PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 11-09

Solving the Generalized Steiner Problem in Edge-survivable Networks

Pablo Sartor **Franco Robledo**

2011

Solving the generalized Steiner Problem in edge-survivable networks
Sartor, Pablo; Robledo, Franco
ISSN 0797-6410
Reporte Técnico RT 11-09
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, 2011

Solving the Generalized Steiner Problem in Edge-survivable Networks

Pablo Sartor and Franco Robledo

Instituto de Computación, Facultad de Ingeniería, Universidad de la República.
Julio Herrera y Reissig 565, Montevideo, Uruguay. CP 11.300.
psartor@um.edu.uy;frobledo@fing.edu.uy
http://www.fing.edu.uy

Abstract. The Generalized Steiner Problem with Edge-Connectivity constraints (GSP-EC) consists of computing the minimal cost subnetwork of a given feasible network where some pairs of nodes must satisfy edge-connectivity requirements. It can be applied in the design of communications networks where connection lines can fail and is known to be an NP-Complete problem. In this paper we introduce an algorithm based on GRASP (Greedy Randomized Adaptive Search Procedure), a combinatorial optimization metaheuristic that has proven to be very effective for such problems. Promising results are obtained when testing the algorithm over a set of heterogeneous network topologies and connectivity requirements; in all cases with known optimal cost, optimal or near-optimal solutions are found.

Keywords: Network Design; Edge-connectivity; Survivability; Steiner Problems; Metaheuristics; GRASP

1 Introduction

The design of communication networks often involves two antagonistic goals. On one hand the resulting design must have the lowest possible cost; on the other hand, certain survability requirements must be met, i.e. the network must be capable to resist failures of some of its components. One way to do it is by specifying a connectivity level (a positive integer) and constraining the design process to only consider topologies that have at least that amount of disjoint paths (either edge or node disjoint) between each pair of nodes. In the most general case, the connectivity level can be fixed independently for each pair of nodes (heterogeneous connectivity requirements), some of them having even no requirement at all. This problem is known as Generalized Steiner Problem (GSP) [10] and it is an NP-Complete problem [18]. Some references on the GSP and related problems are [1], [2], [3], [4], [6], [11], [12], [19], [20] most of them using polyhedral approaches and addressing particular cases (specific types of topology and or connectivity levels).

Topologies verifying edge-disjoint path connectivity constraints ensure that the network can survive failures in the connection lines; while node-disjoint path

constraints ensure that the network can survive failures both in switch sites as well as in connection lines. Finding a minimal cost subnetwork satisfying edge-connectivity requirements is modeled as a GSP edge-connected (GSP-EC) problem; whereas the corresponding problem involving node-connectivity requirements is known as the GSP-NC (node-connected) problem.

The remainder of this paper is organized as follows. Notation, auxiliary definitions and formal definition of the GSP-EC are introduced in Section II. The GRASP metaheuristic and the particular implementation that we propose for the GSP-EC are presented in Section III. Experimental results obtained when applying the algorithms on a test set of GSP-EC instances with up to one hundred nodes and four hundred edges are presented in Section IV. Finally conclusions are presented in Section V.

2 Problem Formalization and Definitions

We use the following notation to formalize the GSP-EC:

- $G = (V, E, C)$: Simple undirected graph with weighted edges;
- V : Nodes of G;
- E : Edges of G;
- $C : E \to \mathbb{R}^+$: Edge weights;
- $T \subseteq V$: Terminal nodes (the ones for which connectivity requirements exist);
- $R : R \in \mathbb{Z}^{|T|\times|T|}$: Symmetrical integer matrix of connectivity requirements; $r_{ij} = r_{ji} \geq 0, \forall i, j \in T; r_{ii} = 0, \forall i \in T$.

The set V models existing sites among which a certain set E of feasible links could be deployed, being the cost of including a certain link in the solution given by the matrix C. The set T models those sites for which at least one connectivity requirement involving other site has to be met; these requirements are specified using the matrix R. Nodes in the set $V \setminus T$ (known as "Steiner nodes") model sites that can potentially be used (because doing so reduces the total topology cost or because it is impossible to avoid using them when connecting a given pair of terminals) but for which no requirements exist.

Using this notation the GSP-EC can be defined as follows:

Definition 1. ***GSP-EC****. Given the graph G with edge weights C, the teminals set T and the connectivity requirements matrix R, the objective is to find a minimum cost subgraph $G_T = (V, E_T, C)$ of G where every pair of terminals i, j is connected by r_{ij} edge-disjoint paths.*

3 The GRASP Metaheuristic

GRASP (Greedy Randomized Adaptive Search Procedure) is a metaheuristic that proved to perform very well for a variety of combinatorial optimization problems. A GRASP is an iterative "multistart local optimization" procedure which performs two consecutive phases during each iteration:

- Construction Phase *(ConstPhase)*: it builds a feasible solution that chooses (following some randomized criteria) which elements to add from a list of candidates defined with some greedy approach;
- Local Search Phase *(LocalSearchPhase)*: it explores the neigborhood of the feasible solution delivered by the Construction Phase, moving consecutively to lower cost solutions until a local optimum is reached.

Figure 1 presents a generic GRASP pseudo-code. The procedure inputs include metaparameters $MetaParams$ which set the size of the list of candidates and other behaviour of the $ConstPhase$ procedure; the amount of iterations to run $MaxIter$; and a seed for random number generation. After having run $MaxIter$ iterations the procedure returns the best solution found. Details of this metaheuristic can be found in [21], [13]. In the next sections we introduce algorithms for implementing the Construction and Local Search Phases, as well as the main GSP algorithm invoking both phases to solve the GSP-EC.

Procedure GRASP$(MetaParams, MaxIter, RndSeed)$

```
1: bestSol ← NIL
2: for k = 1 to MaxIter do
3:    greedySol ← ConstPhase(MetaParams, RndSeed)
4:    localSearchSol ← LocalSearchPhase(greedySol)
5:    if cost(localSearchSol) < cost(bestSol) then
6:       bestSol ← localSearchSol
7:    end if
8: end for
9: return bestSol
```

Fig. 1. GRASP pseudo-code

3.1 Construction Phase Algorithm

The algorithm, shown in Figure 2, proceeds by building a graph which satisfies the requirements of the matrix R; it starts with an edgeless graph and in each iteration one new path is added to the solution under construction. It takes as inputs the graph G of feasible edges, the edge costs C, the set of terminal nodes T and the matrix of requirements R. In line 1 we initialize the solution graph under construction G_{sol} with the nodes of T and no edges; the matrix $M = (m_{ij})_{i,j \in T}$ which records the amount of connection requirements not yet satisfied in G_{sol} between the terminal nodes i and j; the sets P_{ij} that will be used to record the r_{ij} disjoint paths found for connecting the nodes i, j; and an auxiliary matrix $A = \{A_{ij}\}$ used to record how many times it was impossible to find one more path between two terminal nodes i, j whose requirements r_{ij} were not yet covered. In line 2 we alter the costs of the matrix C as we explain below

to satisfy a desirable property that previous algorithms did not satisfy.
Loop 3-15 is repeated until all terminal nodes have their connectivity requirements satisfied, or until for a certain pair of terminals i, j, the algorithm fails to find a path a certain number of times MAX_ATTEMPT.
Each iteration works the following way. Line 4 selects two terminal nodes i, j at random for which there are pending connectivity requirements. Line 5 computes the graph obtained by removing from G the edges of all paths already computed to connect i and j; thus, any path computed in G' will be edge-disjoint from the former $(i...j)$ paths in P_{ij}. In line 6, the edges already present in the solution under construction are given cost 0; by doing this, they will be taken as costless when considering the cost of any new path, enabling edge-reusing among different pairs of terminals. Line 7 computes the shortest path (regarding costs) connecting i and j, considering as feasible the edges from G' and with costs given by C'. In case this turns to be impossible, this is acknowledged in line 9 by incrementing the counter A_{ij} and resetting the path set P_{ij}, hoping that computing a different sucession of paths for i, j allow to satisfy the r_{ij} requirements. In case a path p was found, it becomes part of the solution under construction (lines 11-12), and the general-update-matrix procedure on line 13 updates the pending connection requirements of the matrix M, by applying the Ford-Fulkerson's algorithm with all capacities equal to 1, to detect if the adoption of the new path turned to satisfy other requirements besides the one for the pair i, j. Finally, the algorithm ends by returning the feasible solution G_{sol} together with the path set P which "certifies" that all requirements R were satisfied.

Procedure ConstPhase(G, C, T, R)

1: $G_{sol} \leftarrow (T, \emptyset); m_{ij} \leftarrow r_{ij} \forall i, j \in T; P_{ij} \leftarrow \emptyset \forall i, j \in T; A_{ij} \leftarrow 0 \forall i, j \in T$
2: $C \leftarrow$alter-costs(C)
3: **while** $\exists m_{ij} > 0 : A_{ij} <$MAX_ATTEMPT **do**
4: let i, j be any two terminals with $m_{ij} > 0$
5: $G' \leftarrow G \setminus P_{ij}$
6: let $C' = (c'_{uv}) : c'_{uv} \leftarrow [0$ if $(u, v) \in G_{sol}; c_{uv}$ otherwise $]$
7: $p \leftarrow$ shortest-path(G', C', i, j)
8: **if** $\nexists p$ **then**
9: $A_{ij} \leftarrow A_{ij} + 1; P_{ij} \leftarrow \emptyset; m_{ij} \leftarrow r_{ij}$
10: **else**
11: $G_{sol} \leftarrow G_{sol} \cup \{p\}$
12: $P_{ij} \leftarrow P_{ij} \cup \{p\}; m_{ij} \leftarrow m_{ij} - 1$
13: $[P, M] \leftarrow$ general-update-matrix(G_{sol}, P, M, p, i, j)
14: **end if**
15: **end while**
16: **return** G_{sol}, P

Fig. 2. ConstPhase pseudo-code

The algorithm here proposed satisfies the following property provided an appropriate function alter-costs is used in line 2.

Property 1.

$$\lim_{iterations \to \infty} probability(get\ an\ optimal\ solution) = 1$$

in other words, we can guarantee that any desired level of certainty of getting an optimal solution can be reached provided as many iterations as needed are run.

Proof. The complete proof can be found in Appendix A.

Similar construction phases previously proposed in [15], [14] do not satisfy this property, which can be shown even with trivial instances of the GSP as can be seen in Appendix A where we also show that the property is satisfied when the alter-costs function is such that all altered edge costs take values in $(0, +\infty)$ with any probability distribution that assigns non-zero probabilities to any open subinterval of $(0, +\infty)$. In our tests we used an exponential distribution with parameter $1/c$ being c the real cost of the edge. Moreover, by altering costs the proposed algorithm proceeds by just computing the shortest path in its main loop, instead of computing a set of "simultaneously disjoint shortest paths" and then randomly choosing one (as in previous algorithms), thus involving less computing.

3.2 Local Search Phase Algorithms

The local search phase starts with a feasible solution obtained from the construction phase and proceeds by consecutively moving to neighbour solutions which reduce the cost of the solution graph until it reaches a local optimum. Any local search algorithm needs a precise definition of the neighbourhood concept; we propose two different ones, which we chain inside our suggested LocalSearchPhase algorithm. They are defined in terms of a certain structural decomposition of graphs that we define below, together with some other auxiliary definitions.

Definition 2. key-node: *Given a GSP-EC instance and a feasible solution G_{sol}, we define a* ***key-node*** *as a non-terminal node with degree at least three in G_{sol}.*

Definition 3. key-path: *Given a GSP-EC instance and a feasible solution G_{sol}, we define a* ***key-path*** *as a path in G_{sol} such that all intermediate nodes are non-terminal with degree two in G_{sol} and whose endpoints are either terminal nodes or key-nodes.*

Definition 4. key-star: *Given a GSP-EC instance, a feasible solution G_{sol} and any node v of G_{sol}, we define as the* ***key-star*** *associated to v the subgraph of G_{sol} obtained through the union of all key-paths with v as an endpoint.*

Path-Based Local Search Neighbourhood Our first neighbourhood is based on the replacement of any key-path k by another key-path with the same endpoints, built with any edge from the feasible connections graph G (even some of G_{sol}), provided no connectivity levels are lost when reusing edges. Let k be a key-path of a certain solution G_{sol} and P a set of paths which "certificates" its feasibility (as the one returned by ConstPhase). We will denote by $J_k(G_{sol})$ the set of paths $\{p \in P : k \subseteq p\}$. These are the paths which contain the key-path k. We will also denote by $\chi_k(G_{sol})$ the edge set

$$\chi_k(G_{sol}) = \bigcup_{q=i...j \in J_k(G_{sol})} E(P_{ij} \setminus q)$$

where $i...j$ stands for a path with extremes i and j. These are the edges that, if used to replace the key-path k in P (obtaining a path set P'), would turn to be shared by some paths from G_{sol} with the same endpoints, thus invalidating the resulting set P' as a feasibility certificate. We can now define our first neighbourhood.

Definition 5. Neighbourhood1*: Given a GSP-EC instance and a feasible solution G_{sol}, it is the set of all graphs obtained by replacing any key-path k of G_{sol} by another path p such that $cost(p) < cost(k)$ and the edges of p are chosen from the set $E \setminus \chi_k(G_{sol})$ and/or k. (Recall that E represents the feasible edges between nodes).*

Based on these definitions we built the path-based local search algorithm LocalSearchPhase1 shown in Figure 3. The algorithm receives as inputs the graph G of feasible connections, the edge cost matrix C, the terminals set T and a path set S which build up a feasible solution. Line 1 initializes the flag *improve* which indicates wheter an improved solution has been found or not. Line 2 computes the decomposition in key-nodes and key-paths of the set S. Loop 3-20 looks for succesive cost improvements until no more can be done. Each iteration proceeds as follows. The loop 5-19 analyzes each key-path k trying to find a suitable replacement with lower cost. Line 6 computes the edge set $E(k) \cup (E \setminus \chi_k(S))$, where edges are to be chosen from to build the replacing key-path. This set is such that, as seen above, ensures no loss of connectivity levels in the new solution obtained, while allowing the reuse of edges already present in the current solution S. Line 7 computes a new cost matrix C', zeroing the cost of all edges of S that are not included in the key-path k, to reflect the fact that using any of those edges to build the replacing key-path adds no extra cost to the modified solution. Line 8 computes the path with lower cost according to the matrix C' over the subgraph computed in line 6. Line 9 verifies if the adoption of the new key-path implies a cost reduction. If so, it is acknowledged by the flag *improve* and k is replaced in all paths of S which included k. Care is taken to remove cycles and recompute the k-decomposition if a certain node happens to have a degree greater than two after the replacement (lines 12-14); if the latter does not happen, line 16 simply updates the k-decomposition by replacing the key-path (and a full new k-decomposition is avoided). After exiting the main loop, line

21 returns a feasible solution whose cost can no more be reduced by moving to neighbour solutions.

Procedure LocalSearchPhase1(G, C, T, S)

1: $improve \leftarrow$ TRUE
2: $\kappa \leftarrow$ k-decompose(S)
3: **while** *improve* **do**
4: $improve \leftarrow$ FALSE
5: **for all** kpath $k \in \kappa$ with endpoints u, v **do**
6: $G' \leftarrow$ the subgraph induced from G by $E(k) \cup (E \setminus \chi_k(S))$
7: $C' \leftarrow (c'_{ij})/c'_{ij} = 0$ if $(i,j) \in S \setminus k; c'_{ij} = c_{ij}$ otherwise
8: $k' \leftarrow$ shortest-path(G', C', u, v)
9: **if** cost$(k', C') <$ cost(k, C') **then**
10: $improve \leftarrow$ TRUE
11: update $S : \forall p \in J_k(S)(p \leftarrow (p \setminus k) \cup k')$
12: **if** $\exists z \in V(k'), z \notin \{u, v\}$, degree$(z) \geq 3$ in S **then**
13: remove-cycles$(J_k(S))$
14: $\kappa \leftarrow$ k-decompose(S)
15: **else**
16: $\kappa \leftarrow \kappa \setminus \{k\} \cup \{k'\}$
17: **end if**
18: **end if**
19: **end for**
20: **end while**
21: **return** S

Fig. 3. LocalSearchPhase1 pseudo-code

Key-Star-Based Local Search Neighbourhood Our second neighbourhood is based on the replacement of key-stars, which frequently allow to improve feasible solutions that are locally optimal when only considering Neighbourhood1. In the case of the GSP-NC, as no node sharing is allowed among disjoint paths, all key-stars are trees (named *key-trees*); a key-tree replacement neighbourhood for the GSP-NC can be found in [15], [14]. Due to the possibilty of sharing nodes among edge-disjoint paths, when working with GSP-EC problems, we have to work with key-stars, and unlike [15] we will allow the root node to be a terminal node in order to get a broader neighbourhood. In the GSP-NC any key-tree can be replaced by any tree with the same leaves with no loss of connectivity levels. In the GSP-EC, if the replacing structure is also a key-star the same holds true; but it does not for other general structures (non-star trees included). We propose an algorithm that given a key-star k, deterministically seeks for the lowest cost replacing key-star k' able to "repair" the paths from P broken when removing the edges of k.

Let k be a key-star in a certain feasible solution G_{sol} and P a set of paths which "certificates" its feasibility (as the one returned by ConstPhase). For allowing as much reusing of edges as possible, we can extend our previous definition of $J_k(G_{sol})$ and $\chi_k(G_{sol})$ to consider key-stars k instead of key-paths; and thus we can define the key-star based neighbourhood as follows.

Definition 6. Neighbourhood2*: Given a GSP-EC instance and a feasible solution G_{sol}, it is the set of all graphs obtained by replacing any key-star k of G_{sol} by the lowest possible cost key-star k' such that k' preserves the same connectivity among the leaves of k and its terminal nodes, and the edges of k' are chosen from the set $E \setminus \chi_k(G_{sol})$ and/or k.*

We present the star-based local search algorithm LocalSearchPhase2 in Figure 4.

Procedure LocalSearchPhase2(G, C, T, S)

```
 1: improve ← TRUE
 2: κ ← k-decompose(S)
 3: while improve do
 4:    improve ← FALSE
 5:    for all kstar k ∈ κ do
 6:       [k', newCost] ← BestKeyStar(G, C, T, S, k)
 7:       if newCost < cost(k, C) then
 8:          improve ← TRUE
 9:          replace k by k' in all paths from S
10:          κ ← k-decompose(S)
11:          abort for all
12:       end if
13:    end for
14: end while
15: return S
```

Fig. 4. LocalSearchPhase2 pseudo-code

Line 1 initializes the flag *improve* which indicates wheter an improved solution has been found or not. Line 2 computes the decomposition in key-nodes and key-paths of the set S. Loop 3-14 looks for succesive cost improvements until no more can be done. Each iteration proceeds as follows. The loop 5-13 analyzes each key-star k trying to find a suitable replacement with lower cost. Line 6 determines the lowest cost key-star k' that could replace k and its cost (computed assuming that edges from the current solution not in k have no cost to promote edge reusing). To do so it uses the procedure $BestKeyStar$ described later. Line 7 verifies if the replacing key-star has lower cost than k; if it does, lines 8-11 acknowledge the fact, the replacement is done over the set S, the k-decomposition is recomputed and the "for all" loop is aborted to restart looking for improvements.

Figure 5 presents the algorithm BestKeyStar. Given a key-star k we denote by θ_k its root node; by ψ_k the set of its leaf nodes; and by $\hat{\delta}_{k,m}$ (being m the root node of k or one of its leaves) the highest amount of key-paths that join m in k with any other node that is root or leaf in k.

Procedure BestKeyStar(G, C, T, S, k)

1: $G' \leftarrow$ the subgraph induced from G by $E(k) \cup (E \setminus \chi_k(S))$
2: $C' \leftarrow (c'_{ij})/c'_{ij} = 0$ if $(i,j) \in S \setminus k; c'_{ij} = c_{ij}$ otherwise
3: add a "virtual node" w to G'
4: $\Omega \leftarrow \psi_k$
5: **if** $\theta_k \in T$ **then**
6: $\quad \Omega \leftarrow \Omega \cup \{\theta_k\}$
7: **end if**
8: **for all** $m \in \Omega$ **do**
9: $\quad$ add $\hat{\delta}_{k,m}$ parallel edges (w, m) to G' with cost 0
10: **end for**
11: $c_{min} \leftarrow 0$; $k_{min} \leftarrow k$
12: **for all** $z \in V(G)$ **do**
13: $\quad k' \leftarrow$ simult-shortest-paths$(G', \delta_{G',w}, z, w)$
14: $\quad$ **if** k' has $\delta_{G,w}$ paths $\wedge$ $cost(k', C') < c_{min}$ **then**
15: $\quad\quad c_{min} \leftarrow cost(k', C')$; $k_{min} \leftarrow k'$
16: $\quad$ **end if**
17: **end for**
18: **return** $[k_{min}, c_{min}]$

Fig. 5. BestKeyStar pseudo-code

The algorithm is based on the idea of building key-stars by employing the simult-shortest-paths algorithm (in our tests we used the one introduced in [5]). Lines 1-2 compute the subgraph of G obtained by removing the edges that could cause loss of connectivity level if reused; the altered cost matrix C' with cost zero for reused edges; and adds a virtual node w whose purpose is explained below. Lines 4-7 determine the set of leaf nodes that the key-star to build must have. Lines 8-10 connect each of the latter to w with an appropriate number of parallel zero-cost edges totalling $\delta_{G',w}$ (degree of w in G') edges. The loop 12-17 considers nodes of G that could be potencial roots z of the key-star to be found, and then builds the lowest-cost one with root node z through the application of the already mentioned simult-shortest-path algorithm on the graph G' in line 13; $\delta_{G',w}$ edge-disjoint paths connecting z and w are requested. If found (lines 14-16) and with lower cost than k then the new key-star and its associated cost are recorded as the best ones so far found. After having considered all possible root nodes, line 18 returns both the best key-star and its cost according to C'.

Figure 6 depicts the process of determining which the best key-star to replace a given one is. It illustrates (a) the feasible graph G with a key-star that keeps

connected the leaf nodes t, u, v; (b) the graph G' obtained after adding the virtual nodes w linked with cost zero to each of t, u, v by the appropriate amount of edges (as many as the degree of each in the key-star) and having chosen a "candidate" root node z; (c) the shortest paths found to connect z and w (the sum of degrees of t, u, v are requested); and (d) the new key-star obtained after removing the virtual node w. For example, the node v has degree three in the key-star (a); so it is connected by three parallel zero-cost edges to the virtual node w (b); then, when requesting the simultaneously disjoint paths to link z and w in G', three disjoint paths will join z and v (if possible) (c); finally, those three paths will be part of the new key-star with new root z (d).

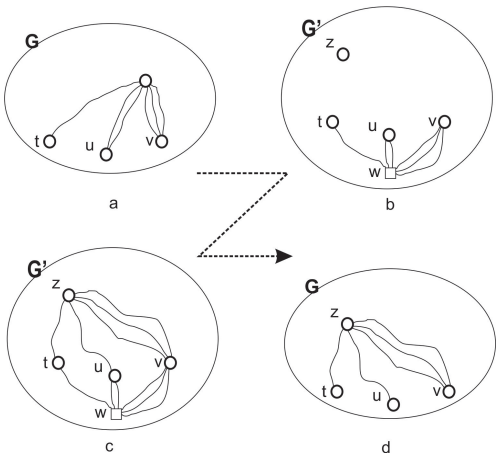

Fig. 6. Computing the best key-star

3.3 GRASP algorithm description

Now we are able to put the pieces together and build a GRASP algorithm for solving the GSP-EC. Figure 7 shows the resulting pseudo-code.

Basically the local search phase of this algorithm applies key-path replacement based movements until no further improvements are possible; then it tries to apply the best key-star replacement movement (once); if the latter is done with a cost reduction, then key-path replacements are tried again, and so on, until no further improvements are possible for both kinds of movements.
In line 1 the minimum cost found c_{min} is initialized to ∞ and an empty path set S_{opt} is initialized. The main loop (2-18) is executed $iters$ times and then the best solution found is returned. Line 3 builds a feasible solution employing our ConstPhase greedy randomized adaptive algorithm; being S the path set that certifies feasibility. If the set S has less paths than the so far found best solution

Procedure GRASP_GSP$(G, C, T, R, iters)$

```
1: c_min ← ∞; S_opt ← ∅
2: for i = 1 to iters do
3:    [G_sol, S] ←ConstPhase(G, C, T, R)
4:    if |S| ≥ |S_opt| then
5:       flag ←TRUE
6:       OptLoop: [G_sol, S'] ←LocalSearchPhase1(G, C, S)
7:       if flag ∨ cost(S') <cost(S) then
8:          flag ←FALSE
9:          [G_sol, S''] ←LocalSearchPhase2(G, C, S')
10:         if cost(S'', C) <cost(S', C) then
11:            S ← S''; go to OptLoop
12:         end if
13:      end if
14:      if cost(S', C') < c_min then
15:         c_min ←cost(S'', C); S_opt ← S
16:      end if
17:   end if
18: end for
19: return S_opt
```

Fig. 7. GRASP_GSP pseudo-code

S_{opt} (line 4) this iteration is discarded. This could happen if the last call to ConstPhase was not able to satisfy all requirements of R and a previous call was able to do it (or at least to satisfy a greater number); our first objective is to satisfy as much requirements of R as possible. Line 6 applies the key-path based movements by calling LocalSearchPhase1. If this was the first local search or if a cost reduction was achieved then a best key-star movement is tried in line 9. In case the latter succeeds in reducing the cost (verified in line 10), the execution flow resumes at line 6, for trying a new cycle of chained improvements. When no further local improvements are possible, lines 14-16 update the best known solution in case an improvement was achieved.

4 Performance tests

This section presents the results obtained after testing our algorithms with twenty-one test cases. The algorithms were implemented in C/C++ and tested on a 2 GB RAM, Intel Core 2 Duo, 2.0 GHz machine running Microsoft Windows Vista. All instances were run with the parameter $iters$ set to 100.

4.1 Test set description

To our best knowledge, no library containing benchmark instances related to the GSP-NC nor GSP-EC exists; we have built a set of twenty-one test cases that are based in cases found in the following public libraries:

– Steinlib [7]: instances of the Steiner problem; in many cases the optimal solution is known, in others the best solution known is available;
– Tsplib [8]: instances of diverse graph theory related problems, including a "Traveling Salesman Problem" section.

Case	V	E	T	St	Redund.	Opt
b01-r1	50	63	9	41	1-EC	82
b01-r2	50	63	9	41	2-EC	NA
b03-r1	50	63	25	25	1-EC	138
b03-r2	50	63	25	25	2-EC	NA
b05-r1	50	100	13	37	1-EC	61
b05-r2	50	100	13	37	2-EC	NA
b11-r1	75	150	19	56	1-EC	88
b11-r2	75	150	19	56	2-EC	NA
b17-r1	100	200	25	75	1-EC	131
b17-r2	100	200	25	75	2-EC	NA
cc3-4p-r1	64	288	8	56	1-EC	2338
cc3-4p-r3	64	288	8	56	3-EC	NA
cc6-2p-r1	64	192	12	52	1-EC	3271
cc6-2p-r2	64	192	12	52	2-EC	NA
cc6-2p-r123	64	192	12	52	1-EC(11),2-EC(36),3-EC(19)	NA
hc-6p-r1	64	192	32	32	1-EC	4003
hc-6p-r2	64	192	32	32	2-EC	NA
hc-6p-r123	64	192	32	32	1-EC(171),2-EC(189),3-EC(136)	NA
bayg29-r2	29	406	11	18	2-EC	NA
bayg29-r3	29	406	11	18	3-EC	NA
att48-r2	48	300	10	38	2-EC	NA

Table 1. Characteristics of the Test Cases

The main characteristics of the twenty-one test cases are shown in Table 1. For each case we show the amount of nodes (V), feasible edges (E), terminal nodes (T) and Steiner (non terminal) nodes (St). We also show the level of edge-connectivity requirements "Redund." (one, two, three or mixed) and the optimal costs when available. GSP poblems solved with connectivity level one are Steiner problems and in those cases we got the optimal solution cost from Steinlib. Problems b01, b03, b05, b11 and b17 were taken from Steinlib's problem instances set "B" and are cases randomically generated with integer uniform costs ranging from 1 to 10. The case cc3-4p belongs to Steinlib's instance set "PUC"; eigth terminal nodes are terminal and we solved two instances with uniform connectivity requirements one and three. The cases cc6-2p and hc-6p belong also to Steinlib's instance set "PUC"; twelve and thirty-two terminal nodes are terminal and we solved three instances for each one with connectivity requirements one, two, and a mix of one to three (in the latter case, the table

shows the amount of terminal pairs with connectivity level required one, two and three). Finally the cases bayg29 and att48 were taken from the library Tsplib; both correspond to real cases (twenty-nine cities from Bavaria, Germany; and 48 cities from USA). Source data of the twenty-one instances as well as the best solutions found are available in [17].

Case	Reqs.	t(ms)	Cost	%LSI
b01-r1	36	77	82	3.0
b01-r2	42	80	98	3.4
b03-r1	300	2611	138	10.6
b03-r2	378	3108	188	4.1
b05-r1	78	298	61	9.2
b05-r2	144	1389	120	5.2
b11-r1	171	1477	88	13.8
b11-r2	324	4901	180	3.4
b17-r1	300	6214	131	10.2
b17-r2	531	15143	244	3.0
cc3-4p-r1	28	388	2338	10.0
cc3-4p-r3	84	2221	5991	4.6
cc6-2p-r1	66	2971	3271	2.4
cc6-2p-r2	132	4801	5962	10.2
cc6-2p-r123	140	6317	8422	9.8
hc-6p-r1	496	25314	4033	6.8
hc-6p-r2	992	28442	6652	3.5
hc-6p-r123	957	26551	7930	5.2
bayg29-r2	110	975	6856.88	4.6
bayg29-r3	165	2413	11722	4.2
att48-r2	90	1313	23214	13.0
Averages	265	6524	-	6.7

Table 2. Numerical Test Results

4.2 Numerical Results

Computational results of the tests can be found in Table 2. Here follows the meaning of each column:

- Reqs.: total amount of terminal-to-terminal disjoint paths found in the best solution (which, if the feasible connections network is compatible with all the requirements given by R, should amount $\sum_{i,j} r_{ij}/2$.;
- t(ms): the average running time (in ms) per iteration;
- Cost: the cost of the best solution found;
- LSI: "local search improvement": the percentage of cost improvement achieved by the local search phase when compared to the cost of the solution delivered by the construction phase, for the best solution found.

In all cases with connectivity requirements equal to one (1-EC) for all pairs of terminals (for which the optimal costs are known) every best solution found is optimal, with the exception of the case hc-6p-r1 (found cost 4033 being the optimal cost 4003). Note also that the average cost improvement over the solution delivered by ConstPhase (LSI) amounts to 6.7% (when computed only for the best solutions found). All solutions found are edge-minimal regarding feasibility (no edge can be supressed without losing required connectivity levels); and in all cases the maximum possible number of requirements are satisfied (i.e. for all pairs of terminals i, j, whether their requirement r_{ij} was satisfied or f_{ij} disjoint paths were found being f_{ij} the maximum achievable amount of disjoint paths joining i and j given by the topology of the feasible connections graph G).

5 Conclusion

The algorithm GRASP_GSP was shown to find good quality solutions to the GSP-EC when applied to a series of heterogeneous test cases with up to 100 nodes and up to 406 edges. It was also shown to guarantee for all instances a non-zero probability of finding an optimal solution avoiding local optima traps. For all cases with known optimal cost the algorithm was able to find solutions with costs no more than 0,74% higher than the optimal cost. Significant cost reductions are achieved after applying the local search phase over the greedy solutions built by the construction phase. Execution times were comparable to the ones of previous similar works like [15] for the node-connected version of the GSP.

A Appendix: Proof of Property 3.1

A.1 Local Optima Traps in Previous Algorithms

Previous algorithms introduced in [15], [14] do not satisfy the Property 3.1 as evidenced by the example shown in Figure 8. In (a) an instance of the GSP-EC is shown with three terminal nodes (black), one Steiner node (white) and nine edges labelled with costs. Let us assume that the matrix R requires one path to connect every pair of terminals. If the algorithm is run with the GRASP parameter "candidate list size" (used in the mentioned previous works) set to two, the 2-shortest-paths will always be paths with length one given by the edges with costs 10 and 11. Some possible outcomes are shown in (b). The algorithm will never find the optimal solution (with cost 18) built with the three edges that have cost 6. Furthermore, none of the solutions buildable by the Construction Phase can be transformed in the optimal solution through the movements introduced in previous nor in this work (replacement of key-paths, key-trees and key-stars). Therefore, there exist instances (even trivial ones) of the GSP and parameterizations of previously introduced GRASP algorithms for which an optimal solution will never be attained.

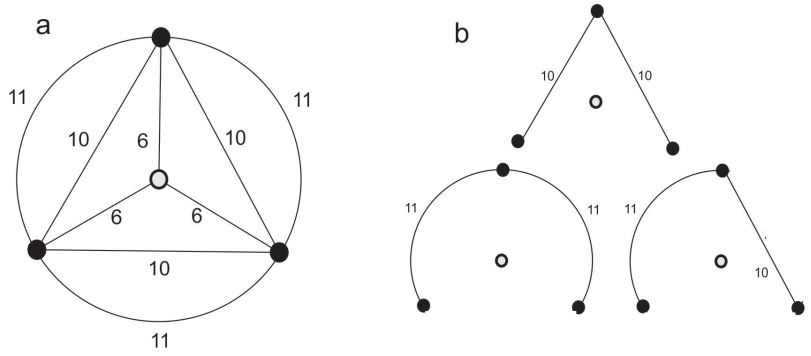

Fig. 8. Counterexample for Property 3.1

A.2 Altering Costs

In order to introduce randomness in the Construction Phase as well as avoid the local optima trap our algorithm begins by altering the edge costs by means of a function that we called alter-costs. The idea is to leave with cost zero the edges already having cost zero and at the same time altering the other costs in a way where generally (but not always) edges having lower costs end up with lower altered costs. Next we formalize these ideas and give a possible definition for this function (the one we applied in our tests).We will seek that the following properties be satisfied by the alter-costs function:

- If the edge cost is zero, the altered cost is also zero.
- Otherwise:
  - Each edge cost will be altered independently from the others;
  - The altered cost will be any value in $(0, +\infty)$ following a certain probability distribution that assigns non-zero probabilities to any open subinterval of $(0, +\infty)$. We will denote as $\hat{C}_{ij}$ the random variable that follows this distribution for a certain edge (i, j);
  - The expected value of the cost will be the original edge cost;
  - $(c_{ij} < c_{hk}) \rightarrow (p(\hat{C}_{ij} < \hat{C}_{hk}) > \frac{1}{2})$ for any pair of edges $(i, j), (h, k)$.

The last condition is an heuristic one though not necessary for the proof given below.

As a possible alternative we suggest employing the "exponential distributionl", defined as follows (with a parameter λ):

$$f_\lambda(x) = \lambda e^{-\lambda x}$$

As parameter we will employ the reciprocal of the original edge costs; for each edge (i, j) with original cost c_{ij} the suggested alter-costs function will return:

- 0 if $c_{ij} = 0$
- a value given by a random variable with distribution $f_{1/c_{ij}}$ if $c_{ij} > 0$

The suggested distribution assigns values in $(0, +\infty)$ to each edge with non-zero cost, satisfying all the above properties; its expected value equals the original cost (which, for the exponential distribution, is the reciprocal of its parameter i.e. c_{ij}). It also satisfies the last condition as can be seen analyzing the joint distribution:

$$p(\hat{C}_{ij} < \hat{C}_{kh}) = \iint_{x<y} (1/c_{ij}) e^{-x/c_{ij}} (1/c_{kh}) e^{-y/c_{kh}} \, dx \, dy = \frac{c_{ij}^{-1}}{c_{ij}^{-1} + c_{hk}^{-1}}$$

where we see that the probability of (i, j) getting an altered cost lower than the one of (h, k) grows as c_{ij} diminishes with respect to c_{hk}. Finally, the exponential distribution assigns non-zero probabilities to any open subinterval of $(0, +\infty)$ which is vital for the proof of the Property 3.1 given below. Next we show that altering costs with such distributions make our algorithms satisfy the Property 3.1.

A.3 Buildable Solutions Space and Global Optima

Theorem 1. *For all instances of the GSP, the algorithm ConstPhase has non-zero probability of reaching an optimal solution; in other words it satisfies the Property 3.1.*

Proof. Let S be any example solution with optimal cost for the problem. Let us suppose that the probability p of building exactly this solution when running the algorithm is strictly positive i.e. $p \in (0, 1]$. Then, the probability of the algorithm not finding that solution even once after running k times would be $(1-p)^k$, and the probability of building that solution at least once after k executions would be:

$$p_k(S) = 1 - (1-p)^k > 0$$

Then,

$$\lim_{k \to \infty} p_k(S) = 1$$

what would complete the proof.

Now we will see that given any instance of the GSP and an optimal solution S, the probability p of building S is different than zero. Let $(S)_i = s_1, s_2, ..., s_r$ with $r = \sum_{i,j \in T} r_{ij}$ be an ordering of the paths in S obtained the following way:

- s_1 is (any of) the path(s) in S having less non-zero cost edges;
- s_k is (any of) the path(s) where the amount of non-zero cost edges not belonging to any of the previous paths $s_1 ... s_{k-1}$ is minimal.

Denoting as $l^*(s)$ the amount of non-zero cost edges in a set s, and $l_1^*, ..., l_r^*$ the results from applying l^* on a certain ordering of paths, formally $(S)_i$ is any ordering such that

$$l^*\left(s_k \setminus \bigcup_{i=1...k-1} s_i\right) = \min_{s \in S \setminus \{s_1,...,s_{k-1}\}} l^*\left(s \setminus \bigcup_{i=1...k-1} s_i\right)$$

The amount of possible orderings (arrangements with repetitions) of the terminals pairs sequence (i, j) that the algorithm can follow (at random) to generate the paths is given by

$$\frac{\left(\sum_{i<j} r_{ij}\right)!}{\prod_{i<j}(r_{ij}!)}$$

and thus the probability that the algorithm generate the example paths following a source-destination terminals order coinciding with the one of $(S)_i$ is

$$p_{order}(S) = \frac{\prod_{i<j}(r_{ij}!)}{\left(\sum_{i<j} r_{ij}\right)!} > 0$$

Let us assume now that the algorithm will generate the paths in the same terminals order given by $(S)_i$. We will see now that the probability of generating the path s_i (or other with the same cost) in place i is non-zero; which would allow to complete our proof finding a lower bound for p. To do so we build a family of functions $\hat{C}$ for the altered costs that will make the algorithm choose exactly the intended paths; and we will show that the probability of the original alter-costs function resulting in one belonging to the family is non-zero.

Given a certain ϵ such that $0 < \epsilon < \frac{1}{2|E|}$, and a certain $k \in \mathbb{N}$, let $\hat{C}_k = (\hat{c}_{k,ij})$ be any cost assignment for the feasible edges satisfying:

- $\hat{c}_{k,ij} = 0$ if $c_{ij} = 0$;
- $\hat{c}_{k,ij} \in (k - \epsilon, k + \epsilon)$ if $c_{ij} > 0$.

In what follows we will work with costs given by $\hat{C}_k$ and denote $\hat{c}_k(q)$ the cost of a path q according to $\hat{C}_k$. In general, a path s with $l = l^*(s)$ non-zero cost edges will have a cost $\hat{c}_k(s) \in (kl - l\epsilon, kl + l\epsilon)$ and thus $\hat{c}_k(s) \in (kl - 1/2, kl + 1/2)$. Therefore, given two paths q and q' with $l^*(q) < l^*(q')$, it will always be true that $\hat{c}_k(q) < \hat{c}_k(q')$.

Let us build $\hat{C}$ the following way. We will assign a cost according to $\hat{C}_1$ to every edge in s_1. We will assign a cost according to $\hat{C}_2$ to every edge in $s_2 \setminus s_1$. Similarly we will keep assigning costs according to $\hat{C}_k$ to every edge in $s_k \setminus (s_1 \cup s_2 \cup ... \cup s_{k-1})$. Finally we assign any cost higher than any already assigned to the edges in $G \setminus S$. With this altered costs $\hat{C}$, the first path to be built will necessarily be s_1; because any different path will have at least the same amount of non-zero cost edges, and for every edge "interchanged" among it and s_1 the cost would be kept or increased. Similarly, the second path built can not be other than s_2, because the only way to minimize the cost is by adding its edges. Extending this reasoning we see that the algorithm will generate the exact path sequence $(S)_i$.
Then we have:

- A non-zero probability $p_{order}(S)$ of randomically shuffling a terminals pairs ordering equal to the one of $(S)_i$;
- A non-zero probability $p_{alter}(S)$ that the altering costs function assign costs to the feasible edges of G according to our definition of $\hat{C}$ (because, for any edge with non-zero original cost, there is a non-zero probability that our altering costs function assign some cost in $(k - \epsilon, k + \epsilon)$).

The probability of getting both facts at the same time is $p_{order}(S)p_{alter}(S) > 0$; and being these cases a subset of the sampling space of all "shuffle and alter" scenarios that lead to building S, we conclude that we have found a non-zero lower bound for the probability p, thus completing the proof.

References

1. Ajit Agrawal, Philip Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized steiner problem on networks. *SIAM Journal on Computing*, 24(3):440–456, 1995.
2. M. Baïou and A.R. Mahjoub. Steiner 2-edge connected subgraph polytope on series-parallel graphs. *SIAM Journal on Discrete Mathematics*, 10(1):505 – 514, 1997.
3. Mourad Baïou. *Le problème du sous-graphe Steiner 2-arête connexe : Approche polyédrale*. PhD thesis, Université de Rennes I, Rennes, France, 1996.
4. Mourad Baïou. On the dominant of the Steiner 2-edge connected subgraph polytope. *Discrete Applied Mathematics*, 112(1-3):3 – 10, 2001.
5. R. Bhandari. Optimal physical diversity algorithms and survivable networks. In *Computers and Communications, 1997. Proceedings., Second IEEE Symposium on*, pages 433 –441, July 1997.
6. C.R. Coullard, A. Rais, D.K. Wagner, and R.L. Rardin. Linear-time algorithms for the 2-Connected Steiner Subgraph Problem on Special Classes of Graphs. *Networks*, 23(1):195 – 206, 1993.
7. Thorsten Koch. Konrad-Zuse-Zentrum fr Informationstechnik Berlin. Steinlib test data library. http://steinlib.zib.de/steinlib.php.
8. Ruprecht-Karls-Universitat Heidelberg. Tsplib network optimization problems library. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95.

9. H. Kerivin and A. Mahjoub. Design of Survivable Networks: A survey. *Networks*, 46(1):1 – 21, 2005.
10. J. Krarup. The generalized steiner problem. Technical report, DIKU, University of Copenhagen, 1979.
11. A.R. Mahjoub and P. Pesneau. On the Steiner 2-edge connected subgraph polytope. *RAIRO Operations Research*, 42(1):259–283, 2008.
12. C.L. Monma, B.S. Munson, and W.R. Pulleyblank. Minimum-weight two connected spanning networks. *Mathematical Programming*, 46(1):153 – 171, 1990.
13. M.G.C. Resende and C.C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Kluwer Academic Publishers, 2003.
14. F. Robledo and E. Canale. Designing backbone networks using the generalized steiner problem. In *Design of Reliable Communication Networks, 2009. DRCN 2009. 7th International Workshop on*, volume 1, pages 327 – 334, October 2009.
15. Franco Robledo. *GRASP heuristics for Wide Area Network design.* PhD thesis, IRISA, Université de Rennes I, Rennes, France, february 2005.
16. Pablo Sartor. *Problema General de Steiner en Grafos: Resultados y Algoritmos GRASP para la versin Arista-Disjunta.* M.Sc. thesis, Universidad de la República, Montevideo, Uruguay, 2011.
17. Universidad de Montevideo. GSP-EC test set with best results found. http://www2.um.edu.uy/psartor/grasp-gsp-ec.zip.
18. Pawel Winter. Steiner problem in networks: a survey. *Networks.*, 17(2):129–167, 1987.
19. M. Stoer. Design of survivable networks. In *Lecture Notes in Mathematics. Springer Verlag, 1992.*, volume 1531.
20. H. Kerivin and R. Mahjoub. Design of survivable networks: A survey. *Networks*, 46(1):1 – 21, 2005.
21. T.A. Feo and M.G.C. Resende Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6:109 – 133, 1995.