A BIASED RANDOM-KEY GENETIC ALGORITHM FOR THE CAPACITATED MINIMUM SPANNING TREE PROBLEM

E. RUIZ, M. ALBAREDA-SAMBOLA, E. FERNÁNDEZ, AND M.G.C. RESENDE

Abstract. This paper focuses on the capacitated minimum spanning tree (CMST) problem. Given a central processor and a set of remote terminals with specified demands for traffic that must flow between the central processor and terminals, the goal is to design a minimum cost network to carry this demand. Potential links exist between any pair of terminals and between the central processor and the terminals. Each potential link can be included in the design at a given cost. The CMST problem is to design a minimum-cost network connecting the terminals with the central processor so that the flow on any arc of the network is at most Q. A biased random-key genetic algorithm (BRKGA) is a metaheuristic for combinatorial optimization which evolves a population of random vectors that encode solutions to the combinatorial optimization problem. This paper explores several solution encodings as well as different strategies for some steps of the algorithm and finally proposes a BRKGA heuristic for the CMST problem. Computational experiments are presented showing the effectiveness of the approach: Seven new best-known solutions are presented for the set of benchmark instances used in the experiments.

1. Introduction

Minimal spanning trees are among the most studied structures in combinatorial optimization. From a theoretical point of view, the interest of studying minimum-cost spanning trees (MSTs) lies in their particularity as mathematical objects. Due to its matroid structure, a MST can be found in polynomial time with greedy algorithms, like the ones of Prim (1957) or Kruskal (1956). From a practical point of view, the connectivity property of MSTs is very useful in multiple applications. For this reason, along with the traveling salesman problem (TSP), MSTs problems are considered to lie in the core of network systems design and of a wide variety of scheduling and routing applications.

Often, TSPs and MST problems take into account additional restrictions, being capacity constraints among the most frequent ones. The problem that consists of finding an MST that satisfies additional capacity constraints is called the *Capacitated Minimum Spanning Tree* (CMST) problem. From now on, we will refer to this problem simply as the CMST. The study of the CMST is of interest because the mere addition of capacity constraints transforms the MST into an NP-hard problem. The CMST often arises in telecommunication network design, but it also has applications in distribution, transportation, and logistics. For example, it is related to vehicle routing problems, due to the influence that MSTs have in constructive heuristics: the TSP heuristic of Christofides (1976) is based on spanning

Key words and phrases. Optimization, combinatorial optimization, networks, graphs, trees, spanning trees, capacitated minimum spanning tree, heuristics, biased random-key genetic algorithm.

trees and similar heuristics were developed for arc routing problems in Frederickson et al. (1979) and Frederickson (1979). Because vehicle routing problems usually consider capacity constraints, the design of more effective or efficient algorithms for the CMST can also play a role in the design of efficient methods to obtain feasible solutions for the capacitated vehicle routing problem. Amberg et al. (2000) showed that it is possible to transform a multicenter capacitated arc routing problem (M-CARP) into a capacitated minimum spanning tree with additional arc constraints.

Many authors have proposed integer programming formulations for the MST and its extensions (Araque et al., 1990; Gouveia, 1993; 1995; 1996; Hall, 1996; Gouveia and Martins, 2000; Gouveia and Hall, 2002; Gouveia and Lopes, 2005; Gouveia and Martins, 2005; Ruiz, 2013). For the particular case of the CMST, lower bounds can be found in Gavish (1982; 1983; 1985), Gouveia (1995), and Uchoa et al. (2008). To date, the most successful exact method is the one of Uchoa et al. (2008). However, due to its NP-hard nature (see Papadimitriou, 1978) the solution of the CMST with exact methods is usually very time consuming and even impossible, already for moderate size instances. This explains why heuristic methods, based on the greedy paradigm (Esau and Williams, 1966; Chow and Kershenbaum, 1974; Gavish and Altinkemer, 1986), neighborhood exploration (Amberg et al., 1996; Sharaiha et al., 1997; Ahuja et al., 2001; 2003; Souza et al., 2003), or dynamic programming (Gouveia and Paixão, 1991), have been widely used. More recent heuristics include the second-order algorithm of Martins (2007), in which subproblems of the original problem including a set of constraints are solved with the Essau Williams heuristic, the heuristic of Rego et al. (2010), which projects dual solutions into the primal feasible space and obtains primal feasible solutions by simple tabu searches and metaheuristics such as the ant colony algorithm by Reimann and Laumanns (2006) or the filter-and-fan algorithm by Rego and Mathew (2011).

In this paper we propose a biased random-key genetic algorithm (BRKGA) for the CMST, which stems from the Ph.D. Thesis of E. Ruiz (2013). BRKGA is a metaheuristic for combinatorial optimization first proposed in Gonçalves and Resende (2011). A BRKGA evolves a population of random vectors that encode solutions to the combinatorial optimization problem. BRKGA heuristics have been used to tackle a wide range of problems, such as traffic congestion (Buriol et al., 2010), telecommunications (Buriol et al., 2005; Noronha et al., 2011; Reis et al., 2011; Resende et al., 2012), container loading problems (Gonçalves and Resende, 2012), scheduling (Mendes et al., 2009; Gonçalves et al., 2011), and arc routing (Martinez et al., 2011). In many of these applications, BRKGA was shown to produce better solutions than other heuristics. It is important to note that capacity constraints are present in many of these applications, just as they are in the CMST. These two observations were the main motivations for exploring the suitability of the BRKGA for the CMST.

The main contributions of this paper are:

- We propose a new BRKGA for the CMST. We test our heuristic on 126 well-known benchmark instances. We are not aware of any other exact or heuristic algorithm for the CMST, tested on such an extensive set of instances. Using a fixed set of parameters values, our BRKGA consistently produces good results with quite modest computing requirements, independently of the type of test instance. The numerical results of our extensive computational experiments indicate that our BRKGA outperforms other heuristics for the CMST both in terms of average deviations from best-known solutions and number of best-known solutions found.

- We study various potential ingredients for the BRKGA and analyze their individual contribution to the overall algorithm. Two alternative decoders are proposed to identify the most effective way of transmitting genetic information from parents to offsprings for the CMST. We are not aware of any code in the literature that represents spanning trees and takes into account capacity constraints. Both decoders are enhanced with an improvement phase which incorporates a local search involving four different neighborhoods. Strategic oscillation is applied as well. In all cases, alternative strategies are considered and compared.
- Our BRKGA is able to improve the best-known solution for seven out of the 25 instances in our benchmark set with unknown optimal solution. This is remarkable taking into account that these are very well-known benchmark instances, which have been much-used by different authors.

The paper is organized as follows. In Section 2, we describe the CMST. In Section 3, we briefly recall biased random-key genetic algorithms. In Section 4, we describe the two decoders as well as the improvement phase that we have incorporated into our BRKGA for the CMST. Different implementation alternatives and reinforcements for the decoders and search strategies are successively presented. The individual impact of each of the proposed ingredients is analyzed in the first part of Section 5. The section concludes with the computational results of the overall proposed BRKGA. The paper ends in Section 6 with some concluding remarks.

2. Notation and problem description

Let $G = (V, E)$ be a given simple graph, with $V = \{0, 1, \dots, n\}$, where 0 is a central processor and $V^+ = \{1, \dots, n\} \subset V$ is a set of n terminals. Associated with each edge $e = (i, j) \in E$ there is a cost $c_{ij} > 0$. Each terminal i, $i = 1, \dots, n$, has an associated demand $w_i \geq 0$.

Given a spanning tree $T \subset E$ of G, rooted at 0, the *cost* of T is naturally defined as $c(T) = \sum_{e \in T} c_e$. We denote by *subroot* of T any vertex directly connected to the root vertex 0. A *subtree* of T rooted at vertex $i \in V$ is denoted by T_i. A subtree $T_i \subseteq T$ where i is a subroot of T is called *s-tree*. We use the notation $V(T_i) \subset V$ to denote the set of terminals that are part of subtree T_i and $w(T_i) = \sum_{j \in V(T_i)} w_j$ to denote the *demand* of subtree T_i.

Definition 1. *Given a graph $G = (V, E)$ with a distinguished vertex $0 \in V$, a demand w_i associated with each terminal vertex $i \in V^+$, a nonnegative cost c_{ij} associated with each edge $(i, j) \in E$, and a capacity $Q > 0$, the CMST is to find a minimum-cost spanning tree of G, rooted at 0, such that the demand of no s-tree exceeds Q.*

3. Biased random-key genetic algorithms

Genetic algorithms with random keys, or *random-key genetic algorithms* (RKGA), were introduced by Bean (1994) for solving combinatorial optimization problems involving sequencing and other optimization problems where the solution can be represented as a permutation vector. In a genetic algorithm, solutions are often referred to as *individuals* or *chromosomes*. In a RKGA individuals are represented as vectors of n random keys, i.e. n real numbers independently generated at random in the uniform interval $[0,1)$. Parameter n is problem dependent. A *decoder* is a deterministic algorithm that takes as input a vector of random keys and produces from it a feasible solution for which an objective value or fitness can be computed.

A RKGA evolves a population of random-key vectors over a number of iterations, called *generations*. The initial population is made up of p vectors of n random keys. In generation k the fitness of each individual is computed by the decoder. The population is then partitioned into two groups of individuals: a small group of $p_e < p/2$ *elite* individuals, i.e. those with the best fitness values, and the remaining set of $p - p_e$ *non-elite* individuals. To evolve the population, a new generation of individuals must be produced. This is done in three steps.

In step 1, all elite individuals of the population of generation k are copied without modification to the population of generation $k+1$. RKGAs implement mutation by introducing *mutants* into the population. A mutant is simply a vector of n random keys generated as the individuals of the initial population. The role of mutants is to inject noise into the population with the goal of avoiding getting stuck at a locally optimal solution. In step 2, p_m mutants are introduced into the population of generation $k+1$. With the p_e elite individuals and the p_m mutants accounted for in population $k+1$, $p - p_e - p_m$ additional individuals need to be produced to complete the p individuals that make up the new population. This is done in step 3 by producing $p - p_e - p_m$ offspring through the process of mating or crossover. Bean (1994) selects two parents at random from the entire population and combines them using *parameterized uniform crossover* (Spears and DeJong, 1991).

A *biased random-key genetic algorithm,* or BRKGA (Gonçalves and Resende, 2011; Gonçalves et al., 2014), differs from a RKGA in the way parents are selected for mating and what role each parent plays in crossover. Unlike in a RKGA, where parents are selected at random from the entire population, in a BRKGA each offspring is generated combining one individual selected at random from the elite partition of the population and another from the non-elite partition. As in a RKGA, repetition in the selection of mates is allowed and therefore an individual can produce more than one offspring in the same generation. Let ρ_e be the probability that an offspring inherits the random key of its elite parent. In order to try to keep its good quality, this probability is typically taken as $\boldsymbol{\rho_e > 0.5}$. If n is the number of random keys in an individual, then for $i = 1, \ldots, n$, the i-th component $o[i]$ of the offspring vector o takes on the value of the i-th component $a[i]$ of the elite parent a with probability ρ_e and the value of the i-th component $b[i]$ of the non-elite parent b with probability $1 - \rho_e$.

When the next population is complete, i.e. when it has p individuals, fitness values are computed by the decoder for all of the newly created random-key vectors and the population is partitioned into elite and non-elite individuals to start a new generation.

Algorithm 1 shows pseudo-code for a BRKGA. In line 1, the initial population $\mathcal{P}$ of p random-key vectors of size n is generated. Evolution takes place in the loop from line 2 to line 19. In line 3, each newly generated individual is decoded and its fitness computed. In the first generation all individuals are decoded. In subsequent generations, only mutants and offspring need to be decoded. In line 4, the p_e most fit individuals are placed in the elite set $\mathcal{P}_e$ while the remaining $p - p_e$ individuals are placed in the non-elite set $\mathcal{P}_{\bar{e}}$. In line 5, $\mathcal{P}^+$, the population of the next generation is initialized with $\mathcal{P}_e$, the elite individuals of the current population.

```
procedure BRKGA
   Input: $n, p, p_e, p_m, \rho_e$
   Output: Best individual $\mathcal{X}^*$
1  Generate population $\mathcal{P}$ with $p$ individuals, each having $n$ keys
   generated at random in the interval $[0, 1)$;
2  while stopping criterion is not satisfied do
3      Apply decoder to evaluate fitness of each new individual in $\mathcal{P}$;
4      Partition $\mathcal{P}$ into sets $\mathcal{P}_e$ with $p_e$ most fit individuals and $\mathcal{P}_{\bar{e}}$
       with $p - p_e$ remaining individuals;
5      Initialize next population: $\mathcal{P}^+ \leftarrow \mathcal{P}_e$;
6      Generate set $\mathcal{P}_m$ with $p_m$ mutants, each having $n$ random keys
       generated uniformly at random in interval $[0, 1)$ ;
7      Add mutants to next population: $\mathcal{P}^+ \leftarrow \mathcal{P}^+ \cup \mathcal{P}_m$;
8      for $i \leftarrow 1$ to $p - p_e - p_m$ do
9          Select parent $a$ at random from $\mathcal{P}_e$;
10         Select parent $b$ at random from $\mathcal{P}_{\bar{e}}$;
11         for $j \leftarrow 1$ to $n$ do
12             Toss biased coin having probability $\rho_e > 0.5$ of heads;
13             if Toss is heads then $o[j] \leftarrow a[j]$;
14             else $o[j] \leftarrow b[j]$;
15         end
16         Add offspring $o$ to population: $\mathcal{P}^+ \leftarrow \mathcal{P}^+ \cup \{o\}$;
17     end
18     $\mathcal{P} \leftarrow \mathcal{P}^+$;
19 end
20 return $\mathcal{X}^* \leftarrow \mathbf{argmin}\{f(\mathcal{X}) \mid \mathcal{X} \in \mathcal{P}\}$
```

Algorithm 1: Biased random-key genetic algorithm.

In line 6 the p_m mutants of set $\mathcal{P}_m$ are generated and, in line 7, they are added to the population of the next generation. Crossover, or mating, takes place in the loop from line 8 to line 17. A total of $p - p_e - p_m$ offspring are generated. In lines 9 and 10, parents a and b are selected at random from, respectively, the elite set $\mathcal{P}_e$ and non-elite set $\mathcal{P}_{\bar{e}}$. In the loop from line 11 to line 17, uniform parameterized crossover takes place, combining elite parent a and non-elite parent b to produce offspring o. The offspring is added to the next population in line 16. In line 18, the generation counter is incremented by moving $\mathcal{P}^+$ to $\mathcal{P}$. In line 20, $\mathcal{X}^*$, the best solution in population $\mathcal{P}$ is returned.

A BRKGA explores the solution space of the combinatorial optimization problem indirectly by searching over the continuous n-dimensional hypercube, using the decoder to map solutions in the hypercube to solutions in the solution space of the combinatorial optimization problem where the fitness is evaluated. Therefore, a BRKGA has a problem-independent component, where the random-key vectors are evolved, and a problem-dependent component, the decoder. Furthermore, BRKGA can easily take advantage of parallel computing environments, since step 3 of the pseudo-code in Algorithm 1 can be done in parallel with each thread decoding a different vector of random keys. An object-oriented application programming interface (API) for the BRKGA framework was proposed in Toso and Resende (2014). This cross-platform library automatically handles a large portion of problem-independent modules that are part of the framework, including population management and evolutionary dynamics. The user only needs to implement the problem-dependent decoder. The implementation is written in the C++ programming language and can benefit from shared-memory parallelism when available. We make use of this API in our proposed implementation of the BRKGA for CMST.

A BRKGA only requires the specification of the solution encoding, the decoder, the parameters that determine population size, size of elite and mutant sets, probability offspring inherits key of elite parent, and the stopping criterion. The termination criteria can be defined either in terms of total iterations, iterations without improvement, time, or a target objective function value. Depending on the problem one of these criteria is chosen. We next define a BRKGA for the CMST.

4. A BRKGA heuristic for the CMST

In this section we describe the ingredients of our BRKGA heuristic for the CMST. We start with two alternative decoders, that we call *subroot assignment* and *predecessor assignment*, respectively. Then, we describe an improvement phase which is common to both decoders.

Different codes have been proposed in the literature to represent spanning trees (Thompson et al., 2007; Rothlauf et al., 2002; Raidl and Julstrom, 2003; Prüfer, 1918; Neville, 1953). To the best of our knowledge, none takes into account the capacity constraint present in the CMST. Indeed the decoders proposed in this paper produce trees that are feasible to the CMST and, thus satisfy such constraints. In addition to the instance data (the graph $G = (V, E)$, demands w_i, $i \in |V^+|$, costs c_{ij}, $(i, j) \in E$, and capacity Q), both decoders take as input a vector $\mathcal{X}$ of n random keys, where $n = |V^+|$ and the i-th random key corresponds to the i-th terminal, and start by applying an assignment to the components of $\mathcal{X}$. The main difference between the two decoders is the meaning of the assignment: while the subroot assignment decoder allocates each component to the subroot of the s-tree the corresponding terminal belongs to in the decoded solution, the predecessor assignment decoder allocates each component to its immediate predecessor in the decoded tree. Other decoders for the CMST have also been proposed in the Ph.D. of E. Ruiz (2013). The numerical results in his thesis indicate that they are outperformed by the decoders presented below so we omit them from our study.

4.1. Subroot Assignment Decoder.

The subroot assignment decoder returns an n-dimensional integer assignment vector a, where $a_i = k$ indicates that vertex $i \in V^+$ is assigned to s-tree s-T_k. Therefore, $a_k = k$ implies that vertex k is a subroot. Recall that a subtree T_i is a subgraph of a tree rooted at vertex $i \in V$, whereas an s-tree s-T_k is a subtree rooted at subroot $k \in V$. The algorithm uses a vector s to keep the residual capacities of the partial s-trees; $s_k = q$ indicates that s-T_k can still accommodate another q units before its Q units are fully used up.

Vertices are scanned in increasing order of their random keys in vector $\mathcal{X}$. Each scanned vertex i is assigned to the closest existing s-tree with sufficient available capacity. If none exists, the scanned vertex is declared as the subroot of a new s-tree to which it is allocated. Here, the distance from a vertex i to an s-tree s-T_k is defined as $\min\{c_{ij} : j \in V(s\text{-}T_k)\}$.

```
procedure subroot assignment
   Input: $\mathcal{X}, G, w, c, Q$
   Output: Assignment vector with subroots a
 1 $a_i \leftarrow 0$, $s_i \leftarrow Q$, for $i = 1, \ldots, n$;
 2 Initialize list VERTICES with vertices $1, \ldots, n$ in increasing order of $\mathcal{X}$;
 3 Initialize empty list ASSIGNED;
 4 $i \leftarrow FIRST$(VERTICES);
 5 while $i \neq$ nil do
                                  /* Try to assign $i$ to an existing $s$-Tree */
 6     Sort vertices $j$ in ASSIGNED in increasing order of $c_{ij}$;
 7     $j \leftarrow FIRST$(ASSIGNED);
 8     while $j \neq$ nil and $a_i == 0$ do
 9         $k \leftarrow a_j$;
10         if $s_k \geq w_i$ then
11             $a_i \leftarrow k$;
12             $s_k \leftarrow s_k - w_i$;
13         end
14         $j \leftarrow$ NEXT(ASSIGNED);
15     end
16     if $a_i == 0$ then
                                                  /* Set $i$ as a new subroot  */
17         $a_i \leftarrow i$;
18         $s_i \leftarrow s_i - w_i$;
19     end
20     Add $i$ to ASSIGNED list;
21     $i \leftarrow$ NEXT(VERTICES);
22 end
23 return
```

Algorithm 2: Pseudo-code for `subroot assignment` decoder.

Algorithm 2 gives pseudo-code for the `subroot-assignment` decoder. In line 1, the assignment vector a and the available capacity vector s are initialized. The procedure makes use of lists `VERTICES`, `CANDIDATE`, and `SUBROOT`. List `VERTICES` is used to scan the vertices in increasing order of the random keys in vector $\mathcal{X}$. It is initialized in line 2. List `CANDIDATE` determines the order in which vertices are considered to become subroot vertices. It is initialized in line 3 and is also ordered according to the sorted random keys in vector $\mathcal{X}$. List `SUBROOT` stores the subroot vertices with nonzero available capacity. It is initialized empty in line 4. The vertex i to be assigned is scanned in the loop from line 5 to line 22. In line 6 the ordered list `ASSIGNED` is built with the vertices which are already assigned. The loop in lines 8 to 15 assigns vertex i to the s-tree s-T_k containing its closest vertex j among the ones that can fit its demand. In line 10 the available capacity at s-T_k is verified and if $s_k \geq w_i$, vertex i is assigned to s-T_k in line 11 and s_k updated in line 12. If there is no s-tree with sufficient capacity to accommodate vertex i, then i is set as a new subroot in lines 16 to 19. In any case, i is added to list `ASSIGNED` in line 20.

Sorting the list of already-assigned vertices with respect to their distances to the vertex being currently scanned (line 6) is computationally expensive, yielding an overall runtime complexity $O(n^2 \log(n))$. For this reason, in our implementation of this algorithm, one heap is maintained for each vertex, containing the distances from that vertex to all already-assigned vertices. For clarity, however, we only describe the basic version of the algorithm, without heaps.

4.2. **Predecessor Assignment Decoder.** The `predecessor assignment` decoder returns an n-dimensional integer assignment vector a, where $a_i = j$ indicates that the predecessor of vertex $i \in V^+$, is vertex $j \in V$, i.e. edge $(i,j) \in T$, and vertex j is in the only path in T from the root vertex 0 to i.

This decoder has a preprocessing phase in which the possible predecessors of each vertex $i \in V^+$ are stored in a list l_i. This list only considers vertices $j \in V^+$ such that $c_{ij} < c_{0i}$, where c_{0i} is the cost of edge $(0,i)$ if $(0,i) \in E$, or $c_{0i} = \infty$, otherwise. Therefore, $l_i = \{j : c_{ij} < c_{0i}\}$.

We define a function which enables us to identify the predecessor of terminal i from its random key, $\mathcal{X}_i$. For this, the interval $(0,1]$ is divided into $|l_i|$ subintervals, each of which is assigned to an element of l_i. Let $f(l_i, \mathcal{X}_i)$ be the function that selects the $\lceil |l_i| \times \mathcal{X}_i \rceil$-th vertex of l_i.

```
procedure predecessor assignment
   Input: $\mathcal{X}, G, w, c, Q$
   Output: Assignment vector with predecessors $a$
1  $a_i \leftarrow f(l_i, \mathcal{X}_i)$ for $i = 1, \ldots, n$;
2  Build $T$ using predecessor vector $a$ ;
3  for $i = 1, \ldots, n$ do
4      if ($T_i$ is infeasible) then
5          $T \leftarrow T \setminus T_i$
6          Execute feasibility recovery on $T_i$;
7          $T \leftarrow T \cup T_i$
8      end
9  end
10 $a \leftarrow T$;
11 return
```

Algorithm 3: Pseudo-code for `predecessor assignment` decoder.

Algorithm 3 shows pseudo-code for the `predecessor assignment` decoder. In line 1, the assignment vector a is initialized with the predecessor $a_i = f(l_i, \mathcal{X}_i)$ of each terminal i. In line 2 the assignment vector a is used to build tree T, which, as we explain below, can be infeasible. In the loop from line 3 to line 9, every subtree T_i is checked for possible feasibility violations. If needed, `feasibility recovery` is applied in line 6 to restore the feasibility of T_i, and T is updated with the restored T_i in line 7. The assignment vector a is defined in line 10 according to T.

4.2.1. *Feasibility Recovery for the Predecessor Assignment Decoder.* The structure T obtained in line 2 of Algorithm 3 can be infeasible because of two reasons. First, it may not be connected (thus containing some loop). Second, it may contain some s-tree violating the capacity constraint. For a given $i \in V^+$, both types of infeasibilities are easy to detect. In the first case, $a_i \neq 0$ and when tracing the predecessors of i we never reach the root vertex 0 and instead we "return" to vertex i. In this case we denote by T_i the connected component of T containing terminal i. In the second case, $a_i = 0$ and $\sum_{j:a_j=i} w_j > Q$.

In addition to the problem data, the input of the feasibility recovery phase is T_i and T. Its output is a feasible s-tree s-T_i and an updated structure T, which still may contain some infeasibility associated with some vertex not in $V(T_i)$.

First, if T_i violates the connectivity constraint, i is connected to the root (by setting $a_i = 0$) so T_i becomes an s-tree s-T_i. Next, the procedure checks if s-T_i violates the capacity constraint, i.e. $w(s\text{-}T_i) > Q$. If so, subtrees $T_j \subset s\text{-}T_i$ with $w(T_j) \leq Q$, are repeatedly removed from s-T_i until $w(s\text{-}T_i) \leq Q$. Every removed subtree T_j is either connected to the root or to the closest s-tree s-T_k ($k \neq i$) with enough capacity to accommodate T_j. The distance from T_j to an s-tree s-T_k, is defined as the minimum distance between any vertex of T_j and any vertex of s-T_k:

$$\text{dist}(T_j, s\text{-}T_k) = \min\{c_{lm} \ : \ l \in V(s\text{-}T_k), m \in V(T_j)\}.$$

After identifying the s-tree s-T_k, with $d(s\text{-}T_k) + d(T_j) \leq Q$, closest to T_j, then T_j is connected to s-T_k, provided that $\text{dist}(T_j, s\text{-}T_k) \leq c_{0j}$. Otherwise, such s-tree does not exist or $\text{dist}(T_j, s\text{-}T_k) > c_{0j}$, so T_j is connected to the root.

Algorithm 4 shows the `feasibility recovery` phase. In line 1, the procedure checks if T_i is an s-tree. If not, T_i is connected to the root in line 2. The loop from line 4 to line 14 is entered when s-tree s-T_i violates the capacity constraint, to make it feasible with respect to capacity. Line 5 displays the search for a subtree T_j, with $j \in V(s\text{-}T_i)$, and an s-tree s-T_k such that $w(s\text{-}T_k) + w(T_j) \leq Q$ and $\text{dist}(T_j, s\text{-}T_k)$ is minimum. If $\text{dist}(T_j, s\text{-}T_k) \leq c_{0j}$ (line 8), T_j is connected to s-T_k in line 9 and the accumulated demand of s-T_k is updated in line 10. Otherwise, T_j is connected to the root in line 12. Since the complexity of the feasibility recovery procedure is $O(n^2)$, the overall runtime complexity of the predecessor decoder is $O(n^3)$.

```
procedure feasibility recovery
  Input: Infeasible subtree $G, w, c, Q$, $T_i$, $T$
  Output: Feasible subtree $T_i$
1  if $a_i \neq 0$ then
2      $a_i \leftarrow 0$;
3  end
4  while $w(T_i) > Q$ do
5      Find $T_j \subset T_i$ and subroot $k \in V^+$ such that $w(s\text{-}T_k) + w(T_j) \leq Q$ and
           $\text{dist}(T_j, s\text{-}T_k)$ is minimum;
6      $T_i \leftarrow T_i \setminus T_j$;
7      $w(T_i) \leftarrow w(T_i) - w(T_j)$;
8      if $\text{dist}(T_j, s\text{-}T_k) \leq c_{0j}$ then
9          $T_k \leftarrow T_k \cup T_j$;
10         $w(s\text{-}T_k) \leftarrow w(s\text{-}T_k) + w(T_j)$;
11     else
12         $a_j \leftarrow 0$;
13     end
14 end
15 return
```

Algorithm 4: Pseudo-code for feasibility recovery procedure.

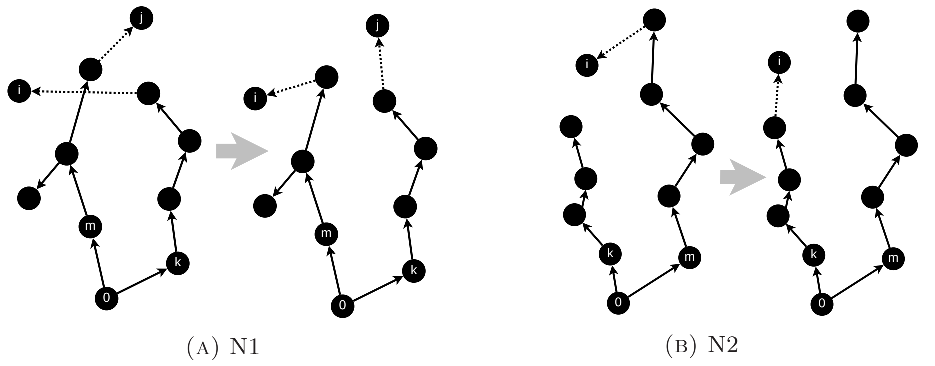


FIGURE 1. Neighborhoods *N1*, exchange of vertices, and *N2*, vertex reassignment

4.3. **Local Search.** Similar to other BRKGA implementations (e.g., Reis et al. 2011 and Resende et al. 2012), we extended the proposed decoders with an improvement phase, which is applied to the solution found in the initial phase. The main ingredient of the improvement phase is a local search, which consists of a multi-neighborhood local improvement procedure. Four neighborhoods (*N1*, *N2*, *N3*, and *N4*) are explored sequentially until a solution is found which is locally optimal in all four neighborhoods. Only when a solution cannot be further improved using the current neighborhood the algorithm jumps to the next one; after *N4* it jumps back to *N1*. The local search finishes when no further improvements can be attained with any of the neighborhoods.

Below we describe the four neighborhoods. In all of them moves are only allowed if the resulting solution remains feasible with respect to the capacity constraint.

N1: involves swapping two vertices in different s-trees, i.e. $i \in V(s\text{-}T_k)$ and $j \in V(s\text{-}T_m)$, with $k \neq m$. Figure 1A illustrates a move in *N1*.

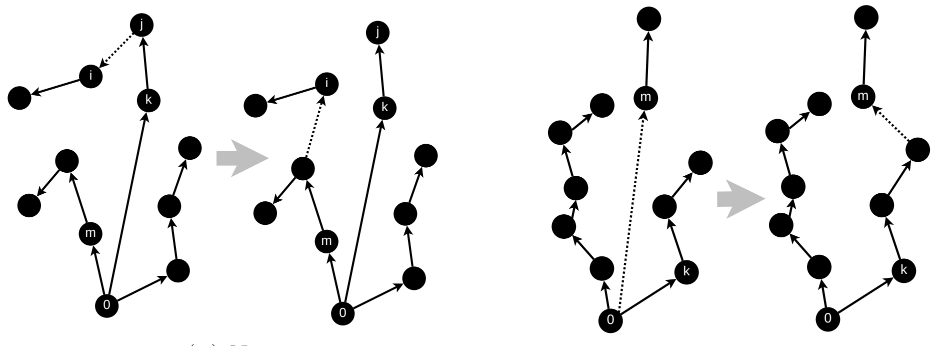
N2: includes reassignments in which vertex $i \in V(s\text{-}T_k)$ is reassigned to another s-tree, say $s\text{-}T_m$. Figure 1B illustrates a move in neighborhood *N2*.

N3: is a generalization of *N2*, where a subtree T_i of an s-tree $s\text{-}T_k$ is reassigned to a different s-tree $s\text{-}T_m$. We restrict moves to subtrees T_i which are not s-trees. Figure 2A illustrates a move in neighborhood *N3*.

N4: merges two s-trees, $s\text{-}T_k$ and $s\text{-}T_m$, into a single s-tree $s\text{-}T_r$. The new s-tree is obtained as the MST on the set of vertices $\{0\} \cup V(s\text{-}T_k) \cup V(s\text{-}T_m)$. This operation also gives the new s-root r, which need not be either m or k. *N4* can be very useful after the feasibility recovery step (see Algorithm 4) in which "small" s-trees can be created. Figure 2B illustrates a move in *N4*.

In the general case, the size of all neighborhoods is $O(n^2)$. In our implementation each neighborhood is explored using a first-improvement policy so, in practice, the actual size of the explored neighborhoods was usually considerably smaller. The order in which the neighborhoods are explored and the strategy for exploring them (sequential vs nested) were decided based on the results of preliminary testing.

Neighborhoods similar to *N1* and *N2* have been used in Amberg et al. (1996) and neighborhoods similar to *N3* and *N4* have been used in Sharaiha et al. (1997). More sophisticated neighborhoods have been explored in Ahuja et al. (2001; 2003).



(A) N3

(B) N4

FIGURE 2. Neighborhoods *N3*, sub-tree reassignment, and *N4*, *s*-tree merging

4.4. **Minimum Spanning Tree Stage.** Any s-tree, $s\text{-}T_k$, can be reoptimized by computing the MST in the graph induced by $V(s\text{-}T_k) \cup \{0\}$, since $w(V(s\text{-}T_k)) \leq Q$. However, with the exception of *N4*, the above neighborhoods do not guarantee that the obtained s-trees are optimal MSTs in the graph induced by their vertices. Since MSTs can be easily computed using, for instance, Kruskal's algorithm (Kruskal, 1956), a natural implementation of the above local search leads to terminate the exploration of $N1$-$N3$ with a minimum spanning tree stage (MST-stage), reoptimizing the obtained s-trees. If the solution is improved by the MST, then we continue exploring the neighborhood. Otherwise, we jump to the next neighborhood. This policy is referred to as *MST-at-end*. In the exploration of $N1$-$N3$, we have also applied the MST-stage according to the following alternative policies:

MST-at-change: whenever an improvement is found, an MST-stage is executed for the s-trees involved in the move.

all-MST: every time a movement is considered, an MST-stage is applied to the involved subtrees. That is, to evaluate each movement we tentatively make it and then reoptimize the resulting subtrees. If the movement improves the solution cost, it is kept, otherwise it is discarded.

4.5. **Strategic Oscillation.** Strategic oscillation (Glover and Laguna, 1997) has shown to improve the results of many heuristics. It can be especially effective in problems where feasibility may restrict the exploration of neighborhoods, as it allows the local search to cross the border between feasibility and infeasibility. We propose the use of strategic oscillation in our BRKGA, allowing a violation of the capacity of up to *MaxQ*, which is penalized in the objective function. In our basic strategic oscillation, *descent-SO*, the penalty term is updated at the end of each iteration using a descent policy. When it reaches zero, no further update is done.

We also propose two alternative policies for updating the penalty term. The first one, *descent-ascent-SO*, uses upper and lower bounds on the value of the penalty term, which decreases when the upper bound is reached and increases when it is equal to the lower bound. In the second alternative policy, *alternate-SO*, which starts as *ascent-descent-SO*, when the penalty term reaches zero, it is set to either $+\alpha$ or $-\alpha$, where the value of α is close to zero. The idea of a negative penalty value is to encourage moves into the infeasible area to escape from local minima.

4.6. **Neighborhood Reduction.** Indeed the most time consuming policy for the MST-stage is *all-MST*, especially when exploring neighborhoods *N1* and *N2*. For these neighborhoods we implemented a neighborhood reduction policy, in which the only moves explored for interchange or reassignment are the ones involving vertices i and j with $\beta c_{ij} \leq c_{0i} + c_{0j}$, i.e. the distance between i and j multiplied by a certain factor β, does not exceed the sum of the distances between each of these vertices and the root. Otherwise the move is not considered. We are not aware of any CMST heuristic where a similar neighborhood reduction has been applied.

5. Experimental results

5.1. **Benchmark instances.** In our computational experiments we use the sets of well-known CMST benchmark instances available at `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/capmstinfo.html`, as well as the instances proposed in Martins (2007). These test instances are divided into two main classes according to the type of demand. The first class contains instances with unitary demands (*UD*), whereas instances in the second class have non-unitary demands (*non-UD*). Instances labeled as *tc*, *te*, and *td* are in the first class, while instances of type *cm* are in the second one. Instances of types *tc*, *te* and *td* have Euclidean distances (*EU*). The main difference among them is the location of the root: at the center of a rectangular region in set *tc*, at a corner of a rectangular region in set *te*, and far apart outside the rectangular region where the rest of the terminals are placed in set *td*. Instances of sets *tc*, and *te* have a number of vertices, excluding the root, $n \in \{80, 120, 160\}$, whereas all instances in set *td* have $n = 80$ terminals. Instances in *tc*, *td*, and *te* have capacity values $Q \in \{5, 10, 20\}$. Set *cm* contains instances with non-Euclidean distances with $n \in \{49, 99, 199\}$. They have non-unitary demands, with values ranging from 1 to 100, and capacities $Q \in \{200, 400, 800\}$. There are five instances for each combination (n, Q), except for sets with $n = 160$, which contain one instance per combination (n, Q). This gives a total of 126 instances. A summary of the characteristics of these instances can be found in Table 1.

Table 1. Characteristics of test instances.

Set	n	demand	distances	root	Q
*tc*80	80	*UD*	*EU*	Center	5, 10, 20
*tc*120	120	*UD*	*EU*	Center	5, 10, 20
*tc*160	160	*UD*	*EU*	Center	5, 10, 20
*te*80	80	*UD*	*EU*	Corner	5, 10, 20
*te*120	120	*UD*	*EU*	Corner	5, 10, 20
*te*160	160	*UD*	*EU*	Corner	5, 10, 20
*td*80	80	*UD*	*EU*	Out of grid	5, 10, 20
*cm*50	49	*non-UD*	*non-EU*	Center	200, 400, 800
*cm*100	99	*non-UD*	*non-EU*	Center	200, 400, 800
*cm*200	199	*non-UD*	*non-EU*	Center	200, 400, 800

5.2. **Implementation details.** All the tested versions of the BRKGA were implemented in `C++` using the BRKGA API of Toso and Resende (2014). Codes were compiled with `g++` with flags "`-c`" and "`-O3`". All runs were executed on a Pentium Core2 at 3.1 GHz computer with 2 GB of RAM. The parameter values for the BRKGA API were $p = 100$, $p_e = 0.25p$, $p_m = 0.10p$, and $\rho_e = 0.65$. The maximum number of generations without improvement was used as stopping criterion. It was set to $\frac{n-20}{Q}$ and to $\frac{45Q}{Q-100}$, for UD and non-UD instances, respectively.

This setting was motivated by previous experiments, which indicated that the number of vertices and the capacity of the instances affected the number of iterations without improvement before finding the best solutions. In addition, for *non-UD* instances, the average demand also had some influence.

5.3. **Preliminary experiments.** To test the effectiveness of the possible ingredients of our BRKGA, we ran some preliminary experiments to evaluate their contribution to the overall performance of the algorithm. All these preliminary tests were run on the five $tc80$ instances and on the five $cm50$ instances with $Q = 200$.

5.3.1. *Decoder comparison.* The first experiment was to run the two decoders proposed proposed in Sections 4.1 and 4.2 respectively within a generic BRKGA API with no further enhancement. For this, each of them was separately embedded in the previously mentioned BRKGA API of Toso and Resende (2014). A summary of the obtained results is given in Table 2 under columns *Without LS.* Entries in columns *Subr.* and *Predec.* correspond to `subroot assignment` and `predecessor assignment`, respectively. The numerical results showed that plain `subroot assignment` and `predecessor assignment` are quite similar in terms of both, quality solution and time requirements. However, on average, `subroot assignment` was faster, although `predecessor assignment` found somehow better solutions. We believe this is due to the fact that `predecessor assignment` better transmits genetic information: similar keys yield similar solutions. In contrast, genetic information is not well transmitted with `subroot assignment`, since small changes in the key can substantially modify the resulting solution. In any case, we should note that the above results were obtained by simply embedding the proposed plain decoders into a generic BRKGA API with no further enhancement.

5.3.2. *Performance of Local Search and impact of MST-stage.* A summary of the results obtained when the decoders were extended with the basic local search (without neighborhood reduction) under the *MST-at-end* policy is given in Table 2 under columns *With LS.* As can be seen, the local improvement stage significantly reduced the average gap of both decoders for all instances. The increase of the computing times is moderate, taking into account the improvement in the quality of the solutions. This confirms the effectiveness of our local search in which relatively simple neighborhoods are explored efficiently. It is worth noting that the impact of the local search on the predecessor assignment decoder was substantially larger than on the subroot assignment decoder. This cannot be attributed to the quality of the initial solution, since, as we have seen, originally both decoders produced solutions with quite similar values. On the contrary, this difference highlights the role played by the employed decoder for providing good starting points to the local search.

Taking into account the results obtained so far, only the predecessor assignment decoder was used in all remaining experiments.

The results obtained with the predecessor assignment decoder and the *MST-at-change* and *all-MST* policies are summarized under columns *MST policies* of Table 2. The *all-MST* strategy produced the smallest gaps, at the expense of much longer CPU times (about one order of magnitude). These longer CPU times are the result of the large number of MSTs that are computed (for each movement considered, at

Table 2. Summary of preliminary results

		Without *LS*		With *LS*		*MST* policies		*Descent Strat. Osc.*			*Neigh.*
Group		*Subr.*	*Pred.*	*Subr.*	*Pred.*	*change*	*all*	*end*	*change*	*all*	*Red.*
tc80											
	%gap	9.38	8.78	3.52	0.29	0.34	0.00	0.22	0.28	0.00	0.00
	CPU	1.59	2.13	17.96	17.82	38.10	341.92	17.98	37.93	361.05	99.05
cm50 $Q = 200$											
	%gap	5.27	3.50	1.33	0.53	0.63	0.10	0.42	0.51	0.10	0.10
	CPU	2.26	2.52	16.54	12.10	28.18	335.05	14.62	32.18	345.59	103.21

least one MST is computed). On average, the *MST-at-change* provides the worst solutions and its CPU times are longer than those of *MST-at-end*, which showed a good balance between time and solution quality.

5.3.3. *Impact of Strategic Oscillation.* After some tuning, the maximum allowed violation was set to $MaxQ = 1.1 \times w(V^+)/n$. Initially, the penalty term is set to $1 + \frac{n}{Q}$. It is updated at the end of each BRKGA iteration by steps of $-\frac{10}{Q}$. In the *descent-SO* policy no further update is done when it reaches zero.

The results of *descent-SO* with the basic local search are summarized under column *Descent Strat. Osc.* of Table 2. As can be seen, for the *all-MST* policy, *SO-descent* increased the computing times producing deviations relative to best-known solutions similar to those without strategic oscillation. In contrast, for both *MST-at-change* and *MST-at-end*, *SO-descent* allowed to reduce the deviations, although these gaps were still worse than those of *all-MST* without strategic oscillation.

5.3.4. *Impact of Neighborhood Reduction.* None of the experiments described so far applied the neighborhood reduction in the local search. The results obtained when applying the neighborhood reduction of Section 4.6 in the local search with the *all-MST* strategy (without strategic oscillation) are summarized under column *Neighborhood Reduction* of Table 2. As can be seen, by constraining the moves in the neighborhoods, the CPU times decreased significantly without compromising the quality of the solutions obtained. Even if these times are still larger than those of the other two strategies, the solutions obtained are considerably better.

On the other hand, we observed that if the neighborhood reduction was applied with the other MST-stage policies, the quality of the solutions obtained was seriously deteriorated. This led us to use the *all-MST* strategy in all the remaining experiments and, in particular, in the final algorithm.

5.3.5. *Strategic Oscillation combined with Neighborhood Reduction.* Finally, we analyze the joint effect of strategic oscillation and neighborhood reduction. Because the results of neighbor reduction without strategic oscillation on instances of sets *tc80* and *cm50* with $Q = 200$ were already quite good, the joint effect of both ingredients on this instances was not clear. Therefore, these experiments were run on the larger instances of sets *cm100* with $Q = 200$ and *cm200* with $Q = 400$.

The results obtained are summarized in Table 3. *Alternate-SO* obtained the best results for both groups of instances. Surprisingly, the versions of the algorithm without strategic oscillation where better than both *descent-ascent-SO* and *SO-descent* for the instances in the largest set. This is probably due to the magnitude of the penalty term, which sometimes is too big relative to the cost of the tree. This also suggests why *alternate-SO* does better. In this procedure the penalty term oscillates between $[-\alpha, \alpha]$, which is very close to zero.

Table 3. Results of Strategic Oscillation with Neighborhood Reduction

		without-SO	*descent-SO*	*ascent-SO*	*alternate-SO*
cm100 $Q = 200$					
	%gap	1.04	0.98	1.08	0.95
	CPU	365.16	365.13	368.12	366.68
cm200 $Q = 400$					
	%gap	0.54	0.84	0.78	0.42
	CPU	3,580.03	3,468.41	3,548.88	3,501.86

5.4. **Numerical results of the BRKGA and analysis.** Below we present and analyze the numerical results obtained with our BRKGA in the final set of computational experiments. According to the results of Section 5.3, the ingredients of the BRKGA used for these experiments are the predecessor decoder with local search under the *all-MST* strategy and neighborhood reduction, and *Alternate-SO*. Now we used the complete set of test instances described in Section 5.1. To evaluate the robustness of the BRKGA, it was run seven times on each test instance.

Detailed results of the BRKGA for all the instances can be found in Tables 9, 10, and 11 of Appendix A. The values of best-known/optimal solutions have been taken from different sources: Ahuja et al. (2001) for instances *tc80* and *te80*; Martins (2007) for instances *tc120*, *te120*, *tc160*, *te160*, and *td80*; and Ahuja et al. (2003) and Uchoa et al. (2008) for the *cm* instances. The optimality of most of these solutions was proven in Uchoa et al. (2008), which is the most successful exact method proposed to date. There are however, 25 instances in our set of benchmark instances with unknown optimal solutions. These are the instances whose entries are not in boldface in columns *Best-known Solution* of Tables 9, 10, and 11.

Table 4 summarizes the results for the different groups of instances. The first 6 columns show deviations of the obtained solutions with respect to best-known solutions. In particular, figures under *mean %gap* refer to average values over the seven runs on each instance, while those under *best %gap* refer to the run(s) where the algorithm produced its best solution. In both cases, columns under *avg.*, *min.* and *max.* give, respectively, the average, minimum, and maximum value, over all the instances in the row group. Column under *stdev* gives averages of the standard deviations of the values of the obtained solutions. The last column shows the average CPU runtime (among both, instances and runs) until termination. Rows labeled *UD* and *non-UD* give the average values over all the *UD* and *non-UD* instances, respectively.

Table 4. Summary of BRKGA results

	mean %gap			best %gap			stdev	CPU time
Group	avg.	min.	max.	avg.	min.	max.	avg.	(secs.)
*tc*80	0.005	0.000	0.069	0.000	0.000	0.000	0,101	99.77
*te*80	0.005	0.000	0.035	0.000	0.000	0.000	0,248	217.88
*td*80	0.004	0.000	0.022	0.000	0.000	0.000	0,216	259.01
*tc*120	0.040	0.000	0.224	0.000	0.000	0.000	0,392	324.55
*te*120	0.092	-0.129	0.457	0.013	-0.259	0.245	1,095	694.73
*tc*160	0.165	0.115	0.197	-0.025	-0.076	0.000	2,656	1,753.70
*te*160	0.010	0.000	0.025	0.000	0.000	0.000	0,289	1,587.30
UD	**0.034**	**-0.020**	**0.158**	**0.001**	**-0.051**	**0.045**	**0.489**	**419.285**
*cm*50	0.072	0.000	0.524	0.000	0.000	0.000	0,374	98.97
*cm*100	0.351	0.000	2.072	0.171	0.000	1.541	0,588	459.50
*cm*200	0.821	-0.107	2.894	0.412	-0.358	2.666	2,342	3,275.26
non-UD	**0.415**	**-0.036**	**1.83**	**0.194**	**-0.119**	**1.402**	**1.101**	**1,277.910**

As can be seen, our results are good. In general, the algorithm was robust in the sense that there were only small variations in the output of different runs on each instance. This can be appreciated by comparing columns under *mean %gap* and *best %gap*. Furthermore, for each instance, we computed the percent deviation from the worst to the best solutions found in the seven runs. This deviation was 0 for 71 out of the 126 considered instances, and on average 0.21%.

The average percent deviation from the best-known solution over all instances and runs was 0.17%. For the *UD* instances, the average percentage gap never exceeded 0.46% and for 50 out of the 81 *UD* instances the BRKGA found the best-known solution in all seven executions. The performance of the algorithm is, however, less predictable on the *non-UD* instances (see Appendix A). The algorithm worked remarkably well on instances with larger capacities ($Q = 800$) where it also produced a best-known solution in all seven runs for the ten instances with $n \in \{49, 99\}$, and improved the best-known solution for two out of the five instances with $n = 199$. Similar results were obtained for the instances with medium capacities ($Q = 400$) where again it always found a best-known solution for nine out the ten instances with $n \in \{49, 99\}$. Furthermore, it improved the value of the best-known solution for three out of the five instances with $n = 199$. However, for the other two instances in this group, it was not able to find a solution with the best-known value. The performance of the algorithm somewhat declines on some of the instances with smaller capacities ($Q = 200$), where BRKGA finds an optimal solution for all of the small 50-vertex instances, and two out of the five instances with 100 vertices, but it fails in finding a best-known solution for the remaining eight instances. Indeed we could improve the performance of our algorithm for this specific subset of instances by tailoring the values of the BRKGA parameters. In order to highlight the robustness of our algorithm, we chose not to follow this path at the expense of not obtaining the best possible results.

The computing times until termination of the BRKGA are quite modest for instances of these sizes and difficulty. Nearly all *UD* instances with Euclidean distances and 80 vertices ($tc80$, $td80$, $te80$) terminated in less than five minutes. The CPU time to termination for instances in the same class with more than 80 vertices ($tc120$, $te120$, $tc160$, and $te160$) was on average less than 2,200 seconds, which is the CPU time required by the most time consuming instance in this group ($te160$ with $Q = 10$). Overall, the most demanding instances were the *non-UD* $cm200$ instances where the average computing time is slightly over 3,000 seconds. Recall that the stopping criterion is a limit on the number of iterations without improvement. This partially explains the variability on the CPU times required for instances of the same size (see the Appendix); although the improvement phase of the decoder plays an important role in the algorithm, in many instances, the incumbent solution keeps improving generation after generation before stabilizing. This suggests that the considered CMST instances tend to have multiple poor local optima with respect to the considered neighborhoods.

For the seven instances listed in Table 5, our best solutions improved the previous best-known ones. This is remarkable, because there are only 25 test instances for which the optimality of the previous best-known solution was not proven, so our improvements affect to 28% such instances. Note also that a number of well-known heuristics already have been applied to the same instances. Moreover, five such instances (all but the two with $Q = 800$) were also used to test the exact algorithm of Uchoa et al. (2008), which for these instances had to be aborted after more than 200,000 seconds with solutions outperformed by the ones we have obtained.

Table 5. New best-known solutions.

Instance	Q	Previous UB	New UB	% improvement
$tc160$-1	10	1319	1318	0.08
$te120$-4	20	773	771	0.26
$cm200$-2	400	476	475	0.21
$cm200$-3	400	559	557	0.36
$cm200$-4	400	389	388	0.26
$cm200$-2	800	294	293	0.34
$cm200$-3	800	361	360	0.28

We close this section by comparing our results with those produced by other heuristics. For this comparison we chose: (*i*) the VLNS heuristic of Ahuja et al. (2003) which, in our opinion, is the heuristic which has produced the best results, (*ii*) the ant colony heuristic (ACO) of Reimann and Laumanns (2006), (*iii*) the enhanced second-order algorithm (ESO) of Martins (2007), and (*iv*) the RAMP heuristic of Rego et al. (2010). Tables 6 and 8 summarize the results of the compared methods, taken from the original papers. The VLNS results for instances $tc120$, $te120$, $tc160$, and $te160$ have been taken from Martins (2007; 2014), and the 'value of ESO'+1 has been used for the instances labeled as "NI" in Martins (2007).

Table 6. Average gaps and CPU times (in seconds) for different heuristics

	VLNS (2003)		ACO(2006)		ESO (2007)		RAMP(2010)		BRKGA	
Group	%gap	time	%gap	time	%gap	time	%gap	time	%gap	time
$tc80$	0.000	1,800	0.205	6.5	0.078	3,600	0.018	267.3	0.005	99.8
$te80$	0.000	1,800	0.085	18.8	0.179	3,600	0.217	1,842.7	0.005	217.9
$td80$	-	-	-	-	0.075	3,600	-	-	0.004	259.0
$tc120$	$0.147^{\dagger}$	-	0.074^{*}	66.0	0.474	5,400	-	-	0.040	324.6
$te120$	$0.256^{\dagger}$	-	0.766^{*}	178.0	0.458	5,400	-	-	0.092	694.7
$tc160$	0.716	-	0.867	543.3	0.412	7,200	-	-	0.165	1,753.7
$te160$	0.583	-	0.358	545.0	0.216	7,200	-	-	0.010	1,587.3
UD	**0.284**	**1,800**	**0.251**	**104.2**	**0.257**	**4,533**	**0.118**	**1,055.0**	**0.034**	**419.3**
$cm50$	0.020	1,000	-	-	-	-	0.146	850.5	0.072	99.0
$cm100$	0.407	1,800	-	-	-	-	$0.314^{\dagger}$	35,800.0	0.351	459.5
$cm200$	1.021	3,600	-	-	-	-	-	-	0.821	3,275.3
non-UD	**0.483**	**2,133**	-	-	-	-	**0.230**	**18,325.3**	**0.415**	**1,277.9**
Computer	Pentium 4		Pentium M		AMD Athlon		Pentium P4		Pentium Core2	
CINT2006	11.5(0.52)		9.04(0.41)		12.9(0.58)		11.5(0.52)		22.1(1)	

- information not available.
Information for only some instances of the group: * Only 3 instances, †Only 11 instances.

For the *UD* instances in sets $tc80$ and $te80$, the best results are obtained with the VLNS, although the results of the BRKGA are also very good, since in both cases the mean percentage gaps are nearly zero (0.005%). On these instances the ACO, ESO, and RAMP are outperformed by VLNS and BRKGA. For the $td80$ instances, the results of the BRKGA are better than those of the ESO, the only heuristic that can be compared on this set. For the *UD* instances with $n \in \{119, 159\}$ ($tc120$, $te120$, $tc160$, and $te160$) the best gaps are obtained with the BRKGA, with averages of 0.07% and 0.09% for the 120- and 160-vertex instances, respectively. For the *non-UD* set $cm50$, again, the VLNS outperforms BRKGA as well as the other algorithms in our study. For *non-UD* instances in set $cm100$, the RAMP algorithm obtains the best average gaps, although these results are restricted to instances with $Q = 200$. For these specific instances, the BRKGA obtains very close results. Note, in addition, that the average CPU times of the RAMP algorithm for this set are very large. For the complete set of instances in $cm100$ with $Q \in \{200, 400, 800\}$, the BRKGA outperforms the VLNS, which was so far the heuristic with the best results for the whole set. Finally, for the instances in $cm200$ the BRKGA obtained better average results than any other method in our study.

Comparing the BRKGA and the VLNS in terms of the average gaps, it can be observed that BRKGA performs better for larger instances (100, 120, 160, and 200 vertices) and slightly worse for instances with 80 vertices. The BRKGA always outperformed the ACO in terms of the average gap. This could be justified by the fact that the ACO was designed to obtain fairly good results with little computational effort. As explained in Reimann and Laumanns (2006), the results of the ACO for the *non-UD* instances remain unpublished because of ACO's poor performance. The average gaps of the BRKGA are also better than those of the ESO. Finally, the BRKGA, on average, also obtained better solutions than RAMP for all comparable groups of instances except group $cm100$ as was previously explained.

To assess the goodness of the results obtained with BRKGA we performed statistical tests comparing the deviations from the best known solution, of the solutions produced by each algorithm. For the BRKGA we used the solution obtained in the first of the seven runs with each instance. According to the probability plots of the corresponding paired samples, these do not follow a Normal distribution. Therefore, Wilcoxon signed-rank tests were performed. In all cases the alternative hypothesis is that the mean of BRKGA is smaller than the mean of the compared algorithm. Table 7 summarizes the obtained results. Since different data are available for the different algorithms, we give the number of instances on which each algorithm can be compared to BRKGA (N), as well as the number of instances where the two algorithms did not yield the same solution ($N_{\neq}$). The value of the test statistic is given as U, and the corresponding p-value is given in the last row. As can be seen,

TABLE 7. Wilcoxon Singned-Rank Tests of the hypothesis

	VLNS	ACO	ESO	RAMP
N	58	42	81	50
$N_{\neq}$	20	20	48	21
U	28	0	55	81
p-val	1.e-3	1.e-5	2.e-8	0.12

in all cases except for RAMP the superiority of BRKGA is statistically significant. This is not the case with RAMP that, as mentioned above, provided excellent solutions but only for some of the instances (group $cm50$ and a subgroup of $cm100$), at the expenses of extremely large CPU times.

The running times of the CMST heuristics used in our comparison were measured on different (older) computers. To give the reader an idea of how much faster our machine is as compared to the machines used in the tests of the other heuristics, the last two rows of Table 6 give the computer used in each case and its SPECint2006[1]

[1]`http://www.spec.org/cpu2006/results/cint2006.html`

base score, together with an estimate of the computing times reduction factor, if runs were performed in our computer (in parentheses). When the available information on a computer was not enough to determine its processor, we chose the slowest possible option to make sure their CPU times were not overestimated. According to the obtained estimates, ACO is the fastest algorithm for UD instances although, as mentioned above, it yields worse solutions than the other algorithms. Taking into account the average CPU times in the different instance groups, our BRKGA was faster than any other algorithm in all groups where data are available, except in $cm200$ where, on the average, VLNS took about 60% as much time as BRKGA. On the contrary, the CPU time of VLNS was about five times and twice that of BRKGA in groups $cm50$ and $cm100$, respectively. The comparison of CPU times between VLNS and BRKGA in UD instances is even more favorable to BRKGA.

Table 8. Number of best-known solutions found by the different heuristics

Group	instances	VLNS	ACO	ESO	RAMP	BRKGA
$tc80$	15	15	8	11	13	15
$te80$	15	15	10	7	6	15
$td80$	15	NA	NA	5	NA	15
$tc120$	15	5	2*	7	NA	15
$te120$	15	3	1*	1	NA	11
$tc160$	3	0	0	0	NA	3
$te160$	3	0	0	0	NA	3
$cm50$	15	14	NA	NA	11	15
$cm100$	15	9	NA	NA	3**	11
$cm200$	15	5	NA	NA	NA	7

* – Results only for $tc120$-1 and $te120$-1
NA – No available information
** – Results only for instances with $Q = 200$

Finally, Table 8 shows the number of best-known solutions obtained by each of the heuristics for each group of test instances. These results show that none of the other heuristics was able to find more best-known solutions on any group of test instances than those found by the BRKGA. On the *UD* instances, the BRKGA found a best-known solution on 77 of the 81 test instances. On the *non-UD* sets, the algorithm found a best-known solution on 34 out of 45 instances.

6. Concluding remarks

This paper presented a BRKGA for the CMST, which is a difficult combinatorial optimization problem with multiple applications, mainly in telecommunications network design. Despite the difficulty introduced by the capacity constraints, the algorithm showed to be efficient and robust for solving instances with up to 200 vertices and various characteristics (*UD* and *non-UD* demands; Euclidean and non-Euclidean distances) without practically changing the setup parameters.

By comparing the two proposed decoders we observed that the one based on predecessor assignment better transmitted genetic information from parents to offsprings than the one based on subroot assignment. This turned out to be crucial for obtaining good solutions: while the quality of the solutions produced by both decoders was not so different, the impact of the local search on the predecessor assignment decoder was substantially larger than on the subroot assignment decoder.

Another element that contributed significantly to the success of our algorithm is an effective local search, which efficiently explores four relatively simple neighborhoods. This is particularly true when the MST-stage is applied after every move and the size of the neighborhoods is restricted by only considering promising moves.

The results produced by the BRKGA on a set of 126 instances illustrate its robustness and effectiveness. Small differences are observed among seven different runs on each instance. New best-known solutions were found for seven out of the 25 instances with unknown optimal values. On the remaining instances, small average percentage gaps were obtained in moderate computational times. A comparison of the BRKGA with four other CMST heuristics in the literature highlights the efficacy of our proposal in terms of both, solution quality and computational burden.

Acknowledgement

This research was partially supported through grant MTM2012-36163-C06-05 of the Spanish Ministry of Economy and Competitiveness and ERDF funds.

References

R.K. Ahuja, J.B. Orlin, and D. Sharma. Multiexchange neighborhood structures for the capacitated minimum spanning tree problem. *Mathematical Programming*, 91:71–97, 2001.

R.K. Ahuja, J.B. Orlin, and D. Sharma. A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem. *Operations Research Letters*, 31:185–194, 2003.

A. Amberg, W. Domeschke, and S. Voss. Capacitated minimum spanning trees: algorithms using intelligent search. *Combinatorial Optimization: Theory and Practice*, 1:9–33, 1996.

A. Amberg, W. Domeschke, and S. Voss. Multiple center capacitated arc routing problems: a tabu search algorithm using capacitated trees. *European J. of Operational Research*, 124:360–376, 2000.

J.R. Araque, L.A. Hall, and T.L. Magnanti. Capacitated trees, capacitated routing, and associated polyhedra. Technical Report SOR-90-12, Program in Statistics and Operations Research, Princeton University, Princeton, NJ, 1990.

J.C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA J. on Computing*, 6:154–160, 1994.

L.S. Buriol, M.G.C. Resende, C.C. Ribeiro, and M. Thorup. A hybrid genetic algorithm for the weight setting problem in OSPF/IS-IS routing. *Networks*, 46: 36–56, 2005.

L.S. Buriol, M.J. Hirsch, T. Querido, P.M. Pardalos, M.G.C. Resende, and M. Ritt. A biased random-key genetic algorithm for road congestion minimization. *Optimization Letters*, 4:619–633, 2010.

A. Chow and W. Kershenbaum. A unified algorithm for designing multidrop teleprocessing networks. *IEEE Transactions on Communications*, COM-22:1762–1772, 1974.

A. Christofides. Worst case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.

L.R. Esau and K.C. Williams. On teleprocessing system design, Part II: A method for approximating the optimal network. *IBM Systems J.*, 5:142–147, 1966.

G.N. Frederickson. Approximation algorithms for some postman problems. *J. of the ACM*, 26:538–554, 1979.

G.N. Frederickson, M.S. Hecht, , and K. Chul. Approximation algorithms for some routing problems. *SIAM J. on Computing*, 7:178–193, 1979.

B. Gavish. Topological design of centralized computer networks formulations and algorithms. *Networks*, 12, 1982.

B. Gavish. Formulations and algorithms for the capacitated minimal directed tree problem. *J. of the Association for Computing Machinery*, 30:118–132, 1983.

B. Gavish. Augmented Lagrangean based algorithms for centralized network design. *IEEE Transactions on Communications*, 33:1247–1257, 1985.

B. Gavish and K. Altinkemer. Parallel savings heuristics for the topological design of local access tree networks. In *Proceedings of the IEEE Conference on Communications*, pages 130–139, 1986.

F. Glover and M. Laguna. *Tabu Search*. Kluwer, 1997.

J.F. Gonçalves and M.G.C. Resende. Biased random-key genetic algorithms for combinatorial optimization. *J. of Heuristics*, 17:487–525, 2011.

J.F. Gonçalves and M.G.C. Resende. A parallel multi-population biased random-key genetic algorithm for a container loading problem. *Computers and Operations Research*, 39:179–190, 2012.

J.F. Gonçalves, M.G.C. Resende, and J.J.M. Mendes. A biased random-key genetic algorithms with forward-backward improvement for the resource constrained project scheduling problem. *J. of Heuristics*, 17:467–486, 2011.

J.F. Gonçalves, M.G.C. Resende, and R.F. Toso. An experimental comparison of biased and unbiased random-key genetic algorithms. *Pesquisa Operacional*, 34: 143–164, 2014. doi: 10.1590/0101-7438.2014.034.02.0143.

L. Gouveia. A comparison of directed formulations for the capacitated minimal spanning tree problem. *Telecommunication Systems*, 1:51–76, 1993.

L. Gouveia. A $2n$-constraint formulation for the capacitated minimal spanning tree problem. *Operations Research*, 43:130–141, 1995.

L. Gouveia. Multicommodity flow models for spanning trees with hop constraints. *European J. of Operational Research*, 95:178–190, 1996.

L. Gouveia and L. Hall. Multistars and directed flow formulations. *Networks*, 40: 188–201, 2002.

L. Gouveia and M. Lopes. The capacitated minimum spanning tree problem: On improved multistar constraints. *European J. of Operational Research*, 160:47–62, 2005.

L. Gouveia and P. Martins. A hierarchy of hop-indexed models for the capacitated minimal spanning tree problem. *Networks*, 35:1–16, 2000.

L. Gouveia and P. Martins. The capacitated minimum spanning tree problem: revisiting hop-indexed formulations. *Computers and Operations Research*, 32: 2435–2452, 2005.

L. Gouveia and J. Paixão. Dynamic programming based heuristics for the topological design of local access networks. *Annals of Operations Research*, 33:305–327, 1991.

L. Hall. Experience with a cutting plane algorithm for the capacitated spanning tree problem. *INFORMS J. on Computing*, 8:219–234, 1996.

J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7-1:8–50, 1956.

C. Martinez, I. Loiseau, M.G.C. Resende, and S. Rodriguez. BRKGA algorithm for the capacitated arc routing problem. *Electronic Notes in Theoretical Computer Science*, 281:69–83, 2011.

P. Martins. Enhanced second order algorithm applied to the capacitated minimum spanning tree problem. *Computers and Operations Research*, 34:2495–2519, 2007.

P. Martins, 2014. Personal Communication.

J.J.M. Mendes, J.F. Gonçalves, and M.G.C. Resende. A random key based genetic algorithm for the resource constrained project scheduling problem. *Computers and Operations Research*, 36:92–109, 2009.

E.H. Neville. The codifying of tree-structure. *Mathematical Proceedings of the Cambridge Philosophical Society*, 49:381–385, 1953.

T.F. Noronha, M.G.C. Resende, and C.C. Ribeiro. A biased random-key genetic algorithm for routing and wavelength assignment. *J. of Global Optimization*, 50: 503–518, 2011.

C. Papadimitriou. The complexity of the capacitated tree problem. *Networks*, 8: 217–230, 1978.

R.C. Prim. Shortest connection matrix network and some generalizations. *Bell System Technical J.*, 36:1389–1401, 1957.

H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv der Mathematik und Physik*, 27:742–744, 1918.

G.R. Raidl and B.A. Julstrom. Edge sets: An effective evolutionary coding of spanning trees. *IEEE Transactions on Evolutionary Computation*, 7:225–239, 2003.

C. Rego and F. Mathew. A filter-and-fan algorithm for the capacitated minimum spanning tree problem. *Computers and Industrial Engineering*, 60:187–194, 2011.

C. Rego, F. Mathew, and F. Glover. RAMP for the capacitated minimum spanning tree problem. *Annals of Operations Research*, 181:661–681, 2010.

M. Reimann and M. Laumanns. Savings based ant colony optimization for the capacitated minimum spanning tree problem. *Computers and Operations Research*, 33:1794–1822, 2006.

R. Reis, M. Ritt, L.S. Buriol, and M.G.C. Resende. A biased random-key genetic algorithm for OSPF and DEFT routing to minimize network congestion. *International Transactions in Operational Research*, 18:401–423, 2011.

M.G.C. Resende, R.F. Toso, J.F. Gonçalves, and R.M.A. Silva. A biased random-key genetic algorithm for the Steiner triple covering problem. *Optimization Letters*, 6:605–619, 2012.

F. Rothlauf, D. Goldberg, and A. Heinzl. Network random keys – A tree representation scheme for genetic and evolutionary algorithms. *Evolutionary Computation*, 10:75–97, 2002.

E. Ruiz. *The capacitated spanning tree problem.* PhD thesis, Department of Statistics and Operations Research. Universitat Politcnica de Catalunya, 2013.

Y.M. Sharaiha, M. Gendreau, G. Laporte, and I.H. Osman. A tabu search algorithm for the capacitated shortest spanning tree problem. *Networks*, 29:161–167, 1997.

M.C. Souza, C. Duhamel, and C.C. Ribeiro. A GRASP heuristic for the capacitated minimum spanning tree problem using memory-based local search strategy. *Applied Optimization*, 86:627–658, 2003.

W.M. Spears and K.A. DeJong. On the virtues of parameterized uniform crossover. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236, 1991.

E. Thompson, T. Paulden, and D.K. Smith. The dandelion code: A new coding of spanning trees for genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 11:91–100, 2007.

R.F. Toso and M.G.C. Resende. A C++ application programming interface for biased random-key genetic algorithms. *Optimization Methods and Software*, 2014. doi: 10.1080/10556788.2014.890197. Published online 13 March.

E. Uchoa, R. Fukasawa, J. Lysgaard, A. Pessoa, M. Poggi de Aragão, and D. Andrade. Robust branch-cut-and-price algorithm for the capacitated minimum spanning tree problem over a large extended formulation. *Matemathical Programming*, 112:443–472, 2008.

Appendix A. Complete results

Tables 9, 10, and 11 give detailed information of the BRKGA results on all the tested instances. Columns *Mean %gap* and *CPU* show, respectively, the average over the seven runs of the deviation of the obtained solution from the optimal/best-known one, and the CPU time to termination, in seconds. Best-known solutions are given next. Boldfaced values correspond to proven optimal solutions, while best-known values that have been improved in this work are marked with an asterisk. The last four columns give the mean of the seven (often repeated) solution values, their standard deviation, and the values of the best and worst solutions obtained in the seven runs, respectively.

Table 9. Results for instances with $n = 80$ and unitary demands

Group	Q	Instance number	Mean %gap	CPU (seconds)	Best-known Solution	Mean BRKGA	Stdev BRKGA	Best BRKGA	Worst BRKGA
$tc80$	5	1	0.00	38.84	**1099**	1099.0	0.00	1099	1099
		2	0.00	99.04	**1100**	1100.0	0.00	1100	1100
		3	0.00	106.49	**1073**	1073.0	0.00	1073	1073
		4	0.00	105.53	**1080**	1080.0	0.00	1080	1080
		5	0.00	119.08	**1287**	1287.0	0.00	1287	1287
	10	1	0.00	53.06	**888**	888.0	0.00	888	888
		2	0.00	105.36	**877**	877.0	0.00	877	877
		3	0.00	112.78	**878**	878.0	0.00	878	878
		4	0.07	121.26	**868**	868.6	1.51	868	872
		5	0.00	126.02	**1002**	1002.0	0.00	1002	1002
	20	1	0.00	61.63	**834**	834.0	0.00	834	834
		2	0.00	117.17	**820**	820.0	0.00	820	820
		3	0.00	102.85	**828**	828.0	0.00	828	828
		4	0.00	103.56	**820**	820.0	0.00	820	820
		5	0.00	123.88	**916**	916.0	0.00	916	916
$te80$	5	1	0.00	180.05	**2544**	2544.0	0.00	2544	2544
		2	0.03	341.61	**2551**	2551.7	1.89	2551	2556
		3	0.01	270.51	**2612**	2612.3	0.76	2612	2614
		4	0.03	277.84	**2558**	2558.9	1.07	2558	2560
		5	0.00	199.06	**2469**	2469.0	0.00	2469	2469
	10	1	0.00	186.13	**1657**	1657.0	0.00	1657	1657
		2	0.00	215.17	**1639**	1639.0	0.00	1639	1639
		3	0.00	221.18	**1687**	1687.0	0.00	1687	1687
		4	0.00	218.45	**1629**	1629.0	0.00	1629	1629
		5	0.00	201.22	**1603**	1603.0	0.00	1603	1603
	20	1	0.00	185.14	**1275**	1275.0	0.00	1275	1275
		2	0.00	190.74	**1224**	1224.0	0.00	1224	1224
		3	0.00	187.42	**1267**	1267.0	0.00	1267	1267
		4	0.00	216.44	**1265**	1265.0	0.00	1265	1265
		5	0.00	177.27	**1240**	1240.0	0.00	1240	1240
$td80$	5	1	0.00	279.23	**6068**	6068.3	0.76	6068	6070
		2	0.00	266.01	**6019**	6019.0	0.00	6019	6019
		3	0.01	285.57	**5994**	5994.9	1.46	5994	5997
		4	0.00	206.99	**6012**	6012.0	0.00	6012	6012
		5	0.00	197.89	**5977**	5977.0	0.00	5977	5977
	10	1	0.00	341.31	**3223**	3223.0	0.00	3223	3223
		2	0.00	228.83	**3205**	3205.0	0.00	3205	3205
		3	0.00	317.28	**3212**	3212.0	0.00	3212	3212
		4	0.00	323.99	**3203**	3203.0	0.00	3203	3203
		5	0.00	237.86	**3180**	3180.0	0.00	3180	3180
	20	1	0.00	196.10	**1832**	1832.0	0.00	1832	1832
		2	0.02	262.74	**1829**	1829.3	0.49	1829	1830
		3	0.00	230.66	**1839**	1839.0	0.00	1839	1839
		4	0.02	243.12	**1834**	1834.4	0.53	1834	1835
		5	0.00	267.65	**1826**	1826.0	0.00	1826	1826

TABLE 10. Results for instances with $n = 120, 160$ and unitary demands

Group	Q	Instance number	Mean %gap	CPU (seconds)	Best-known Solution	Mean BRKGA	Stdev BRKGA	Best BRKGA	Worst BRKGA
$tc120$	5	1	0.00	128.07	**1291**	1291.0	0.00	1291	1291
		2	0.00	189.76	**1189**	1189.0	0.00	1189	1189
		3	0.03	274.47	**1124**	1124.3	0.76	1124	1126
		4	0.03	250.22	**1126**	1126.3	0.76	1126	1128
		5	0.02	235.08	**1158**	1158.3	0.49	1158	1159
	10	1	0.00	168.30	**904**	904.0	0.00	904	904
		2	0.00	355.37	**756**	756.0	0.00	756	756
		3	0.00	372.05	**722**	722.0	0.00	722	722
		4	0.00	276.17	**722**	722.0	0.00	722	722
		5	0.17	355.59	**761**	762.3	1.89	761	765
	20	1	0.00	298.73	**768**	768.0	0.00	768	768
		2	0.00	420.21	**569**	569.0	0.00	569	569
		3	0.00	442.56	**536**	536.0	0.00	536	536
		4	0.13	549.16	**571**	571.7	0.49	571	572
		5	0.22	552.47	**581**	582.3	1.50	581	585
$te120$	5	1	0.03	413.70	**2197**	2197.6	1.51	2197	2201
		2	0.06	520.85	**2134**	2135.3	1.38	2134	2137
		3	0.03	343.81	**2079**	2079.7	0.49	2079	2080
		4	0.05	464.19	**2158**	2159.0	0.00	2159	2159
		5	0.04	532.96	**2017**	2017.7	1.50	2017	2021
	10	1	0.00	575.22	**1329**	1329.0	0.00	1329	1329
		2	0.45	659.87	**1225**	1230.6	2.94	1228	1235
		3	0.17	809.37	1195	1197.0	1.83	1195	1200
		4	0.23	702.35	1230	1232.9	2.19	1231	1237
		5	0.26	767.34	**1164**	1167.0	2.16	1165	1171
	20	1	0.00	632.49	**920**	920.0	0.00	920	920
		2	0.09	1034.26	785	785.7	0.95	785	787
		3	0.02	1058.64	749	749.1	0.38	749	750
		4	-0.13	920.74	773*	772.0	0.58	771	773
		5	0.08	985.22	746	746.6	0.53	746	747
$tc160$	5	1	0.20	1034.13	**2077**	2081.1	2.97	2077	2084
	10	1	0.18	2197.84	1319*	1321.4	3.05	1318	1327
	20	1	0.12	2029.13	960	961.1	1.95	960	964
$te160$	5	1	0.03	1211.09	**2789**	2789.7	0.49	2789	2790
	10	1	0.01	2141.96	1645	1645.1	0.38	1645	1646
	20	1	0.00	1408.86	1098	1098.0	0.00	1098	1098

Table 11. Results for instances with non-unitary demands

Group	Q	Instance number	Mean %gap	CPU (seconds)	Best-known Solution	Mean BRKGA	Stdev BRKGA	Best BRKGA	Worst BRKGA
*cm*50*r*	200	1	0.00	88.78	**1098**	1098.0	0.00	1098	1098
		2	0.53	108.15	**974**	979.1	2.27	974	980
		3	0.00	102.22	**1186**	1186.0	0.00	1186	1186
		4	0.00	121.37	**800**	800.0	0.00	800	800
		5	0.00	101.38	**928**	928.0	0.00	928	928
	400	1	0.13	106.54	**679**	679.9	1.07	679	681
		2	0.00	62.16	**631**	631.0	0.00	631	631
		3	0.35	74.30	**732**	734.6	1.13	732	735
		4	0.08	124.69	**564**	564.4	1.13	564	567
		5	0.00	91.65	**611**	611.0	0.00	611	611
	800	1	0.00	131.13	**495**	495.0	0.00	495	495
		2	0.00	108.15	**513**	513.0	0.00	513	513
		3	0.00	90.67	**532**	532.0	0.00	532	532
		4	0.00	81.30	**471**	471.0	0.00	471	471
		5	0.00	92.04	**492**	492.0	0.00	492	492
*cm*100*r*	200	1	1.04	439.57	**509**	514.3	2.87	509	517
		2	2.08	368.24	**584**	596.1	1.57	593	597
		3	0.26	484.89	**540**	541.4	0.53	541	542
		4	0.33	383.76	**435**	436.4	0.98	435	437
		5	0.89	428.98	**418**	421.7	1.80	420	425
	400	1	0.17	439.19	**252**	252.4	0.53	252	253
		2	0.36	397.48	**277**	278.0	0.00	278	278
		3	0.18	457.12	**236**	236.4	0.53	236	237
		4	0.00	523.62	**219**	219.0	0.00	219	219
		5	0.00	495.37	**223**	223.0	0.00	223	223
	800	1	0.00	489.72	**182**	182.0	0.00	182	182
		2	0.00	479.85	**179**	179.0	0.00	179	179
		3	0.00	466.91	**175**	175.0	0.00	175	175
		4	0.00	485.28	**183**	183.0	0.00	183	183
		5	0.00	552.53	**186**	186.0	0.00	186	186
*cm*200*r*	200	1	1.67	3122.37	994	1010.6	5.00	1003	1015
		2	1.96	3007.78	1188	1211.3	9.11	1202	1225
		3	2.89	3061.93	1313	1351.0	2.45	1348	1355
		4	1.34	3561.27	917	929.3	7.18	919	937
		5	2.11	3236.41	948	968.0	3.32	964	974
	400	1	0.91	3097.51	391	394.6	1.90	392	397
		2	0.30	3186.94	476*	477.4	1.51	475	480
		3	-0.10	4040.28	559*	558.4	1.13	557	560
		4	0.29	3447.51	389*	390.1	1.35	388	392
		5	0.85	4046.50	418	421.6	0.79	421	423
	800	1	0.00	2778.14	254;	254.0	0.00	254	254
		2	-0.05	3133.56	294*	293.9	0.38	293	294
		3	-0.08	3187.92	361*	360.7	0.49	360	361
		4	0.00	3043.11	275;	275.0	0.00	275	275
		5	0.20	3177.69	292	292.6	0.53	292	293

(Efraín Ruiz) Departament d'Estadística i Investigació Operativa, Universitat Politècnica de Catalunya. BarcelonaTech, Barcelona, Spain.
E-mail address: `efrain.ruiz@upc.edu`

(Maria Albareda) Departament d'Estadística i Investigació Operativa, Universitat Politècnica de Catalunya. BarcelonaTech, Carrer Colom, 11, 08222 Terrassa, Spain
E-mail address: `maria.albareda@upc.edu`

(Elena Fernández) Departament d'Estadística i Investigació Operativa, Universitat Politècnica de Catalunya. BarcelonaTech, Campus Nord, C5-208, Jordi Girona, 1-3, 08034 Barcelona, Spain.
E-mail address: `e.fernandez@upc.edu`

(Mauricio G.C. Resende) Network Evolution Research Department, AT&T Labs Research, 200 S. Laurel Avenue, Room 5A-1F34, Middletown, NJ 07748 USA.
E-mail address: `mgcr@research.att.com`