, F. Glover, F.J. Rodríguez, M. Lozano, R. Martí, Strategic oscillation for the quadratic multiple knapsack problem, Comput. Opt. Appl. 58 (1) (2014) 161–185.
[17] A. Hiley, B. Julstrom, The quadratic multiple knapsack problem and three heuristics approaches to it, in: Proc. of the Genetic and Evol. Comput. Conf., 1992, pp. 547–552.
[18] U. Aickelin, K.A. Dowsland, An indirect genetic algorithm for a nurse-scheduling problem, Comput. Oper. Res. 31 (5) (2004) 761–778.
[19] P. Borisovsky, A. Dolgui, A. Eremeev, Genetic algorithms for a supply management problem: MIP-recombination vs greedy decoder, Eur. J. Oper. Res. 195 (3) (2009) 770–779.
[20] P.A. Borisovsky, X. Delorme, A. Dolgui, Genetic algorithm for balancing reconfigurable machining lines, Comput. Oper. Res. 66 (3) (2013) 541–547.
[21] M. Resende, R. Martí, M. Gallego, A. Duarte, GRASP and path relinking for the max–min diversity problem, Comput. Oper. Res. 37 (3) (2010) 498–508.
[22] M. Lozano, F. Herrera, J.R. Cano, Replacement strategies to preserve useful diversity in steady-state genetic algorithms, Inf. Sci. 178 (23) (2008) 4421–4433.
[23] G. Sywerda, Uniform crossover in genetic algorithms, in: J.D. Schaffer (Ed.), Proc. of the Int. Conf. on Genetic Algorithms, 1989, pp. 2–9.
[24] D. Whitley, The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best, in: Proc. of the Genetic and Evol. Comput. Conf., 1989, pp. 116–121.
[25] D.E. Goldberg, K. Deb, A Comparative study of Selection Schemes Used in Genetic Algorithms, Vol. 1 of Foundations of Genetic Algorithms, Morgan Kaufmann, 1991, pp. 69–93.
[26] K.A. Jong, W.M. Spears, A formal analysis of the role of multi-point crossover in genetic algorithms, Ann. Math. Artif. Intell. 5 (1) (1992) 1–26.
[27] F. Herrera, M. Lozano, A. Sánchez, Hybrid crossover operators for real-coded genetic algorithms: an experimental study, Soft Comput. 9 (4) (2005) 280–298.
[28] G.E. Liepins, M.D. Vose, Characterizing crossover in genetic algorithms, Ann. Math. Artif. Intell. 5 (1) (1992) 123–134.
[29] H. Someya, Striking a mean- and parent-centric balance in real-valued crossover operators, IEEE Trans. Evol. Comput. 17 (6) (2013) 737–754.
[30] K. Deb, A. Anand, D. Joshi, A computationally efficient evolutionary algorithm for real-parameter optimization, Evol. Comput. 10 (4) (2002) 371–395.
[31] L.J. Eshelman, J.D. Schaffer, Real-coded genetic algorithms and interval-schemata, in: L.D. Whitley (Ed.), Found. of Genetic Algorithms 2, Morgan Kaufmann, 1992, pp. 187–202.
[32] C. García-Martínez, M. Lozano, F. Herrera, D. Molina, A. Sánchez, Global and local real-coded genetic algorithms based on parent-centric crossover operators, Eur. J. Oper. Res. 185 (3) (2008) 1088–1113.
[33] I. Ono, S. Kobayashi, A real-coded genetic algorithm for function optimization using unimodal normal distribution crossover, in: Proc. of the Genetic and Evol. Comput. Conf., 1997, pp. 246–253.
[34] S. Tsutsui, M. Yamamura, T. Higuchi, Multi-parent recombination with simplex crossover in real coded genetic algorithms, in: Proc. of the Genetic and Evol. Comput. Conf., 1999, pp. 657–664.
[35] M. Črepinšek, S.-H. Liu, L. Mernik, M. Mernik, Is a comparison of results meaningful from the inexact replications of computational experiments? Soft Comput. 20 (1) (2014) 223–235.
[36] R. Martí, M. Resende, C. Ribeiro, Multi-start methods for combinatorial optimization, Eur. J. Oper. Res. 226 (1) (2013) 1–8.
[37] J.P. Hart, A.W. Shogan, Semi-greedy heuristics: an empirical study, Oper. Res. Lett. 6 (1987) 107–114.
[38] F. Herrera, M. Lozano, Gradual distributed real-coded genetic algorithms, IEEE Trans. Evol. Comput. 4 (1) (2000) 43–63.
[39] D. Molina, M. Lozano, C. García-Martínez, F. Herrera, Memetic algorithms for continuous optimization based on local search chains, Evol. Comput. 18 (1) (2010) 27–63.
[40] N. Krasnogor, J. Smith, A tutorial for competent memetic algorithms: model, taxonomy and design issues, IEEE Trans. Evol. Comput. 9 (5) (2005) 474–488.

Please cite this article in press as: M. Lozano, et al., A genetic algorithm for the minimum generating set problem, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.07.020