GRASP Algorithms for the Edge-survivable Generalized Steiner Problem

Pablo Sartor and Franco Robledo

Instituto de Computación, Facultad de Ingeniería, Universidad de la República
Julio Herrera y Reissig 565, Montevideo, Uruguay, CP 11300
Tel. (+598)27114244, Fax (+598)27110469
{psartor, frobledo}@fing.edu.uy

Abstract

The design of a communication network where the connectivity between nodes must be kept even after failure of some links can be modeled as a Generalized Steiner Problem. It consists of computing the minimal cost subnetwork of a given feasible network where certain pairs of nodes must satisfy a number of disjoint connectivity requirements and is known to be an NP-Complete problem. This paper introduces an heuristic based on the combinatorial optimization metaheuristic "Greedy Randomized Adaptive Search Procedure" (GRASP) to solve the edge-survivable version of the problem, where nodes are perfect but links can fail. The algorithm is tested with a set of heterogeneous network topologies and connectivity requirements obtaining promising results; in all cases with known optimal cost, optimal or near-optimal solutions are found.

Keywords: *Edge-Connectivity, Survivability, Steiner Problem, Metaheuristics, GRASP*

1. Introduction

The design of communication networks involves two antagonistic goals. On one hand the resulting design must have the lowest possible cost; on the other hand, certain survivability requirements must be met i.e. the network must be capable to resist failures in some of its components. One way to achieve this is by specifying a connectivity level (a positive integer) and constraining the design process to only consider topologies that have (at least) such 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) [1] and is an NP-Complete problem [2]. Most references on the GSP and related problems [3-9] apply polyhedral approaches and address particular cases (specific types of topology and 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. Due to the intrinsic complexity of the problem, heuristic approaches have to be used to cope with general real-sized instances. This paper presents one such heuristic inspired in the ideas of recent research [10-12]. The remainder of this paper is organized as follows. Notation, auxiliary definitions and formal definition of the GSP-EC are introduced in Section 2. The GRASP (Greedy Randomized Adaptive Search Procedure) metaheuristic and the particular implementation that we suggest for the GSP-EC are presented in Section 3. 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 4. Finally conclusions are presented in Section 5.

2. Definitions and Formalization

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;
- $C: E \to \Re^+$: edge weights;
- $T \subseteq V$: Terminal nodes (the ones for which connectivity requirements exist);
- $R: R \in Z^{|T| \times |T|}$: Symmetrical integer matrix of connectivity requirements, $r_{ii} = 0\ \forall i \in T$.

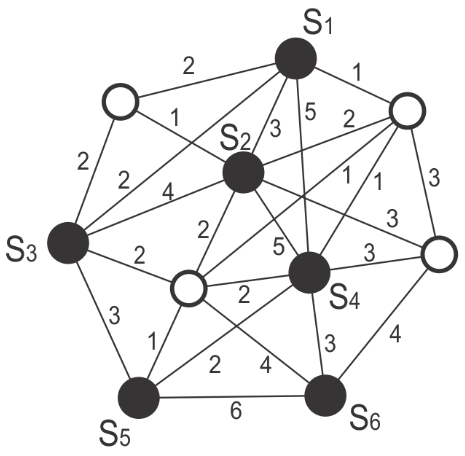
V models existing sites among which a certain set E of feasible links could be deployed, being the cost of including a certain link in the solution given by the matrix C. T models those sites for which at least one connectivity requirement involving other site has to be met; these requirements are specified by the matrix R. Nodes in $V \backslash T$ (named 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 requirement exist. Using this notation the GSP-EC can be defined as follows:

Definition 2.1: 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 $G_T = (V, E_T, C)$ where every pair of terminals i, j is connected by (at least) r_{ij} edge-disjoint paths.

An example instance of the GSP is shown in Figure 1. There are six terminal nodes, colored black and labeled S_1, S_2, S_3, S_4, S_5 and S_6. There are four non-terminal nodes, colored white. The connections (links) that can be potentially deployed are shown in the figure annotated with their costs. The matrix R shows the connectivity requirements among the terminals, ranging in this case from two to three. Figure 2 shows a solution for this instance having a total cost of 29; note that only three of the four non-terminals were used in this solution. Due to the enormous intrinsic complexity of the GSP, exact algorithms to solve it (i.e. that guarantee that an optimal solution is built) can only be applied for certain topologies or limited-size instances. Therefore, to deal with real general problems, the use of heuristic algorithms conceived to generate good quality solutions within reasonable resource consumption turns to be mandatory.

3. A GRASP Heuristic

GRASP [13] 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, as shown in Figure 3. The “construction phase” (ConstPhase) builds a feasible solution that chooses (following some randomized criterion) which elements to add from a greedy-generated list of candidates. The “local search phase” (LocalSearchPhase) then explores the neighborhood of the feasible solution delivered by ConstPhase, moving consecutively to lower cost solutions until a local optimum is reached. Typical parameters are the size of the list of candidates, 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. In the following sections we introduce algorithms that implement the Construction and Local Search phases as well as the main algorithm invoking both phases to solve the GSP-EC.



$$R = \begin{pmatrix} - & 2 & 2 & 2 & 2 & 2 \\ 2 & - & 3 & 3 & 2 & 2 \\ 2 & 3 & - & 2 & 2 & 2 \\ 2 & 3 & 2 & - & 2 & 2 \\ 2 & 2 & 2 & 2 & - & 2 \\ 2 & 2 & 2 & 2 & 2 & - \end{pmatrix}$$

Figure 1. Example Instance for the GSP

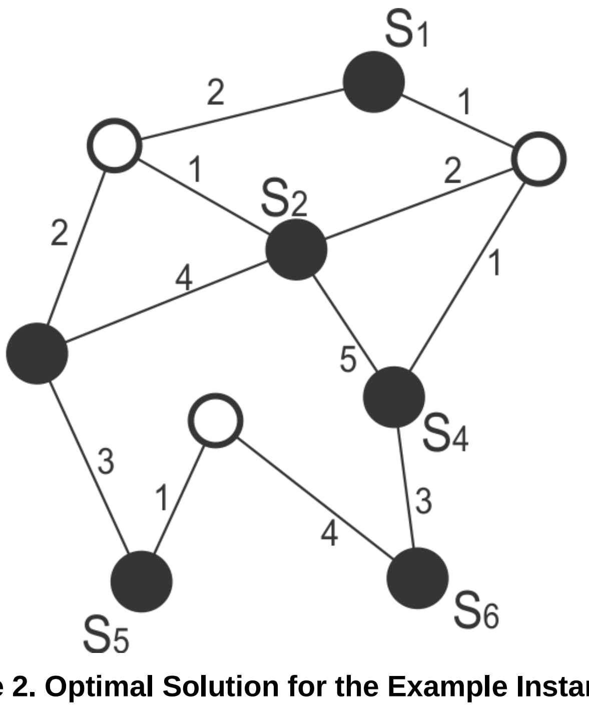


Figure 2. Optimal Solution for the Example Instance

```
Procedure GRASP($MetaParams, MaxIter, RndSeed$)
$bestSol \leftarrow NULL$
for $k = 1$ to $MaxIter$ do
    $greedySol \leftarrow$ ConstPhase($MetaParams, RndSeed$)
    $localSearchSol \leftarrow$ LocalSearchPhase ($greedySol$)
    if cost($localSearchSol$) <cost($bestSol$) then
        $bestSol \leftarrow localSearchSol$
    end if
end for
return $bestSol$
```

Figure 3: Pseudo-code for the GRASP Metaheuristic

3.1. Construction Phase Algorithm

The algorithm “ConstPhase”, shown in Figure 4, is an adaptation of a previous one [10] to the edge-connected case. It takes as inputs the graph G of feasible edges, the edge costs matrix C, the set of terminals T and the matrix of requirements R. It proceeds by building a graph which satisfies the requirements of the matrix R, starting with an edgeless graph and adding one new path in each iteration to the solution under construction G_{sol}. The matrix $M = (m_{ij})$ records the amount of connection requirements not yet satisfied in G_{sol} between the terminals i, j. The sets P_{ij} record r_{ij} disjoint paths found for connecting the nodes i, j. The auxiliary matrix $A = (a_{ij})$ is used to record how many times it was impossible to find one more path between two terminals i, j whose requirement r_{ij} was not yet completely satisfied. One improvement that this algorithm introduces over the one in [10] is to alter the costs of the matrix C to introduce randomness and enable the chance to build an optimal solution no matter what the problem instance is. We have proven (see the proof in Appendix A) that a sufficient condition is that all edges have their costs altered independently from each other and the altered costs take values in $(0, +\infty)$ with any probability distribution that assigns non-zero probabilities to any open subinterval of $(0, +\infty)$. The *while* loop is repeated until all terminal nodes have their connectivity requirements satisfied or until the algorithm fails to find a path a certain number of times MAX ATTEMPT for any pair of terminals i, j. Within the loop, each iteration works the following way. First, two terminals i, j for which there are pending connectivity requirements are randomly chosen. Then a graph G' is built from G by suppressing the edges of all paths already computed to connect i, j; thus, any path computed in G' will be edge-disjoint from the former paths connecting i, j in P_{ij}. Then the edges already present in the solution under construction G_{sol} are given cost zero; by doing this, they will be taken as costless when computing the cost of any new path, enabling edge-reusing among different pairs of terminals. Now, the shortest path (in terms of total cost) connecting i, j is computed, considering as feasible the edges of G' and with costs given by C'. In case this turns to be impossible this is acknowledged by incrementing the counter A_{ij} and resetting the path set P_{ij}, hoping that computing a different succession of paths for i, j allows to satisfy the r_{ij} requirements. In case a path p was found, it becomes part of the solution under construction and a procedure named general-update-matrix is used to update the pending connection requirements of the matrix M by applying the Fold-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 that "certifies" that the requirements specified by R were met.

```
Procedure ConstPhase($G, C, T, R$)
$G_{sol} \leftarrow (T, \emptyset); m_{ij} \leftarrow r_{ij}, \forall i,j \in T; P_{ij} \leftarrow \emptyset, \forall i,j \in T; A_{ij} \leftarrow 0, \forall i,j \in T$
$C \leftarrow$alter-costs($C$)
while $\exists m_{ij} > 0: A_{ij}$ <MAX_ATTEMPT do
    let $i,j$ be any pair of terminals with $m_{ij} > 0$
    $G' \leftarrow G \setminus P_{ij}$
    let $C' = (c'_{uv})$: $c'_{uv} = [0$ if $(u,v) \in G_{sol}$; $c_{uv}$ otherwise]
    $p \leftarrow$shortest-path $(G', C', i, j)$
    if $\nexists p$ then
        $A_{ij} \leftarrow A_{ij} + 1; P_{ij} \leftarrow \emptyset; m_{ij} \leftarrow r_{ij}$
    else
        $G_{sol} \leftarrow G_{sol} \cup \{p\}$
        $P_{ij} \leftarrow P_{ij} \cup \{p\}; m_{ij} \leftarrow m_{ij} - 1$
        $[P, M] \leftarrow$general-update-matrix $(G_{sol}, P, M, p, i, j)$
    end if
end while
return $G_{sol}, P$
```

Figure 4: Algorithms for Construction Phase

3.2. Local Search Phase Algorithm

The local search phase starts with a feasible solution delivered by the construction phase and proceeds by consecutively moving to neighbor solutions until it reaches a local optimum. Any local search algorithm needs a precise definition of the neighborhood concept; we propose two different ones that we will chain for our suggested GRASP implementation. They are defined in terms of the structural decomposition of graphs in "key-nodes" and "key-paths" [10].

Definition 3.2.1. Key-node: Given a GSP-EC instance and a feasible solution G_{sol}, a key-node is any non-terminal node with degree at least three in G_{sol}.

Definition 3.2.2. Key-path: Given a GSP-EC instance and a feasible solution G_{sol}, a key-path is any path in G_{sol} such that all intermediate nodes are non-terminals with degree two in G_{sol} and whose endpoints are either terminals or key-nodes.

A new structural component is now defined that will be used to implement our second alternative of neighborhood.

Definition 3.2.3. Key-star: Given a GSP-EC instance, a feasible solution G_{sol} and any of its nodes v, the key-star associated to v is the subgraph of G_{sol} obtained through the union of all key-paths having v as an endpoint.

With the above definitions we are now able to define two neighborhoods that will be used in a chained way to build the main GRASP algorithm.

Definition 3.2.4. Path-based Local Search Neighborhood1: Our first neighborhood 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 $\{p \in G_{sol}: k \subseteq p\}$. 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 $i \ldots j$ 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. The algorithm LocalSearchPhase1 (shown in Figure 5) then considers the replacement of key-paths k by other paths p such that cost(p) <cost(k) and the edges of p are chosen from the set $(E\setminus\chi_k(G_{sol})) \cup k$. The algorithm starts by computing the decomposition in key-nodes and key-paths of the set S. Then the *while* loop looks for successive cost improvements until no more can be done. The iterations proceed by analyzing each key-path k, trying to find a suitable replacement with lower cost for it. The graph G' is built from G by considering only the edges in $E(k) \cup (E \setminus \chi_k(S))$. 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 solution S. The new cost matrix C' is computed by zeroing the cost of all edges in $S \setminus k$ and then the path with lowest cost according to G' and C' is computed to link u and v. If the adoption of the new path implies a cost reduction over the previous one, this 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 replacements; if this does not happen, the k-decomposition is simply updated by replacing the key-path (thus avoiding recomputing a new k-decomposition). After exiting the main loop, a feasible solution S is returned, having a cost that can no more be reduced by moving to neighbor solutions.

```
**Procedure LocalSearchPhase1**$(G, C, T, S)$
$improve \leftarrow TRUE$
$\kappa \leftarrow$k-decompose$(S)$
**while** $improve$ **do**
    $improve \leftarrow FALSE$
    **for all** kpath $k \in \kappa$ with endpoints $u, v$ **do**
        $G' \leftarrow$ subgraph of $G$ induced by $E(k) \cup (E \setminus \chi_k(S))$
        $C' \leftarrow (c'_{ij}): c'_{ij} = 0$ if $(i,j) \in S \setminus k$; $c'_{ij} = c_{ij}$ otherwise
        $k' \leftarrow$shortest-path $(G', C', u, v)$
        **if** cost $(k', C')$ <cost$(k, C')$ **then**
            $improve \leftarrow TRUE$
            update $S: \forall p \in J_k(S)(p \leftarrow (p \setminus k) \cup k')$
            **if** $\exists z \in V(k'), z \notin \{u, v\}$,degree $(z) \geq 3$ in $S$ **then**
                remove-cycles $(J_k(S))$
                $\kappa \leftarrow$k-decompose $(S)$
            **else**
                $\kappa \leftarrow \kappa \setminus \{k\} \cup \{k'\}$
            end if
        end if
    end for
end while
return $S$
```

Figure 5. Algorithm LocalSearchPhase1

Definition 3.2.5. Key-star-based Local Search Neighborhood2: This is a second neighborhood based on the replacement of key-stars, which frequently allows to improve feasible solutions that are locally optimal when only considering Neighborhood1. In the case of the GSP node-connected (GSP-NC), all key-stars are trees (key-trees), as no node sharing is allowed among disjoint paths. Due to the possibility of sharing nodes among edge-disjoint paths, when working with GSP-EC problems, we deal with key-stars. Unlike previous works [10,11] we allow the root node to be a terminal node, thus getting a broader (and richer) neighborhood. 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-shaped 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 that get broken when removing the edges of k. 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. Figure 6 presents the LocalSearchPhase2 algorithm. It looks at the key-stars given by the k-decomposition of S, determining for each one the best key-star able to replace it until an improvement is obtained.

The LocalSearchPhase2 algorithm makes use of a procedure named BestKeyStar whose pseudo-code is shown in Figure 7. 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. The BestKeyStar algorithm is based on the idea of building key-stars by employing a "simult-shortest-path" polynomial-time algorithm to compute a given number of edge-disjoint paths with minimized total cost [14]. It starts by computing the subgraph G' of G obtained by removing the edges that could cause loss of connectivity levels if reused; the altered costs matrix C' with zeroed costs for reused edges; and it adds a virtual node w whose purpose is explained below. The set of nodes Ω is computed so that it includes the leaf nodes that the key-start to build must have. Then each of the latter are connected to w with an appropriate number of parallel zero-cost edges totaling $\delta_{G',w}$ (degree of w in G') edges. Then the second *for all* loop considers nodes of G that could be potential roots z of the key-star to be found and then builds the lowest-cost one with root node z by the application of the already mentioned simult-shortest-path algorithm. To do so, $\delta_{G',w}$ edge-disjoint paths connecting z and w are requested. If found with cost lower than k then the new key-start and its associated cost are recorded as the best one so far found. After having considered all possible root nodes, the best key-start and its cost according to C' are returned. All the process is depicted in Figure 8. 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 to t, u, v by the appropriate amount of edges, and a "candidate" root node z; (c) the shortest paths found to connect z and w; and (d) the new key-star obtained after removing the virtual node w.

```
Procedure LocalSearchPhase2($G, C, T, S$)
$improve \leftarrow TRUE$
$\kappa \leftarrow$k-decompose($S$)
while $improve$ do
    $improve \leftarrow FALSE$
    for all kstar $k \in \kappa$ do
        $[k', newCost] \leftarrow$BestKeyStar($G, C, T, S, k$)
        if $newCost <$cost $(k, C)$ then
            $improve \leftarrow TRUE$
            replace $k$ by $k'$ in all paths of $S$
            $\kappa \leftarrow$k-decompose $(S)$
            abort for all
        end if
    end for
end while
return $S$
```

Figure 6. Algorithm LocalSearchPhase2

```
Procedure BestKeyStar $(G, C, T, S, k)$
$G' \leftarrow$subgraph induced from $G$ by $E(k) \cup (E \setminus \chi_k(S))$
$C' \leftarrow (c'_{ij})$: $c'_{ij} = 0$ if $(i, j) \in S \setminus k$; $c'_{ij} = c_{ij}$ otherwise
add a virtual node $w$ to $G'$
$\Omega \leftarrow \psi_k$
if $\theta_k \in T$ then
    $\Omega \leftarrow \Omega \cup \{\theta_k\}$
end if
for all $m \in \Omega$ do
    add $\hat{\delta}_{k,m}$ parallel edges $(w, m)$ to $G'$ with cost 0
end for
$c_{min} \leftarrow 0; k_{min} \leftarrow k$
for all $z \in V(G)$ do
    $k' \leftarrow$simult-shortest-paths$(G', \delta_{G,w}, z, w)$
    if $k'$ has $\delta_{G,w}$ paths $\wedge$ cost$(k', C') < c_{min}$ then
        $c_{min} \leftarrow$cost$(k', C'); k_{min} \leftarrow k'$
    end if
end for
return $[k_{min}, c_{min}]$
```

Figure 7. Algorithm BestKeyStar

3.3. The Bundled GRASP Algorithm

Now we are able to put the pieces together and build a GRASP algorithm to solve the GSP-EC. Figure 9 shows the corresponding 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-start 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. The algorithm receives as inputs the graph G of feasible connections, the cost matrix C, the terminals T, the redundancy requirements matrix R and a number of iterations $iters$ to perform. First the minimum found cost $c_{mín}$ is initialized to ∞ and an empty path set S_{opt} is initialized. The main loop is executed $iters$ times and then the best solution found is returned. When starting an iteration, the ConstPhase is invoked to build a feasible solution; the returned set S contains those paths that certify feasibility of the solution returned. If the set S has less paths than the so far found best solution S_{opt} this iteration is discarded. This could happen if the last call to ConstPhase was not able to satisfy all requirements of R while a previous call was able to do it (or at least to satisfy a greater number); our first objective is to satisfy as many requirements of R as possible, and only then minimize the total cost. Then, Neighborhood1 is applied by invoking

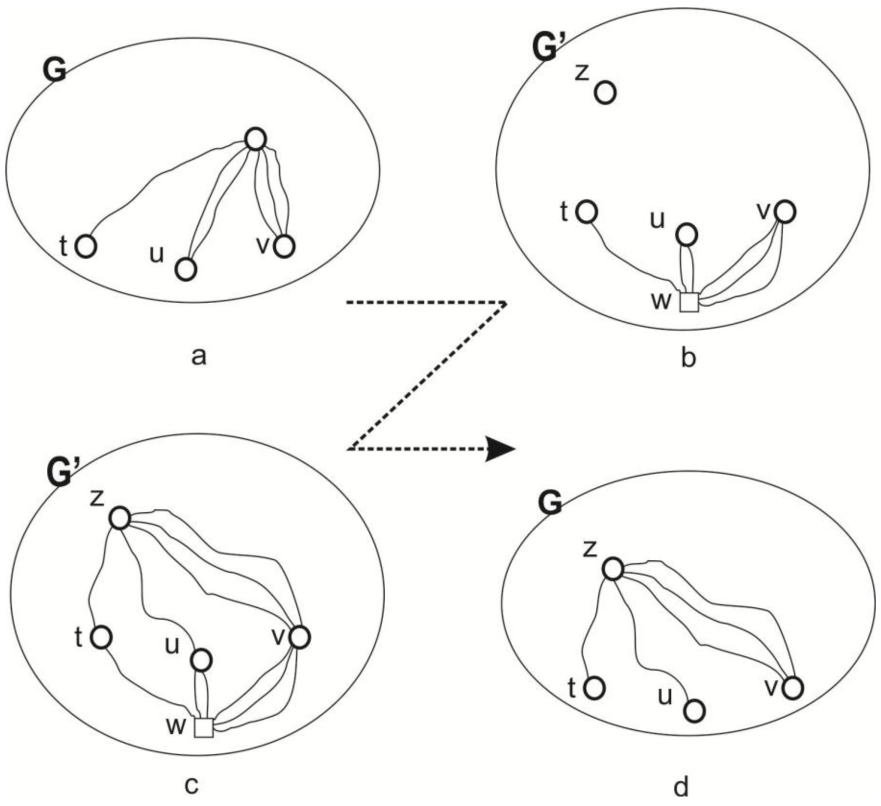


Figure 8: How BestKeyStar Proceeds

LocalSearchPhase1. If this was the first local search or if a cost reduction was achieved then a best key-star movement (Neighborhood2) is tried by invoking LocalSearchPhase2. In case the latter succeeds to reduce the cost the execution flow resumes at *OptLoop* for trying a new cycle of chained improvements. When no further local improvements are possible, the best known solution is updated (in case an improvement was achieved) and returned.

```
Procedure GRASP_GSP($(G, C, T, R, iters)$)
$c_{min} \leftarrow \infty; S_{opt} \leftarrow \emptyset$
for $i = 1$ to $iters$ do
    $[G_{sol}, S]$ ←ConstPhase $(G, C, T, R)$
    if $|S| \geq |S_{opt}|$ then
    $flag \leftarrow TRUE$
    OptLoop: $[G_{sol}, S']$ ← LocalSearchPhase1 $(G, C, S)$
        if $flag \vee$ cost$(S', C)$ <cost$(S, C)$ then
            $flag \leftarrow FALSE$
            $[G_{sol}, S'']$ ← LocalSearchPhase2 $(G, C, S')$
            if cost$(S'', C)$ <cost$(S', C)$ then
                $S \leftarrow S''$;  go to OptLoop
            end if
        end if
        if cost$(S', C') < c_{mín}$ then
            $c_{mín}$ ← cost $(S'', C); S_{opt} \leftarrow S$
        end if
    end if
end for
return $S_{opt}$
```

Figure 9. Main GRASP Algorithm

4. Performance Tests

This section presents the results obtained after testing our algorithms with twenty-one 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. One hundred iterations were run for every instance.

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 [16]: instances of the Steiner problem; in many cases the optimal solution is known, in others the best known solution is available;
- Tsplib [17]: instances of diverse graph theory related problems, including a “Traveling Salesman Problem” section.

The main characteristics of the twenty-one test cases are shown in Table 1. For each case we show:

- total number of nodes (V);
- number of feasible edges (E);
- number of terminal nodes (T);

- number of Steiner (non-terminal) nodes (St);
- the level of edge-connectivity requirements (one, two, three or mixed) (Redund);
- the optimal costs when known (Opt).

GSP problems solved with connectivity level one are Steiner problems and in those cases we got the optimal solution cost reported by 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"; eight nodes were taken as 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 nodes were taken as terminal and we solved three instances for each one with connectivity requirements one, two and a mix of one to three. Finally the cases bayg29 and att48 where taken from the library Tsplib; both correspond to real cases (twenty-nine cities from Bavaria, Germany and forty-eight cities from USA).

Table 1. Test Cases

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	1,2,3-EC	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	1,2,3-EC	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

4.2. Numerical Results

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

- total amount of requirements satisfied by the best solution found (Reqs);
- the average running time in milliseconds per iteration (t);
- the cost of the best solution found (Cost);
- "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 (LSI).

Table 2. Numerical Results

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	6857	4.6
bayg29-r3	165	2413	11722	4.2
att48-r2	90	1313	23214	13.0
Avg.	**265**	**6524**	**-**	**6.7**

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 hp-6p-r1 (found cost 4033 being the optimal 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 found solutions). All found solutions are edge-minimal regarding feasibility (no edge can be suppressed without losing required connectivity levels). In all cases the maximum possible number of requirements are satisfied. This means that 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. Figures 10 and 11 show the best 3-connected network found to connect the sites of bayg29 and the best 2-connected network found to connect the sites of att48. Source data of the twenty-one instances as well as the best solutions found are available here [17].

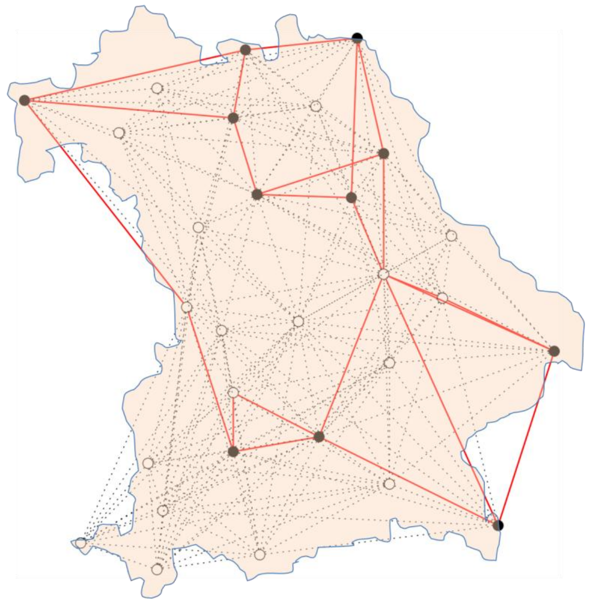


Figure 10. Best Found Solution for bayg29

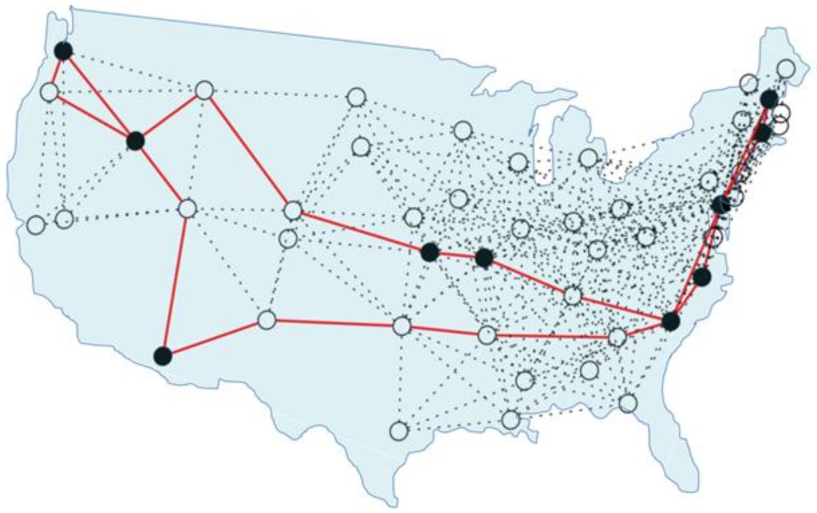


Figure 11. Best Found Solution for att48

5. Conclusions

In this paper we introduced an algorithm for addressing the edge-disjoint version of the GSP that extends ideas from previously proposed algorithms for the node-disjoint version. Our algorithm was shown to find good quality solutions when applied to a series of heterogeneous test cases with up to one hundred nodes and four hundred and six edges. 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 (averaging 6.7%) 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 [10,11] for the node-connected version of the GSP. In all cases the maximum possible amount of connection requirements (allowed by the topology of the original graph G and the requirements in R) was reached and edge-minimal solutions were always returned.

Appendix

Appendix 1. Local Optima Traps

Previous algorithms [10,11] do not satisfy the property mentioned in Subsection 3.1 as evidenced by the example shown in Figure 12. In (a) an instance of the GSP-EC is shown with three terminal nodes (black), one Steiner node (white) and nine edges labeled with their costs. Let us assume that the matrix R requires one path to connect every pair of terminals. If those algorithms were run with the GRASP parameter "candidate list size" (used in the mentioned 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 algorithms will never find the optimal solution (with cost 18) built with the three edges that have cost 6. Furthermore, none of the solution 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 as this one) of the GSP and parameterizations of previously introduced GRASP

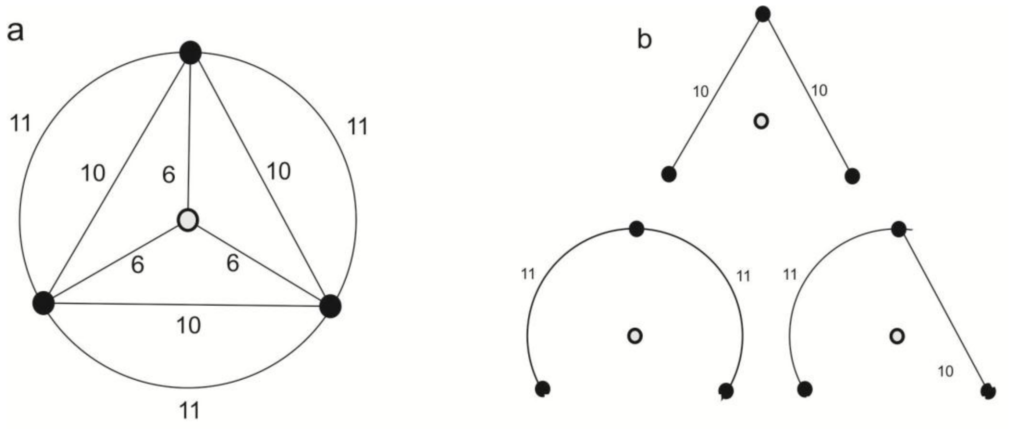


Figure 12: Local Optima Traps

algorithms for which an optimal solution will never be attained.

Appendix 2. Buildable Solutions Space

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 named alter-costs. Edges having original cost zero keep that cost. The remaining edges have their costs altered independently from each other taking random 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 function whose parameter is the reciprocal of the original cost. Next we prove that this is enough to ensure that for all instances of the GSP our algorithm ConstPhase has non-zero probability of reaching an optimal solution.

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 that the algorithm does not find this solution even once after running k times would be $(1-p)^k$ and the probability of building this solution at least once would be $p_k(S) = 1-(1-p)^k > 0$. Then, $\lim_{k\to\infty} p_k(S) = 1$ what would complete the proof.

Let us see that given any instance of the GSP and an optimal solution S, the probability p of building S is different that zero. Let $(S)_i = s_1, s_2, \dots, s_r$ 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 a minimal amount of 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 $s_1 \dots s_{k-1}$ is minimal.

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

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

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

$$\left(\sum_{i<j} r_{ij}\right)! \left(\prod_{i<j} r_{ij}!\right)^{-1}$$

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

$$p_{order}(S) = \left(\left(\sum_{i<j} r_{ij}\right)!\right)^{-1} \left(\prod_{i<j} r_{ij}!\right) > 0$$

Let us now suppose that the algorithm generates the paths in the same order given by $(S)_i$ with regard to the extreme nodes. We will now see 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 probability distribution 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 $0 < \epsilon < 1/2|E|$ and a certain integer k let $\hat{C}_k = \left(\hat{c}_{k,ij}\right)$ be any cost assignement for the feasible edges satisfying $\hat{c}_{k,ij} = 0$ if $c_{ij} = 0$ and $\hat{c}_{k,ij} \in (k - \epsilon, k + \epsilon)$ if $c_{ij} > 0$. In what follows we 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 assign a cost according to $\hat{C}_1$ to every edge in s_1. We assign a cost according to $\hat{C}_2$ to every edge in $s_2 \setminus s_1$. Similarly we assign costs according to $\hat{C}_k$ to every edge in $s_k \setminus (s_1 \cup s_2 \cup \ldots \cup s_{k-1})$. 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 “swaped” edge 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 $(S)_i$. So 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 the family $\hat{C}$ (because for any edge with non-zero original cost there is a non-zero probability that our altering costs function assigns 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] Krarup, “The generalized steiner problem”, Technical report, DIKU, University of Copenhagen, **(1979)**.

[2] Winter, “Steiner problem in networks: A survey”, Networks, vol. 17, no. 2, **(1987)**, pp. 129–167.

[3] Agrawal, Klein and Ravi, “When trees collide: An approximation algorithm for the generalized Steiner problem on networks”, SIAM Journal on Computing, vol. 24, no. 3, **(1995)**, pp. 440–456.

[4] Baïou, “Le problème du sous-graphe Steiner 2-arête connexe: Approche polyédrale”, PhD thesis, Université de Rennes I, Rennes, France, **(1996)**.

[5] Baïou, Mahjoub, “Steiner 2-edge connected subgraph polytope on series-parallel graphs”, SIAM Journal on Discrete Mathematics, vol. 10, no. 1, **(1997)**, pp. 505–514.

[6] Baïou, “On the dominant of the Steiner 2-edge connected subgraph polytope”, Discrete Applied Mathematics, vol. 112, no. 1-3, **(2001)**, pp. 3–10.

[7] Kerivin, Mahjoub, ‘Design of Survivable Networks: A survey”, Networks, vol. 46, no. 1, **(2005)**, pp. 1–21.

[8] Mahjoub, Pesneau, “On the Steiner 2-edge connected subgraph polytope”, RAIRO Operations Research, vol. 42, no. 1, **(2008)**, pp. 259–283.

[9] Monma, Munson, Pulleyblank, “Minimum-weight two connected spanning networks”, Mathematical Programming, vol. 46, no. 1, **(1990)**, pp. 153–171.

[10] Robledo, Canale, “Designing backbone networks using the generalized steiner problem”, Proceedings of the 7th International Workshop on Design of Reliable Communication Networks, DRCN 2009, vol. 1, **(2009)**, pp. 327–334.

[11] Robledo, “GRASP heuristics for Wide Area Network design”, PhD thesis, IRISA, Université de Rennes I, Rennes, France, **(2005)**.

[12] Sartor, "Problema general de Steiner en grafos: resultados y algoritmos GRASP para la versión arista-disjunta", M.Sc. thesis, Universidad de la República, Montevideo, Uruguay, **(2011)**.

[13] Resende, Ribeiro, "Greedy randomized adaptive search procedures", In F. Glover and G. Kochenberger, editors, Handbook of Metaheuristics, **(2003)**, pp. 219–249. Kluwer Academic Publishers.

[14] Bhandari, "Optimal physical diversity algorithms and survivable networks", Proceedings of the Second IEEE Symposium on Computers and Communications, **(1997)**, pp. 433–441.

[15] Koch, Konrad-Zuse-Zentrum für Informationstechnik Berlin. Steinlib test data library. Available at http://steinlib.zib.de/steinlib.php.

[16] Ruprecht-Karls-Universitat Heidelberg, Tsplib network optimization problems library. Available at http://comopt.ifi.uniheidelberg.de/software/TSPLIB95.

[17] Universidad de Montevideo, GSP-EC test set with best results found. Available at http://www2.um.edu.uy/psartor/grasp-gsp-ec.zip.

Authors

Pablo Sartor

- PhD (c) Computer Science, PEDECIBA, Facultad de Ingeniería-UDELAR, Uruguay (2011-).
- MSc Computer Science, PEDECIBA, Facultad de Ingeniería-UDELAR, Uruguay (2011).
- MBA, IEEM, Universidad de Montevideo, Uruguay (2009).
- Systems Engineer, Facultad de Ingeniería, UDELAR, Uruguay (1999).

Franco Robledo

- PhD Computer Science, INRIA/Université de Rennes I, France. (2005).
- PhD Computer Science, PEDECIBA, Facultad de Ingeniería-UDELAR, Uruguay (2005).
- MSc Computer Science, PEDECIBA, Facultad de Ingeniería-UDELAR, Uruguay (2000).
- MBA focused on Finance, Centro de Posgrado de la Facultad de CCEE y Adminstración-UDELAR, Uruguay.
- Systems Engineer, Facultad de Ingeniería, UDELAR, Uruguay (1997).