The Capacitated Minimum Spanning Tree Problem

PhD Student: Héctor Efraín Ruiz y Ruiz

Thesis directed by

Maria Albareda Sambola

Elena Fernández Aréizaga

Department of Statistics and Operations Research
Universitat Politècnica de Catalunya. BarcelonaTech

Acknowledgments

Along the development of this thesis I've had the pleasure of working with some researchers from whom I learnt a lot. I am specially grateful to Elena Fernández and Mari Albareda for their unconditional support and patience, specially in the most difficult moments. I also want to thank Mauricio Resende for his contributions to do part of this work.

Most of this work was developed in the *Departament d'Estadística i Investigació Operativa* of the *Universitat Politècnica de Catalunya*. I want to thank the department and all its members for all the facilities and help I found here. Specially to Sonia Navarro and Laura Marí for their help during the thesis deposit.

I also want to thank the *CONACYT* for their financial support.

Finally, I want to thank my parents for the gift of life; my brothers and sister for all the time and experiences that we have shared; and all my friends for their support and friendship.

Contents

Introduction

Choices are part of our lives. In some occasions, the choice has to be made among a small number of alternatives, and intuition and common sense can be enough to make a good decision. However, when it comes to choose some elements of a large set that altogether satisfy our needs in the best possible way, the complexity of the decision can notably increase. In daily life situations, like choosing the best combination of possible dishes and drinks to design a banquet, the impact of the decision is relative, and most of the acceptable combinations are almost equally good. But when the decision affects the economy of a company (as for instance, when selecting the particular goods to be sold in a supermarket in order to produce the highest possible yields) or the cost and quality of a public service (e.g. when deciding which bus stops to set and the links between them to form the bus lines) it becomes very important to take all available information into account to make the best possible decision. In this case, mathematics are a very useful support.

The study of sets as mathematical objects has always been a relevant field of mathematics that is called Combinatorics. Although some tools in this field, as the combinatorial coefficients, were already known in the 12th century, most of the relevant developments were carried out during the 17th century and afterwards. Indeed, the term combinatorics itself was not introduced until 1666, when Gottfried Leibniz published his book *De Arte Combinatoria*.

Going beyond the study of the sets by themselves and their structural properties, many other fields of mathematics are strongly related to sets and make intensive use of results from combinatorics. Among them, we find, for instance, probability theory, order theory, graph theory and combinatorial optimization. The focus of this last field are optimization problems that are defined over discrete sets and whose feasible solutions are those subsets that fulfill some established conditions. If the ground set has a small number of elements, it is possible to explicitly enumerate all feasible solutions, and then, choose the most attractive one. However, as the ground set becomes larger, the number of subsets explodes and the time required just to enumerate the feasible solutions of a problem can be unaffordable for the decision maker. Therefore, specific tools are required to solve these problems.

Given the fact that, as just mentioned, the optimization problems faced in combinatorial optimization are discrete and that the size of the set of solutions to consider can be extremely large, in most cases, the only way to solve these problems efficiently is to exploit the specific structure of each particular problem. For this reason, there is a large variety of problems that have been studied within the scope of combinatorial optimization. Many of these problems can be expressed in terms of graphs, which often allows to make use of results from graph theory to have a deeper knowledge of the properties of the involved sets.

A very important class of graphs is that of connected graphs. In many problems defined over a graph, solutions are subgraphs that are required to be connected. For instance, if one seeks to find a route for traveling between two cities, that route cannot be formed by unconnected links; also, in the design of a telecommunications network, all the nodes need to be accessible from the server. Since the most elemental connected graph is a tree, tree structures are often essential to guarantee connectivity in the solution of combinatorial optimization problems.

For this reason, Minimum Spanning Tree (MST) problems are considered to be among the core problems in combinatorial optimization. These problems have been widely studied by researchers who have identified some of their structural properties and have developed efficient algorithms for solving them. This type of problems has applications in telecommunications, vehicle routing, network design, facility location, etc.

From the theoretical point of view, the interest of studying the MST lies in its inherent particularity as a mathematical object. The MST problem considers a set of vertices which need to be all connected in the best possible way through the use of edges or arcs with an associated weight. This problem can be solved exactly in polynomial time. This can be done using greedy algorithms as the ones proposed by Kruskal (1956) and Prim (1957). From the practical point of view, as noted above, the connecting structures of the MST are of great utility in graph-related applications. Along with the traveling salesman problem, it is considered to be the basis of network systems design and a wide variety of scheduling and routing applications.

Many results are known about integer programming formulations of this problem and their structure. Besides, many extensions of the MST problem have been studied, which consider other constraints in addition to the connectivity ones. Among these additional constraints the most common are the capacity constraints, as they are present in many real life applications and also in many problems that have the MST as a subproblem. Capacity constraints can be useful to model different types of situations. Most of them arise when the tree has a root, or special vertex, from which some communication flow must be sent (or received) to/from the rest of vertices. Depending on the situation they model, capacity constraints may be expressed in many ways. For instance, as a parameter over the links in the graph, when a limit exists on the flow that can circulate on each of the arcs of a tree. Also as a vertex degree limit, as a maximum number of links between two vertices, or as a maximum number of vertices on each of the branches of the rooted tree.

Depending of how we express such capacity constraints we obtain different extensions of the problem. Among such extensions we can mention the k-minimum spanning tree, the degree-constrained minimum spanning tree, the hop constrained minimum spanning tree, and the capacitated minimum spanning tree. The k-minimum spanning tree problem deals with finding the MST which connects at least k vertices, while in the degree-constrained minimum spanning tree the number of edges connecting any vertex in the resulting graph is limited. When we are interested in trees with a limited depth, we talk about the hop constrained minimum spanning tree. In this thesis we focus on the Capacitated Minimum Spanning Tree (CMST), an extension of the MST which considers a central or root vertex which receives and sends commodities (information, goods, etc) to a group of terminals. Such commodities flow through links which have capacities that limit the total flow they can accommodate.

These capacity constraints over the links result of interest because in many applications the capacity limits are inherent. We find the applications of the CMST in the same areas as

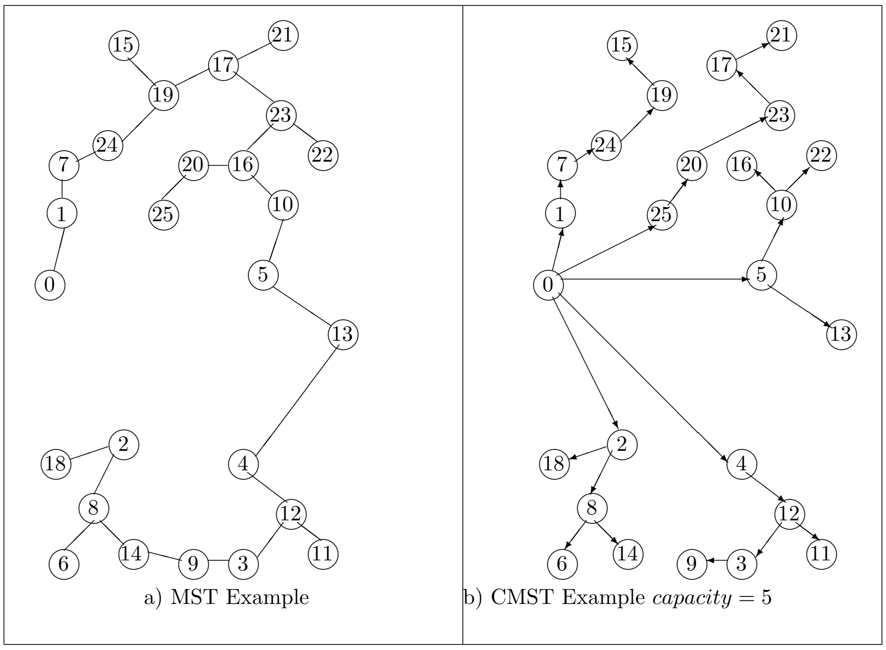

Figure 1: MST and CMST solution comparison

the applications of the MST; telecommunications network design, facility location planning, and vehicle routing.

The CMST arises in telecommunications networks design when the presence of a central server is compulsory and the flow of information is limited by the capacity of either the server or the connection lines. Its study also results specially interesting in the context of the vehicle routing problem, due to the utility that spanning trees can have in constructive methods. As it is well known, the best heuristics for the vehicle routing problem from the point of view of the worst case behavior are based on the use of spanning trees (Christofides 1976). The same idea has also been successfully transferred to arc routing problems (Frederickson, Hecht, and Kim 1976) and it has already been shown how solving the CMST can help in the solution of capacitated arc routing problems (Amberg, Domeschke, and Voß 2000).

By the simple fact of adding capacity constraints to the MST problem we move from a polynomially solvable problem to a non-polynomial one (See Figure 1). Its theoretical interest and applications, make the CMST, a problem of profound interest. Because the CMST is NP-hard, it is very difficult to find provable optimal solutions with exact methods. That explains why heuristic methods have been widely used in many of the works previous to this thesis. Nonetheless, there are also many exact methods, some of them with fairly good results.

Given the difficulty of the CMST, our capability of solving different sets of instances relies to a large extent on our capability of proposing tight formulations. In this context, one further difficulty which limits the sizes of the instances that can be successfully addressed is the number of variables that the tighter formulations involve. From this point of view, finding an equilibrium between the tightness of the formulation and its memory requirements may become crucial. Another issue that becomes critical from an algorithmic point of view, is the study of various classes of valid inequalities to reinforce the formulations, together with their separation problems.

From a different point of view, any formulation can be reinforced by incorporating the information derived from good quality solutions. This information can be used to reduce the size of the instances with elimination tests, or to reduce the size of the enumeration trees by incorporating good incumbent values from the beginning. This reinforces the interest of heuristic methods.

All the above issues are studied in this thesis, where we focus on formulations and solutions methods for the CMST. We present two new formulations that somehow relate the CMST with a location problem in which the vertices connected to the root of the tree must be selected. For these formulations, we develop different families of valid inequalities that can be used to reinforce the linear programming relaxation of the formulations and we study their corresponding separation problems. We also propose a Biased Random Key Genetic heuristic which exploits the structure of the problem and produces very good quality solutions, improving in some cases the best solution known so far.

This thesis has the following structure. In Chapter one, we define the problem and present an extensive literature review about the CMST. In the second Chapter, we describe two previous formulations of the problem present in the literature, and review the most important families of valid inequalities that have been developed. Two new formulations for the CMST are introduced in Chapter three, as well as solution algorithms for both of them. A Biased Random Key Genetic heuristic for the CMST is presented in Chapter four. In Chapter five we present the computational results obtained by the heuristic method and the solution algorithms of the proposed formulations. Finally, in Chapter six we present the conclusions for this thesis.

Chapter 1

Problem definition and literature review

Extensive research has been carried out on the capacitated minimum spanning tree (CMST) and related problems. Various formulations have been proposed as well as solution methods, including exact and heuristic approaches. Since the problem is of type NP-Hard, many heuristic algorithms have been developed. Because the CMST is a minimization problem, heuristics are typically used to find upper bounds (any feasible solution yields an upper bound). Additionally, efficient exact methods have been developed. In this chapter we describe and define the problem, introduce some notation, and present a review of the existing literature. In such review we include formulations and exact methods as well as the most relevant heuristic approaches.

1.1 Problem definition

As mentioned in the introduction, in combinatorial problems defined over graphs, trees become essential structures when connectivity is an issue. A tree is defined as any connected graph without cycles. A spanning tree is a tree in which every vertex of the original graph has at least one incident link. In the CMST we look for spanning trees that satisfy additional constraints. In principle, trees are structures defined on non directed graphs. However in the CMST, some additional conditions require a given vertex to be a reference for the others. For this reason, it is convenient to work on a directed graph and to call the reference vertex the *root*, and *terminals* the other vertices.

Let $G = (V, A)$ be a directed network without loops, with $V = \{v_0, v_1, \ldots, v_n\}$, where v_0 represents the root and $V^+ = \{v_1, \ldots, v_n\} \subset V$ the set of terminals. Abusing notation we will indistinctively use $v_i \in V$ or $i \in V$ (respectively $v_i \in V^+$ or $i \in V^+$). Each terminal i has an associated demand d_i, which must be sent from the root. Each arc $a = (i, j) \in A$ has an associated cost $c_{ij} > 0$. Consider now the following notation. Let $T \subset A$ denote a given spanning tree of G rooted at v_0:

- Subroot: Vertex connected to v_0 in T.
- Subtree ($T_i \subseteq T$): Subgraph of T which is a tree rooted at vertex $i \in V$. Note that

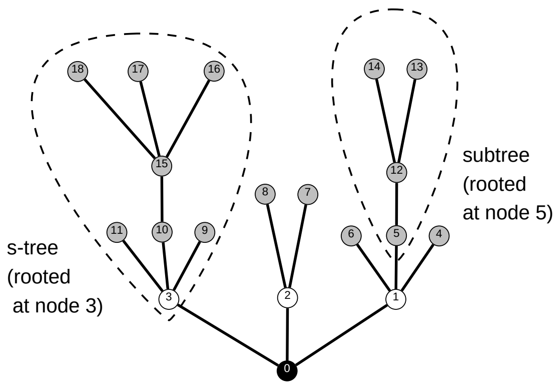

Figure 1.1: Tree T rooted at vertex 0.
Nodes 1, 2, and 3 are subroots of T. Leaves are vertices 4, 6, 7, 8, 9, 11, 13, 14, 16, 17, and 18.The subtree $\{(5, 12), (12, 13), (12, 14)\}$ is a rooted arborescence, rooted at vertex 5. The subtree $\{(3, 9), (3, 10), (3, 11), (10, 15), (15, 16), (15, 17), (15, 18)\}$ is an s-tree, rooted at subroot 3.

since T is acyclic, T_i is also acyclic. A particular case of subtrees are those rooted at subroots of T for which we will use a different notation .

- s-tree (s-$T_k \subset T$): Subtree rooted at k where k is a subroot of T (Figure 1.1).
- $V(T_i) \subseteq V^+$: Set of terminals connected by subtree T_i.
- $d(S) = \sum\limits_{j \in S} d_j$: Sum of the demands of the vertices in set $S \subseteq V^+$. When $S = V(T_i)$ for a given subtree T_i we simply write $d(T_i) = d(V(T_i)) = \sum\limits_{j \in V(T_i)} d_j$.
- Cost of T, $c(T) = \sum\limits_{(i,j) \in T} c_{ij}$.

Feasible solutions to the CMST are spanning trees of G rooted at v_0 such that the total demand of any s-tree does not exceed a given capacity $Q > 0$. That is, for any s-tree s-$T_k \subset T$ a capacity constraint is satisfied $d(s\text{-}T_k) \leq Q$

Then, the ***capacitated minimum spanning tree problem*** is to find a spanning tree rooted at v_0, satisfying the capacity constraints, of minimum total cost.

A particular case of special interest arises when all the terminal demands have the same value. This case is called the unitary demand (UD) case. In this case, without loss of generality it can be assumed that $d_i = 1$ for all $i \in V^+$. The general case is also called the non-unitary demand case (non-UD) and can be always transformed into the UD case. The transformation is achieved by replacing original vertices i with associated demand $d_i > 1$ with a set M of artificial vertices with $|M| = d_i$. Every new vertex $m \in M$ has an associated demand $d_m = 1$. Also two types of artificial arcs replace the original arcs incident with vertex i. The first type connect every pair of vertices $m, p \in M$ and have cost $c_{pm} = 0$. The second type, have cost $c_{mj} = c_{ij}$ for every pair of vertices $m \in M, j \in V^+$.

The complexity of the CMST is of type NP-hard when $3 \leq Q \leq n/2$ for the UD case as was shown by Papadimitriou (1978). This also applies for the general general case, as it can always be transformed into the UD case.

The CMST is an extension of the MST with supplementary capacity restrictions (Figure 1). By the simple fact of adding these capacity constraints, the complexity of the problem increases notably. As in any NP-hard problem, optimal solutions for the CMST are difficult to find. Nevertheless, in the UD case, there are special values of Q for which the optimal solution is trivial or can be found using a polynomial time algorithm. These values are:

- $Q = 1$. The optimal solution in this case is trivial, and is given by the star solution (all terminals are subroots).
- $Q = 2$. An optimal solution for this value of Q can be obtained by solving a perfect matching problem on a modified graph (Papadimitriou 1978).
- $Q = n - 1$. To obtain an optimal solution for this case, we compute the MST for all subsets $S \subset V^+$ with $\mid S \mid= n - 1$ and connect the node $i = V^+ \setminus S$ directly to the root. An optimal solution is the one with minimal cost.
- $Q \geq n$. For this particular case, the capacity constraints are redundant. Therefore, an optimal solution can be found by solving the associated MST problem. As we have mentioned, this is achieved using a greedy algorithm such as Kruskal's or Prim's.
- In the general case (non-UD) of the CMST, when the capacity is greater than or equal to the accumulated demand of all the terminals ($Q \geq d(V^+)$), an optimal solution is also obtained by computing an MST. Note that having $Q \geq d(V^+)$ in the general case, is equivalent to have $Q \geq n$ in th UD case.

Some well know extensions of the CMST are:

- The *multilevel capacitated minimum spanning tree* considers a feasible set of capacities available for the arcs (Gamvros, Golden, and Raghavan 2006; Uchoa, Toffolo, de Souza, Martins, and Fukasawa 2012). The cost of each arc depends on its length and the selected capacity, which needs to be greater than or equal to the flow passing through the arc.
- The *topological network design problem* (Gavish 1989) differs from the CMST in the objective function, which considers both, setup and traffic costs for the arcs.
- The *delay constrained capacitated minimum spanning tree* (Lee and Atiquzzaman 2005) considers capacity constraints as well as traffic and delay constraints over the s-trees.
- In this work we deal with deterministic data. Indeed, another extension of the CMST, is the one that considers uncertainty on the demand of each vertex and on the cost of the arcs (Öncan 2007).

Also, as an extension of the MST, the CMST is related to many other problems. Among them we can mention the following:

- The *K-minimum spanning tree* is to find an spanning tree which connects at least k vertices of set V at minimum cost (Katoh, Ibaraki, and Mine 1981).
- The *degree-constrained minimum spanning tree problem* (Narula and Ho 1980), includes constraints on the number of edges incident with any non-leaf vertex. The *min-degree constrained minimum spanning tree* (de Almeida, Martins, and de Souza 2012) is an extension of the degree- constrained minimum spanning tree problem, in which non-leaf vertices are required to have a degree greater than a previously defined degree.
- The *hop constrained minimum spanning tree problem* (Balakrishnan and Altinkemer 1992; Gouveia 1995b), includes constraints to limit the maximum number of hops allowed when designing the network to connect all vertices. The depth is defined as the number of arcs in the path from the root to any terminal.
- The *Minimum spanning tree with conflict constraints* (Zhang, Kabadi, and Punnen 2011) includes additional conflict constraints among pairs of vertices on the graph. The solution has to include at most one edge linking each pair of vertices with conflicts.

1.2 Literature review

1.2.1 Formulations and exact methods

The first exact methods developed for solving the CMST were enumeration procedures based on combinatorial bounds. Many of these branch and bound methods are based on the relaxation of the capacity constraints of the CMST. To avoid confusions, in this section the term node is related to a search tree in a branch and bound procedure, while vertex is related to the graphs in which combinatorial problems are defined. Also it is important to mention, that many of the first solution methods were developed to solve the UD case of the CMST. Unless specified, the solution methods or formulations presented in this section were proposed for the UD case.

The MST is a well known relaxation of the CMST where capacity constraints are not considered. The main reason to use this relaxation is the ability of algorithms like Kruskal's (Kruskal 1956), and Prim's (Prim 1957) to optimally solve the problem in a short computing time. In these algorithms the search tree is explored focusing either on arcs or vertices. For the arc oriented algorithms enumeration is done by fixing arcs (i, j) which are included in $(x_{ij} = 1)$ or excluded from the solution $(x_{ij} = 0)$. For the vertex oriented algorithms vertices are subject to a clustering process, which determines whether or not a pair of vertices is in the same cluster. In both of these approaches a MST is solved at every node of the search tree. When the solution of such MST is feasible for the CMST, it is optimal for the current subproblem and therefore such node is discarded for further branching. Chandy and Russell (1972) presented an arc oriented branch and bound in which the branching process starts using the infeasible arc (an arc that violates the capacity constraint) closest to the root node. Every time a variable is fixed, the cost matrix is modified. As explained before, nodes of the search tree are discarded for further branching when the MST solution of the subproblem is feasible for the CMST. The cost matrix is updated taking into account the fixed variables. Later, Elias and Ferguson (1974) improved the algorithm introducing logical tests, dominance criteria and improved lower bounds. Logical tests allow to reduce the number of variables, while the lower bound is improved adding a degree constraint on the root node. Dominance criteria are used to fix certain arcs included in the initial solution. After fixing those arcs, the enumeration process starts. As in Chandy and Russell (1972) the cost matrix is updated considering the fixed variables.

The first vertex oriented branch and bound algorithm was proposed by Chandy and Lo (1973) and is based on Litlle's algorithm (Little, Murty, Sweeney, and Karel 1963) for the Travelling Salesman Problem (TSP). The subproblems are branched on by considering pairs of vertices which either belong to the same subtree or not. Another vertex oriented algorithm was developed by Kershenbaum and Boorstyn (1983) based on the construction of subtrees. The algorithm initially considers every terminal of the graph as a subtree. Branching is done using a LIFO policy. Using the same approach of subtree construction, an arc oriented branch and bound algorithm is also presented. Another arc oriented algorithm was designed by Han, McMahon, and Sugden (2002) in which arcs are sorted according to their cost and a binary search engine is used to explore the search tree.

Many formulations have been proposed to model the CMST. Among the mixed integer formulations we can mention the flow formulation by Chandy and Russell (1972). This formulation has been extensively used to develop exact algorithms and also as a base for other types of formulations. Using this formulation Gavish (1982) proposed a Benders decomposition approach with disappointing results, although, a new set of generalized cut constraints was introduced. This family was equivalent to the well-known generalized subtour elimination (GSE) constraints used for the the TSP. Better results were obtained by Gavish (1983) using a Lagrangean relaxation with a modified version of this formulation. In that work, the Lagrangean relaxation dualizes the capacity constraints and includes a degree constraint for the root, which improves the lower bound and thus the results of the algorithm. A Dantzig-Wolfe approach, a linear relaxation and a subgradient optimization procedure are compared. To compare the performance of the different approaches, the heuristic by Elias and Ferguson (1974) is used to find upper bounds. Using the flow formulation as a base, other formulations and solution methods have been developed, like the single commodity flow formulation proposed by Gavish (1989). Also, Frangioni, Pretolani, and Scutellà (1999) used the flow formulation to model the CMST as a minimum cost flow problem. Lower bounds are found in a very short computing time using Lagrangean relaxation, although the quality of the bounds is not good.

Among the binary formulations for the CMST, one of the most important ones, is the directed formulation by Gavish (1985). This integer formulation included a new set of constraints equivalent to the GSE constraints (derived from (Gavish 1982)). These constraints are used for controlling the connectivity and the capacity constraints on the solution. The above constraints are relaxed in a Lagrangean fashion and the resulting model is solved with an augmented Lagrangean procedure. A dual ascent algorithm is used to identify violated constraints. The results showed small optimality gaps and fairly good CPU times. Malik and Yu (1993) introduced new valid inequalities for this formulation, which were separated using a heuristic. These inequalities are stronger than those of Gavish (1982). Lagrangean relaxation was applied to these inequalities and the resulting problem was solved using the subgradient method. Toth and Vigo (1995) also proposed a Lagrangean relaxation strengthened in a cutting plane fashion by iteratively adding violated constraints to the Lagrangean problem. Additionally, they presented another lower bound procedure based on the solution of minimum cost flow problems. They also proposed an overall additive lower bounding procedure, which was included within a branch and bound algorithm.

A branch and bound algorithm was developed by Hall (1996) using the undirected version of the formulation by Gavish (1985). In this formulation arcs $a_{ij} = (i, j)$ are replaced by edges $e_{ij} = (i, j)$. The algorithm was used to solve instances for the general case as well as for the UD case. It incorporates a cutting plane method in which multistar and rootcutset inequalities are incorporated. These strong inequalities are separated using heuristic methods based on the contraction of vertices over the fractional graph induced by the solution of the LP relaxation of the model. The used valid inequalities are based on the previous polyhedral study of Araque, Hall, and Magnanti (1990), where it is shown that under certain conditions, multistar and rootcutset inequalities are facet defining. Zhang (1993) proved similar results but for the directed version of such inequalities. A review of directed formulations at that time can be found in Gouveia (1993).

Based on the flow formulation, Gouveia (1995a) developed the *$2n$-formulation* for the CMST. Using capacity indexed variables and following the idea of the flow conservation rules, Gouveia presented a binary formulation with $2n$ constraints and Qn^2 variables. He also proposed valid inequalities and four solution methods based on Lagrangean relaxation. Two different types of test instances were introduced for the UD case. Such test instances became a standard, which started to be used by other researchers. Using the same benchmark instances has helped to compare the performance of the different algorithms. A complete description of those instances is given in Chapter 5. Gouveia and Martins (2000) presented another binary formulation, the hop indexed formulation, also based on the flow formulation. Valid inequalities based on the GSE inequalities were presented as well as a heuristic separation method. A cutting plane algorithm was used to solve the LP relaxation of the formulation obtaining fairly good results. A comparison between different types of formulations and methods was presented in that work. Using the same formulation, Gouveia and Martins (2005) enhanced their previous cutting plane algorithm using stronger valid inequalities as well as a new separation method that gave improved results. In Chapter 2 we present a more extensive review of such formulation.
A different formulation for the CMST, *the q-arb formulation* proposed by Uchoa, Fukasawa, Lysgaard, Pessoa, de Aragao, and Andrade (2008), is based on a type of arborescence structures called *q-arbs* which are used to define the variables. The model is solved using a branch-and-cut-and-price algorithm. Using this algorithm, many of the best known solutions of the test instances were improved or confirmed to be optimal. To the best of our knowledge, at present, such algorithm still produces the best results among all exact methods.

1.3 Heuristic methods

Because of the theoretical and empirical difficulty of the CMST, many approximate solution methods have been developed to find upper bounds for the problem. Many of these methods include heuristics, metaheuristics and other approaches that have shown to be able to find good feasible solutions for the CMST.

Among the different heuristics that have been developed for the CMST we can distinguish two types: Constructive heuristics, which build a solution, and improvement heuristics, which start with a given solution and try to improve it. Among constructive algorithms we can mention the modified versions of Prim's, Kruskal's and Vogel's approximation algorithm

for the MST (Chandy and Russell 1972; Kershenbaum 1974). Modified Prim and modified Kruskal algorithms work as their original version, but include capacity checks to avoid the inclusion of edges that violate the capacity constraint. Vogel's approximation algorithm works on a cost matrix which is modified using a trade-off function that considers the first and second closest vertex of every terminal.
The above mentioned construction algorithms focus on arcs. In contrast, Sharma and El-Bardai (1970) and McGregor and Shen (1977) designed construction algorithms focused on vertices. The procedure in Sharma and El-Bardai (1970) uses polar coordinates to create clusters of vertices. Polar coordinates are estimated using the root as the main reference. A greedy method considering capacity constraints is used to build the solutions. McGregor and Shen (1977) proposed a vertex contraction procedure. When two vertices are selected to be linked by an edge such vertices are contracted and replaced by one new vertex and distances to other vertices are recomputed. Infeasible contractions of vertices are not allowed. Kershenbaum (1974) designed a heuristic based on Kruskal's modified algorithm, that assigns weights to terminals. These weights are used to compute a modified cost matrix which is used to build solutions.

One of the most important heuristic, among the improvement ones, is the one by Esau and Williams (1966). The initial solution for this procedure is the star tree (all terminals are subroots) which is subject to improvement using a savings strategy. It is simple and fast, and the quality of solutions is good. Due to the algorithm's efficiency and simplicity, many other authors have developed enhanced versions. Whitney's savings heuristic (1970) is very similar to the Esau-Williams algorithm. The starting solution is the same and the only difference lies in the savings strategy criterion used in the improvement phase. While Esau-Williams considers for replacement only the edges directly connected to the root, Whitney's procedure considers all the edges in the current solution. The algorithm by Lee and Atiquzzaman (2005) generates an initial solution using Esau-Williams and then applies a local search based on vertex exchange. Battarra, Öncan, Altinel, Golden, Vigo, and Phillips (2011) presented a genetic algorithm for setting the parameters of the Esau-Williams algorithm.

Gavish and Altinkemer (1986) proposed a parallel savings heuristic also based on Esau-Williams. In this heuristic multi-exchange of edges is allowed while in Esau-Williams only a pair of edges is exchanged at each step. Unfortunately, this algorithm has the tendency to produce solutions with a big number of s-trees with wasted capacity. For that reason Gavish (1991) introduced dummy vertices to cleverly control the maximum number of matches allowed at each iteration and thus reduce the number of s-trees with wasted capacity.

Another important and natural improvement heuristic is to compute the MST of the s-trees which are part of the solution obtained by a construction algorithm. Esau and Williams (1966) and Whitney (1970) used such an approach to improve the solution when their algorithm is unable to find further improvement. This improvement procedure can be applied to any feasible solution.

The improvement heuristic by Elias and Ferguson (1974), also based on Esau-Williams, starts using the MST solution of the problem instead of the star tree. The initial solution is infeasible (if it is feasible, then it is optimal) and unused edges replace edges in the current solution. Instead of using a savings strategy a least cost strategy is used. The algorithm stops when a feasible solution is found.

Karnaugh (1976) designed a second order greedy algorithm for the CMST. Second order algorithms create subproblems modifying the original model (changing data or including constraints), which are solved using a heuristic procedure. In that work the cost matrix is perturbed at each step and the resulting subproblem is solved with a modified version of Esau-Williams. An inhibition procedure is also included to help in the diversification of solutions. Martins (2007) proposed an enhanced second order algorithm based on Esau-Williams. The difference with a second order algorithm, is that the enhanced version is able to drop some of the previously defined constraints to find improved solutions. Dropped constraints are included in a tabu list to avoid local optima.

As metaheuristics started to appear, many were applied to the CMST. One of the first attempts was done by Amberg, Domeschke, and Vob (1996) who used simulated annealing and tabu search. Their algorithm considers two neighborhoods: Vertex exchange and vertex shifting. Using also tabu search Sharaiha, Gendreau, Laporte, and Osman (1997) developed a procedure where the explored neighborhood is related to subtrees. Infeasible solutions (according to capacity) are allowed and an aspiration criterion is also applied. Ahuja, Orlin, and Sharma (2001) developed another tabu search algorithm that included a new type of neighborhood search structures. The new structures are based on neighborhoods previously defined in Amberg, Domeschke, and Vob (1996) and in Sharaiha, Gendreau, Laporte, and Osman (1997) and include vertex shifting, vertex exchange and subtrees shifting. The difference lies in the fact that the neighborhoods are explored using a cyclic multi-exchange strategy. Using a customized improvement graph, the algorithm is able to find cyclic multi-exchanges to improve the solution. Further improvements of this algorithm were carried out in Ahuja, Orlin, and Sharma (2003) by creating a composite very large scale neighborhood in which cyclic multi-exchange improvements are found. To the best of our knowledge, the best heuristic results are obtained by this algorithm, which was applied both to UD and non-UD instances. Combining tabu and scatter search, Fernández, Díaz, and Ruiz (2005) designed a heuristic in which initial solutions are generated using a randomized version of Esau-Williams. Two arc related neighborhoods are explored during the tabu search and solutions are combined during the scatter search phase to create a support graph, which is transformed into a CMST solution by a vertex contraction procedure.

Patterson, Pirkul, , and Rolland (1999) solved the CMST using the adaptive reasoning technique. This algorithm introduces special strategies to exploit construction neighborhoods combined with some memory functions of tabu search. The procedure is based on Esau-Williams with additional constraints that help to avoid local optima. GRASP was applied by Souza, Duhamel, and Ribeiro (2003) on the CMST. Their GRASP algorithm explores an edge exchange neighborhood and uses Esau-Williams to improve solutions.

Genetic algorithms have also been used on the CMST, although with not so good results. One of the problems faced by genetic algorithms is the way solutions are coded for crossover. The predecessor coding, which is a common way to represent CMST solutions, has the difficulty that crossover can lead to infeasible offsprings. To overcome this problem Zhou, Cao, Cao, and Meng (2007) used a different encoding. On the other hand, Raidl and Drexel (2000) and de Lacerda and de Medeiros (2006) used complicated crossover techniques to obtain feasible solutions.

Ant colony optimization (ACO) has been used by Reimann and Laumanns (2006), who developed a fast procedure to solve the UD case. Using a TSP savings procedure, solutions for the capacitated vehicle routing problem (CVRP) are built and used to create clusters of vertices which are transformed into CMST solutions using Prim's algorithm.

A dual-primal relaxation adaptive memory programming heuristic was proposed by Rego, Mathew, and Glover (2010) to solve the CMST. This type of algorithm explores the dual solution space by incorporating cutting planes of relaxed surrogate constraints. Using tabu search and projecting dual solutions into the primal space, primal feasible solutions are obtained. Further improvement is accomplished by using a scatter search procedure.

Rego and Mathew (2011) proposed a filter and fan algorithm that combines local search with tree search procedures. Diversification of the solutions is achieved by a fan procedure and a filter procedure helps to reduce the size of the search tree. The filter and fan algorithm makes use of tabu search adaptive memory as well as strategic oscillation.

In addition, there are other types of approaches for finding bounds on the solution of the CMST like the one introduced by Gouveia and Paixão (1991). Using clustering and decomposition techniques the original problem is transformed to obtain smaller size problems, which are solved using dynamic programming. Altinkemer and Gavish (1988) designed a heuristic algorithm based on the partitioning of a Hamiltonian tour that provides constant error guarantees for the bounds.

Chapter 2

Formulations and valid inequalities for the CMST

In this chapter we give some preliminaries for the work we have developed in this thesis. First we present two basic formulations that use binary variables, which allow us to highlight some of the main issues related to the CMST. The chapter ends with an overview of the most used valid inequalities and their separation. Again, the aim is not to be exhaustive but to give the basis of the developments on this work.

2.1 Two basic binary formulations

When modeling the CMST there are two important issues that need to be considered; connectivity of the tree and capacity constraints. An easy way to address these issues is the use of flow variables and flow conservation constraints. With such constraints we assure connectivity, while capacity issues are solved setting up an upper limit on the value of the flow variables. The formulations based on flow variables are mainly mixed integer programming (MIP) models as flow variables are essentially continuous. A common alternative to model the CMST is the use of integer (binary) variables associated with arcs in conjunction with cutset inequalities that guarantee connectivity and capacity constraints.

Experience has shown that the linear relaxations (LP) of binary formulations provide tighter lower bounds than those of MIP models. Unfortunately, in binary formulations the number of constraints is exponential on the number of vertices of the problem and it is impossible to solve the CMST using the full formulation. Therefore, these constraints need to be relaxed and iteratively separated and added to the formulation.

Formulations are also influenced by the CMST variant that is studied. The UD case allow for tighter formulations with the inconvenience that they are not useful or need to be adapted to solve the general case. Here we present two formulations, the first one considers the general demand case while the second one considers only the UD case. The first formulation is the classical one, commonly used by researchers to define the CMST. The second one is a three index formulation, which has been useful to develop part of the research presented in this thesis.

To facilitate comprehension through this and the next chapter in some situations we will abuse notation using:

$$\sum_{i\in V}\sum_{j\in V} x_{ij} \text{ to express } \sum_{\substack{(i,j)\in A\\ i\in V, j\in V}} x_{ij}. \tag{2.1}$$

2.1.1 Directed formulation

A classical mathematical programming formulation for the CMST is the one proposed by Gavish (1983). In this directed formulation, arcs leave from the root towards the leaves. This formulation includes a set of inequalities based on the general subtour elimination constraints (GSE) (Malone and Bellmore 1971) to guarantee connectivity and capacity limit.

The decision variables are the following:

$$x_{ij} = \begin{cases} 1, & \text{if arc } (i,j) \text{ is part of the solution,} \quad (i,j)\in A \text{ with } j\in V^+ \\ 0, & \text{otherwise.} \end{cases}$$

Then, the $CMST$ is formulated as:

$$\min \sum_{i\in V}\sum_{j\in V^+} c_{ij}x_{ij} \tag{2.2}$$

$$\text{s.t.} \sum_{(i,j)\in A} x_{ij} = 1 \qquad j\in V^+ \tag{2.3}$$

$$\sum_{i\notin S}\sum_{j\in S} x_{ij} \geq \left\lceil \frac{d(S)}{Q} \right\rceil \qquad S\subseteq V^+, |S|\geq 2 \tag{2.4}$$

$$x_{ij}\in\{0,1\} \qquad (i,j)\in A \text{ with } j\in V^+ \tag{2.5}$$

With equations (2.3) we guarantee that for every vertex there is only one incoming arc. Constraints (2.4) are the classical connectivity constraints, as extended by Laporte. and Nobert (1983) for the Vehicle Routing Problem (VRP) to take capacities into account. Also called bin-packing inequalities, with these constraints the connectivity of the solution is assured and the accumulated demand of the subtrees is limited so each subtree does not exceed the capacity in any solution. Recall that $d(S) = \sum_{j\in S} d_j$ denotes the overall demand of the vertices of set S.

The above formulation has n^2 variables, and a number of constraints, which is exponential on $|V^+|$. Note that there are some variables which are not defined (i.e. arcs going from terminal vertices to the root). Additionally, it is possible to eliminate some other variables based on optimality criteria. Variants of this formulation have been also studied, like the non-directed version (Hall 1996).

2.1.2 Hop indexed formulation

This formulation was presented in Gouveia and Martins (2000), and improved in Gouveia and Martins (2005). It is based on the single commodity network formulation by Gavish (1983) and applies only to the UD case. It takes into account the fact that every arc has a certain depth in relation to the root. Consider an arc (i, j) with depth t which is part of the solution. If $t = 1$, we have that such arc has as origin the root $(i = 0)$ and as destination the subroot j. When $t = 2$ the arc (i, j) is originated at the subroot i and arrives to vertex j, which is neither the root nor a subroot. Arcs with $t \geq 3$, link vertices which are neither the root nor a subroot. Notice that the depth of a vertex is determined by the depth of its incoming arc (Figure 2.1).

The following variables are defined:

$$u_{ijt} = \begin{cases} 1, & \text{if arc } (i,j) \text{ with depth } t \text{ is in the solution, } (i,j) \in A \text{ with } j \in V^{+}, t = 1, ..., Q \\ 0, & \text{otherwise} \end{cases}$$

$z_{ijt} \geq 0,$ The flow passing through arc (i, j), $(i, j) \in A$ with $j \in V^{+}, t = 1, ..., Q$ when this arc has depth t

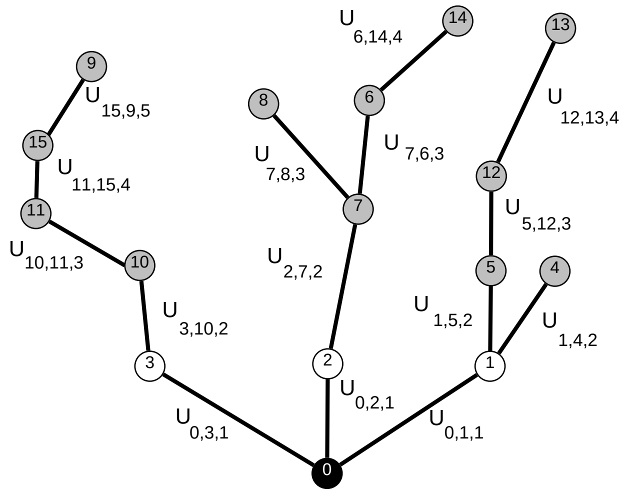

Figure 2.1: Example of hop indexed variables

It is important to notice that index t for variables z and u, has values ranging from 1 to Q because no feasible solution would have any arc with depth greater than Q. Note that, since this formulation is used to model the UD case, the capacity constraints are in fact cardinality constraints. Therefore, in this situation the capacity gives directly the maximum depth in a subtree.

Then using the previously defined variables, the hop indexed formulation is as follows:

$$\text{Min} \quad \sum_{t=1}^{Q} \sum_{(i \in V} \sum_{j \in V^+} c_{ij} u_{ijt} \tag{2.6}$$

$$\text{s.t.} \quad \sum_{t=1}^{Q} \sum_{(i,j) \in A} u_{ijt} = 1 \qquad j \in V^+ \tag{2.7}$$

$$\sum_{i \in V} z_{ijt} - \sum_{i \in V} z_{ji,t+1} = \sum_{i \in V} \sum_{t=1}^{Q} u_{ijt} \qquad j \in V^+, t = 1, ..., Q-1 \tag{2.8}$$

$$u_{ijt} \leq z_{ijt} \leq (Q - t + 1) u_{ijt} \qquad (i,j) \in A, j \in V^+, t = 1, ..., Q-1 \tag{2.9}$$

$$u_{ijt} \in \{0,1\}, \quad z_{ijt} \in 0,..,Q \qquad (i,j) \in A,\ t \in 1, ..., Q \tag{2.10}$$

Constraints (2.7) are the classical ones that require there is exactly one incoming arc to every terminal and they are equivalent to (2.3) for the directed formulation. Constraints (2.8) are a generalization of the flow balance equations and also state that, if an arc (l, j) with depth t enters j, any outgoing flow would use some arc with depth $t + 1$. Finally constraints (2.9) link variables and limit capacity.

This formulation has $2Qn^2$ variables. As in the directed formulation, there are variables which is possible to eliminate based on optimality criteria.

2.2 Well-known valid inequalities

Many valid inequalities have been proposed for the different formulations for the CMST. Among the most important ones, we can mention the bin-packing inequalities (Hall 1996), multistar inequalities (Hall 1996; Gouveia and Lopes 2005), rootcutset inequalities (Araque, Hall, and Magnanti 1990), other based on the GSE inequalities (Gouveia and Martins 2005), and extended capacity cuts (Uchoa, Fukasawa, Lysgaard, Pessoa, de Aragao, and Andrade 2008). We will use the directed formulation to present these inequalities, except for the ones derived from another type of formulation. It is important to notice that many of these inequalities can be extended to different formulations. Let P and P_d denote the convex hull of feasible solutions to the undirected and directed CMST formulations respectively.

2.2.1 Bin-packing inequalities

Bin-packing inequalities apply both to the general and UD cases, and can be expressed as connectivity constraints, or as subtour elimination constraints. These inequalities have been widely used in vehicle routing and network design problems. An example of these inequalities is presented in Figure 2.2 considering $Q = 5$. Their expression for the general case as connectivity constraints, was given in (2.4) for the directed formulation. Their expression as a subtour elimination constraint is:

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq d(S) - MT(S) \qquad S \subseteq V^+, |S| \geq 2 \tag{2.11}$$

where $S \subseteq V^+$ is a subset of terminals and $MT(S) = \left\lceil \frac{d(S)}{Q} \right\rceil$ represents the minimum number of s-trees (or the minimum number of incoming arcs) required to accommodate the demand of the vertices contained in subset S, taking into account the capacity constraint. For the UD case we have that $MT(S) = \left\lceil \frac{|S|}{Q} \right\rceil$ as $d(S) = |S|$. For the same case, these inequalities are facet defining for P and P_d whenever $|S|$ is not a multiple of Q (Hall 1996), and (2.11) can be rewritten as:

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - MT(S) \qquad S \subseteq V^+, |S| \geq 2 \tag{2.12}$$

or alternatively as a connectivity constraints:

$$\sum_{i \notin S} \sum_{j \in S} x_{ij} \geq MT(S) \qquad S \subseteq V^+, |S| \geq 2 \tag{2.13}$$

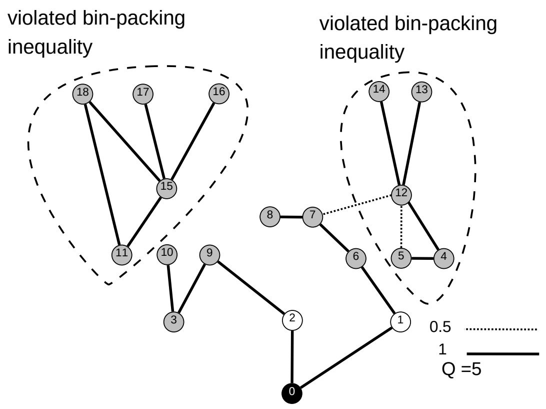
violated bin-packing inequality
violated bin-packing inequality
18
17
16
15
11
10
9
3
2
0
14
13
12
8
7
6
5
4
1
0.5
1
Q =5

Figure 2.2: Example of violated bin-packing inequalities with UD and Q=5

Note that in the UD case $MT(S) = \left\lceil \frac{|S|}{Q} \right\rceil$, because we have unitary demands. However, in the general case, computing $MT(S)$ resorts to solving a bin-packing problem, which is itself a NP-hard problem. The separation problem of this type of inequalities was solved heuristically by Hall (1996) using a vertex contraction procedure over the graph induced by the fractional solution.

2.2.2 Multistar inequalities

The multistar inequalities were first introduced by Araque, Hall, and Magnanti (1990) for capacitated tree and capacitated routing problems. These inequalities are capacity-connectivity constraints, that have different expressions depending on the type of demand considered on the terminals (UD case or general). In Figure 2.3, an example of this family of inequalities is presented. The example considers a capacity parameter of $Q = 5$. For the UD case these inequalities are facet defining under certain conditions (Hall 1996) and have the following form:

$$\sum_{i\in S}\sum_{j\in S} Qx_{ij} + \sum_{\substack{i\in S\\ i\neq 0}}\sum_{\substack{j\notin S\\ i\neq 0}}(x_{ij}+x_{ji}) \leq |S|(Q-1) \qquad S\subseteq V^+, |S|\geq 2 \tag{2.14}$$

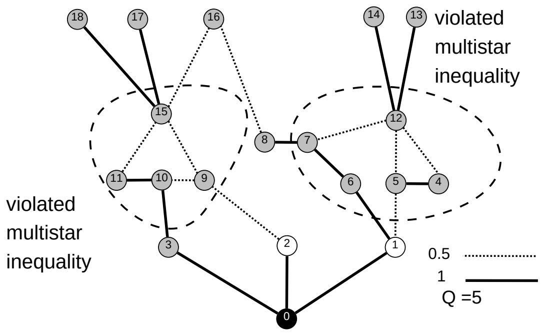

Figure 2.3: Example of violated multistar inequalities with UD and Q=5

The right hand side of (2.14) represents the maximum potential number of arcs between vertices in S and leaving/entering S. The first term in the left hand side considers the arcs between vertices in S and the second term refers to arcs leaving/entering S that do not come from the root. In total, if all the vertices in S are subroots, then at most $|S|(Q-1)$ arcs can leave S. If two vertices in S are connected by one arc, it is clear that one of such vertices is not a subroot. Therefore the number of potential outgoing arcs from S is reduced by Q. For the general demand case, Hall (1996) proposed a generalization based on the weighted version of (2.14). Gouveia and Lopes (2005) proposed another version of these inequalities as well as a reinforcement procedure to improve them. They are expressed as follows:

$$\sum_{i\notin S}\sum_{j\in S}(Q-d_i)x_{ij} - \sum_{i\notin S}\sum_{j\in S} d_i x_{ji} \geq d(S)$$

The left side of these inequalities represents the available capacity entering the set of vertices in S. This capacity has to be enough so as to satisfy the sum of the demands of the vertices in S plus the demand of the vertices outside S directly connected with an outgoing arc from S.

To separate these inequalities, Hall (1996) used a vertex contraction procedure over the graph induced by the fractional solution. Letchford and Salazar-González (2006) presented an exact separation procedure for these inequalities. The exact separation is achieved by creating an auxiliary graph and then solving a maximum flow-minimum cut problem.

2.2.3 Rootcutset inequalities

The rootcutset inequalities were first proposed by Araque, Hall, and Magnanti (1990). Hall (1996) developed the version of these inequalities for the UD case while Zhang (1993) presented them for the general case. These inequalities help to enforce the connectivity of the subtrees to the root. For a given set $S \subseteq V^+$, two types of incoming arcs (or edges) to S are considered: subroot arcs between the root and any subroot, and arcs between terminals. The version of these constraints for the UD case considers a set of vertices $|S| \geq Q$, $MT(S)$ that as before represents the minimum number of s-trees needed to contain the vertices in S, and $b = max\{0, |S| - MT(S)(Q-1)\}$ the maximum number of saturated s-trees in S. Their expression is as follows:

$$\sum_{i \in S} \sum_{j \in S} b x_{ij} \leq b(|S| - MT(S) - 1) + \sum_{j \in S} x_{0j} \tag{2.15}$$

The constant on the right hand side of inequalities (2.15), represents the maximum number of arcs (or edges) that can link vertices in subset S assuming that there are no subroots in S. For each subroot contained in S, the number of potential arcs linking vertices in S is increased in one unit (variable part of the right hand side).

For the general case inequalities (2.15) can be extended to:

$$\frac{MT(S)+1}{MT(S)} \sum_{i \in \bar{B}} \sum_{j \in S} x_{ij} + \sum_{i \in B} \sum_{j \in S} x_{ij} \geq MT(S) + 1 \tag{2.16}$$

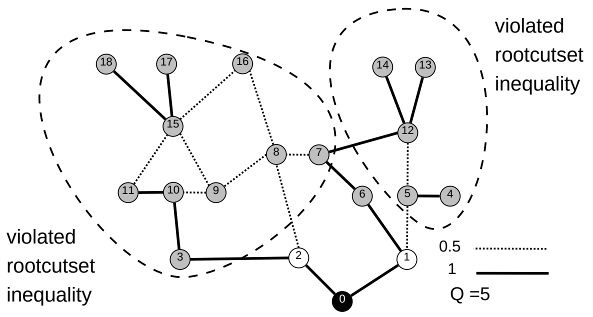

Figure 2.4: Example of violated rootcutset inequalities with UD and Q=5

where $B = \left\{ i \notin S \mid \left\lceil \frac{d(S)+d_i}{Q} \right\rceil > MT(S) \right\}$ is the set of vertices not in S, that if included in S, would require an additional s-tree, and $\overline{B} = V^+ \setminus \{S \cup B\}$ the set of vertices that would not. As in the UD case, the connectivity to the root is enforced with this family of inequalities.

These inequalities can be separated heuristically using a vertex contraction procedure over the graph induced by the fractional solution (Hall 1996). To the best of our knowledge, no exact separation method is available for these inequalities. An example for this family of inequalities is presented in Figure 2.4 considering $Q = 5$.

2.2.4 GSEh inequalities

Another set of valid inequalities are the GSEh inequalities from Gouveia and Martins (2005) that are based on the well known GSE inequalities. These inequalities only apply to the hop indexed formulation by Gouveia and Martins (2000) presented in Section 2.1.2. Their expression is the following:

$$\sum_{t\geq h+1}\sum_{i\in S}\sum_{j\in S}u_{ijt}\leq |S|-\left\lceil\frac{|S|}{Q-h+1}\right\rceil \qquad S\subseteq V^{+},|S|\geq 2,1\leq h\leq Q-1 \qquad (2.17)$$

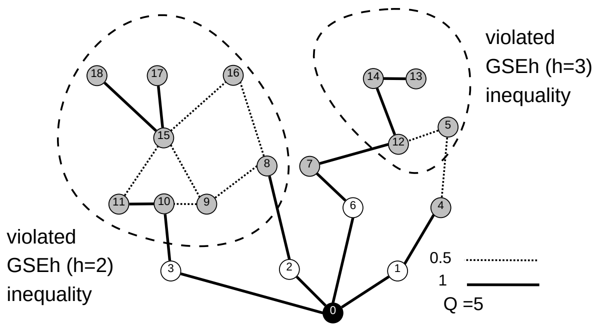

Figure 2.5: Example of violated GSEh inequalities with UD and Q=5

Since variables in (2.17) indicate the depth of the arc, we can use these inequalities for finding GSE inequalities at different depths h, and consequently, enhance the connectivity in the solution. The right hand side of the expression represents the maximum number of arcs considering the cardinality of S ($|S|$) at a fixed depth h. In Figure 2.5, an example of this family of inequalities is presented.

Gouveia and Martins (2005) separate heuristically inequalities GSEh using a drop strategy.

Hop ordering are a type of inequalities specially designed for the hop indexed formulation of Gouveia and Martins (2005). These inequalities state that an arc (i, j) can be at depth $t + 1$ only if there is an entering arc (m, i) at depth t. They are expressed as follows:

$$\sum_{\substack{m\in V\\ m\neq j}} x_{mit}\geq u_{ij,t+1} \qquad (i,j)\in A, t=1,..,Q-1 \qquad (2.18)$$

and they can be separated by enumeration using the graph induced by the fractional solution.

Finally, we will mention that Uchoa, Fukasawa, Lysgaard, Pessoa, de Aragao, and Andrade (2008) introduced a new family of inequalities called extended capacity cuts (ECC). They use the capacity indexed variables proposed by Gouveia (1995a). The rootcutset inequalities are a subset of the ECC. Another subset of this family of inequalities are the homogeneous extended capacity cuts (HECC). The HECC only consider entering variables with the same capacity having the same coefficient.

Chapter 3

New formulations for the CMST

In this chapter we present two new formulations for the CMST which are based on the identification of subroots (vertices directly connected to the root). One way of characterizing CMST solutions is by identifying the subroots and the vertices assigned to them. Both formulations use binary decision variables y to identify the subroots. Additional decision variables x are used to represent the elements (arcs) of the tree. In the second formulation the set of x variables is extended to indicate the depth of the arcs in the tree. For each formulation we present families of valid inequalities and address the separation problem in each case. Also a solution algorithm is proposed.

The type of variables used in both formulations are easy to understand and the identification of subroot gives to both formulations the capacity of building stronger inequalities which can lead to improved lower bounds. Additionally, this type of variables have similarities with the ones used in location-assigment problems, since they identify the subroots and the vertices assigned to those subroots.

3.1 The subroot formulation

This formulation is based on the subroots and the s-trees which are subtrees rooted at a subroot. Assignment variables y help us to identify which vertices belong to each s-tree, and link each terminal $j \in V^+$ to a subroot $k \in V^+$. Additionally, variables x, related to the arcs, specify whether an arc is part of an s-tree rooted at a given subroot. Because it is assumed that the graph of the problem is a network without loops, variables x are not defined when $i = j$. Also, since we consider that flow goes from the root to the terminals, arcs going from terminal to the root are not defined either. Now we can define the following variables for the subroot formulation:

$$y_{jk} = \begin{cases} 1, & \text{if vertex } j \text{ belongs to the } s\text{-tree rooted at vertex } k, \quad j,k \in V^+ \\ 0, & \text{otherwise.} \end{cases}$$

$$x_{ijk} = \begin{cases} 1, & \text{if arc } (i,j) \text{ is part of the } s\text{-tree rooted at vertex } k, \quad (i,j) \in A, \quad j,k \in V^+ \\ 0, & \text{otherwise.} \end{cases}$$

Then the CMST can be formulated as follows:

$$
\begin{aligned}
\text{Min} \quad & \sum_{k\in V^+}\sum_{i\in V}\sum_{j\in V^+\setminus\{k\}} c_{ij}x_{ijk} && && (3.1)\\
\text{s.t.} \quad & \sum_{k\in V^+} y_{jk} = 1 && j \in V^+ && (3.2)\\
& \sum_{(i,j)\in A} x_{ijk} = y_{jk} && j,k \in V^+ && (3.3)\\
& y_{kk} = x_{0kk} && k \in V^+ && (3.4)\\
& \sum_{j\in V^+, j\setminus\{k\}} d_j y_{jk} \le (Q-d_k)y_{kk} && k \in V^+ && (3.5)\\
& \sum_{k\in V^+}\sum_{i\notin S}\sum_{j\in S} x_{ijk} \ge \left\lceil \frac{d(S)}{Q} \right\rceil && S \subseteq V^+ && (3.6)\\
& y_{jk} \in \{0,1\} && j \in V^+,\, k \in V^+ && (3.7)\\
& x_{ijk} \in \{0,1\} && (i,j) \in A,\, k \in V^+ && (3.8)
\end{aligned}
$$

With (3.2) we assign each vertex to a subroot, while with equation (3.3) we impose that each vertex has an entering arc part of the s-tree that the vertex is assigned to. Constraints (3.4) require a subroot to be directly connected to the root. In fact such constraints can be removed by replacing variables x_{0kk} by variables y_{kk}. The capacity is controlled by constraints (3.5) and the connectivity of the solution is guaranteed by (3.6), the extended version of the generalized connectivity constraints (2.4) presented in Section 2.1.1.

3.2 Valid inequalities for the subroot formulation

Many families of the valid inequalities presented in Section 2.2 are valid for the directed formulation by Gavish (1985). Since the subroot formulation (3.1)- (3.8) is based on that formulation, those families can be extended to this new formulation. In this section we present the extended versions of the multistar and rootcutset families. Additionally, a new reinforced version of the multistar family is presented and its validity is proved.

3.2.1 Subroot multistar inequalities

For the CMST we consider that each vertex has a demand, and the sum of these demands is produced at the root which is the only production vertex. The total production at the root is given by $d(V^+)$. The demand for each vertex j travels through the arcs in the only path that connects j to the root. The flow through each arc (i,j) is the sum of the demands of the vertices in the tree that have arc (i,j) in the only path connecting them to the root.

The family of multistar inequalities, which was presented in Section 2.2, has shown to be important for the solution of the CMST and CVRP (Letchford, Eglese, and Lysgaard

2002). In this section we present a new family of inequalities which go beyond adapting the multistar inequalities to the subroot variables. The new inequalities have an additional term which considers extra demand associated to s-trees not considered in the original ones.

Notice that the multistar inequalities are a particular case of the inequalities presented in the paper by Letchford and Salazar-González (2006):

$$\sum_{i\in S}\sum_{j\in V\setminus S}(\bar{d}_{ij}x_{ij}-\underline{d}_{ij}x_{ij})\geq d(S), \tag{3.9}$$

where $\bar{d}$ and $\underline{d}$, respectively denote a lower and an upper bound of the flow entering j.

Proposition 1. *For any given set $S\subset V^+$ the following subroot multistar inequality is valid for the CMST:*

$$\sum_{k\in S}Qy_{kk}+\sum_{k\notin S}\sum_{j\in S}(Q-d_k)x_{kjk}+\sum_{k\notin S}\sum_{\substack{i\notin S\\ i\neq k}}\sum_{j\in S}(Q-d_k-d_i)x_{ijk}$$
$$-\sum_{k\notin S}\sum_{i\in S}\sum_{j\notin S}d_jx_{ijk}-\sum_{k\in S}\sum_{i\notin S}d_iy_{ik}\geq d(S) \tag{3.10}$$

Proof:

To prove validity we use auxiliary variables z_{ij}, associated with binary variables x_{ij}, representing the flow passing through the arc (i,j). We first present the flow balance equation for each terminal $i\in V^+$ using variables z_{ij} and then we substitute variables z_{ij} by lower and upper bounds defined in terms of variables x_{ijk} of the subroot formulation. For each vertex, the flow balance equation is: "incoming flow = outgoing flow + demand". That is:

$$\sum_{i\neq j}z_{ij}=d_j+\sum_{i\neq j}z_{ji} \tag{3.11}$$

Given a feasible solution (x,y) for formulation (3.1)- (3.8), let $K=\{k\in V^+|x_{0k}=1\}$ be the set of subroot vertices and $R=V^+\setminus K$ be the set of non-subroot vertices. We further partition the set of non-subroot vertices $R=R_1\cup R_2$, into $R_1=\{i\in R|\sum_{k\in K}x_{kik}=1\}$ which contains the vertices directly connected to a subroot, and $R_2=\{i\in R|\sum_{k\in K}x_{kik}=0\}$, which contains the vertices with no connection to a subroot.

1. *From (3.11), the flow balance equation for any subroot $k\in K$ is:*

$$z_{0k}=d_k+\sum_{i\neq k}z_{ki} \tag{3.12}$$

Variables z_{0k} in the previous equation are bounded above by Q. Additionally, for the flow variables associated with arcs leaving from a subroot k, the upper bound is given by $(Q-d_k)$. The lower bound of the flow leaving a subroot to a vertex i is given by d_i (the demand of the *destination vertex i). Then, we have the following inequality for the flow entering a subroot $k \in S$:*

$$z_{0k} \leq Q y_{kk} \tag{3.13}$$

2. *Now, let us consider the flow balance equation for the non subroot vertices $j \in R$:*

$$\sum_{i \neq 0, i \neq j,} z_{ij} = d_j + \sum_{i \neq j} z_{ji} \tag{3.14}$$

When $j \in R_1$, an upper bound on z_{ij} is $Q - d_k$, and when $j \in R_2$ an upper bound on z_{ij} is $(Q - d_k - d_i)$. For variables z_{ji} for $j \in R$ (the flow leaving j), a lower bound is d_i, the demand of the destination vertex i .

3. *Now, adding up all the constraints* (3.11) *associated with a subset of vertices $S \subset V^+$, $|S| \geq 2$ we have that:*

$$\sum_{i \notin S} \sum_{j \in S} z_{ij} = \sum_{j \in S} d_j + \sum_{j \in S} \sum_{i \notin S} z_{ji} \tag{3.15}$$

Considering that the vertices included in S belong to only one of the subsets K, R_1 or R_2, it is possible to separate the flow entering S and the flow leaving S depending on the type of vertex where the flow comes from. First, we split the flow entering S using equations (3.12) *and* (3.14)*, and obtain:*

$$\sum_{j \in K \cap S} z_{0j} + \sum_{i \in R_1 \setminus S} \sum_{j \in S} z_{ij} + \sum_{i \in R_2 \setminus S} \sum_{j \in S} z_{ij} = \sum_{j \in S} d_j + \sum_{i \notin S} \sum_{j \in S} z_{ji} \tag{3.16}$$

Then we split the flow leaving S according to the type of source vertex (whether or not it is a subroot) using (3.13) *and* (3.14)*:*

$$\sum_{k \in K \cap S} z_{0k} + \sum_{i \in R_1 \setminus S} \sum_{j \in S} z_{ij} + \sum_{i \in R_2 \setminus S} \sum_{j \in S} z_{ij} = \sum_{j \in S} d_j + \sum_{j \in K \cap S} \sum_{i \notin S} d_i y_{ij} + \sum_{j \in R \cap S} \sum_{i \notin S} z_{ji} \tag{3.17}$$

Finally, we replace variables z_{ij} with variables y_{jk} and x_{ijk}, using the upper bounds as coefficients for the terms on the left hand side of the equation, and using the lower bounds as coefficients in the the terms on the right hand side of the equation, and we obtain the following valid inequality:

$$\begin{aligned}
&\sum_{k \in S} Q y_{kk} + \sum_{i \in R_1 \setminus S} \sum_{j \in S} \sum_{k \in K \setminus S} (Q - d_k) x_{ijk} + \sum_{k \in K \setminus S} \sum_{i \in R_2 \setminus S} \sum_{j \in S} (Q - d_i - d_k) x_{ijk} \\
&\geq \sum_{j \in S} d_j + \sum_{k \in K \cap S} \sum_{j \notin S} d_j y_{jk} + \sum_{j \in R \cap S} \sum_{i \notin S} \sum_{k \in K \setminus S} d_i x_{jik}
\end{aligned} \tag{3.18}$$

which can be rewritten as (3.10). ■

Enhancing the subroot multistar inequalities

Proposition 2. *For any given set $S \subset V^+$ the following enhanced subroot multistar inequality is valid for the CMST:*

$$\sum_{k\in S}\sum_{j\in S} d_j y_{jk} + \sum_{k\notin S}\sum_{j\in S}(Q-d_k)x_{kjk} + \sum_{k\notin S}\sum_{\substack{i\notin S\\ i\neq k}}\sum_{j\in S}(Q-d_k-d_i)x_{ijk} - \sum_{k\notin S}\sum_{i\in S}\sum_{j\notin S} d_j x_{ijk} \geq d(S) \tag{3.19}$$

Proof:

The proof of the validity of these inequalities is similar to that of inequalities (3.10). *We only have to replace inequality* (3.13), *the flow entering the subroots in S, by the equality:*

$$z_{0k} = \sum_{j\in V^+} d_j y_{jk} \tag{3.20}$$

since the flow entering a subroot $k \in K$ should be equal to the demand of k plus the sum of the demands of the non-subroot vertices assigned to k. The resulting inequality is:

$$\begin{aligned}&\sum_{k\in S}\sum_{j\in V^+} d_j y_{jk} + \sum_{i\in R_1\setminus S}\sum_{j\in S}\sum_{k\in K\setminus S}(Q-d_k)x_{ijk} + \sum_{k\in K\setminus S}\sum_{i\in R_2\setminus S}\sum_{j\in S}(Q-d_i-d_k)x_{ijk}\\ &\geq \sum_{j\in S} d_j + \sum_{k\in K\cap S}\sum_{j\notin S} d_j y_{jk} + \sum_{j\in R\cap S}\sum_{i\notin S}\sum_{k\in K\setminus S} d_i x_{jik}\end{aligned} \tag{3.21}$$

which after some manipulation leads to:

$$\begin{aligned}&\sum_{k\in S}\sum_{j\in S} d_j y_{jk} + \sum_{i\in R_1\setminus S}\sum_{j\in S}\sum_{k\in K\setminus S}(Q-d_k)x_{ijk} + \sum_{k\in K\setminus S}\sum_{i\in R_2\setminus S}\sum_{j\in S}(Q-d_i-d_k)x_{ijk}\\ &\geq \sum_{j\in S} d_j + \sum_{j\in R\cap S}\sum_{i\notin S}\sum_{k\in K\setminus S} d_i x_{jik}\end{aligned} \tag{3.22}$$

that is equivalent to (3.19). ■

Note that we can derive inequalities (3.19) from inequalities (3.10) by replacing

$$\sum_{k\in S} Q y_{kk} \qquad \text{with} \qquad \sum_{k\in S}\sum_{j\in V^+} d_j y_{jk}$$

and since,

$$\sum_{j\in V^+} d_j y_{jk} \leq Q y_{kk}$$

it is clear that (3.19) is stronger than (3.10).

3.2.2 s-tree multistar inequalities

Using variables of the subroot formulation, another family of valid inequalities associated with s-trees can be derived. This family of inequalities arises from the disaggregation of inequalities (3.10).

Proposition 3. *For any given set $S \subset V^+$ and $k \in V^+ \setminus S$, the following s-tree multistar inequality is valid for the CMST:*

$$\sum_{j \in S}(Q - d_k)x_{kjk} + \sum_{\substack{i \notin S \\ i \neq k}} \sum_{j \in S}(Q - d_k - d_i)x_{ijk} - \sum_{i \in S}\sum_{j \notin S} d_j x_{ijk} \geq \sum_{j \in S} d_j y_{jk} \tag{3.23}$$

Proof: *The result follows since for any s-tree rooted in k the overall capacity Q must cover the demand of k, of all vertices directly connected to k and at a distance* 2 *or more from k. These are respectively the terms in the left hand side of inequalities* (3.23).

■

3.2.3 Subroot rootcutset inequalities

Using subroot variables a modified version of the rootcutset inequalities (2.16) is obtained and has the following form:

$$\begin{aligned} &\frac{MT(S)+1}{MT(S)}\sum_{k \in S} y_{kk} + \frac{MT(S)+1}{MT(S)}\sum_{k \notin S}\sum_{i \in \bar{B}}\sum_{j \in S} x_{ijk} \qquad S \subset V^+, |S| \geq 2 \\ &+ \sum_{k \notin S}\sum_{i \in B}\sum_{j \in S} x_{ijk} \geq MT(S) + 1 \end{aligned} \tag{3.24}$$

where $MT(S)$ is the minimum number of subroot vertices required to handle the accumulated demand of the vertices in S, $B = \left\{ i \notin S \mid \left\lceil \frac{d(S)+d_i}{Q} \right\rceil > MT(S) \right\}$ is the set of vertices outside S that if included in S would require the inclusion of an extra subroot arc, and $\overline{B} = V^+ \setminus \{S \cup B\}$.

3.2.4 Other inequalities

There are also other sets of valid inequalities for the subroot formulation. Here, we present a list of them, as well as a brief description:

- Minimum number of subroots constraint. Proposed by Elias and Ferguson (1974) this inequality imposes the minimum number of subroots required to handle the total demand of all vertices.

$$\sum_{k \in V^+} y_{kk} \geq \left\lceil \frac{d(V^+)}{Q} \right\rceil \tag{3.25}$$

- Inequalities to restrict the values of variables y_{jk}, using the upper bound defined by variables y_{kk}.

$$y_{jk} \leq y_{kk} \qquad k \in V^+, j \in V^+, j \neq k \tag{3.26}$$

- Inequalities to restrict values of variables x_{ijk}, using the upper bound defined by variables y_{ik} and y_{jk}. They state that arc (i,j) can only be part of the s-tree s-T_k only if both vertices are assigned to the s-tree in k.

$$x_{ijk} + x_{jik} \leq y_{ik} \qquad (i,j) \in A,\ i \in V^+, k \in V^+, i \neq k \tag{3.27}$$
$$x_{ijk} + x_{jik} \leq y_{jk} \qquad (i,j) \in A,\ j \in V^+, k \in V^+, j \neq k \tag{3.28}$$

- Constraints that relate the values of variables y_{jk} with those of variables associated with arcs leaving the subroot.

$$y_{jk} \leq \sum_{i \in V^+} x_{kik} \qquad k \in V^+, j \in V^+, j \neq k \tag{3.29}$$

3.2.5 Solution algorithm for the subroot formulation

Next we propose a cutting plane algorithm to for the CMST. It is an iterative algorithm in which each iteration considers a subproblem with only a subset of the constraints of formulation (3.1)-(3.8), and solves its linear programming (LP) relaxation. In addition, at each iteration of the algorithm, separation procedures are applied to identify inequalities of formulation (3.1)-(3.8), which are violated by the current LP solution. All violated inequalities found are incorporated into the current subproblem. When the separation procedures do not detect any violated inequalities, then a variable elimination procedure is applied. The algorithm stops when no cuts are found and no variable is eliminated, or after a certain number of iterations without improvement. A pseudocode for the algorithm is presented in Algorithm 1.
The initial subproblem includes constraints (3.2), (3.3), (3.5) and (3.25). Constraints (3.27) and (3.28) are not considered because they are dominated by s-tree inequalities (3.23). At each iteration we try to separate violated inequalities of types (3.10), (3.23) and (3.24).

Variables Elimination

The algorithm includes a two variable elimination procedures. The first is based on optimality criteria, while the second is based on a lower bound (LB), an upper bound (UB) and reduced costs. The LB is obtained computing a MST with the minimum subroot vertices constraint (3.25). The UB is obtained by using a heuristic method ($UB = h(V, A, d, c)$). Let T_{LB} and T_{UB} denote the trees which produce the bounds LB and UB, respectively.
Using an optimality criterion, variables $x_{ijk}, \quad i \in V, j \in V^+, k \in V^+ \quad$ such that $\quad c_{ij} \geq c_{0j}$ are eliminated. The reason is that, if the arc (i,j) is part of an optimal solution, there is an alternative optimal solution that does not include this arc (Gouveia and Martins 2000). In other words, the arc (i,j) can be always replaced by the arc $(0,j)$ without deteriorating the objective function value, since there is no limit on the number of subtrees. Also based on a optimality criteron, variables $x_{ijk} \quad i \in V, j \in V^+, k \in V^+ \quad$ such that $\quad c_{ij} \geq c_{kj}$, can be eliminated for the same reason as in the previous case. This time the rationale is applied to the arcs associated with the s-tree rooted in k. If in an optimal solution, the arc (i,j) is part of the s-tree rooted in k, arc (i,j) can be always replaced by the arc (k,j), without deteriorating the objective function value. Finally using the LB and UB, we can eliminate

the variables x_{ijk} with $\quad i \in V, j \in V^+, k \in V^+ \quad$ such that $LB + c_{ij} - c_{pj} > UB$, where $p = p_{LB}(j)$ is the predecessor of vertex j in T_{LB}.

For non-UD instances there is a third elimination procedure which eliminates variables $x_{ijk}, \quad i \in V, j \in V^+, k \in V^+ \quad$ with $d_i + d_j >= Q$ or $d_k + d_i + d_j >= Q$

Auxiliary network for the separation of inequalities

Before presenting the separation procedures for inequalities (3.19), (3.23) and (3.24) we introduce some additional notation. In each case $\overline{N}(\bar{x}, \bar{y}) = (\overline{V}, \overline{A})$ denotes the auxiliary graph which is used for the separation procedure. It always contains all the vertices and arcs of the subgraph induced by the fractional solution $(\bar{x}, \bar{y})$. It also always contains two pseudo-vertices, a source vertex s' and a sink vertex u'. Depending on the case, $\overline{N}(\bar{x}, \bar{y})$ will contain additional vertices and arcs. In most cases the separation resorts to finding a minimum cut between s' and u' in the corresponding network.

Separation of subroot multistar inequalities

As stated in Theorem 3 in Letchford and Salazar-González (2006) there is an exact separation algorithm for inequalities (3.9), based on the solution of a maximum flow problem on an auxiliary network induced by the fractional solution. Extending this theorem, subroot multistar inequalities can also be separated exactly in a similar way, using the auxiliary network $\overline{N}(\bar{x}, \bar{y})$. For this family of inequalities we include in $\overline{V}$, the original set of terminals V^+, a source vertex s', a sink vertex u', and a set of artificial vertices $\overline{V}_j$ associated with each vertex $j \in V^+$. For a given j, $\overline{V}_j$ contains a copy k_j of each subroot k such that $\bar{y}_{jk} > 0$, $j \neq k$. That is, $\overline{V}_j = \{k_j : y_{jk} > 0\}$. Then, $\overline{V} = V^+ \cup \{s', u'\} \cup \{\overline{V}_j, j \in V^+\}$. $\overline{A}$ contains arcs of the following types: (i) arcs (s', k), with $k \in V^+$; (ii) arcs (k, u') , with $k \in V^+$; (iii) arcs linking two original vertices $i, j \in V^+$; (iv) arcs (j, k_j) linking original vertices $j \in V^+$ with artificial vertices $k_j \in V_j$; and, arcs (k_j, i) connecting artificial vertices $k_j \in V_j$ and original vertices $i \in V^+ (i \neq j)$.

The weight w_{ij} associated with each arc $a \in \overline{A}$ depends on the type of arc. The weight of the arcs between the source s' and the original vertices $k \in V^+$ is given by $\bar{y}_{kk} + \sum_{i \in V^+} \sum\limits_{j \in V^+} d_j \bar{x}_{ijk} + \sum\limits_{i \in V^+} d_k \bar{y}_{ki}$. The weight of the arcs linking original vertices $k \in V^+$ with the sink u' is defined as $d_k + \sum\limits_{j \in V^+} d_j \bar{y}_{jk}$. The weight of the arcs linking two original vertices $i \in V^+$, $j \in V^+$ is given by $(Q - d_i)\bar{x}_{kik} + \sum\limits_{j \in V^+} d_i \bar{x}_{jik} - d_i \bar{y}_{ik}$. The arcs linking original vertices $j \in V^+$ with artificial vertices $k_j \in \overline{V}_j$ have a weight of Q. Finally, the weight of the arcs between artificial vertices $k_j \in \overline{V}_j$ and original vertices $i \in V^+$ $(i \neq j)$ is defined as $(Q - d_k - d_i)\bar{x}_{ijk} - d_j \bar{x}_{ijk}$. Figure 3.1 illustrates how the weights of the arcs $(i, j) \in \overline{A}$ are defined.

On this network we find the minimum cut $\overline{S} \subseteq \overline{V}$ with $u' \in \overline{S}$ and $s' \notin \overline{S}$. Then, there is a multistar inequality violated by the fractional solution $(\bar{x}, \bar{y})$, if and only if, the capacity of $\overline{S}$, $W(\overline{S})$, is smaller than:

$$L = \sum_{i \in V^+} d_i + \sum_{k \in V^+} \sum_{i \in V^+} \sum_{j \in V^+} d_j \bar{x}_{ijk} + \sum_{k \in V^+} \sum_{j \in V^+} \bar{y}_{jk}$$

.

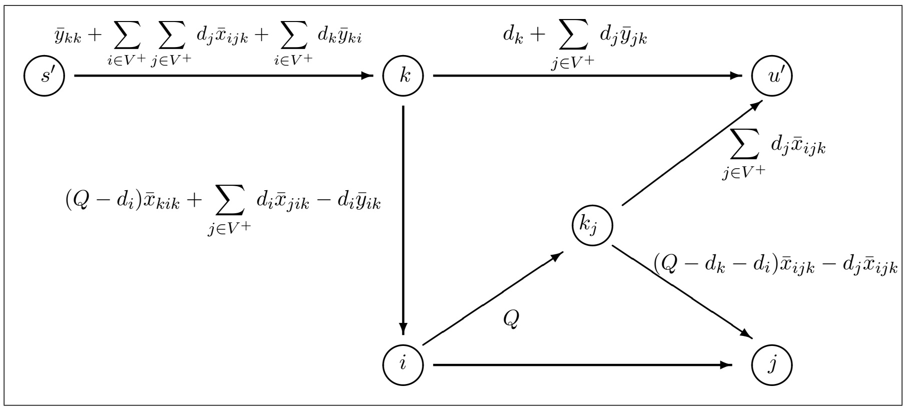
$\bar{y}_{kk} + \sum_{i\in V^+}\sum_{j\in V^+} d_j\bar{x}_{ijk} + \sum_{i\in V^+} d_k\bar{y}_{ki}$
$d_k + \sum_{j\in V^+} d_j\bar{y}_{jk}$
s'
k
u'
$\sum_{j\in V^+} d_j\bar{x}_{ijk}$
$(Q-d_i)\bar{x}_{kik} + \sum_{j\in V^+} d_i\bar{x}_{jik} - d_i\bar{y}_{ik}$
k_j
$(Q-d_k-d_i)\bar{x}_{ijk} - d_j\bar{x}_{ijk}$
Q
i
j

Figure 3.1: Auxiliary network for the exact separation of subroot multistar inequalities

The subset of vertices $S \subseteq V^+$ which produces a violated inequality (3.19) is given by $S = \overline{S} \cap V^+$.

Separation of the s-tree multistar inequalities

The separation for the family of inequalities (3.23) is solved exactly in a similar way to the subroot multistar ones (3.19). In this case we generate a different network $N^k(\bar{x},\bar{y}) = (\overline{V},\overline{A})$ for each vertex k with $\bar{y}_{kk} > 0$ in the fractional solution $(\bar{x},\bar{y})$. For this family of inequalities we include in $\overline{V}$, the subroot k, the terminals $\overline{V}_k = \{j : y_{jk} > 0,\ j \in V^+\}$, a source vertex s', and a sink vertex u'. Then, $\overline{V} = \overline{V}_k \cup \{k, s', u'\}$. $\overline{A}$ contains arcs of the following types: (i) arcs (s',k), where $k \in V^+$ is a subroot; (ii) arcs (k,j) connecting subroot k and terminal $j \in V_k$; (iii) arcs (j,u') linking terminal vertices $j \in V_k$ for some subroot k with the sink u'; and, (iv) arcs (i,j) linking two original vertices $i,j \in V_k$ for some subroot k.

The weight w_{ij} associated with each arc $(i,j) \in \overline{A}$ depends on the arc. The weight of the arcs between the source s' and subroot k is given by Q. For arcs between k and vertices $j \in \overline{V}_k$, w_{kj} is defined as $(Q-d_k)x_{kjk}$. For the arcs linking vertices $j \in \overline{V}_k$ with the sink u' their weight is defined as $\sum_{j\in V^+} d_j\bar{x}_{ijk}$. The weight of the arcs linking two vertices $i \in \overline{V}_k$, $j \in \overline{V}_k$ is $(Q-d_k-d_i)\bar{x}_{ijk} - d_j\bar{x}_{ijk}$. Figure 3.2 shows how the weights for arcs $(i,j) \in \overline{A}$ on the auxiliary network $N(\bar{x},\bar{y})$ are defined.

Then for every vertex k with $\bar{y}_{kk} > 0$ we find the minimum cut $\overline{S} \subseteq \overline{V}$ with $u' \in \overline{S}$ and $s' \notin \overline{S}$ using the auxiliary graph $N^k(\bar{x},\bar{y})$. Then, there is a s-tree multistar inequality violated by the fractional solution $(\bar{x},\bar{y})$, if and only if, the capacity of $\overline{S}$, $W(\overline{S})$, is smaller than:

$$L = \sum_{j\in S} d_j y_{jk}$$

.

In this case, the subset of vertices $S \subseteq V^+$ which produces a violated inequality (3.23). is given by $S = \overline{S} \setminus u'$.

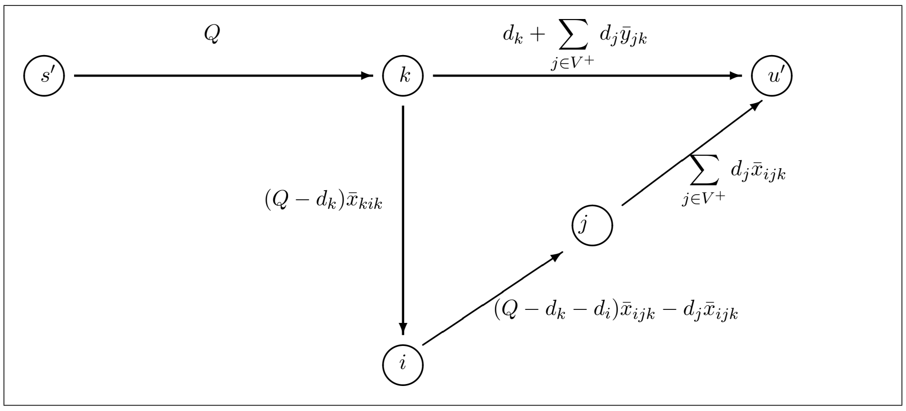
Q
$d_k + \sum_{j \in V^+} d_j \bar{y}_{jk}$
$\sum_{j \in V^+} d_j \bar{x}_{ijk}$
$(Q - d_k)\bar{x}_{kik}$
$(Q - d_k - d_i)\bar{x}_{ijk} - d_j \bar{x}_{ijk}$
s'
k
u'
j
i

Figure 3.2: Auxiliary network for the exact separation of s-tree multistar inequalities

Separation of the subroot rootcutset inequalities

The separation problem for the subroot rootcutset inequalities is solved heuristically using the network $N(\bar{x}, \bar{y}) = (\overline{V}, \overline{A})$ induced by the fractional solution $\bar{x}, \bar{y}$. The set of vertices $\overline{V}$ includes the terminals V^+, a source vertex s', and a sink vertex u' ($\overline{V} = V^+ \cup \{s', u'\}$). $\overline{A}$ contains arcs of the following types: (i) arcs (s', k), where $k \in V^+$ is a subroot; (ii) arcs (i, j) connecting terminal $i \in V^+$ with terminal $j \in V^+$; and (iii) arcs (j, u') linking terminal vertices $j \in V^+$ with the sink u'. Weights w_{ij} are defined using $\sum_k x_{ijk}$, $(i, j) \in \overline{A}$. In this case we apply a vertex shrinking procedure to $N(\bar{x}, \bar{y})$, to find violated subroot rootcutset inequalities. It is important to notice that in any feasible solution of the CMST, every vertex in V^+ has only one predecessor (Figure 1) since we assume that the root is the origin. Vertices are shrunk with their predecessor and the shrinking starts by the leaves (vertices with no successors) and continues until all non subroot vertices have been shrunk. Every time two vertices are shrunk, a new artificial vertex containing a list of the shrunk vertices and its accumulated demand is created. Using the list of vertices and the accumulated demand in the artificial vertex we check for violated subroot rootcutset inequalities. Since in fractional solutions is common to find vertices with two or more predecessors, the shrinking is done with the vertex with greater w_{ij} value (Figure 3.3).

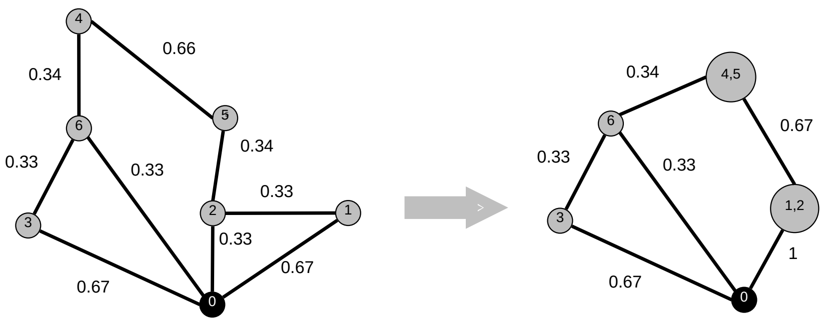

Figure 3.3: Vertex shrinking for the separation of subroot rootcutset inequalities

3.3 Subroot hop indexed formulation.

This new formulation results from the combination of subroot and hop indexed variables (described in Section 2.1.2) and is designed to solve the UD case. It has two types of variables, y variables which are the same as in the subroot formulation, and x variables, which are arc variables that take into account the subroot of the s-tree they belong to and its depth relative to the root. The new model has an increased number of variables $Q \cdot n^3 + n^2$ and we will refer to this new formulation as *Subroot Hop Indexed Formulation* (SRHIF).

The subroot hop indexed variables are defined as follows:

$$y_{jk} = \begin{cases} 1, & \text{if vertex } j \text{ belongs to the } s\text{-tree rooted at vertex } k, \\ 0, & \text{otherwise.} \end{cases}$$

$$x^t_{ijk} = \begin{cases} 1, & \text{if arc } (i,j) \text{ is in depth } t \text{ in the subtree rooted at vertex } k, \\ 0, & \text{otherwise.} \end{cases}$$

$$\min \sum_{t=1}^{Q} \sum_{k \in V^+} \sum_{(i \in V} \sum_{j \in V^+} c_{ij} x^t_{ijk} \tag{3.30}$$

$$\text{s.t.} \sum_{k \in V^+} y_{jk} = 1 \qquad j \in V^+ \tag{3.31}$$

$$\sum_{t=1}^{Q} \sum_{k \in V^+} \sum_{i \in V^+} x_{ijk} = y_{jk} \qquad j,k \in V^+ \tag{3.32}$$

$$y_{kk} = x^1_{0kk} \qquad j \in V^+ \tag{3.33}$$

$$\sum_{t=1}^{Q} x^t_{ijk} + \sum_{t=1,..,Q} x^t_{jik} \leq y_{ik} \qquad (i,j,k) \in V^+ i \neq j, j \neq k \tag{3.34}$$

$$\sum_{t=1}^{Q} x^t_{ijk} + \sum_{t=1,..,Q} x^t_{jik} \leq y_{jk} \qquad (i,j,k) \in V^+ i \neq j, j \neq k \tag{3.35}$$

$$\sum_{\substack{j \in V^+ \\ j \neq k}} d_j y_{jk} \leq (Q - d_k) y_{kk} \qquad k \in V^+ \tag{3.36}$$

$$\sum_{t=1}^{Q} \sum_{k \in V^+} \sum_{i \notin S \cup \{0\}} \sum_{j \in S} x^t_{ijk} \geq \left\lceil \frac{d(S)}{Q} \right\rceil \qquad S \subseteq V^+ \tag{3.37}$$

$$\sum_{k \in V^+} y_{kk} \geq \left\lceil \frac{d(V^+)}{Q} \right\rceil \tag{3.38}$$

$$y_{jk} \in \{0,1\} \qquad j \in V^+, k \in V^+ \tag{3.39}$$

$$x^t_{ijk} \in \{0,1\} \qquad a = (i,j) \in A, k \in V^+, t \in 1,..,Q \tag{3.40}$$

With equation (3.31) we assign each vertex to a subroot and with (3.32) we guarantee that there is one incoming arc for each vertex. Equations (3.33) imposes subroot vertices to be directly connected to the root. Inequalities (3.34) and (3.35) state that an arc can only be used only if both, its origin and end vertices, belong to the same s-tree. The capacity is controlled by constraints (3.36) and the connectivity of the solution is guaranteed by (3.37).

For this new formulation we can adapt the different families of inequalities which we have already presented for the subroot formulation in the previous section. The coefficients of the multistar (3.19), s-tree multistar (3.23) and rootcutset (3.24) inequalities are improved with this new formulation. Additionally, a modified version of the GSEh inequalities (2.17) from Gouveia and Martins (2005) using the subroot hop indexed variables, is shown.

Using subroot hop indexed variables, adapted multistar inequalities derived from (3.19) are expressed as follows:

$$\sum_{k\in S}\sum_{j\in S} y_{jk} + \sum_{t\geq 2}^{Q}\sum_{k\notin S}\sum_{i\notin S}\sum_{j\in S}(Q-t+1)x_{ijk}^t - \sum_{t\geq 3}\sum_{k\notin S}\sum_{i\in S}\sum_{j\notin S} x_{ijk}^t \geq |S| \qquad S\subset V^+, |S|\geq 2 \tag{3.41}$$

Using subroot hop indexed variables, s-tree multistar inequalities (3.23) have the following expression:

$$\sum_{j\in S}(Q-dk)x_{kjk}^2 + \sum_{t\geq 3}^{Q}\sum_{i\notin S}\sum_{j\in S}(Q-t+1)x_{ijk}^t - \sum_{i\in S}\sum_{j\notin S} d_j x_{ijk}^t \geq \sum_{j\in S} d_j y_{jk} \qquad S\subset V^+, |S|\geq 2 \tag{3.42}$$

Another family of inequalities that can be expressed using the subroot hop indexed variables are subroot rootcutset inequalities (3.24), that have the following expression:

$$\frac{MT+1}{MT}\sum_{k\in S} y_{kk} + \frac{MT+1}{MT}\sum_{t\geq 2}^{Q}\sum_{k\notin S}\sum_{i\in A}\sum_{j\in S} x_{ijk}^t \qquad S\subset V^+, |S|\geq Q \tag{3.43}$$
$$+\sum_{t\geq 2}^{Q}\sum_{k\notin S}\sum_{i\in B}\sum_{j\in S} x_{ijk}^t \geq MT+1$$

GSEh inequalities (2.17) can also be rewritten using the subroot hop indexed variables. The subroot GSEh inequalities have the following expression:

$$\sum_{t\geq h+1}\sum_{k\in V^+}\sum_{i\in S}\sum_{j\in S} x_{ijk}^t \leq |S| - \left\lceil \frac{|S|}{Q-h+1} \right\rceil \qquad S\subseteq V^+, |S|\geq 2, 1\leq h\leq Q-1 \tag{3.44}$$

As in the original inequalities (2.17), the right hand side of the equation represents the maximum number of arcs considering the cardinality of S ($|S|$) and a fixed depth h. They are used to enhance the connectivity of the solution by finding GSE inequalities at different depths h.

If we compare the previous inequalities (3.44) with (2.15), is easy to see that the last ones are a special case of the GSEh inequalities for $h = 1$. As the modified subroot rootcutset inequalities (3.43) are a modified version of (2.15), we can conclude that (3.43) are also a special case of the GSEh inequalities for $h = 1$, which have been subjected to a lifting procedure.

Hop ordering inequalities from Gouveia and Martins (2005) can also be expressed in terms of the subroot hop indexed variables. They have the following form:

$$\sum_{k\in V^+}\sum_{\substack{m\in V\\ m\neq j}} x^t_{mik} \geq \sum_{k\in V^+} x^{t+1}_{ijk} \qquad (i,j)\in V^+, t=1,..,Q-1 \tag{3.45}$$

As mentioned in Section 2.2, these inequalities state that an arc (i,j) can be in depth $t+1$ only if there is an entering arc (m,i) in depth t.

3.3.1 Solution algorithm for the subroot hop indexed formulation

We also propose a cutting plane algorithm for the CMST. It is an iterative algorithm in which each iteration considers a subproblem with only a a subset of constraints of formulation (3.30)-(3.40), and solves its linear programming (LP) relaxation. The initial subproblem includes constraints (3.31), (3.32), (3.36) and (3.38). Constraints (3.34) and (3.35) are not considered because they are dominated by s-tree inequalities (3.42). In addition, at each iteration, we try to separate violated inequalities of types (3.41), (3.42), (3.44), (3.43) and (3.45), violated by the current LP solution. All violated inequalities found are incorporated to the current subproblem. When the separation procedures do not detect any violated inequalities then a variable elimination procedure is applied. The algorithm stops when no cuts are found and no variable is eliminated, or after a certain number of iterations without improvement. The pseudocode for the algorithm is almost the same as the one for the subroot formulation, we just need to replace line 17 from Algorithm 1 by:

"Separate violated extended subroot multistar, extended subroot rootcutset, extended s-tree multistar, extended GSEh and extended hop ordering inequalities"

Variables Elimination

As in the the subroot formulation, the algorithm includes two procedures to reduce the number of variables for the subroot hop indexed formulation. The procedures are a slight adaptation of the ones for the subroot formulation. The UB and LB are calculated as mentioned in Section 3.2.5. Again, using an optimality criterion, variables $x_{ijkt}, \quad i\in V, j\in V^+, k\in V^+ \quad, t\in\{1,...,Q\}$ such that $\quad c_{ij}\geq c_{0j}$ are eliminated as well as variables $x_{ijkt} \quad i\in V, j\in V^+, k\in V^+ \quad$ such that $\quad c_{ij}\geq c_{kj}$. The reasons for these eliminations follow are the same ones stated in Section 3.2.5. Additionally, variables $x_{kjkt}, \quad k\in V, j\in V^+ \quad, t\neq 2$ are eliminated, since the arcs leaving a subroot can be only at depth 2. the previous case. This time the rationale is applied to the arcs associated with the s-tree rooted in k. If in an optimal solution, the arc (i,j) is part of the s-tree rooted in k, arc (i,j) can be always replaced by the arc (k,j), without deteriorating the objective function value. Finally using the LB and UB, we can eliminate the variables x_{ijkt} with $\quad i\in V, j\in V^+, k\in V^+ \quad, t\in\{1,...,Q\}$ such that $LB + c_{ij} - c_{pj} > UB$, where $p = p_{LB}(j)$ is the predecessor of vertex j in T_{LB}.

For non-UD instances there is a third elimination procedure which eliminates variables $x_{ijk}, \quad i\in V, j\in V^+, k\in V^+ \quad$ with $d_i + d_j >= Q$ or $d_k + d_i + d_j >= Q$

The initial subproblem includes constraints (3.31), (3.32), (3.36) and (3.38). At each iteration we try to separate violated inequalities of types (3.41), (3.42) and (3.43).

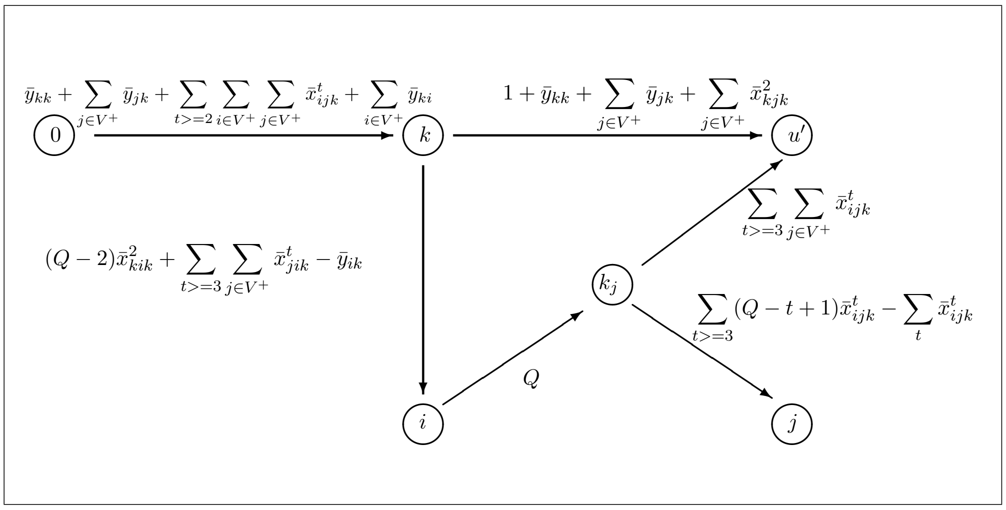
$\bar{y}_{kk} + \sum_{j\in V^+} \bar{y}_{jk} + \sum_{t>=2}\sum_{i\in V^+}\sum_{j\in V^+} \bar{x}^t_{ijk} + \sum_{i\in V^+} \bar{y}_{ki}$
$1 + \bar{y}_{kk} + \sum_{j\in V^+}\bar{y}_{jk} + \sum_{j\in V^+}\bar{x}^2_{kjk}$
0
k
u'
$\sum_{t>=3}\sum_{j\in V^+}\bar{x}^t_{ijk}$
$(Q-2)\bar{x}^2_{kik} + \sum_{t>=3}\sum_{j\in V^+}\bar{x}^t_{jik} - \bar{y}_{ik}$
k_j
$\sum_{t>=3}(Q-t+1)\bar{x}^t_{ijk} - \sum_{t}\bar{x}^t_{ijk}$
Q
i
j

Figure 3.4: Auxiliary network for the exact separation of extended subroot multistar inequalities

The separation procedures for inequalities (3.41) is similar to the one used for (3.19). For this family we also use the network induced by the fractional solution $N(\bar{x},\bar{y}) = (\overline{V},\overline{A})$. The set of vertices $\overline{V}$ and the set of arcs $\overline{A}$ are the same as the ones defined for the network to separate inequalities (3.19). Remember that $\overline{V} = V^+ \cup \{s', u'\} \cup \{\overline{V}_j, j \in V^+\}$. The weight w_{ij} associated to each arc $a \in \overline{A}$, is defined using the customized graph presented in Figure 3.4. On this network we find the minimum cut $\overline{S} \subseteq \overline{V^+}$ with $u' \in \overline{S}$ and $s' \notin \overline{S}$. Then, there is a multistar inequality violated by the fractional solution $(\bar{x},\bar{y})$, if and only if, the capacity of $\overline{S}$, $W(\overline{S})$, is smaller than:

$$L = \sum_{i\in V} d_i + \sum_{k\in V^+}\sum_{i\in V^+}\sum_{j\in V^+} d_j \bar{x}_{ijk} + \sum_{k\in V^+}\sum_{j\in V^+} \bar{y}_{jk}$$

.

Again, in this case, the subset of vertices $S \subseteq V^+$ which produces a violated inequality (3.41). is given by $S = \overline{S} \cap V^+$.

The separation problem for the adapted s-tree multistar inequalities (3.42) is solved exactly in a similar way to inequalities (3.23). In this case we also generate a network $N^k(\bar{x},\bar{y}) = (\overline{V},\overline{A})$ for each vertex k with $\bar{y}_{kk} > 0$ in the fractional solution $(\bar{x},\bar{y})$. The set of vertices $\overline{V}$ and the set of arcs $\overline{A}$ are the same as the ones used for inequalities (3.23). Remember that $\overline{V} = V_k + \cup\{k, s', u'\}$. The weight w_{ij} associated with each arc $(i,j) \in \overline{A}$ is defined using the auxiliary graph presented in Figure 3.5.

For every vertex k with $\bar{y}_{kk} > 0$, we find the minimum cut $\overline{S} \subseteq \overline{V}$ with $u' \in \overline{S}$ and $s' \notin \overline{S}$ using the corresponding auxiliary graph $N^k(\bar{x},\bar{y})$. Then, there is a s-tree multistar inequality violated by the fractional solution $(\bar{x},\bar{y})$, if and only if, the capacity of $\overline{S}$, $W(\overline{S})$, is smaller than

$$L = \sum_{j\in S} d_j y_{jk}$$

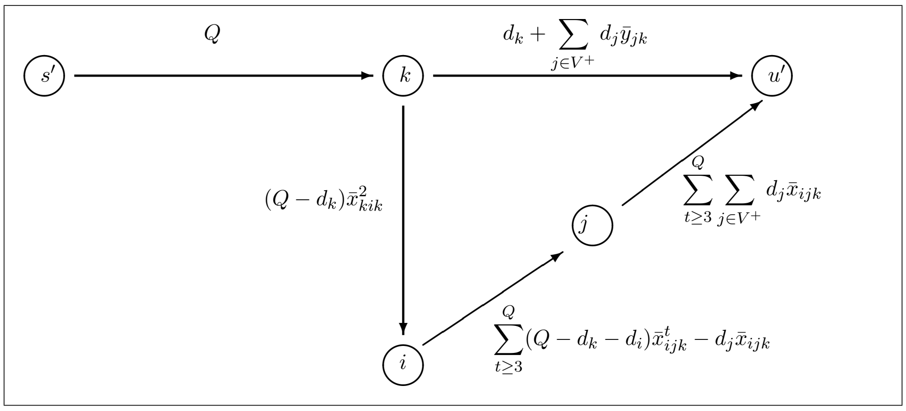
Q
$d_k + \sum_{j \in V^+} d_j \bar{y}_{jk}$
s'
k
u'
$(Q - d_k)\bar{x}^2_{kik}$
$\sum_{t \geq 3}^{Q} \sum_{j \in V^+} d_j \bar{x}_{ijk}$
j
$\sum_{t \geq 3}^{Q} (Q - d_k - d_i)\bar{x}^t_{ijk} - d_j \bar{x}_{ijk}$
i

Figure 3.5: Auxiliary network for the exact separation of extended s-tree multistar inequalities

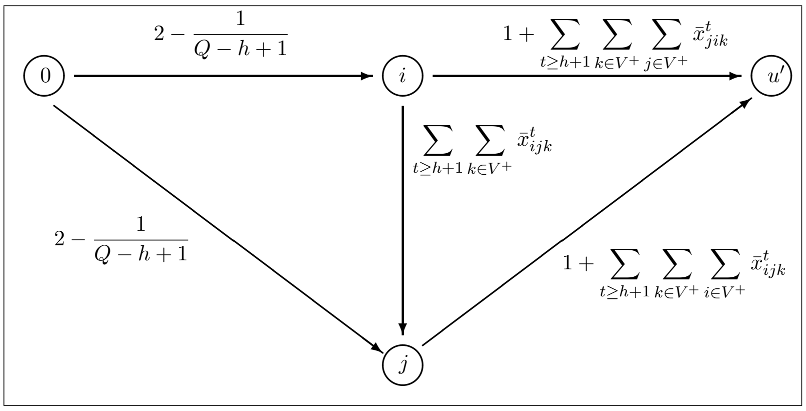
$2 - \frac{1}{Q-h+1}$
$1 + \sum_{t \geq h+1} \sum_{k \in V^+} \sum_{j \in V^+} \bar{x}^t_{jik}$
0
i
u'
$\sum_{t \geq h+1} \sum_{k \in V^+} \bar{x}^t_{ijk}$
$2 - \frac{1}{Q-h+1}$
$1 + \sum_{t \geq h+1} \sum_{k \in V^+} \sum_{i \in V^+} \bar{x}^t_{ijk}$
j

Figure 3.6: Auxiliary network for the heuristic separation of GSEh inequalities

.
The subset of vertices $S \subseteq V^+$ which produces a violated inequality (3.23) is given by $S = \overline{S} \setminus u'$.

Subroot GSEh inequalities 3.44 can be separated heuristically by using the auxiliary network $N^h(\bar{x}, \bar{y}) = (\overline{V}, \overline{A})$ for every value of h $(1 \leq h < Q)$. Remember that h represents the depth of the arcs relative to the root. The set of vertices $\overline{V}$ includes the terminals V^+, a source vertex s', and a sink vertex u' $(\overline{V} = V^+ \cup \{s', u'\})$. $\overline{A}$ contains arcs of the following types: (i) arcs (s', k), where $k \in V^+$ is a subroot; (ii) arcs (i, j) connecting terminal $i \in V^+$ with terminal $j \in V^+$; and (iii) arcs (j, u') linking terminal vertices $j \in V^+$ with the sink u'.
The weight of the arcs linking the source s' and a terminal $j \in V^+$ is $w_{ij} = 2 - \frac{1}{Q-h+1}$. The arcs linking terminals $i \in V^+$ and $j \in V^+$ have a weight of $\sum_{t \geq h+1} \sum_{k \in V^+} \bar{x}^t_{ijk}$. Finally, the weight of the arcs between terminals $j \in V^+$ and and sink s' is defined as $1 + \sum_{t \geq h+1} \sum_{k \in V^+} \sum_{j \in V^+} \bar{x}^t_{jik}$.
Figure 3.6 illustrates how the weights of the arcs $(i, j) \in \overline{A}$ are defined.

Finally the adapted rootcutset inequalities (3.43) are separated using the network used for the GSEh inequalities (3.44) when $h = 1$. This separation heuristic is different to the one proposed for inequalities (3.24). As this family of inequalities is known to be facet defining for other formulations, improving the effectiveness of its separation is important. Most of the methods for separating these inequalities are usually based on shrinking procedures on the graph induced by a fractional solution. The drawback of such methods is that during contraction good inequalities can be avoided. However, using the auxiliary network defined for inequalities (3.44) when $h = 1$ in practice leads to finding more subsets S violating inequalities (3.43).

```
Cutting Plane algorithm
  Input: $V, A, d, c$.
  Output: $Z$ and $X^*$
1 Compute upper bound (UB) using a heuristic method;
2 Using UB perform initial variable elimination procedures;
3 Build initial subproblem; $Z$:=0; STOP = false;
4 Define $Z'$as the objective function of the current LP subproblem; while
  not STOP do
5     Solve LP;
6     if $Z <$ F(LP) then
7         Update $Z$:= $F(LP)$;
8         Iterations without improvement:=0;
9     end
10    else
11        Increase the counter for the number of iterations without
          improvement;
12    end
13    if $Z$ = UB then
14        STOP := true;
15    end
16    else
17        Apply the corresponding separation procedure to the following
          types of inequalities:
18        - subroot multistar (3.19);
19        - subroot rootcutset (3.24);
20        - $s$-tree multistar inequalities (3.23);
21        if No cuts found then
22            Perform variable elimination procedure;
23            if No variable eliminated then
24                STOP := true;
25            end
26        end
27    end
28    if Iterations with out improvement $> M$ then
29        STOP := true;
30    end
31 end
32 return
```

Algorithm 1: Pseudo-code of the solution algorithm of subroot formulation.

Chapter 4

A BRKGA heuristic for the CMST

In this chapter we present a biased random-key genetic algorithm (BRKGA) for the CMST. BRKGA is a well known population-based metaheuristic, that has been used for combinatorial optimization (Gonçalves and Resende 2011). The BRKGA evolves a population of random vectors that encode solutions of the problem being addressed. As mentioned in Section 1.3, finding a suitable representation of solutions for crossover is one of main difficulties for designing population-based heuristics for the CMST. As it was mentioned, the widely used predecessor coding may lead to infeasible solutions after crossover. The alternative encoding presented (Zhou, Cao, Cao, and Meng 2007) is rather complex and feasibility is only guaranteed for some mutation operations, not for any kind of crossover. Thus, by presenting a BRKGA for the CMST we also face the challenge of finding an efficient solution representation for crossover.

In the first section of this chapter we give a brief description of the BRKGA. Then we present alternative coding/decoding representations. We continue with the description of the improvement phase, which is applied to the solutions after decoding and that has led to interesting computational results. The chapter ends with the presentation of the overall BRKGA that we propose.

4.1 Biased random-key genetic algorithms

Genetic algorithms with random keys, or *random-key genetic algorithms* (RKGA), were first introduced by Bean (1994) for solving combinatorial optimization problems involving sequencing. In a RKGA, problem solutions are represented as vectors of randomly generated real numbers in the interval $[0, 1]$, called chromosomes. A *decoder* is a deterministic algorithm that takes as input a chromosome and associates with it a solution to the problem 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 random-keys. Each component of the solution vector is generated independently at random in the real interval $[0, 1]$. In generation g the fitness of each individual is computed by the decoder and then the population is partitioned into two groups of individuals: a small group of p_e *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. All elite individuals of the population of generation g are copied without modification to the population of generation

$g+1$. RKGAs implement mutation by introducing *mutants* into the population. A mutant is simply a vector of random keys generated in the same way as the elements of the initial population. At each generation, a small number (p_m) of mutants is introduced into the population. With the p_e elite individuals and the p_m mutants accounted for, in population $g+1$, $p - p_e - p_m$ additional individuals need to be produced to complete the p individuals that make up the new population. This is done by producing $p - p_e - p_m$ offsprings through the process of mating or crossover. Bean (1994) selects two parents at random from the entire population to implement mating in a RKGA.

A *biased random-key genetic algorithm,* or BRKGA Gonçalves and Resende (2011), differs from a RKGA in the way parents are selected for mating. In a BRKGA, each element is generated combining one element selected at random from the elite partition in the current population and one from the non-elite partition. Repetition in the selection of a mate is allowed and therefore an individual can produce more than one offspring in the same generation. *Parameterized uniform crossover* (Spears and DeJong 1991) is used to implement mating in BRKGAs. Let $\rho_e > 0.5$ be the probability that an offspring inherits the vector component of its elite parent. Let n denote the number of components in the solution vector of an individual. For $i = 1, \ldots, n$, the i-th component $c(i)$ of the offspring vector c takes on the value of the i-th component $e(i)$ of the elite parent e with probability ρ_e and the value of the i-th component $\bar{e}(i)$ of the non-elite parent $\bar{e}$ 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.

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 points in the hypercube to solutions in the solution space of the combinatorial optimization problem where the fitness is evaluated. The termination criteria for the BRKGA can be defined either in terms of total iterations, iterations without improvement, time or a set objective function. Depending on the problem one of these criteria is chosen.

4.2 Encoding and decoding

As mentioned before, encoding and decoding solutions is one of the most important issues when using genetic algorithms for the CMST. For the algorithm we propose, CMST solutions are encoded as vectors $\mathcal{X}$ of n random keys, where $n = |V| - 1$. The i-th random key corresponds to the i-th terminal vertex. A decoder for a BRKGA for the CMST takes as input a vector $\mathcal{X}$ of n random keys and outputs a capacitated spanning tree and its cost.

For our algorithm we have developed three different decoding procedures. Two of them always produce feasible solutions after crossover, whereas one does not. The difference between the decoders lies in how the vectors $\mathcal{X}$ of random keys are interpreted. For the first two decoders, vectors $\mathcal{X}$ are used to sort the vertices and then an assignment procedure is performed. For the third decoder the i-th random key in vector $\mathcal{X}$ represents the predecessor of vertex i in the solution rooted tree.

For all three decoders we use a modified graph we use the undirected graph $G_U = (V, E)$, where each arc $a = (i, j) \in A$ of the original graph $G = (V, A)$ is substituted by its corresponding undirected edge $e = \{i, j\}$. Each decoder has an assignment phase and an improvement phase. The first decoder, called `direct assignment`, does not use arc costs, whereas the second one , called `cost-based assignment`, does. The predecessor decoder, called `predecessor assignment`, makes use of predecessor lists that are previously defined. In the improvement phase, neighborhoods of solutions are explored for improvements. Next the three decoders are described as well as the improvement phase.

4.2.1 Direct assignment decoder

The first decoding procedure described in this section takes as input the random-key vector $\mathcal{X}$, the graph structure $G_U = (V, E)$ and demands d, and returns an n-dimensional integer assignment vector a, where $a_i = k$ indicates that vertex i is assigned to the s-tree s-T_k and $a_k = k$ indicates that vertex k is a subroot. We remind the reader, that a subtree T_i, is a subgraph of a tree rooted at vertex i, whereas an s-tree s-T_k, is a subtree rooted at subroot k. The algorithm uses a vector, s, to keep the residual capacities of the partial s-trees; it is such that $s_k = q$ indicates that the s-tree s-T_k can still accommodate q units of demand before its Q units are fully used.

In this decoder vertices are considered, one at a time, in increasing order of their corresponding key. For each vertex i, the algorithm first tries to fit it to an already existing partial s-tree with enough residual capacity. If no such s-tree exists, the algorithm sets a new subroot k, and initializes a new s-tree s-T_k covering only k and i. Algorithm 2 shows the pseudo-code for `direct assignment`.

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 random keys in vector $\mathcal{X}$. List `SUBROOT` stores the subroot vertices with nonzero available capacity. It is initialized empty in line 4.

The loop from line 6 to line 34 scans each vertex and assigns it to an appropriate s-tree. Line 5 selects the first vertex in `VERTICES` as the vertex i to be assigned. The loop from line 8 to line 15 attempts to assign i to an already created s-tree if it is possible. To do this it traverses the list `SUBROOT`, in the order subroots have been created, seeking an existing subroot with available capacity. If vertex k is the subroot of an s-tree with enough capacity to accommodate i, i.e. $s_k \geq d_i$, then i is assigned to k in line 10 and it is removed from the list of candidate subroot vertices (`CANDIDATE`) in line 12. The capacity of s-T_k is updated in line 11. If i was successfully assigned, the while loop 8–15 is broken because the second condition in line 8 is no longer met. Otherwise, the next available subroot is assigned to k in line 14 and the loop restarts.

Line 16 assigns to k, the first candidate subroot in list `CANDIDATE`. If there is no available s-tree with sufficient residual capacity to accommodate vertex i, then i is assigned to a new subroot in lines 17 to 32. Loop 17 to 32 is repeated until i is assigned. In case i and k are identical, a new subroot is produced in line 19. If the s-tree s-T_k has enough capacity to accommodate vertices i and k, i.e. $s_k \geq d_i + d_k$, then k is made a subroot and i is assigned to s-tree s-T_k in lines 24 to 29. That is, k is set as a subroot in line 24, and inserted at the end of list `SUBROOT` in line 27. It is also removed from lists `VERTICES` and `CANDIDATE` in line 28. The assignment vector a is updated in line 25. In line 29 i is also removed from list `CANDIDATE`. In line 26 the available capacity of the s-tree s-T_k is updated. If an assignment of i to k is made, then the while loop 17 to 32 is broken since the second condition in line 17 is no longer met. Otherwise, in line 31 the next candidate vertex is considered to become a subroot.

```
procedure direct assignment
   Input: $\mathcal{X}, V, E, Q, d, c$
   Output: Assignment array a
1  $a_i \leftarrow 0$, $s_i \leftarrow Q$, for $i = 1, \ldots, n$;
2  Initialize VERTICES with vertices $1, \ldots, n$ sorted in increasing order of $\mathcal{X}$;
3  Initialize CANDIDATE with vertices $1, \ldots, n$ sorted in increasing order of $\mathcal{X}$;
4  Initialize empty list SUBROOT;
5  $i \leftarrow FIRST$(VERTICES);
6  while $i \neq$ nil do
                                  /* Try to assign $i$ to an existing subroot  */
7      $k \leftarrow FIRST$(SUBROOT);
8      while $k \neq$ nil and $a_i == 0$ do
9          if $s_k \geq d_i$ then
10             $a_i \leftarrow k$;
11             $s_k \leftarrow s_k - w_i$;
12             Remove $i$ from CANDIDATE list;
13         end
14         $k \leftarrow NEXT$(SUBROOT);
15     end
16     $k \leftarrow FIRST$(CANDIDATE);
17     while $a_i == 0$ do
                                  /* Set a new subroot $k$ and assign $i$ to it */
18         if $k == i$ then
19             $a_i \leftarrow i$;
20             $s_i \leftarrow Q - d_i$;
21             Add $i$ to end of SUBROOT list;
22             Remove $i$ from CANDIDATE and VERTICES lists;
23         else if $s_k \geq d_i + d_k$ then
24             $a_k \leftarrow k$;
25             $a_i \leftarrow k$;
26             $s_k \leftarrow Q - d_k - d_i$;
27             Add $k$ to end of SUBROOT list;
28             Remove $k$ from CANDIDATE and VERTICES lists;
29             Remove $i$ from CANDIDATE list;
30         end
31         $k \leftarrow NEXT$(CANDIDATE);
32     end
33     $i \leftarrow NEXT$(VERTICES);
34 end
35 return
```

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

If doubly linked lists are used for implementing `VERTICES`, `CANDIDATE`, and `SUBROOT`, assuming all list insertion and deletion operations can be done in $O(1)$ time, the runtime complexity of `assignment-1` is $O(n^2)$. The efficiency of this algorithm is improved by eliminating from list `SUBROOT` the subroots whose residual capacity becomes smaller than the smallest demand of all the vertices, after lines 11, 20, and 26.

4.2.2 Cost-based assignment decoder

The `direct assignment` decoder described above focuses mostly on the capacity constraints, and makes no use of arc costs for making assignments. We now describe an alternative procedure, `cost-based assignment`, which does. The central idea in this decoder is to scan the vertices in increasing order of the random keys in vector $\mathcal{X}$ and try to assign the scanned vertex i to its closest s-tree among the already existing ones with enough available capacity. Here, the distance from a vertex i to an s-tree $s\text{-}T_k$ is defined by the least c_{ij} value for $j \in V(s\text{-}T_k)$.

```
procedure cost-based assignment
  Input: $\mathcal{X}, V, E, Q, d, c$
  Output: Assignment array a
1 $a_i \leftarrow 0$, $s_i \leftarrow Q$, for $i = 1, \dots, n$;
2 Initialize list VERTICES with vertices $1, \dots, n$ 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 subtree */
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 $s_k \geq d_i$ then
11            $a_i \leftarrow k$;
12            $s_k \leftarrow s_k - d_k$;
13            Remove $i$ from CANDIDATE list;
14            Add $i$ to ASSIGNED list;
15        end
16        $j \leftarrow$ NEXT(ASSIGNED);
17    end
18    if $a_i == 0$ then
                                        /* Set $i$ as a new subroot */
19        $a_i \leftarrow i$;
20        $s_i \leftarrow s_i - d_i$;
21        Add $i$ to ASSIGNED list;
22    end
23    $i \leftarrow$ NEXT(VERTICES);
24 end
25 return
```

Algorithm 3: Pseudo-code for `cost-based assignment` decoder.

Algorithm 3 gives the pseudo-code of `assignment-2`, the cost-based assignment procedure. In line 1, the assignment vector a and available capacity vector s are initialized.

Vectors a and s are similar to those used in `assignment-1`. The lists are initialized in lines 2 and 3.

The vertices to be assigned are scanned in the loop in lines 5 to 24. At each iteration, i denotes the vertex to be assigned (initialized in line 4). In line 7 an ordered list is built using the vertices already assigned which are stored in list `ASSIGNED`. The loop in lines 9 to 16 assigns vertex i to the s-tree s-T_k containing the closest vertex j. In line 11 the available capacity at s-T_k is checked and in case $s_k \geq d_i$, vertex i is assigned to s-tree s-T_k (line 12). s_k is updated in line 13 and i is removed from list `VERTICES` and included in list `ASSIGNED` in lines 13 and 14. If there is no s-tree with available capacity to accommodate i, then i is set as a new subroot and included in list `ASSIGNED` (lines 20 to 23).

Sorting the list of already assigned vertices according to their distance to the vertex being currently scanned (line 6) is computationally expensive. For this reason, in our implementation of this algorithm, one heap is kept for each vertex, containing the distances from that vertex to all already assigned vertices. However, for clarity, we present here the basic version of the algorithm, since it allows a simpler comparison with Algorithm 2.

4.2.3 Predecessor assignment decoder

As in the previously described assignment procedures, the `predecessor assignment` decoder takes as input a random-key vector $\mathcal{X}$, the graph structure $G_U = (V, E)$ and demands d, and returns an n-dimensional integer assignment vector a and predecessor vector p. In this procedure the central idea is to use the key to define the predecessor vertex of each terminal. For a given a tree T, the predecessor p_j of a terminal $j \in V^+$ is the vertex $i \in V$, if the edge (i, j) is part of T ($(i, j) \in T$) and i is in the only path from the root to j.

This decoder has a preprocessing phase in which the possible predecessors of each vertex $j \in V^+$ are stored in a predecessor list for j (l_j). This list does not consider the vertices which have a cost greater than or equal to the cost of connecting vertex j to the root (i.e. $l_j = \{i : c_{ij} < c_{0j}\}$). Then we divide the interval $[0, 1]$ into $|l_j|$ small subintervals, each of which is assigned to a member of the list l_j.

Algorithm 4 shows the pseudo-code for the `predecessor assignment`. In line 1, the assignment vector a is initialized and in line 2 the predecessor p_j for each terminal j is defined using the list l_j and the random-key vector $\mathcal{X}$ ($p_j = f(l_j, \mathcal{X}_j)$). Predecessor vector p is used to build the tree T (line 3), which might be infeasible by two causes. The first is that a given s-tree s-T_k with $p_k = 0$, has an accumulated demand that violates the capacity constraint ($d(s\text{-}T_k) > Q$). The second one is that a given subtree T_i violates the connectivity constraint (no connection to the root and containing a loop). In both cases, a feasibility recovery phase "$R()$" is applied to make s-T_k or T_i a feasible s-tree or subtree respectively. In the loop in lines 4 to 7 we check vertex subtree T_i for infeasibility. If necessary, feasibility of T_i is recovered in line 6, and in line 7 we update tree T' using T_i. Finally assignment list a and predecessor vector p are defined using T' in line 8 to 9.

4.2.4 Feasibility recovery procedure for the predecessor assignment decoder

Once the decoding has ended for the `predecessor assignment`, we check every subtree T_j for possible feasibility violations. Infeasible subtrees T_j either violate the connectivity

```
procedure Predecessor-assignment
   Input: $\mathcal{X}, V, E, Q, w, c$
   Output: Assignment array $a$,predecessor vector $p$
 1 $a_j \leftarrow 0$, for $j = 1, \ldots, n$;
 2 $p_j \leftarrow f(l_j, \mathcal{X}_j)$ for $j = 1, \ldots, n$;
 3 Build $T$ using predecessor vector $p$ ;
 4 for $i = 1, \ldots, n$ do
 5     if $(T_i)$ is infeasible then
 6         Execute feasibility recovery procedure $R(T_i)$;
 7     end
 8     $T' \leftarrow T_i$;
 9 end
10 $a \leftarrow T'$;
11 $p \leftarrow T'$;
12 return
```

Algorithm 4: Pseudo-code for `predecessor assignment` decoder.

constraint, the capacity constraint, or both. First we check if T_j violates the connectivity constraint. If so, we connect T_j to the root and T_j becomes the s-tree s-T_j. Then we check if s-T_j violates the capacity constraint ($d(s\text{-}T_j) > Q$). If so, we remove subtrees T_i from s-T_j ($T_i \subset s\text{-}T_j$,$d(T_i) \leq Q$) until $d(s\text{-}T_j) \leq Q$. Every removed subtree T_i is either connected to the root or to the closest s-tree s-T_k ($k \neq j$) with enough capacity to accommodate T_i. The distance of T_i to an s-tree s-T_k, is defined as the minimum distance between any vertex of T_i and any vertex of $s - T_k$. That is: $dist(T_i, s\text{-}T_k) = \min\{c_{lm} \mid l \in V(s - T_k), m \in V(T_i)\}$. We connect T_i to s-T_k, if $dist(T_i, s\text{-}T_k) \leq c_{0i}$ and $d(s\text{-}T_k)+d(T_i) \leq Q$, or to the root otherwise.

Algorithm 5 shows the pseudocode for this procedure. In line 1 we check if the subtree T_j is an s-tree. If not we connect T_j to the root in line 2. Then if the s-tree s-T_j violates the capacity constraint, the algorithm enters the loop in lines 4 to 15 to make s-T_j become feasible according to capacity. Inside the loop, in line 5, we find a subtree T_i and an s-tree s-T_k with the minimum $dist(T_i, s\text{-}T_k)$ ($i \in V(s\text{-}T_j), k \neq j, k \in V^+$). If $dist(T_i, s\text{-}T_k) \leq c_{0i}$ (line 8) we connect T_i to s-T_k (line 9) and update the accumulated demand of s-T_k (line 10). If not, we connect T_i to the root in line 13.

4.3 Improvement phase

Since decoding is a crucial part of the BRKGA algorithm, some tests were carried out to evaluate the performance of the three different decoders. The complete results and other important details will be presented in Chapter 5. As it will become evident from the numerical results, all decoders required an improvement phase to enhance their performance. Therefore, after decoding, changes are introduced into the solution to look for improvements. The proposed improvement phase considers the use of reoptimization and local search. Since the set of vertices associated with a given s-tree s-T_k satisfies the capacity constraint we can

```
procedure Feasibility Recovery R()
  Input: Infeasible subtree $T_j$, $V, E, Q, d, c, T$
  Output: Feasible subtree $T_j$
1  if $p_j \neq 0$ then
2      $p_j \leftarrow 0$;
3  end
4  while $d(T_j) > Q$ do
5      Find $T_i$ and $s$-$T_k$ such that $\arg\min\{dist(T_i, s\text{-}T_k) : T_i \subset T_j, k \in V^+ subrootwith d(s\text{-}T_k) + d(T_i) \leq Q)\}$;
6      Remove $T_i$ from $T_j$ ;
7      $d(T_j) := d(T_j) - d(T_i)$ ;
8      if $dist(T_i, s\text{-}T_k) \leq c_{0i}$ then
9          Add $T_i$ to $s$-$T_k$ ;
10         $d(s\text{-}T_k) := d(s\text{-}T_k) + d(T_i)$ ;
11     end
12     else
13         $p_i \leftarrow 0$;
14     end
15 end
16 return
```

Algorithm 5: Pseudo-code for feasibility recovery procedure.

reoptimize every s-tree s-T_k computing the MST in $V(s\text{-}T_k) \cup \{0\}$. This reoptimization will be used in two different ways during the improvement phase. We can also modify the s-trees applying local search within different neighborhoods. In particular, the improvement phase consists of two stages: in the first one each s-tree is reoptimized whereas the second stage is a local search that explores several neighborhoods.

4.3.1 Minimum spanning tree stage (MST-stage)

As mentioned, in this stage every s-tree s-T_k in the solution is subject to a reoptimization process. In the output from the decoding phase, the array $a = [a_1, \dots, a_n]$, vertices are partitioned in subsets, which are used to build s-trees. For each s-tree s-T_k, the MST-stage computes a MST. All arcs connecting the root 0 to each subroot are added to the minimum cost spanning trees to jointly produce the capacitated minimum spanning tree. The cost of the tree is the sum of the costs of the s-trees plus the cost of the arcs connecting the subroot vertices to the root. To reoptimize an s-tree we use the set of vertices $V(s\text{-}T_k)$ associated with the s-tree s-T_k plus the root 0. Then, using Kruskal's algorithm we compute the MST for $V(s\text{-}T_k) \cup \{0\}$, which gives as result an s-tree with its associated cost plus the cost of connecting it to the root. Typically Kruskal's algorithm uses a list of the arcs which is sorted by increasing values of their costs. To avoid such sorting we created a data structure using linked lists which reduces significantly the time for computing the MST for each s-tree.

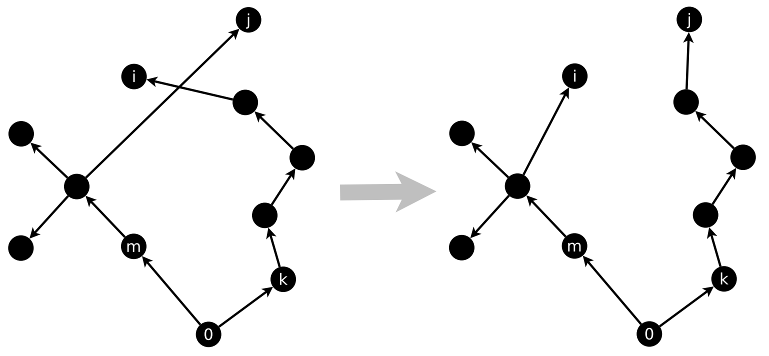

Figure 4.1: Neighborhood *N1*, exchange of vertices.

4.3.2 Local search stage

Once the solution has been reoptimized, a local search stage is used to improve it. In the local search phase neighborhoods are defined as well as exploration policies. In this section we describe the chosen neighborhoods and how they are explored.
We have considered four different neighborhoods. Every neighborhood is explored using a first improvement policy. The first proposed neighborhood $N1$, involves the swap of two vertices i and j. Vertices i and j involved in the swapping, belong to different s-trees, s-T_k and s-T_m respectively. Swaps are only allowed if capacity restrictions are not violated $d(s\text{-}T_k) - d_i + d_j \leq Q$ and $d(s\text{-}T_m) + d_i - d_j \leq Q$. A second neighborhood $N2$, is a move neighborhood, where a vertex i assigned to s-tree s-T_k is reassigned to another s-tree, say s-T_m. Such a move can only be made if s-T_m has sufficient available capacity to accommodate vertex i ($d(s\text{-}T_m) + d_i \leq Q$).

The third neighborhood $N3$ also corresponds to a move. In this case instead of reassigning a vertex i, we reassign the subtree T_i with origin in the non-subroot vertex i which is part of the s-tree s-T_k, to another s-tree s-T_m. The move is done only when the s-tree s-T_m has enough capacity to include the subtree T_i without violating capacity constraints ($d(s\text{-}T_m)+d(T_i) \leq Q$). Finally the fourth neighborhood considers the merging of two s-trees, s-T_k

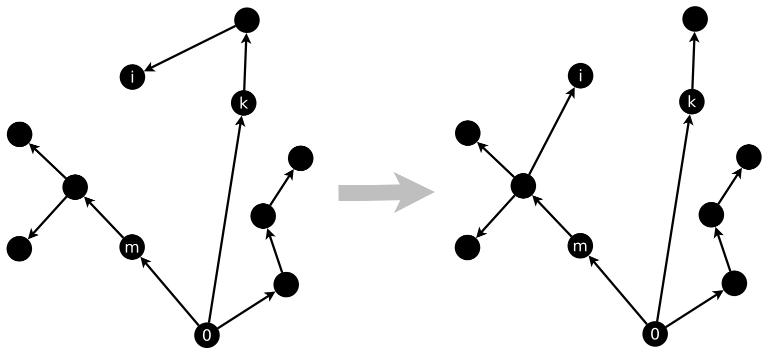

Figure 4.2: Neighborhood *N2*, vertex reassignment.

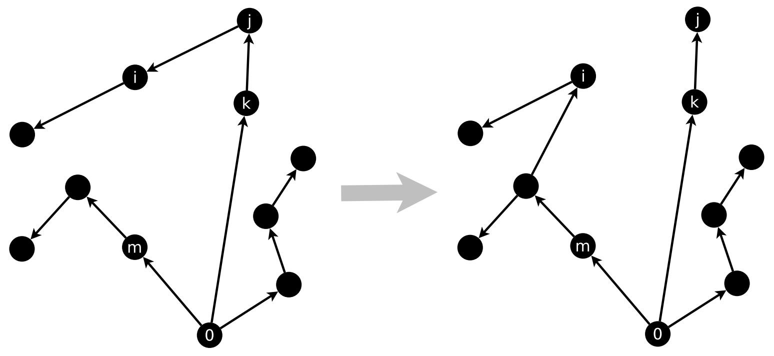

Figure 4.3: Neighborhood *N3*, sub-tree reassignment.

and s-T_m, into one s-tree s-T_r. The merging is allowed when the sum of the accumulated demand of both s-trees does not exceeds capacity parameter Q ($d(s\text{-}T_k) + d(s\text{-}T_m) \leq Q$). For every move in any neighborhood, the residual capacity (array s) used during the assignment phase and accumulated demands, are updated for the vertices and subtrees involved in the move. Figures 4.1- 4.4 allow to visualize neighborhoods $N1 - N4$ described above.

A variable neighborhood search (VNS) (Mladenović and Hansen 1997) strategy was used to search for improved solutions in the different proposed neighborhoods. VNS is a metaheuristic which systematically changes neighborhoods, to avoid local optima. We used the most simple version of VNS in which neighborhoods are explored sequentially in a given order. The neighborhoods are explored in the order ($N1$, $N2$, $N3$, $N4$) until no further improvement is reached in any neighborhood. The pseudocode for VNS is shown in Algorithm 6. The VNS receives the solution tree T and returns the improved solution tree. In line 1 the algorithm computes the tree cost (Z) and in line 2 an the auxiliary variable Z' is initialized. Then the algorithm enters the loop in lines 3 to 9 where the 4 neighborhoods are explored sequentially. After exploring each neighborhood, the cost (Z) of the tree is updated. The algorithm leaves the loop when the solution cost Z is not improved.

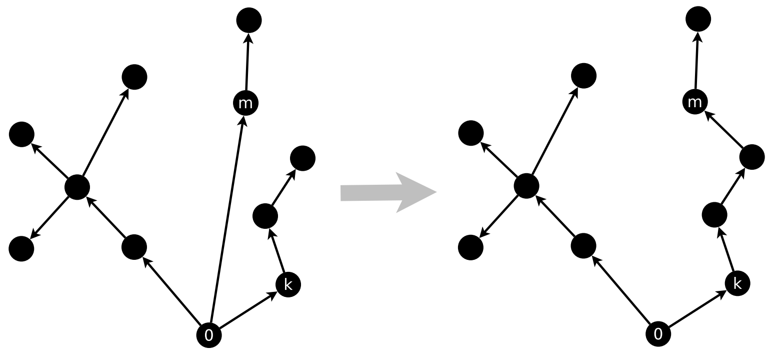

Figure 4.4: Neighborhood *N4*, s-tree merging.

```
procedure VNS
   Input: Solution tree $T$, $V, E, Q, d, c$
   Output: Updated solution tree $T$
1  Compute solution value $Z$ using solution tree $T$
2  end=false;
3  while end==false do
4      Explore Neighborhood $N1$;
5      Explore Neighborhood $N2$;
6      Explore Neighborhood $N3$;
7      Explore Neighborhood $N4$;
8      if $T$ has not been updated then
9          end=true;
10     end
11 end
12 return
```

Algorithm 6: Pseudocode for Variable Neighborhood search

4.3.3 Impact of MST computations

To explore each of the neighborhoods we have presented above, we propose three different strategies. In the first one we apply a MST-stage if no further improvement is found after exploring a neighborhood. If the solution is improved, we continue exploring the neighborhood, if not we jump to the next neighborhood. We will refer to this strategy as *MST-at-end*. In the second strategy that we will call *MST-at-change*, when an improvement is found a MST-stage is executed for the s-trees involved in such improvement. This is done in all explored neighborhoods. In the third strategy every time a movement is considered, we run a MST-stage for the subtrees involved in the movement. In other words, to evaluate each movement we tentatively make the movement and then the resulting subtrees are reoptimized. If the movement improves the solution cost, the movement is kept, otherwise it is discarded. As this strategy requires the computation of two MSTs to evaluate every movement, it is much more time consuming than the two other strategies. We refer to this third strategy as *All-MST*.

4.3.4 Strategic oscillation

Strategic oscillation has shown to be helpful to improve the results of heuristic methods, so we have used it in our BRKGA algorithm. Strategic oscillation allows the local search procedure to alternatively cross the border between the feasible and unfeasible regions. The idea is to use strategic oscillation to prevent the local search stage to get trapped at a local optima. In our case we propose the relaxation of capacity constraints using a penalty term and a maximum value of capacity violation $MaxQ$. The maximum violation allowed is set using the average weight of the vertices $d(V^+)/n$ and is computed using the following formula $MaxQ = 1.1d(V^+)/n$. Initially, for updating the penalty term, we only considered a descent policy (*Descent-SO*), but later we also considered two other oscillation strategies. All three are explained next.

The *Descent-SO* strategy, uses a fixed starting value for the penalty term which is updated until it reaches the zero value. After reaching this value, no further update is done to the penalty term. The second approach, *Descent-ascent-SO*, uses an upper and a lower limit for the penalty term, which decreases when the upper limit is reached and increases when it is equal to the lower limit. The initial value for the penalty is the upper limit. The third approach, *Alternate-SO*, starts as *Descent-SO* but when it reaches the zero value, the penalty term is set to either $+\alpha$ or $-\alpha$. The value of α is close to zero, and the idea of having a negative value is to encourage the solutions to move in the infeasible area to avoid local optima.

4.3.5 Neighborhood reduction

As mentioned previously, strategy *All-MST* is more time consuming than the other two. For this reason we applied a neighborhood reduction for this strategy. To apply this reduction we made some tests and observed the time spent by the algorithm in each of the neighborhoods. We found out that the most time consuming ones were $N1$ and $N2$. For those two neighborhoods we implemented a neighborhood reduction policy considering two factors: Distance between vertices, and the number of times that the movement was successful in improving the solution in the history of the search. If the distance between a pair of vertices (i, j) multiplied by a certain factor β, is smaller than or equal to the sum of the distances between such vertices and the root ($\beta c_{ij} \leq c_{0i} + c_{0j}$), then the move (interchange or reassignment) is explored. Otherwise the move is not considered for exploration. The complete results are presented in the next chapter.

4.4 BRKGA algorithm for the CMST

A framework designed for the BRKGA with the capability of handling parallel populations was used with the 2-phase predecessor decoder (see Section 4.2.3). In the first phase keys are decoded and the resulting solutions are then subjected to an improvement phase. The improvement phase has two stages, in the first one each s-tree is reoptimized computing its MST (see 4.3.1). In the second one local search is applied using the VNS of Algorithm 6 with the *All-MST* exploration strategy of Section 4.3.3, the Neighborhood Reduction of Section 4.3.5 and the Strategic Oscillation *Alternate-SO* of section 4.3.4 (see Algorithm 7). For each population a set of elite solutions is kept. After a certain number of generations a exchange of these elite sets is made among the different populations. After initial testing a set of parameters was chosen to run the algorithm in the different sets of test instances.

After testing different combinations of the features and options proposed in the previous sections for the decoder and improvement phase, we arrived to a final version for the algorithm. Based on preliminary computational testing we selected the following alternatives. For the decoding phase we chose the *predecessor assignment* decoder. In the improvement phase we apply VNS, exhaustive exploring strategies (*All-MST*), neighborhood reduction and strategic oscillation. The pseudocode is presented in Algorithm 7 and details are given next.

```
Algorithm BRKGACMST
  Input: $V, E, c, d, Q, H, L, ItEx, StopCriterion$
  Output: Best solution tree $T^*$, $Z^*$
1  $Z^* := \infty$; $it := 0$;
2  Generate first generation of $L$ populations containing $M$ random keys;
3  while $StopCriterion == FALSE$ do
4      $it := it + 1$;
5      for $l = 1, \ldots, L$ do
6          for $h = 1, \ldots, H$ do
7              Obtain tree $T$ using predecessor decoder on key $h$;
8              Apply MST-stage to $T$;
9              Apply VNS search to $T$;
10             Compute $Z_h$ using $T$;
11             if $Z_h < Z^*$ then
12                 $Z^* := Z_h$; $T^* \leftarrow T$;
13             end
14             Recode key $h$ using $T$;
15         end
16         Sort elements of population $l$ according to their fitness value $Z_h$
17         Execute Crossover;
18         Add $p_m$ mutants to population $l$
19     end
20     if $it$ mod $ItEx == 0$ then
21         Interchange elite solutions between populations;
22     end
23 end
24 return
```

Algorithm 7: Pseudocode for the final algorithm

The input of the algorithm includes the set of vertices V, the set of edges E, the cost matrix c, the demand vector d, the capacity parameter Q, the number of populations L, the number of elements in each population H, the number of iterations between interchange of elite elements of populations $ItEX$, and the stopping rule *StopCriterion.*

First, the algorithm introduces L populations, each one with H random keys (line 2), which are part of the first generation. Each random key h of each population l is decoded to obtain a solution T for the CMST (line 7). If T is infeasible, it is modified to make it feasible and then is subject of an improvement phase that stops when it is not possible to improve it. In the improvement phase, T is first reoptimized applying to it a MST-stage (line 8). Then it enters into the local search stage (Section 4.3.2), where VNS is used to search sequentially in neighborhoods, $N1$, $N2$, $N3$ and $N4$, which are explored using the *All-MST* strategy (line 9). For $N1$ and $N2$ a neighborhood reduction policy is additionally used in order to reduce the computational time (Section 4.3.3). At a certain point, when is difficult to improve the best solution, strategic oscillation is use within the VNS search. The strategic oscillation is performed using the *alternate-SO policy* (Section 4.3.4)

When no further improvement can be found, the solution value Z_h is computed using T (line 10), which is recoded and stored in key h (line 15). If Z_h is smaller than the current best solution value Z^*, the current best solution T^* is updated (line 12). When all the keys of the current population have been decoded, then it is divided into two groups; the elite and the regular population. To produce the next generation, children are created after the crossover of one elite parent with a non elite parent (line 17). Additionally a group of mutants (new random keys) is included in the new generation (line 18). The resulting population is then subjected to decoding, improvement and crossover to produce the next generation. Every certain number of iterations $ItEx$, the elite solutions of the L populations are interchanged (line 21). The process is repeated until a stop criterion is reached. In this algorithm the stop criterion is the number of generations passes without improving the overall best solution.

This final version was used to solve with all the test instances (explained in Chapter 5) and the results are presented in Section 5.2.

Chapter 5

Numerical Experience

In the two previous chapters three solution methods for the CMST were described. The first two are cutting plane algorithms based in two different formulations, the third one is a heuristic procedure that obtains upper bounds for the problem. In this chapter we will start by introducing the different sets of test instances that we have used for the CMST, to continue describing how the BRKGA heuristic algorithm was developed and the results obtained. In the last part, we present and analyze the results for the cutting plane algorithms.

5.1 Benchmark instances

In the experiments with the proposed algorithms we use different sets of well-known CMST benchmark instances available at `http://people.brunel.ac.uk/~mastjjb/jeb/jeb.html`. These test instances are divided into two main classes according to the type of demand at the vertices. The first class contains test 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 in sets *tc* and *te* have Euclidean distances (E). The difference between them is that in *tc* instances the root is located at the center of a rectangular region, whereas in *te* instances the root is located at a corner of a rectangular region. Instances in set *td* have Euclidean distances for edges connecting terminals, i.e. (i, j) with $i \in V^+$ and $j \in V^+$, and non-Euclidean (*non-E*) distances for edges connecting the root with terminals, i.e. $(0, j)$ with $j \in V^+$. The root of these instances is located at the center of a rectangular region. The number of vertices in instances of sets *tc* and *te* ranges in $\{80, 120, 160\}$, whereas all instances in *td* have 80 vertices. Instances in *tc*, *td* and *te* have capacity values $Q \in \{5, 10, 20\}$. Each of these sets contains 5 instances for each combination (n, Q) with $n \in \{80, 120\}$ and $Q \in \{5, 10, 20\}$, whereas they only contain one instance for each combination (n, Q) with $n = 160$ and $Q \in \{5, 10, 20\}$. As opposite to the other groups, the number of vertices of the test instances in *tc* does not include the root (i.e. $|V| = 81$ and $|V^+| = 80$).
Set *cm* contains instances with non-Euclidean distances (i.e., triangle inequality does not hold in many cases) and a number of vertices $n \in \{50, 100, 200\}$. The capacity values for these instances are $Q \in \{200, 400, 800\}$. Demands are non-unitary with values ranging from 1 to 100. There are five instances for each combination (n, Q). A summary of the characteristics of the 126 test instances can be found in Table 5.1.

Table 5.1: Summary of instances characteristics

Set	n	type of demand	type of distances	location of root	Q	number of instances
$tc80$	81	UD	E	Center	5	5
					10	5
					20	5
$tc120$	121	UD	E	Center	5	5
					10	5
					20	5
$tc160$	161	UD	E	Center	5	1
					10	1
					20	1
$te80$	80	UD	E	Corner	5	5
					10	5
					20	5
$te120$	120	UD	E	Corner	5	5
					10	5
					20	5
$te160$	160	UD	E	Corner	5	1
					10	1
					20	1
$td80$	80	UD	non-E	Center	5	5
					10	5
					20	5
$cm50$	50	non-UD	non- E	Center	200	5
					400	5
					800	5
$cm100$	100	non-UD	non- E	Center	200	5
					400	5
					800	5
$cm200$	200	non-UD	non- E	Center	200	5
					400	5
					800	5

5.2 BRKGA experimental results

As explained in Chapter 4, several alternatives were considered at each stage of the design of the BRKGA algorithm, and choices were made according to preliminary computational experiments. The first stage comprised the development of three different decoders, while in the second one an improvement phase was developed. In the third stage, we incorporated two additional features to the improvement phase; strategic oscillation and neighborhood reduction. During this development, several tests were made. In this section we present the results of such tests as well as their analysis. To perform such tests, we chose two sets of instances from the benchmarks described in Section 5.1. The first one was group $tc80$ with capacity $Q = 5$. The second group selected was $cm50$ with $Q = 200$. As mentioned before, the first group complies with the triangular inequality and has unitary demands, while the second does not satisfy the triangular inequality and has non-unitary demands.

Since solution encoding/decoding is a vital part of the BRKGA, we decided to test the performance of three different decoders. In genetic algorithms it is important to obtain a balance between the decoder efficiency (CPU time spent) and the genetic information

Table 5.2: Decoders comparison

		Direct Assignment		Cost Assignment		Predecessor Assignment	
Group	vertices	Avg gap	CPU	Avg gap	CPU	Avg gap	CPU
*tc*80	80	68.50	0.50	8.78	2.13	9.38	1.59
*cm*50	50	26.33	0.61	3.50	2.52	5.27	2.26

transmission. Table 5.2 compares the results obtained with three basic implementations of BRKGA, using the three decoders presented in Chapter 4, for the selected sets of instances. In particular, we have compared the direct assignment decoder of Section 4.2.1 (see Algorithm 2), the cost-based assignment decoder of Section 4.2.2 (see Algorithm 3), and the predecessor assignment decoder of Section 4.2.3 (see Algorithm 4).
In each case, column *Avg gap* refers to the average percentage deviation from the optimal or the best known solution. Column *CPU* refers to the average CPU time in seconds. The entries in each case are the averages over the five instances in the corresponding group after 3 runs.

In general terms, the presented results indicate that cost assignment and predecessor decoders perform better than the direct assignment decoder. The performance of all the decoders was far from the best known results, demanding the implementation of a improvement phase to enhance the performance of all of them. While the predecessor decoder may be more demanding than the cost assignment one in terms of CPU time, it has the advantage of transmitting better the genetic information, since similar keys yield similar solutions. On the other hand, the cost and direct assignment decoders perform faster but have the disadvantage that genetic information is not very well transmitted, since small changes in the key may modify substantially the solution.

Since the addition of an improvement phase in the algorithm had potential for improving any of the tested decoders, we decided to test them all with the improvement phase. In our experiments we have explored the neighborhoods presented in Section 4.3.2, using VNS as indicated in Algorithm 6. These neighborhoods are: vertices exchange (N1), vertex reassignment (N2), subtree reassignment (N3) and s-tree merging (N4). The results obtained with the addition of the improvement phase are summarized in Table 5.3 following the same structure as the previous table. These results show that the gap (average of all five instances in the group after 3 runs) is significantly reduced for all the decoders using the improvement phase. The smallest gaps are now obtained when using the predecessor assignment decoder. We attribute this fact to the ability of the predecessor assignment decoder of transmitting genetic information to the offspring. As can be seen, the inclusion of the improvement phase to all decoders has increased the computational requirements to solve the instances, giving raise to similar CPU times for the three decoders.

The best results are obtained by the predecessor decoder and the average CPU time is quite similar to all of them. Therefore, we decided to discard the two first decoders and continue the enhancement of the local search procedure using only the predecessor assignment decoder.

Table 5.3: Decoders comparison with improvement phase

		Direct Assignment		Cost Assignment		Predecessor Assignment	
Group	vertices	Avg gap	CPU	Avg gap	CPU	Avg gap	CPU
$tc80$	80	5.49	12.82	3.52	17.96	0.29	17.82
$cm50$	50	2.90	14.90	1.33	16.54	0.53	12.10

Table 5.4: Results with local search using different neighborhood exploring strategies

		MST-at-end		MST-at-change		All-MST	
Group	vertices	Avg gap	CPU	Avg gap	CPU	Avg gap	CPU
$tc80$	80	0.29	17.82	0.34	38.10	0.00	341.92
$cm50$	50	0.53	12.10	0.63	28.18	0.10	335.05

During the local search, MST computations can be done with different intensities in the neighborhood exploration. In particular, we compared three different stragegies (see Section 4.3.3): *MST-at-end*, *MST-at-change*, *All-MST*. All three strategies imply the reoptimization of the solution at different levels. The obtained results are summarized in Table 5.4 in which the entries give the average percentage deviation from the optimal or best-known solution. Again, averages are taken over all the instances in each group after 3 runs for each instance.

From the results in Table 5.4, it is clear , as it could be expected, that the *All-MST* strategy produces the best results although it requires larger CPU times in comparison with the two other strategies. The larger CPU times are due to the large amount of MSTs that are computed (for every considered move, two MSTs are computed). The *MST-at-change* strategy performs the worst in terms of average gap and the times are greater than those of the *MST-at-end* strategy, which showed a good balance between time consumption and solutions quality. In view of the above results we decided to explore two possible further improvements. The first one was to improve the results by introducing a strategic oscillation procedure in the local search. The second one was to reduce the number of moves considered in the exploration of the different neighborhoods.

Among the three different strategic oscillation policies presented in Section 4.3.4, we initially considered the *Descent-SO* in order to have a first impression of the effect of strategic oscillation in the behavior of the algorithm. This strategic oscillation policy was tested with the three different neighborhood exploring strategies. The results are shown in Table 5.5. As can be seen, strategic oscillation lead to improved gaps for strategies *MST-at-change* and *MST-at-end*, without a significant increase of their CPU times. Nevertheless, even with the improvement, these results are still worse than those of the *All-MST*, even if with this strategy the strategic oscillation did not show any improvement. However, as we mentioned before, the CPU times of *All-MST* strategy are larger, and therefore reducing the size of

Table 5.5: Results with local search including strategic oscillation

		MST-at-end		MST-at-change		All-MST	
Group	vertices	Avg gap	CPU	Avg gap	CPU	Avg gap	CPU
*tc*80	80	0.22	17.98	0.28	37.93	0.00	361.05
*cm*50	50	0.42	14.62	0.51	32.18	0.10	345.59

Table 5.6: Results including neighborhood reduction

		MST-at-end		MST-at-change		All-MST	
Group	vertices	Avg gap	CPU	Avg gap	CPU	Avg gap	CPU
*tc*80	80	0.22	17.98	0.28	37.93	0.00	99.05
*cm*50	50	0.42	14.62	0.51	32.18	0.10	103.21

the neighborhoods to be explored seemed a promising tool to improve the overall algorithm behavior.

Using the neighborhood reduction presented in Section 4.3.5 the CPU times for *All-MST* strategy were reduced significantly without compromising the quality of the solutions. Table 5.6 shows a comparison between the *All-MST* strategy with neighborhood reduction and the other two other strategies without neighborhood reduction. As in previous tables, entries indicate percentage average gaps over all the instances in each group after 3 runs. These results show that times are still larger than those of the other two exploring strategies, although they are considerably better in terms of average gap. For that reason we decided to continue working using only *All-MST* neighborhood exploring strategy with the neighborhood reduction.
The gaps obtained with this strategy in the two sets of instances used during the algorithm design were already very small before testing the strategic oscillation. This fact might explain why the strategic oscillation did not show a significant effect on the *All-MST* strategy. For this reason, we decided to further analyze the potential effect of the strategic oscillation in larger instances. In particular, we compared the three types of strategic oscillation and compared the results with the algorithm without strategic oscillation in instances $cm100$ with $Q = 200$ and in the set of the largest instances, $cm200$ with $Q = 400$.

For all the policies the penalty term takes values in the interval $[2.5, 0]$ and its initial value is 2.5. For *SO-descent* and, *Alternate-SO* policies the penalty is updated with steps of $-\beta$, while for *Descent-ascent-SO*, it is updated with steps of $\pm\beta$. In the case of the *Alternate-SO* policy, when the penalty reaches 0, the term oscillates taking the values $\pm\alpha$. β values are defined considering the cost of the current best solution. If such cost is greater than 800, $\beta = 0.1$. If the solution cost is between 550 and 799 $\beta = 0.05$, while for costs smaller than 550 $\beta = 0.03$. Values for alpha are close to zero.

A summary of the results is shown in Table 5.7 where it can be appreciated that *Alternate-SO* produces the best results for both groups of instances in terms of solution quality, with ap-

Table 5.7: Strategic oscillation results for $cm100$ with $Q = 200$ and $cm200$ with $Q = 400$

Strategic Oscillation type		Without		Descent		Descent -ascent		Alternate	
Group	vertices	Avg gap	CPU	Avg gap	CPU	Avg gap	CPU	Avg gap	CPU
$cm100$ $Q = 200$	100	1.04	365.16	0.98	365.13	1.08	368.12	0.95	366.68
$cm200$ $Q = 400$	200	0.54	3580.03	0.84	3468.41	0.78	3548.88	0.42	3501.86

proximately the same computational effort. Surprisingly, *Without-SO* algorithm performed better than *Descent-ascent-SO* and *SO-descent* for the instances in set $cm200$ with $Q = 400$. This is due to the magnitude of the penalty term, which sometimes is too big in relation to account the cost of the tree. This also explains why *Alternate-SO* performs better, because in this procedure the penalty term oscillates between $(-\alpha, \alpha)$ and takes small values more often.

5.2.1 BRKGA implementation details

The BRKGA algorithm for the CMST was implemented in C++. All the testing and the results presented in this section were obtained using a PC with an Intel core 2 at 3.1 GHZ processor and 2 GB of RAM.

The BRKGA framework requires the setting of 7 parameters:

- L, the number of populations to use.
- pe, the percentage of elite elements in each population.
- pm, the percentage of mutants introduced in each population after crossover.
- $ItEx$, the number of iterations between exchanges of elite solutions among populations.
- $ElitetoEx$, the number of elite solutions to exchange.
- ρ, the probability that an offspring inherits the vector component of its elite parent.
- $ItStop$, the number of iterations without improvement in the stopping criterion.

The values used for each parameter are presented in Table 5.8. Since non-UD instances are considered to be more difficult than UD instances, in this case we allowed for larger numbers of iterations both, between population exchanges and without improvement before termination.

In particular, the initial experiments showed that, in the case of UD instances, the difficulty of an instance was related both, to the size of the instance and to the capacity parameter. As opposite, in the case of non-UD instances the main source of complexity was the capacity, and the effect of the instance size was smaller. This evidence led us to the development of the expressions used for setting the parameter *itStop*.

Table 5.8: Parameter settings for the BRKGA framework

Parameter	UD	Non-UD
L	5	
pe	0.25	
pm	0.10	
$ElitetoEx$	1	
ρ	0.65	
$ItEx$	5	40
$ItStop$	$\frac{n-20}{Q}$	$\frac{45Q}{Q-100}$

5.2.2 BRKGA numerical results analysis

To evaluate the performance of Algorithm 7, taking into account that it is not deterministic, it was executed 7 times for each test instance. The solutions obtained were compared with the values of the best solutions known so far for the benchmark instances.

Algorithm 7 was executed 7 times for each test instance to evaluate the performance of the algorithm. The values of best known solutions for instances in sets $tc80$ and $te80$ were taken from (Ahuja, Orlin, and Sharma 2001), for sets $tc120$, $te120$, $tc160$ and $te160$ from Martins (2007) and Uchoa, Fukasawa, Lysgaard, Pessoa, de Aragao, and Andrade (2008). For instances of sets $td80$ from Martins (2007) and Uchoa, Fukasawa, Lysgaard, Pessoa, de Aragao, and Andrade (2008). Finally, for sets $cm50$, $cm100$, and $cm200$, best known solutions are taken from Ahuja, Orlin, and Sharma (2003), Ahuja, Orlin, and Sharma (2001), Ahuja, Orlin, and Sharma (2003) and Uchoa, Fukasawa, Lysgaard, Pessoa, de Aragao, and Andrade (2008). The values of those best known solutions are given for each instance in Tables 5.13, 5.14 and 5.15, under the heading best/opt. Known optima are highlighted in bold.

Table 5.9 summarizes the results obtained with BRKGA for the different groups of instances. Entries are averages taken over all the instances in the same group. Columns *Average mean gap*, *Minimum mean gap* and *Maximum mean gap* give, respectively, the average, the minimum and the maximum, over the instances in the same group, of the average of deviations with respect to the best known solution taken over the 7 runs of the algorithm. Instead, columns *Average best gap*, *Minimum best gap* and *Maximum best gap* give results related to the best of the seven runs of each instance. That is, *Average best gap*, *Minimum best gap* and *Maximum best gap* give the values of the average, minimum and maximum percentage deviation of the solution obtained in the best of the 7 runs of each instance, respectively. Column *Mean avg. CPU* shows the average CPU time until termination for the seven runs in seconds, while column *CPU to best* gives the average CPU time until the best solution was found. The difference between these two last columns, gives the average CPU time spent to fulfill the stopping criterion after the algorithm has found the best UB (for each run).

More detailed results for all the test instances are given in Tables 5.13, 5.14 and 5.15. In particular, columns *Best BRKGA* and *Worst BRKGA* show, respectively, the best UB and the worst UB obtained by the algorithm for each of the instances after the seven runs.

As can be seen, the obtained results are good. Our BRKGA found the best-known solution for 77 out of the 81 UD instances and for 34 out of the 45 non-UD instances. Furthermore, for 7 instances, the algorithm found a solution improving the currently best-known solution. The specific instances and the value of the new best solution we found are given in Table 5.10. In general, our algorithm was robust in the sense that there are small variations in the output of different runs on the same instances. This can be appreciated by comparing columns *Best BRKGA* and *Worst BRKGA* in Tables 5.13-5.15 and also in Figure 5.1 where an histogram is given for the values of the percent deviations of the worst solution found among the seven runs with respect to the best one, for all the instances.

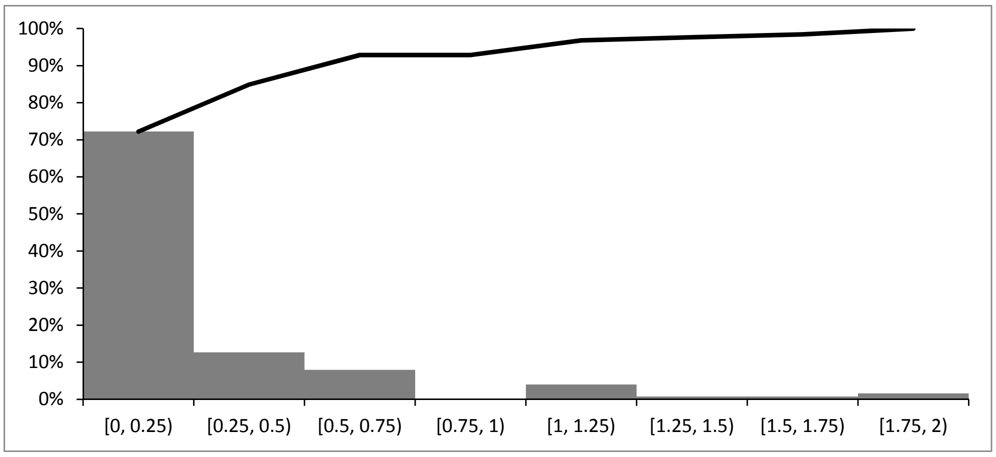

Figure 5.1: Histogram of percent deviations of best with respect to worst solution found in the seven runs.

In fact, there is a considerable number of instances for which BRKGA found the best-known solution in each of the 7 executions. For the UD instances this happens in 38 out of the 45 instances with 80 vertices, 11 out of the 30 instances with 120 vertices and in 1 out of 6 for the 160 vertices instances. The performance of the algorithm is however less predictable with the non-UD instances. The algorithm works remarkably well for the instances with larger capacities ($Q = 800$) where it always produces a best-known solution for all instances with 50 and 100 vertices, and it improves the best-known solution in 2 out of the 5 instances with 200 vertices. Similar results are obtained for the instances with medium capacities ($Q = 400$), where it finds a best-known solution for all instances with up to 100 vertices except one, and it improves the value of the best-known solution for 3 out of the 5 instances with 200 vertices, although for the other 2 instances in this group, it was not able to find a solution with the best-known value. The performance of the algorithm somehow decreases with some of the instances with smaller capacities ($Q = 200$). Now, BRKGA finds an optimal solution for all the small 50 vertex instances, and 2 out of the 5 instances with 100 vertices, but it fails in finding a best known solution for the remaining 8 instances. In fact, we could improve the performance of our algorithm for this specific subset of instances by further fine-tuning the values of the parameters of the BRKGA heuristic, but we preferred to use the same parameters values for all the instances in the same class at the expense of

Table 5.9: Summary BRKGA results

Group	n	Average mean gap	Minimum mean gap	Maximum mean gap	Average best gap	Minimum best gap	Maximum best gap	Mean avg. CPU(secs.)	CPU to best (secs.)
$tc80$	80	0.004	0.005	0.069	0.000	0.000	0.000	99.77	24.45
$td80$	80	0.004	0.000	0.022	0.000	0.000	0.000	259.01	125.76
$te80$	80	0.005	0.005	0.035	0.000	0.000	0.000	217.88	76.76
$tc120$	120	0.039	0.000	0.224	0.000	0.000	0.000	324.55	147.31
$te120$	120	0.091	-0.129	0.457	0.013	-0.259	0.245	694.73	422.37
$tc160$	160	0.168	0.115	0.197	-0.025	-0.076	0.000	1753.70	1210.47
$te160$	160	0.011	0.000	0.025	0.000	0.000	0.000	1587.30	941.33
$cm50$	50	0.072	0.000	0.524	0.000	0.000	0.000	98.97	6.23
$cm100$	100	0.354	0.000	2.072	0.171	0.171	1.541	459.50	122.18
$cm200$	200	0.820	-0.781	2.894	0.412	0.336	2.666	3275.26	2092.95

Table 5.10: New best-known solutions

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

not obtaining the best possible results.
The times required by the BRKGA algorithm to obtain the above results are quite modest. All UD instances with Euclidean distances and 80 vertices ($te80$, $td80$, $te80$) terminated in less than 5 minutes. The average CPU time to termination for instances in the same class with more than 80 vertices ($tc120$, $te120$, $tc160$ and $te160$) was less than haf an hour; indeed the most time consuming instances in this group where $te160$ that took an average of 1753.70 seconds. In total, the most demanding instances were, non-UD instances $cm200$ where the average computing time is 3275.26 seconds (less than an hour). Additionally, in Table 5.9 there is a comparison for each set of instances, between the average CPU times (in seconds) and the average CPU times when the best solution was found. It is clear from the results that many CPU time is spend to fulfill the stopping criterion, as many of the best UB's are found in early iterations. It is important to remember that the stopping criterion was set using the information obtained in the initial experiments which led to the formulas presented in section 5.2.1.

Since the stopping criterion was defined in function of the capacity Q, it was expected that Q will have an effect over the CPU times. Figure 5.2 shows the average CPU times over instances of the same size and with the same capacity. Here, for an easier comparison of the behavior of the algorithm for UD and non-UD instances, capacities are presented in terms of the average number of vertices that fit in a tree (r in the figure). In the case of UD instances $r = Q$, whereas, in the case of non-UD instances $r = Q/50$, since 50 is approximately the average demand of the vertices in the non-UD instances. The figure shows clearly how the time taken by BRKGA to solve the instances increases with the instance size in a similar way for UD and non-UD instances. As opposite, as already noticed in the preliminary tests carried out for tunning the algorithm, the effect of the instance capacities on the computational effort required for their solution is different depending on the type of instance, being much more relevant in the case of UD instances. Nonetheless, in both cases, the instances that seem to

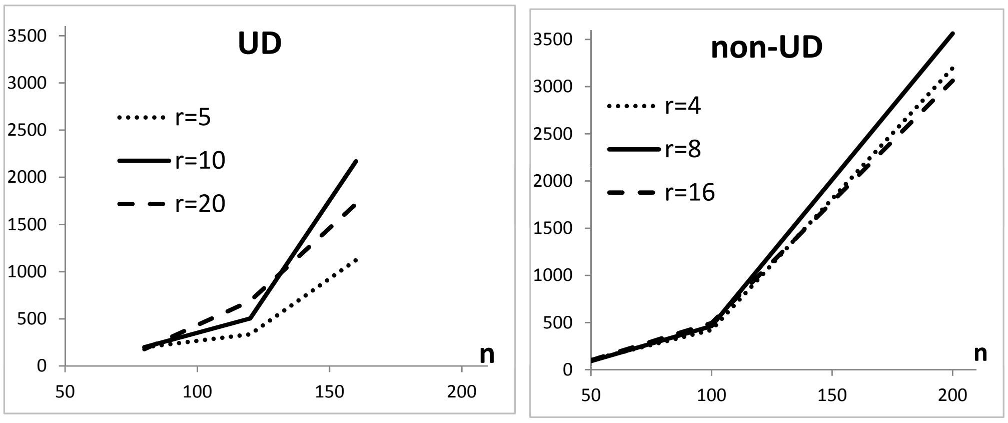

Figure 5.2: CPU time increase with instance size for different values of r, the average number of terminals per tree

require the smallest times are those where capacity constraints are either the tightest or the loosest, while situations in between appear to be the most difficult ones.

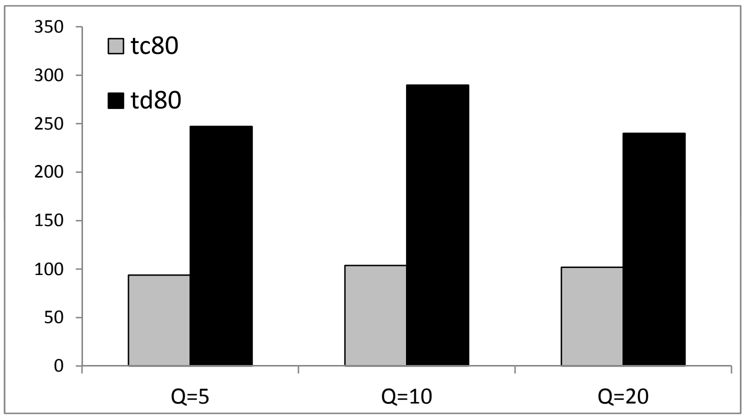

Figure 5.3: Average CPU times Euclidean between sets $tc80$ and $td80$

In order to evaluate the sensitivity of BRKGA to the type of distances we have concentrated in instance sets $tc80$ and $td80$, containing UD instances of the same sizes and the same type of location of the root. The only difference between those sets is the type of distances; they are Euclidean in set $tc80$ and non Euclidean in set $td80$. Figure 5.3 shows the average CPU times taken to solve the problem for the different capacities considered in these sets. The figure shows very different behaviors in the two sets. In particular, the computational effort required to solve instances with non-Euclidean distances is much higher than what is needed for solving instances with Euclidean distances with the same characteristics. As for the effect of the capacity in the instances difficulty, the observed behavior coincides with what was observed in general, solving instances with extreme capacities tends to be a bit less demanding than for moderate capacities, and this effect is larger for non-Euclidean instances.

We close this section by comparing our results with those of other heuristics for the

CMST. For the comparison we have chosen the VLNS heuristic of Ahuja, Orlin, and Sharma (2003) which, in our opinion, is so far the heuristic which has produced the best results, the more recent ant colony heuristic (ACO) of Reimann and Laumanns (2006), the enhanced second order (ESO) algorithm by Martins (2007) and the RAMP heuristic of Rego, Mathew, and Glover (2010).

For instances tc and te with 80 vertices and UD the best results are obtained by the VLNS algorithm, although the results of the BRKGA are also very good since the average mean gaps are really close to zero. The ACO, ESO and RAMP algorithms perform worse than the BRKGA and VLNS. For instances of type td and 80 vertices, the results of the BRKGA are better than those of the ESO in terms of average mean gap. For instances of $tc120$, $te120$, $tc160$ and $te160$ vertices and UD, the best average mean gaps are obtained by the BRKGA with values below 2% for the set $te160$ and below 1% for the other three sets. For the non-UD set with 50 vertices, again, the VLNS performs better than the BRKGA and the other algorithms. For non-UD instances with 100 and 200 vertices, the RAMP algorithm obtains the best average gap, although only instances with $Q = 200$ are considered in that work. The BRKGA obtains very close results in instances with $Q = 200$, and improves the results obtained by VLNS in instances with $Q = 400$ and $Q = 800$. Notice that for this set of instances, the average CPU times of the RAMP algorithm are very large. Finally, for non-UD instances with 200 vertices, the BRKGA obtains better results than the other algorithms in terms of the average mean gap.
Comparing the VLNS with the BRKGA in terms of the mean average gap, we can see that the BRKGA performs better for the instances with larger number of vertices (100, 120, 160 and 200) and slightly worse for the sets with 80 vertices. If compared with the ACO, the BRKGA always performs better in terms of the mean average gap. This is explained by the fact that ACO was designed to obtain fairly good results using a small computation effort (small CPU times). It is important to mention that the results of the ACO for non-UD instances were not published for reasons related to its performance, which are explained in the original article. If compared with the ESO, the BRKGA mean average gaps are also better. Finally, the BRKGA results are also better in terms of the average mean gap for all test instances, when compared to those of the RAMP, except for the $cm100$ as it has been explained. In fact, in general terms, the BRKGA obtains good results without regard of the type of test instance.
Since it is hard to compare the CPU times of the different algorithms, we decided to compile the BRKGA with the previously mentioned compiling options (g++ with the the flags "-c" and "-O3"), on a pentium IV machine with 712 MB of RAM, and solved some of the test instances. The results showed that the pentium core2 was about 10 times faster than the Pentium IV. Therefore, the 1/10 of a second of CPU time on the pentium core2 represents approximately one second of CPU timee on a pentium IV. No test were performed on machines with pentium M or Athlon processors.

In table 5.12 the best results obtained by each algorithm for the different sets of instances are presented. The results shows that the BRKGA obtains the best results among the different algorithms. For the UD sets, the BRKGA found the best know solutions for 77 of the 81 test instances. For the non-UD sets, the algorithm found the best known solution for 33 out of the 45 instances.

Table 5.11: Comparison of average mean GAP and CPU time by instance group with state of the art heuristics

Group	n	VLNS Average mean gap	VLNS Mean avg. CPU(secs.)	ACO Average mean gap	ACO Mean avg. CPU(secs.)	ESO Average mean gap	ESO Mean avg. CPU(secs.)	RAMP Average mean gap	RAMP Mean avg. CPU(secs.)	BRKGA Average mean gap	BRKGA Mean avg. CPU(secs.)
$tc80$	80	0.000	1800	0.205	6.52	0.078	3600	0.018	267.33	0.004	99.77
$td80$	80	NA	NA	NA	NA	0.075	3600	NA	NA	0.004	259.01
$te80$	80	0.000	1800	0.085	18.81	0.179	3600	0.217	1842.67	0.005	217.88
$tc120$	120	0.281	NA	0.074*	66.00	0.474	5400	NA	NA	0.039	324.55
$te120$	120	0.240	NA	0.766*	178.00	0.458	5400	NA	NA	0.091	694.73
$tc160$	160	0.716	NA	0.867	543.33	0.412	7200	NA	NA	0.168	1753.70
$te160$	160	0.583	NA	0.358	545.00	0.216	7200	NA	NA	0.011	1587.30
$cm50$	50	0.020	1000	NA	NA	NA	NA	0.146	850.53	0.072	98.97
$cm100$	100	0.407	1800	NA	NA	NA	NA	0.314**	35800.00	0.354	459.50
$cm200$	200	1.021	3600	NA	NA	NA	NA	NA	NA	0.820	3275.26
Computer used		Pentium 4		Pentium M		AMD Athlon		Pentium P4		Pentium Core2	

*Only $tc120$-1 and $te120$-1 instances were tested.
** Only instances with $Q = 200$ were solved.
NA, no information was found/available for the algorithm and the set of instances.

Table 5.12: Number of best known solutions found by instance group for each state of the art heuristic

Group	n	total number of instances	VLNS	ACO	ESO	RAMP	BRKGA
$tc80$	80	15	15	8	11	13	15
$td80$	80	15	NA	NA	5	NA	15
$te80$	80	15	15	10	7	6	15
$tc120$	120	15	5	2*	7	NA	15
$te120$	120	15	3	1*	1	NA	11
$tc160$	160	3	0	0	0	NA	3
$te160$	160	3	0	0	0	NA	3
$cm50$	50	15	14	NA	NA	11	15
$cm100$	100	15	9	NA	NA	3**	11
$cm200$	200	15	5	NA	NA	NA	8

*Only tc120-1 and te120-1 instances were tested.
** Only instances with $Q = 200$ were solved.
NA, no information was found/available for the algorithm and the set of instances.

Table 5.13: Results for UD instances with $n = 80$ vertices

Group	Q	Instance number	Average gap	CPU (seconds)	Best/Opt* Solution	Best BRKGA	Worst BRKGA
$tc80$	5	1	0.00	38.84	**1099**	1099	1099
		2	0.00	99.04	**1100**	1100	1100
		3	0.00	106.49	**1073**	1073	1073
		4	0.00	105.53	**1080**	1080	1080
		5	0.00	119.08	**1287**	1287	1287
	10	1	0.00	53.06	**888**	888	888
		2	0.00	105.36	**877**	877	877
		3	0.00	112.78	**878**	878	878
		4	0.07	121.26	**868**	868	872
		5	0.00	126.02	**1002**	1002	1002
	20	1	0.00	61.63	**834**	834	834
		2	0.00	117.17	**820**	820	820
		3	0.00	102.85	**828**	828	828
		4	0.00	103.56	**820**	820	820
		5	0.00	123.88	**916**	916	916
$td80$	5	1	0.00	279.23	**6068**	6068	6070
		2	0.00	266.01	**6019**	6019	6019
		3	0.01	285.57	**5994**	5994	5997
		4	0.00	206.99	**6012**	6012	6012
		5	0.00	197.89	**5977**	5977	5977
	10	1	0.00	341.31	**3223**	3223	3223
		2	0.00	228.83	**3205**	3205	3205
		3	0.00	317.28	**3212**	3212	3212
		4	0.00	323.99	**3203**	3203	3203
		5	0.00	237.86	**3180**	3180	3180
	20	1	0.00	196.10	**1832**	1832	1832
		2	0.02	262.74	**1829**	1829	1830
		3	0.00	230.66	**1839**	1839	1839
		4	0.02	243.12	**1834**	1834	1835
		5	0.00	267.65	**1826**	1826	1826
$te80$	5	1	0.00	180.05	**2544**	2544	2544
		2	0.03	341.61	**2551**	2551	2556
		3	0.01	270.51	**2612**	2612	2614
		4	0.03	277.84	**2558**	2558	2560
		5	0.00	199.06	**2469**	2469	2469
	10	1	0.00	186.13	**1657**	1657	1657
		2	0.00	215.17	**1639**	1639	1639
		3	0.00	221.18	**1687**	1687	1687
		4	0.00	218.45	**1629**	1629	1629
		5	0.00	201.22	**1603**	1603	1603
	20	1	0.00	185.14	**1275**	1275	1275
		2	0.00	190.74	**1224**	1224	1224
		3	0.00	187.42	**1267**	1267	1267
		4	0.00	216.44	**1265**	1265	1265
		5	0.00	177.27	**1240**	1240	1240

* Marked in bold are the values that have been proven to be optimal.

Table 5.14: Results for UD instances with $n = 120$ and $n = 160$ vertices

Group	Q	Instance number	Average gap	CPU (seconds)	Best/Opt* Solution	Best BRKGA	Worst BRKGA
$tc120$	5	1	0.00	128.07	**1291**	1291	1291
		2	0.00	189.76	**1189**	1189	1189
		3	0.03	274.47	**1124**	1124	1126
		4	0.03	250.22	**1126**	1126	1128
		5	0.02	235.08	**1158**	1158	1159
	10	1	0.00	168.30	**904**	904	904
		2	0.00	355.37	**756**	756	756
		3	0.00	372.05	**722**	722	722
		4	0.00	276.17	**722**	722	722
		5	0.17	355.59	**761**	761	765
	20	1	0.00	298.73	**768**	768	768
		2	0.00	420.21	**569**	569	569
		3	0.00	442.56	**536**	536	536
		4	0.13	549.16	**571**	571	572
		5	0.22	552.47	**581**	581	585
$te120$	5	1	0.03	413.70	**2197**	2197	2201
		2	0.06	520.85	**2134**	2134	2137
		3	0.03	343.81	**2079**	2079	2080
		4	0.05	464.19	**2158**	2159	2159
		5	0.04	532.96	**2017**	2017	2021
	10	1	0.00	575.22	**1329**	1329	1329
		2	0.45	659.87	**1225**	1228	1235
		3	0.17	809.37	1195	1195	1200
		4	0.23	702.35	1230	1231	1237
		5	0.26	767.34	**1164**	1165	1171
	20	1	0.00	632.49	**920**	920	920
		2	0.09	1034.26	785	785	787
		3	0.02	1058.64	749	749	750
		4	-0.13	920.74	773	771	773
		5	0.08	985.22	746	746	747
$tc160$	5	1	0.20	1034.13	**2077**	2077	2084
	10	1	0.18	2197.84	1319	1318	1327
	20	1	0.12	2029.13	960	960	964
$te160$	5	1	0.03	1211.09	**2789**	2789	2790
	10	1	0.01	2141.96	1645	1645	1646
	20	1	0.00	1408.86	1098	1098	1098

* Marked in bold are the values that have been proven to be optimal.

Table 5.15: Results for non-UD instances with $n = 50$, $n = 100$ and $n = 200$ vertices

Group	Q	Instance number	Average gap	CPU (seconds)	Best/Opt* Solution	Best BRKGA	Worst BRKGA
*cm*50*r*	200	1	0.00	88.78	**1098**	1098	1098
		2	0.53	108.15	**974**	974	980
		3	0.00	102.22	**1186**	1186	1186
		4	0.00	121.37	**800**	800	800
		5	0.00	101.38	**928**	928	928
	400	1	0.13	106.54	**679**	679	681
		2	0.00	62.16	**631**	631	631
		3	0.35	74.30	**732**	732	735
		4	0.08	124.69	**564**	564	567
		5	0.00	91.65	**611**	611	611
	800	1	0.00	131.13	**495**	495	495
		2	0.00	108.15	**513**	513	513
		3	0.00	90.67	**532**	532	532
		4	0.00	81.30	**471**	471	471
		5	0.00	92.04	**492**	492	492
*cm*100*r*	200	1	1.04	439.57	**509**	509	517
		2	2.08	368.24	**584**	593	597
		3	0.26	484.89	**540**	541	542
		4	0.33	383.76	**435**	435	437
		5	0.89	428.98	**418**	420	425
	400	1	0.17	439.19	**252**	252	253
		2	0.36	397.48	**277**	278	278
		3	0.18	457.12	**236**	236	237
		4	0.00	523.62	**219**	219	219
		5	0.00	495.37	**223**	223	223
	800	1	0.00	489.72	**182**	182	182
		2	0.00	479.85	**179**	179	179
		3	0.00	466.91	**175**	175	175
		4	0.00	485.28	**183**	183	183
		5	0.00	552.53	**186**	186	186
*cm*200*r*	200	1	1.67	3122.37	994	1003	1015
		2	1.96	3007.78	1188	1202	1225
		3	2.89	3061.93	1313	1348	1355
		4	1.34	3561.27	917	919	937
		5	2.11	3236.41	948	964	974
	400	1	0.91	3097.51	391	392	397
		2	0.30	3186.94	476	475	480
		3	-0.10	4040.28	559	557	560
		4	0.29	3447.51	389	388	392
		5	0.85	4046.50	418	421	423
	800	1	0.00	2778.14	254	254	254
		2	-0.05	3133.56	294	293	294
		3	-0.08	3187.92	361	360	361
		4	0.00	3043.11	275	275	275
		5	0.20	3177.69	292	292	293

* Marked in bold are the values that have been proven to be optimal.

5.3 Experimental results with the cutting plane algorithms

The two cutting plane algorithms based on the formulations proposed in Section 3.1 and Section 3.3, were implemented in C++ and compiled with *g++*. The software used to solve the LP relaxation during the cutting plane algorithm was *CPLEX 12.1.* The tests were executed in a PC with an intel Dual Core 3.1 GHz processor and 2 GB of RAM. Due to memory limitations in the equipment no tests were performed for instances of more than 100 vertices. As mentioned in Section 3.2.5, both algorithms require the use of an upper bound (UB) to perform some variable elimination tests. In our experiments we have used the outcome of the BRKGA heuristic to obtain the UB. We remind the reader that the LB is obtained computing a MST with the minimum subroots constraint (3.25). In Table 5.16, column *Avg, gap* shows the average deviation of the LB with respect to the optimal solution by group of instances.

Table 5.16: Average percentage deviation for the obtained LB's

Group	vertices	Q	Avg gap
*tc*80	80	5	12.43
		10	3.52
		20	0.64
		Average	5.53
*td*80	80	5	8.31
		10	7.14
		20	3.82
		Average	6.42
*te*80	80	5	32.83
		10	22.80
		20	8.74
		Average	21.45
*cm*50	50	200	32.72
		400	23.43
		800	9.30
		Average	21.82
*cm*100	100	200	17.56
		400	8.46
		800	6.45
		Average	10.82

From the results of Table 5.16 it is clear that better LB's are obtained for higher values of Q, which is something expected, since solutions with big values of Q have more common elements with the solution of the MST with the minimum subroots constraint. Also the type of distances (euclidean and non-euclidean) have an important effect, since instances with non-euclidean distances (*td*) have better gaps than those instances with euclidean distances (*tc*, *te*). The location of the root also has an effect since *tc* and *te* instances have closer gaps than the *te* ones. Finally, it is also clear that for non-UD instances the gap between the LB and the UB increases.

5.3.1 Variable elimination

Special attention was put on the variable elimination procedure because the two formulations presented in Chapter 3 have an important number of variables. For both formulations a large number of variables could be eliminated using the variable elimination procedure presented in Sections 3.3.1 and 3.2.5. Hence, the size of the instances could be initially reduced. In Table 5.17 we present the percentage of variables eliminated for each formulation for the groups of UD instances *tc*80, *td*80 and *te*80. Column *Original* shows the original number of variables by group of instance, while columns $Left$ and %*Eliminated* show, respectively, the average number of variables left and the percentage of eliminated variables after the variable elimination procedure. From this table it is easy to see that an important reduction on the number of variables could be obtained. In particular, for the subroot formulation, the percentage reduction on the number of variables ranges in $80 - 83\%$ for the *te* instances, in $79 - 96\%$ for the *td* instances and in $92 - 99\%$ for the *tc* instances. For the hop indexed subroot formulation, the percentage reduction on the number of variables ranges in $84 - 86\%$ for the *te* instances, in $89 - 97\%$ for the *td* instances and in $95 - 99\%$ for the *tc* instances.

Broadly speaking the percentage reduction was somehow bigger for the hop indexed subroot formulation. However, the number of variables after the elimination procedure is still large for the hop indexed subroot formulation and instances of set *te*. For both types of formulations, it is clear that the reduction of variables for instances of set *te* is not as good as for the set *tc* and *td*. For instances of set *td* most of the variables are eliminated using the LB and UB criterion (many vertices are discarded as subroots), while for *tc* (with $Q \neq 20$) and *te* instances, variables are mostly eliminated using the optimality criteria. For instances of set *tc* with $Q = 20$, variables are eliminated by both of the elimination criteria.

The results in Table 5.17 also show that for the subroot formulation, the percentage of eliminated variables for instances *tc*, *td* and *te* increases as Q grows. This is due to the reduction of the gap between the LB and the UB which allows the elimination of more variables. However, for the subroot hop indexed formulation, this only happens for *tc* and *td* instances. Remember that in this formulation, the number of variables increases as Q grows. For *tc* and *td* instances the variable elimination procedures, are able to compensate the augmentation in the number of variables due to the reduction on the gap between the LB and UB. Nevertheless, for *te* instances, the gap reduction between the LB and UB is not enough to make the elimination procedures compensate the augmentation in the number of variables when Q increases from 5 to 10.

Since the hop indexed subroot formulation is a specialized formulation for the UD case, in Table 5.18 we present the results of the variables elimination procedure for the non-UD instances only for the subroot formulation. The percentage reduction was somehow bigger for the *cm*100 instances than for *cm*50. Again, this might be explained by the fact that the gaps between the LB and UB are tighter for the *cm*100 instances. Most of the elimination of

Table 5.17: Variable Elimination Comparison for UD instances

			Subroot Formulation			Hop Indexed Subroot Formulation		
Group	vertices	Q	Original	Left	%Eliminated	Original	Left	%Eliminated
*tc*80		5	518400	40116	92.26	2566400	103901	95.95
	80	10	518400	30276	94.16	5126400	204977	96.00
		20	518400	4187	99.19	10246400	59823	99.42
*td*80		5	518400	107519	79.26	2566400	267982	89.56
	80	10	518400	31962	93.83	5126400	231556	95.48
		20	518400	16691	96.78	10246400	289130	97.18
*te*80		5	518400	101477	80.43	2566400	345016	86.56
	80	10	518400	101477	80.43	5126400	808980	84.22
		20	518400	84253	83.75	10246400	1456716	85.78

variables in $cm50$ sets is performed based on the optimality criteria, since there are important gaps between the LB and the UB. The same happens for $cm100$ instances with $Q = 200$. For the other two groups of $cm100$ instances, the optimality and LB criteria help reduce the number of variables importantly.

Table 5.18: Variable elimination of subroot formulation for non-UD instances

Group	vertices	Q	Original	Left	%Eliminated
*cm*50	50	200	120050	10428	91.31
		400	120050	12889	89.26
		800	120050	10289	91.43
*cm*100	100	200	980100	73820	92.47
		400	980100	28528	97.12
		800	980100	7936	99.19

5.3.2 Results obtained with the subroot formulation

The final design of Algorithm 1, presented in Section 3.2.5, is the result of a series of preliminary experiments testing alternative configurations. We first analyzed the effect of the different inequalities proposed for the subroot formulation. To carry out this analysis, each of the proposed families of valid inequalities (bin-packing, subroot multistar, s-tree multistar, and subroot rootcutset was considered alone, together with the basic formulation (3.2)-(3.8).

Table 5.19 shows the average LP gaps of the resulting formulations, together with the CPU times taken to find them.

For separating bin-packing and subroot rootcutset the heuristic vertex contraction procedure described in Section 3.2.5 was used. For subroot multistar inequalities and s-tree multistar the exact separation procedures presented in Section 3.2.5 and 3.2.5, respectively, were used. For this test, the stopping criterion was that no new cuts were found by the separation procedures.

The results in Table 5.19, show the impact of the different inequalities on the improvement of the LP lower bound. In all cases, column *Avg gap* shows the average percent deviation by group of instances, of the obtained LP lower bounds from the optimal solution. Note that, as shown in Table 5.13-5.15, optimal solutions are known for all the instances considered in this test. Finally, column *Avg CPU* shows average CPU time by group of instances. It is clear from Table 5.19 that using s-tree multistar inequalities requires sensibly larger CPU times, but results in much better lower bounds, than using any of the other families alone. One of the reasons for having large CPU times, is that at, some point, although we find violated inequalities from this family, the gap does not decrease significantly. The poorest results are for the bin-packing inequalities mainly due to two causes; the first is that the heuristic separation procedure is not able to find many violated inequalities, and the second is that the cuts generated have less effect on increasing the bound. Subroot rootcutset have a more relevant effect on the lower bound, with the inconvenient that the separation procedure is neither very effective for finding violated inequalities. The effect of subroot multistar inequalities on the lower bound is stronger than that of the subroot rootcutset inequalities, at the cost of larger CPU times. Moreover, CPU times are also smaller for the formulation with the subroot rootcutset inequalities.

Using the above results we decided to discard the use of bin-packing inequalities. Then, we developed two algorithms. The first one, *solalg*1 separated inequalities subroot multistar and subroot rootcutset, while the second one, *solalg*2 separated$s$-tree multistar and subroot rootcutset. The separation procedures for each family of inequalities suffered no changes. The results in Table 5.20 show that both algorithms improved the previous results. This is explained by the fact subroot multistar and s-tree multistar are designed to improve the connectivity between terminals, while the subroot rootcutset, help to improve the connectivity of the terminals with the root.

Given that the above results indicate that contributions of multistar and s-tree multistar inequalities to the gap reduction seem to be different, we considered the separation of all three remaining families of valid inequalities (all except bin-packing), giving rise to the final algorithm.

The summary results are presented in Table 5.21 and they show that in terms of gap there was a minimal improvement, but CPU times improved importantly. The reason for the CPU time improvement is that some sets of subroot multistar inequalities help to avoid the separation of many s-tree multistar inequalities for candidate subroots that are not connected to the root in the current solution and therefore the size of the problem does not grows excessively. Complete results are presented in Tables 5.23 and 5.24 for UD and non-UD instances respectively. The column *opt* refers to the value of the optimal solution of the instance. Columns *Bound* and *gap* refer, respectively, to the LP lower bound and percentage deviation gap of the solution obtained by the algorithm. Columns *CPU* denotes the CPU time in seconds that the algorithm used to find the LP lower bound.

Table 5.19: Individual effect of each family of valid inequalities

	LP relaxation		Bin-packing inequalities		Multistar inequalities		s-tree multistar inequalities		Rootcutset inequalities	
Group	Avg gap	CPU	Avg gap	Avg CPU	Avg gap	Avg CPU	Avg gap	Avg CPU	Avg gap	Avg CPU
$cm50$	34.07	0.12	20.30	7.37	7.18	8.92	5.61	21.56	9.71	5.37
$cm100$	12.55	5.22	6.54	299.68	4.54	252.74	4.10	917.58	7.99	139.01
$tc80$	6.64	0.27	5.36	46.83	1.78	11.24	1.20	27.06	4.09	11.02
$td80$	15.18	2.28	10.23	128.35	3.26	216.58	2.95	985.60	2.52	98.37
$te80$	28.27	1.33	20.85	143.11	4.85	297.02	3.10	1790.10	7.55	78.62

Table 5.20: Inequalities combination, effect on the gap by groups of test instances

	*solalg*1		*solalg*2	
Group	Avg gap	CPU	Avg gap	Avg CPU
$cm50$	6.08	38.65	4.06	91.25
$cm100$	3.95	347.02	3.68	1517.93
$tc80$	1.53	17.64	1.13	327.06
$td80$	2.03	480.24	1.78	2100.87
$te80$	3.89	590.55	2.85	3290.10

Table 5.21: Summary results for subroot formulation algorithm

Group	Avg gap	Avg CPU
$tc80$	1.12	30.86
$td80$	1.60	607.99
$te80$	2.84	2765.40
$cm50$	4.01	30.23
$cm100$	3.43	683.48

The results show that the location of the root node and the type of distances have an important effect on the gaps and CPU times on the UD instances. For tc (root in the center and euclidean distances) the average gap is close to one percent and the CPU time is close to 30 seconds. For td instances (root in the center and euclidean distances), the change in the CPU time and the gap reflect the effect of having non euclidean distances. The difference between the results for tc and te instances, reflect the effect produced by the location of the root node. Somehow this was expected, since the number of variables for te instances is greater than for those for the tc and td ones, except for td with $Q = 5$.

For non-UD instances, we can observe in the average gaps the effect of having non unitary demands and non euclidean distances. The gaps increase significantly if compared with the UD instances, although the initial number of variables usually are greater than those for tc instances but smaller than for td and te ones.

The effect generated by the number of vertices, is observed when comparing the results of $cm50$ and $cm100$ instances. The results show that CPU times are significantly greater when

Table 5.22: Comparison with other formulations

Group	*Subroot*		*q − arb*		*HIF*	
	Avg gap	Avg CPU	Avg gap	Avg CPU	Avg gap	Avg CPU
$tc80$	1.12	30.86	0.00	30.42	0.31*	554.5
$td80$	1.60	607.99	0.00	15480.65	NA	NA
$te80$	3.13	2765.40	0.00	3562.17	0.55*	2273
$cm50$	4.01	30.23	0.00	150.77	**	**
$cm100$	3.43	683.48	0.00	23241.27	**	**

* Instances with $Q = 20$ were not solved. **The formulation is designed to solve only UD instances NA, no information was found/available for the algorithm and the set of instances.

the number of vertices increases. Surprisingly, the average gap is greater for $cm50$ instances than for $cm100$. This behavior is explained if we look at the percentage of eliminated variables, which is greater for $cm50$ than for $cm100$ (See Table 5.18) We can conclude that the type of demand (UD and non-UD), the location of the root(centered or not), and the type of distances (euclidean and non-euclidean) have influence over the initial number of variables, which have an effect on the gaps and CPU times.

We close this section by comparing the obtained results with the ones obtained by other formulations. We use for comparison the q-arb formulation by Uchoa, Fukasawa, Lysgaard, Pessoa, de Aragao, and Andrade (2008) and the hop indexed formulation (HIF) by Gouveia and Martins (2005) which are the more recent works. Table 5.22 presents the above mentioned comparison. Columns $Subroot$, $q - arb$ and HIF show, respectively, the results for the subroot, q-arb and hop indexed formulations.

The best results in terms of gap are obtained by the q-arb formulation, which finds the optimal solution of all the instances with significantly higher CPU times than for the other two formulations. The hop indexed formulation gaps for UD instances are between those of the q-arb and subroot formualtion. In general the subroot formulation obtains small gaps at small CPU times.

5.3.3 Results obtained with the subroot hop indexed formulation

To developed the subroot hop indexed formulation the solution algorithm presented we used the information already obtained for the subroot formulation. Therefore, the algorithm included the separation of the extensions of the three families of valid inequalities used above: subroot multistar, s-tree multistar and subroot rootcutset. Additionally, GSEh and hop ordering inequalities are separated. The separation problem for GSEh inequalities is solved heuristically by solving a maximum flow problem over a customized graph (see Section 3.3), while hop ordering inequalities are separated by inspection. A complete description of the algorithm and the separation problems can be found in Section 3.3.1.

Table 5.23: Complete results for UD instances with 80 vertices using subroot formulation

Group	Q	Instance	*Bound*	gap	CPU
*tc*80	5	1	**1099**	1.78	123.91
		2	**1100**	1.81	62.70
		3	**1073**	1.99	38.39
		4	**1080**	2.69	42.78
		5	**1287**	2.31	44.75
	10	1	**888**	1.20	75.39
		2	**877**	0.54	16.80
		3	**878**	1.59	12.66
		4	**868**	0.81	7.13
		5	**1002**	0.97	28.50
	20	1	**834**	0.48	2.84
		2	**820**	0.49	1.17
		3	**828**	0.00	1.50
		4	**820**	0.00	0.87
		5	**916**	0.27	3.44
*td*80	5	1	**6068**	1.93	771.95
		2	**6019**	1.78	772.16
		3	**5994**	1.77	623.85
		4	**6012**	1.46	726.30
		5	**5977**	1.22	651.66
	10	1	**3223**	2.11	637.90
		2	**3205**	1.47	745.45
		3	**3212**	2.12	610.59
		4	**3203**	1.12	471.11
		5	**3180**	1.26	642.07
	20	1	**1832**	1.20	131.51
		2	**1829**	1.53	838.03
		3	**1839**	1.85	383.67
		4	**1834**	1.31	542.67
		5	**1826**	1.92	570.90
*te*80	5	1	**2544**	1.43	3522.47
		2	**2551**	2.28	2738.87
		3	**2612**	2.34	3730.89
		4	**2558**	2.45	2110.47
		5	**2469**	2.22	1987.83
	10	1	**1657**	5.51	5693.27
		2	**1639**	5.66	5802.42
		3	**1687**	4.63	5631.63
		4	**1629**	4.66	4482.54
		5	**1603**	3.35	4206.96
	20	1	**1275**	2.46	205.40
		2	**1224**	2.52	679.20
		3	**1267**	2.64	268.73
		4	**1265**	2.76	309.87
		5	**1240**	2.05	111.28

Table 5.24: Complete results for non-UD instances of up to 100 vertices using subroot formulation

Group	Q	Instance	*Bound*	gap	CPU
$cm50$	200	1	**1098**	5.49	24.78
		2	**974**	5.38	25.23
		3	**1186**	5.86	27.17
		4	**800**	5.08	4.44
		5	**928**	6.41	15.02
	400	1	**679**	4.40	142.09
		2	**631**	2.23	42.92
		3	**732**	6.85	56.61
		4	**564**	4.79	5.73
		5	**611**	4.76	17.52
	800	1	**495**	3.16	18.61
		2	**513**	1.77	36.34
		3	**532**	1.72	29.84
		4	**471**	0.90	1.09
		5	**492**	2.16	6.05
$cm100$	200	1	**509**	7.51	444.76
		2	**252**	2.37	443.31
		3	**182**	0.38	77.19
		4	**584**	6.03	1444.55
		5	**277**	1.47	1079.74
	400	1	**179**	0.98	47.72
		2	**540**	6.97	964.04
		3	**236**	2.75	439.06
		4	**175**	1.61	30.59
		5	**435**	8.70	739.28
	800	1	**219**	3.49	2443.41
		2	**183**	0.41	120.94
		3	**418**	8.50	576.32
		4	**223**	2.56	1372.34
		5	**186**	1.46	29.02

Table 5.25: Summary results for UD instances for subroot hop indexed formulation

Group	vertices	Q	Avg gap	Avg CPU
		5	0.31	1927.03
*tc*80	80	10	0.41	3082.80
		20	0.19	276.39
		Average	0.30	1762.07
		5	0.23	3896.18
*td*80	80	10	0.29	6959.28
		20	0.60	3773.71
		Average	0.37	4876.39
		5	0.58	116799.80
*te*80	80	10	4.64	10783.19
		20	2.07	4248.50
		Average	2.43	43943.39

Since the subroot hop indexed formulation is specially designed for solving the UD case of the problem, the algorithm was tested for the groups of instances $tc80$, $td80$ and $te80$. A summary of the results are presented in Table 5.25. The complete results are shown in Table 5.27

From the results in Table 5.25 we can observe that the average gaps by group of instance obtained by the subroot hop indexed formulation are better for the $tc80$ instances as these are less difficult than the two other groups of instances. Also the CPU times for $tc80$ group, are smaller because the initial number of variables is smaller than for the two other groups (see Table 5.17).
The results for *td* instances are not far from those of *tc* ones, since an important part of the cost of the tree is due to the arcs linking the root with the subroots and somehow the formulation puts special emphasis on these vertices. From Table 5.25 we can see that the gap increases as Q increases (the number of subroots decreases as Q increases). Remember that the proposed formulation focuses on the connectivity of the subroots.
Finally instances of type *te* obtain not so good results as they are the most difficult ones. However, the algorithm obtained small gaps for *te* instances with $Q = 5$, although the CPU times are larger. For this group of instances, the algorithm is capable of finding violated inequalities with the inconvenient that such cuts only represent a small increase in the objective function, thus explaining the large CPU times.
The complete results of the algorithm are shown is Table 5.27. Notice that for 11 of the 45 instances an optimal solution was found.

We close this section by comparing the results of the subroot hop indexed formulation with the subroot formulation, the q-arb and hop indexed formulations. From Table 5.26 we can see that subroot hop indexed formulation obtains better results than those of the subroot formulation but still not as good as the ones by Uchoa, Fukasawa, Lysgaard, Pessoa, de Aragao, and Andrade (2008) and Gouveia and Martins (2005). The gaps obtained for UD instances with the subroot hop indexed formulation are a bit larger than those obtained by Uchoa, Fukasawa, Lysgaard, Pessoa, de Aragao, and Andrade (2008) but the computational burden for obtaining them is, in general, much smaller. The comparison with the results from Gouveia and Martins (2005) is rather poor, since that work does not report experiments on many of the instance groups tested in thi thesis. However, in the groups for which data is available their results are slightly better than those obtained with the subroot hop indexed formulation, specially in what refers to the CPU times. This is probably due to the fact that subroot formulations have a larger number of variables and it requires larger times to solve the LP subproblems.

Table 5.26: Comparison of subroot hop indexed formulation results for UD instances with other formulations

			SRHIF		SR		q-arb		Hop Indexed	
Group	vertices	Q	Avg gap	Avg CPU	Avg gap	Avg CPU	Avg gap	Avg CPU	Avg gap	Avg CPU
$tc80$	80	5	0.31	1927.03	2.11	62.46	0.00	67.46	0.256	276.00
		10	0.41	3082.80	1.02	28.07	0.00	23.50	0.363	833.00
		20	0.19	276.39	0.25	1.96	0.00	0.32	NA	NA
		Average	0.30	1762.07	1.12	30.83	0.00	30.42	0.31	554.50
$td80$	80	5	0.23	3896.18	1.63	709.18	0.00	106.30	NA	NA
		10	0.29	6959.28	1.61	621.42	0.00	45883.30	NA	NA
		20	0.60	3773.71	1.56	493.36	0.00	451.86	NA	NA
		Average	0.37	4876.39	1.60	607.99	0.00	15480.65	NA	NA
$te80$	80	5	0.58	116799.80	2.14	2818.04	0.00	188.84	0.288	786.00
		10	4.64	10783.19	4.76	5163.32	0.00	10452.96	0.811	3760.00
		20	2.07	4248.50	2.49	314.90	0.00	44.72	NA	NA
		Average	2.43	43943.39	3.13	2765.42	0.00	3562.17	0.55	2273.00

Table 5.27: Complete results UD instances for the subroot hop indexed cutting plane algorithm

Group	Q	Instance number	*Bound*	gap	CPU
*tc*80	5	1	**1099**	0.00	797.48
		2	**1100**	0.00	454.76
		3	**1073**	0.00	1140.99
		4	**1080**	0.83	3632.92
		5	**1287**	0.70	3609.00
	10	1	**888**	0.90	4838.04
		2	**877**	0.00	93.79
		3	**878**	1.14	3247.60
		4	**868**	0.00	3776.59
		5	**1002**	0.00	3457.97
	20	1	**834**	0.48	650.07
		2	**820**	0.49	391.94
		3	**828**	0.00	34.59
		4	**820**	0.00	39.75
		5	**916**	0.00	265.60
*td*80	5	1	**6068**	0.33	4531.67
		2	**6019**	0.22	3989.07
		3	**5994**	0.28	3652.14
		4	**6012**	0.17	3665.81
		5	**5977**	0.17	3642.23
	10	1	**3223**	0.37	4912.60
		2	**3205**	0.00	3718.04
		3	**3212**	0.50	4683.27
		4	**3203**	0.03	18451.66
		5	**3180**	0.53	3030.85
	20	1	**1832**	0.33	628.78
		2	**1829**	0.98	5242.27
		3	**1839**	0.49	6303.54
		4	**1834**	0.60	4064.32
		5	**1826**	0.60	2629.65
*te*80	5	1	**2544**	0.00	32714.59
		2	**2551**	0.94	5522.19
		3	**2612**	0.69	5074.56
		4	**2558**	0.90	5521.12
		5	**2469**	0.36	5083.48
	10	1	**1657**	5.37	26899.64
		2	**1639**	5.86	223545.99
		3	**1687**	4.80	57644.98
		4	**1629**	3.93	233945.16
		5	**1603**	3.24	41963.21
	20	1	**1275**	2.82	4510.92
		2	**1224**	2.37	4206.50
		3	**1267**	2.29	4175.73
		4	**1265**	1.74	4156.61
		5	**1240**	1.13	4192.75

Chapter 6

Conclusions and future research

The CMST is one of the most interesting problems in combinatorial optimization and network design. Along with the TSP, the CMST is considered to be the core of network systems design and a wide variety of scheduling and routing applications. In this thesis we focus on the CMST, for which we develop a heuristic method and present two alternative formulations. Both formulations are enhanced with a series of valid inequalities and different procedures are proposed for their separation.

The subroot formulation is the first of the proposed formulations that we have presented. This formulation is based on identifying a special type of vertices on the network;the subroots, which are the vertices directly connected to the root. This type of formulation is aimed at tackling problems in which the location of special equipments in a special type of vertices is of relevance for the design of the network and can be extended in a straight-forward way to situations where there are fixed costs for locating these equipements or different subroot locations have associated different capacities.

Using this formulation, some of the already known valid inequalities are extended. Also new stronger inequalities are presented. These new families of inequalities include the subroot multistar inequalities, the s-tree multistar inequalities and the subroot rootcutset inequalities. The first two families are separated exactly by solving a maximun flow problem. For the third family the subroot rootcutset, a heuristic separation procedure is used.

The cutting plane algorithm for this formulation founds good results, although the gap from the optimal value is not closed in general. A strong point of this algorithm is the variable elimination procedure, which importantly reduces the number of variables of the test instances (95.2% and 81.5% for tc and te instances respectively)

The second proposed formulation is the subroot hop indexed formulation, which is able to obtain better results for the test instances with unitary demands. Extended versions of the subroot multistar, s-tree multistar and subroot rootcutset inequalities are shown. Additionally, GSEh and hop ordering inequalities are included for this formulation. We present exact separation methods for the multistar and s-tree inequalities and heuristic separation methods for the GSEh and subroot rootcutset inequalities.

The solution algorithm for this formulation also implements a variable elimination procedure that yields dramatic reductions on the number of variables. Better lower bounds are obtained using the solution algorithm and for some of the test instances the gap was closed arriving to the optimal solution.

The heuristic method introduced in this thesis, showed to be efficient and robust for solving instances with both UD and non-UD demands, and Euclidean or non Euclidean distances, without practically changing the setup parameters.

By comparing several decoders we have seen that way solutions are encoded has an important effect on the quality of the offspring after crossover. The transmission of genetic information from parents to their offspring showed to be crucial to obtain good quality results, although offsprings may represent infeasible solutions, so a feasibility recovery procedure is needed.

The numerical results presented showed that the BRKGA algorithm is robust, since it obtains good results either for UD and non-UD, euclidean and non euclidean instances. These results are obtained in fairly good CPU times (average CPU times are of less than an hour for the most difficult set of instances).

There are several research opportunities that arise from the results obtained on this thesis that result of interest for future research. For the BRKGA heuristic, reducing the time of MST computations is an issue that could lead to important improvements over the CPU times. The implementation of the BRKGA heuristic to extensions of the problem like the multilevel capacitated minimum spanning tree, also results of interest for future research.

For the two proposed formulations many interesting possibilities for future arise, like the adaptation of the subroot hop indexed formulation to work with the general case and the improvement of the separation methods for the rootcutset and GSEh inequalities. The improvement of the variable elimination procedure, the inclusion of a customized branch and bound algorithm to both of the proposed formulations or the use of column generation or variable aggregation, are interesting issues for future research.

References

Ahuja, R. K., J. B. Orlin, and D. Sharma (2001). Multi-exchange neighborhood structures for the capacitated minimum spanning tree problem. *Mathematical Programming 91*(1), 71–97.

Ahuja, R. K., J. B. Orlin, and D. Sharma (2003). A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem. *Operations Research Letters 31*(3), 185 – 194.

Altinkemer, K. and B. Gavish (1988). Heuristics with constant error guarantees for the design of tree networks. *Management Sci. 34*(3), 331–341. Focussed issue on heuristics.

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

Amberg, A., W. Domeschke, and S. Voß (2000). Multiple center capacitated arc routing problems: A tabu search algorithm using capacitated trees. *European Journal of Operational Research 124*(2), 360–376.

Araque, J., L. Hall, and T. Magnanti (1990). Capacitated trees, capacitated routing, and associated polyhedra, technical report sor-90-12. Master's thesis, Program in Statistics and Operations Research,Princeton University, Princeton, NJ.

Balakrishnan, A. and K. Altinkemer (1992). Using a hop-constrained model to generate alternative communication network design. *ORSA Journal on Computing 4*(2), 192–205.

Battarra, M., T. Öncan, I. Altinel, B. Golden, D. Vigo, and E. Phillips (2011). An evolutionary approach for tuning parametric esau and williams heuristics. *Journal of the Operational Research Society 63*(3), 368–378.

Bean, J. C. (1994). Genetic algorithms and random keys for sequencing and optimization. *ORSA journal on computing 6*(2), 154–160.

Chandy, K. M. and T. Lo (1973). The capacitated minimum spanning tree. *Networks 3*(2), 173–181.

Chandy, K. M. and R. A. Russell (1972, oct.). The design of multipoint linkages in a teleprocessing tree network. *Computers, IEEE Transactions on C-21*(10), 1062 – 1066.

Christofides, A. (1976). Worst case analysis of a new heuristic for the travelling salesman problem report 388. Master's thesis, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA.

de Almeida, A., P. Martins, and M. de Souza (2012). Min-degree constrained minimum spanning tree problem: complexity, properties, and formulations. *International Transactions in Operational Research*.

de Lacerda, E. and M. de Medeiros (2006). A genetic algorithm for the capacitated minimum spanning tree problem. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pp. 725–729.

Elias, D. and M. Ferguson (1974, nov). Topological design of multipoint teleprocessing networks. *Communications, IEEE Transactions on 22*(11), 1753 – 1762.

Esau, L. and K. Williams (1966). On teleprocessing system design, part ii: A method for approximating the optimal network. *IBM Systems Journal 5*(3), 142–147.

Fernández, E., J. Díaz, and E. Ruiz (2005). Búsqueda tabú y búsqueda dispersa para el árbol de expansión capacitado de coste mínimo. *CEDI 2005 9*, 265–279.

Frangioni, A., D. Pretolani, and M. Scutellà (1999). Fast lower bounds for the capacitated minimum spanning tree problem. Technical Report TR-99-05, Dipartamento di informática, Università di Pisa.

Frederickson, G. N., M. S. Hecht, and C. E. Kim (1976, oct.). Approximation algorithms for some routing problems. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pp. 216 –227.

Gamvros, I., B. Golden, and S. Raghavan (2006). The multilevel capacitated minimum spanning tree problem. *INFORMS Journal on Computing 18*(3), 348–365.

Gavish, B. (1982). Topological design of centralized computer networks formulations and algorithms. *Networks 12*(4), 355–377.

Gavish, B. (1983). Formulations and algorithms for the capacitated minimal directed tree problem. *Journal of the Association for Computing Machinery 30*(1), 118–132.

Gavish, B. (1985, dec). Augmented lagrangean based algorithms for centralized network design. *Communications, IEEE Transactions on 33*(12), 1247 – 1257.

Gavish, B. (1989, jan). Topological design of computer communication networks. In *System Sciences, 1989. Vol.III: Decision Support and Knowledge Based Systems Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, Volume 3, pp. 770 –779 vol.3.

Gavish, B. (1991). Topological design of telecommunication networks-local access design methods. *Annals of Operations Research 33*, 17–71. 10.1007/BF02061657.

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

Gonçalves, J. and M. Resende (2011). Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics 17*(5), 487–525.

Gouveia, L. (1993). A comparison of directed formulations for the capacitated minimal spanning tree problem. *Telecommunication Systems 1*, 51–76. 10.1007/BF02136155.

Gouveia, L. (1995a). A 2n-constraint formulation for the capacitated minimal spanning tree problem. *Operations Research 43*(1), 130–141.

Gouveia, L. (1995b). Using the miller-tucker-zemlin constraints to formulate a minimal spanning tree problem with hop constraints. *Computers & Operations Research 22*(9), 959–970.

Gouveia, L. and M. Lopes (2005). The capacitated minimum spanning tree problem: On improved multistar constraints. *European Journal of Operational Research 160*(1), 47–62.

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

Gouveia, L. and P. Martins (2005). The capacitated minimum spanning tree problem: revisiting hop-indexed formulations. *Computers & Operations Research 32*(9), 2435–2452.

Gouveia, L. and J. Paixão (1991). Dynamic programming based heuristics for the topological design of local access networks. *Annals of Operations Research 33*(4), 305–327.

Hall, L. (1996). Experience with a cutting plane algorithm for the capacitated spanning tree problem. *INFORMS Journal on Computing 8*(3), 219–234.

Han, J., G. McMahon, and S. Sugden (2002). A branch and bound algorithm for capacitated minimum spanning tree problem. In B. Monien and R. Feldmann (Eds.), *Euro-Par 2002 Parallel Processing*, Volume 2400 of *Lecture Notes in Computer Science*, pp. 404–407. Springer Berlin / Heidelberg. 10.1007/3-540-45706-2_54.

Karnaugh, M. (1976, may). A new class of algorithms for multipoint network optimization. *Communications, IEEE Transactions on 24*(5), 500 – 505.

Katoh, N., T. Ibaraki, and H. Mine (1981). An algorithm for finding k minimum spanning trees. *SIAM Journal on Computing 10*(2), 247–255.

Kershenbaum, A. (1974). Computing capacitated minimal spanning trees efficiently. *Networks 4*(4), 299–310.

Kershenbaum, A. and R. R. Boorstyn (1983). Centralized teleprocessing network design. *Networks 13*(2), 279–293.

Kruskal, J. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society 7*(1), 48–50.

Laporte., G. and Y. Nobert (1983). A branch and bound algorithm for the capacitated vehicle routing problem. *OR Spectrum 5*, 77–85. 10.1007/BF01720015.

Lee, Y. and M. Atiquzzaman (2005). Least cost multicast spanning tree algorithm for local computer network. *Networking and Mobile Computing*, 268–275.

Letchford, A., R. Eglese, and J. Lysgaard (2002). Multistars, partial multistars and the capacitated vehicle routing problem. *Mathematical Programming 94*(1), 21–40.

Letchford, A. and J. Salazar-González (2006). Projection results for vehicle routing. *Mathematical Programming 105*, 251–274. 10.1007/s10107-005-0652-x.

Little, J., K. Murty, D. Sweeney, and C. Karel (1963). An algorithm for the traveling salesman problem. *Operations research 11*(6), 972–989.

Malik, K. and G. Yu (1993). A branch and bound algorithm for the capacitated minimum spanning tree problem. *Networks 23*(6), 525–532.

Malone, M. and J. C. Bellmore (1971). Pathology of travelling salesman subtour-elimination algorithms. *Operations Research 19*(2), 278–307.

Martins, P. (2007). Enhanced second order algorithm applied to the capacitated minimum spanning tree problem. *Computers & Operations Research 34*(8), 2495–2519.

McGregor, P. and D. Shen (1977, jan). Network design: An algorithm for the access facility location problem. *Communications, IEEE Transactions on 25*(1), 61 – 73.

Mladenović, N. and P. Hansen (1997). Variable neighborhood search. *Computers & Operations Research 24*(11), 1097–1100.

Narula, S. and C. Ho (1980). Degree-constrained minimum spanning tree. *Computers & Operations Research 7*(4), 239–249.

Öncan, T. (2007, oct). Design of capacitated minimum spanning tree with uncertain cost and demand parameters. *Inf. Sci. 177*(20), 4354–4367.

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

Patterson, R., H. Pirkul, , and E. Rolland (1999). A memory adaptive reasoning technique for solving the capacitated minimum spanning tree problem. *Journal of Heuristics 5*(2), 159–180.

Prim, R. (1957). Shortest connection matrix network and some generalizations. *Bell system Tech. J 36*, 1389–1401.

Raidl, G. and C. Drexel (2000). A predecessor coding in an evolutionary algorithm for the capacitated minimum spanning tree problem. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pp. 309–316.

Rego, C. and F. Mathew (2011). A filter-and-fan algorithm for the capacitated minimum spanning tree problem. *Computers & Industrial Engineering 60*(2), 187–194.

Rego, C., F. Mathew, and F. Glover (2010). Ramp for the capacitated minimum spanning tree problem. *Annals of Operations Research 181*(1), 661–681.

Reimann, M. and M. Laumanns (2006). Savings based ant colony optimization for the capacitated minimum spanning tree problem. *Computers & Operations Research 33*(6), 1794–1822.

Sharaiha, Y., M. Gendreau, G. Laporte, and I. Osman (1997). A tabu search algorithm for the capacitated shortest spanning tree problem. *Networks 29*(3), 161–171.

Sharma, R. and M. El-Bardai (1970). Suboptimal communications network synthesis. In *Proceedings of the IEEE International Conference on Communications*, Volume 19, pp. 11–16.

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

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

Toth, P. and D. Vigo (1995). An exact algorithm for the capacitated shortest spanning arborescence. *Annals of Operations Research 61*, 121–141. 10.1007/BF02098285.

Uchoa, E., R. Fukasawa, J. Lysgaard, A. Pessoa, M. P. de Aragao, and D. Andrade (2008). Robust branch-cut-and-price algorithm for the capacitated minimum spanning tree problem over a large extended formulation. *Mathematical Programming 112*(2), 443–472.

Uchoa, E., T. A. M. Toffolo, M. C. de Souza, A. X. Martins, and R. Fukasawa (2012). Branch-and-cut and hybrid local search for the multi-level capacitated minimum spanning tree problem. *Networks 59*(1), 148–160.

Whitney, V. K. M. (1970). *A study of optimal file assignment and communication network configuration in remote-access computer message processing and communication systems.*

Zhang, N. (1993). Facet-defining inequalities for capacitated spanning trees. Master's thesis, Princeton University.

Zhang, R., S. Kabadi, and A. Punnen (2011). The minimum spanning tree problem with conflict constraints and its variations. *Discrete Optimization 8*(2), 191–205.

Zhou, G., Z. Cao, J. Cao, and Z. Meng (2007). A centralized network design problem with genetic algorithm approach. *Computational Intelligence and Security*, 123–132.

Notation and Abbreviations

Notation

b The maximum number of saturated s-trees in S $i \in I$.

c Cost matrix.

c_{ij} Distance between vertex i and vertex j

$c(T)$ Cost of T.

d Demand vector.

d_i Demand associated to terminal $i \in V^+$.

$d(S)$ Sum of the demands of the vertices in set $S \subseteq V^+$.

$d(T_i)$ Sum of the demands of the vertices in subtree T_i.

$dist(T_i, s\text{-}T_k)$ Minimum distance between subtree T_i and s-tree s-T_k.

n Number of components in the solution vector of an individual.

l_j Predecessor list for j.

p_e Number of elite individuals.

p_m Number of mutants.

s' Source vertex.

$s\text{-}T_k$ s-tree rooted at k.

u' Sink vertex.

v_0 Root vertex.

ρ_e Probability that an offspring inherits the vector component of its elite parent.

A Set of arcs.

$\overline{A}$ Set of arcs in auxiliary graph $\overline{N}(\bar{x}, \bar{y})$.

E Set of edges in graph G_U.

$ElitetoEx$ Number of elite solutions to exchange.

$G = (V, A)$ Directed network without loops.

$G_U = (V, E)$ Undirected graph for BRKGA.

$ItEX$ Number of iterations between interchange of elite elements of populations.

$ItStop$ Number of iterations without improvement in the stopping criterion.

L Number of populations.

$MaxQ$ Maximum Value of Capacity Violation for strategic oscillation.

$MT(S)$ Minimum number of s-trees required to accommodate the demand of the vertices contained in subset S.

$N1$ Local search neighborhood 1.

$\overline{N}(\bar{x}, \bar{y})$ Auxiliary graph used for inequalities separation procedures.

Q Capacity parameter.

"$R()$" Feasibility recovery phase for predecessor decoder.

S Subset of vertices.

$StopCriterion$ Stopping rule for BRKGA algorithm.

T Spanning tree subset of A.

T_i Subtree rooted at vertex i.

V Set of vertices.

$\overline{V}$ Set of vertices in auxiliary graph $\overline{N}(\bar{x}, \bar{y})$.

V_+ Set of terminals.

$V(T_i)$ Set of terminals connected by subtree T_i.

$\mathcal{X}$ Vector of n random keys.

Z Tree cost.

Abbreviations

ACO	Ant Colony Optimization
CMST	Capacitated Minimum Spanning Tree
CVRP	Capacitated Vehicle Routing Problem
ECC	Extended Capacity Cuts
ESO	Enhance Second Order algorithm
GSE	Generalized Subtour Elimination
CPLP	Capacitated Plant Location Problem
BRKGA	Biased Random Key Genetic Algorithm
HECC	Homogeneous Extended Capacity Cuts
LIFO	Last Input First Output
LB	Lower Bound
LP	Linear Programming
LIFO	Last Input First Output
MST	Minimum Spanning Tree
MIP	Mixed Integer Programming
NP-Hard	Non Polynomial Hard
Non-UD	Non Unitary Demands
GRASP	Greedy Randomized Adaptive Search Procedure
LS	Local Search
RKGA	Random Key Genetic Algorithm
SRHIF	Subroot Hop Indexed Formulation
TSP	Traveling Salesman Problem
UB	Upper Bound
UD	Unitary Demands
VLNS	Very Large Neighborhood Search heuristic
VNS	Variable Neighborhood Search
VRP	Vehicle Routing Problem

List of Figures

List of Tables

List of Algorithms