"This is the peer reviewed version of the following article: Martins, L., Gomes, T. and Tipper, D. (2017), Efficient heuristics for determining node-disjoint path pairs visiting specified nodes. NETWORKS, 70: 292–307. doi:10.1002/net.21778, which has been published in final form at http://dx.doi.org/10.1002/net.21778. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Self-Archiving."

Efficient Heuristics for Determining Node-Disjoint Path Pairs Visiting Specified Nodes

Lúcia Martins*†, Teresa Gomes*†, David Tipper‡

*Department of Electrical and Computer Engineering, University of Coimbra
Pólo 2, Rua Sílvio Lima 3030-290 Coimbra, Portugal

†INESC Coimbra, Rua Sílvio Lima, 3030-290 Coimbra, Portugal

‡Graduate Telecommunications and Networking Program
University of Pittsburgh, Pittsburgh, PA 15260

lucia,teresa@deec.uc.pt, tipper@tele.pitt.edu

Abstract

A new recursive heuristic is proposed to calculate a shortest simple path, from a source node to a destination node, that visits a specified set of nodes in a network. To provide survivability to failures along the path, the proposed heuristic is modified to ensure that the calculated path can be protected by a node-disjoint backup path. Additionally, the case when both paths in the disjoint path pair are required to visit specific sets of nodes is studied and effective heuristics are proposed. An evaluation of the solutions of the heuristics is conducted by comparing with results from an integer linear programming (ILP) formulation for each of the considered problems, and also with previous heuristics. The ILP solver may require a significant amount of time to obtain a solution, especially in large networks, which justifies the need for effective, computationally efficient heuristics for solving these problems.

Keywords: Resilient routing, visiting a given set of nodes, min-sum, heuristics, node-disjoint path pair, path-based formulation.

1 Introduction

Communication networks have a very important role in today's society. Many services require uninterrupted service even in the presence of challenges [20]. An overview on resilience, strategies for attaining resiliency and survivability in communication networks can be found in [24].

Network service providers, depending on service level agreements, may need to ensure distinct levels of resiliency per service, which leads to the introduction of the concept of Quality of Resiliency classes [5, 6, 23]. Network recovery can be ensured using protection (pre-designed restoration) or rerouting (restoration after fault detection).

Routing with path protection seeks to obtain a pair of node (or arc) disjoint paths (the active path and the protection path). In this context it is important to evaluate the survivability level of a given network, also in a geographical manner, which is of paramount important in the case of disaster-based resilience. Related with this last aspect in [21] a relevant set of graph metrics that characterize path diversity in a given graph, also taking into account geographic aspects, are proposed. The evaluation of the fitness of geographic graph generators for representing realistic physical networks is studied in [3].

Sometimes it is necessary to establish a path with specified nodes. These nodes may have been chosen due to inter-operator's agreements, or due to network management and operational constraints, such as transiting optical nodes that support wavelength conversion. An emerging scenario where specific nodes may be required for some network flows is deep packet inspection of traffic where a subset of network nodes are equipped with deep packet security monitoring capabilities. These security inspections may help to prevent disasters due to malicious attacks on the network or end users. For an overview of network resilience issues see [18] and for an overview of security challenges in communication networks see [12].

Very few works address the problem of calculating a shortest path from a source node to a target node that visits a given set of nodes. This problem was first pointed out by Kalaba [16] and the first attempt to solve this problem is due to Saksena and Kumar [22], where the authors sought to develop an exact algorithm, using Bellman's optimality principle, for calculating a shortest path (possibly with cycles) that visits a specified set of nodes. In their approach, they consider the optimal path must be composed of segments that belong to the set of shortest paths between the nodes in the set of specified nodes and the shortest paths between the specified nodes and the source and target nodes. However their algorithm (designated hereafter SK66) is not correct. Dreyfus [11] questioned their approach because it did not necessarily obtain the minimum cost solution. Bajaj [1] introduced relevant modifications to correct the algorithm by Saksena and Kumar [22]. Additionally he also proposed an algorithm for solving the problem when the order of the nodes to visit is known – in both cases cycles are allowed.

Ibaraki in [15] considered separately the problem of calculating a shortest loopless path that visits a given set of nodes and a shortest path (possibly with cycles) that visits a set of specified nodes. Two approaches to obtain a shortest loopless path visiting a given set of nodes, one based on dynamic programming and the other based on the branch and bound principle, are proposed by Ibaraki [15]. Computational results in [15] suggest that the algorithm based on dynamic programming is less efficient than the algorithm based on branch and bound principle. However for the case of a single mandatory node v, the optimal solution can be easily obtained through an algorithm that computes two disjoint shortest paths, as is the case of Suurballe algorithm [26], adding two fictitious nodes to the graph and also splitting the mandatory node into two nodes v' and v''. One fictitious node is to be connected through fictitious links to the source and to the destination nodes and the other fictitious node is to be connected to v' and to v''. After two disjoint shortest paths between the fictitious nodes are computed, the desired path will be the concatenation of the two paths after removing the fictitious nodes and the respective connection links from those paths.

Vardhan *et al.* [28] propose an algorithm to find a simple path where the sequence of the nodes to visit is given. The authors point out that if the ordering of the set of nodes to visit is relaxed, then a re-ordering of the nodes using a *depth first transversal* can improve the performance of the algorithm. Note that the cost of the path is not taken into account in their approach.

In the context of planning a travel route, arriving in time to meet a deadline can be the main concern. Wu *et al.* [29] formulate a problem with the objective to calculate a path that has 100% probability of punctual arrival and also has the fewest number of road links. Reducing the number of road links should reduce the number of crossings and/or traffic lights which may lower the travel time and its variability. They extend their approach to consider the case where travellers add several fixed locations to visit. To solve this problem they propose a formulation that requires sub-tour elimination, which they implement by adding constraints. They compare their approach with the enumeration method, for one, two and three specified locations to visit, and claim their formulation requires less CPU time.

In [4] the authors evaluated the GeoDivRP routing protocol with minimum-cost and the delay-skew requirement. The geodiverse routes required by this protocol are calculated by the heuristic iWPSP. These routes are a set of k geodiverse paths, where each path is separated by a distance d, with respect to the reference shortest path, while ensuring the skew value (the difference in delay time across the set of paths) is satisfied. To achieve this goal, and using (initially) the shortest path as reference, iWPSP selects an intermediate node (waypoint) where an additional path must pass, to ensure geodiversity. Each geodiverse path is obtained through the concatenation of two shortest paths between the waypoint and two previously obtained end nodes. These end nodes are the source neighbour and destination neighbour nodes that are d distance separated from the source and destination nodes, respectively.

Andrade [7, 8] developed new formulations for addressing the determination of a shortest loopless path, without cycles, from a source node s to a destination node t, that visits only once all nodes of a specified set. Three formulations are presented in [8]: $Q2$, $Q3$ and $Q4$ – note that $Q2$ and $Q3$ had already been introduced in [7]. Model $Q2$ is based on an adapted version of the cycle elimination constraints of the spanning tree polytope; model $Q3$ is a primal-dual based mixed integer formulation; and model $Q4$ is derived from a flow-based compact model for the Steiner traveling salesman problem (TSP). The numerical results presented in [7] show that the new primal-dual based mixed integer formulation, designated $Q3$, is more effective than $Q2$. Numerical experiments in [8] show that $Q3$ is the best compromise model, in terms of execution time and in handling large instances for solving the problem.

In [11], Dreyfus proposes an approach for obtaining a shortest path (possibly with cycles), from a source node to a target node that visits a given set of nodes, and concludes that the problem can not be easier than the traveling salesman problem of dimension k, where $k - 2$ is the number of given nodes to be visited.

In [13] algorithm SK66 was adapted to ensure only loopless paths were considered admissible, and some modifications were introduced to improve the original algorithm effectiveness in the context of loopless path calculation; this new improved version was designated SK. The problem of finding a protected loopless path visiting a given set of nodes was also addressed in [13]. Based on SK two new heuristics were proposed (ASK and BSK); having verified that ASK was computationally more effective than BSK, but that BSK often found solutions ASK was unable to obtain, algorithm ABSK [13] was proposed, where ASK is used followed by BSK if necessary.

In [14] the algorithm VSN was proposed to visit a set S of nodes (in a path from s to t). The main idea of algorithm VSN is to build an auxiliary graph, with node set $\{s,t\} \cup S$, where a shortest paths visiting all nodes in the graph can be obtained using a k-shortest path enumeration algorithm like Yen's [31]. A solution can only be found in a short time if the the number of nodes of S is small (less than 9), otherwise generating paths until finding paths visiting all nodes may take a very long time. The obtained path is expanded into a path in the original network. If the resulting expanded path does not contain any cycle it is a feasible solution to the min-cost loopless path visiting specified nodes, and the algorithm ends. If the obtained path contains a cycle, the k-shortest paths method keeps generating paths as long as they have the same minimum cost. If these actions do not result in finding a loopless path visiting all nodes in S, then the strategy is to successively delete one arc from the original network, and repeat the above procedure (which starts by recomputing the auxiliary graph) until a solution is found, or a certain number of arcs have been deleted, or no paths from s to t can be obtained in the auxiliary graph. In order to provide survivability, a trap avoidance approach, inspired by [30], was incorporated in the VSN heuristic, resulting in the VTA and MSVTA heuristics [14]. VTA ensures the calculated active path can be protected by a node-disjoint path, and MSVTA obtains a min-sum node-disjoint path pair where each path must visit a different set of specified nodes.

The contribution of this work is a recursive heuristic to calculate a shortest path that visits a given set of nodes, designated PSN, and its extension PPSN to provide a node-disjoint backup, which were first proposed in [17]. Here, and based on the mentioned recursive heuristic, the problem of calculating a min-sum pair of node-disjoint simple paths, each visiting a set of nodes, is also addressed, and two new heuristics are proposed for solving it.

Experimental results in [17] and [14] showed that SK and ABSK were the algorithms with the worst performance. Hence in the present work we will evaluate the proposed heuristic only with respect to the ones introduced in [14]. Moreover an improved version of PPSN, designated PSNTA for determining a protected path visiting specified nodes is proposed. This problem is also formulated as path-based optimization model, and results are presented that justify the use of the heuristic in large networks.

The remainder of the paper is structured as follows. In Section 2 the notation is introduced, the problems are formally defined and a path-based optimization model is presented. In Section 3 a new and effective heuristic is proposed for the computation of a loopless path visiting specified nodes. The algorithm is then modified to take into account the constraint that the obtained path must be protected by a node-disjoint path. This approach is further extended to address the problem of calculating a min-sum pair of node-disjoint simple paths, each visiting a set of nodes. The trap avoidance approach, proposed in [14], is also explored for node-disjoint path pair calculation. Computational results are presented in Section 4. Section 5 concludes the paper.

2 Notation and problem formulations

This work addresses three problems. The first one is the calculation of a shortest loopless path, from a source to a destination node, visiting a given set of nodes, designated as problem $\mathcal{P}_0$. The second problem, designated as problem $\mathcal{P}_1$, consists in solving $\mathcal{P}_0$ with the constraint that the obtained path can be protected by a node-disjoint path. The third problem $\mathcal{P}_2$, consists in obtaining a min-sum node-disjoint simple path pair, such that each path must visit a different given set of nodes. A loopless path must visit each of its nodes only once. Hence, unless explicitly stated otherwise, all paths are considered to be loopless.

2.1 Notation

We adopt the following notation. A directed graph $G = (V, A)$ is defined by a set of vertices (or nodes) $V = \{v_1, \ldots, v_n\}$, and a set of directed arcs $A = \{a_1, \ldots, a_m\}$, where n and m are the number of nodes and arcs, respectively, of G. Each arc $a_k = (v_i, v_j)$, with $v_i, v_j \in V$ $(v_i \neq v_j)$, is an ordered pair of elements belonging to V; v_i is the tail (or source) of the arc and v_j is its head (or destination). Arc (v_i, v_j) is said to be emergent from node v_i and incident on node v_j.

A path from a source node s to a destination node t, $(s, t \in V)$, is represented by $p = \langle s \equiv v_1, v_2, \ldots, v_k \equiv t\rangle$, with $(v_i, v_{i+1}) \in A, \forall i \in \{1, \ldots, k-1\}$, where k is the number of different nodes in the path. A path from a node v_i to a node v_j may also be represented by $p_{v_i v_j}$. If a path between a given pair of nodes does not exist, it is represented by the empty set ($\emptyset$). A segment is a continuous sequence of nodes that are part of a path.

The sets of nodes (arcs) of path p will be represented by V_p (A_p). The *concatenation* of paths $p_{v_i v_j}$ and $p_{v_j v_l}$ is the path $p_{v_i v_j} \diamond p_{v_j v_l}$ from v_i to v_l, which coincides with $p_{v_i v_j}$ from v_i to v_j and with $p_{v_j v_l}$ from v_j to v_l. Let p and $\hat{p}$ be two paths such that the first (last) node of p is the last (first) node of $\hat{p}$. Let $p \circledcirc \hat{p}$ represent the concatenation of those paths which will coincide with $p \diamond \hat{p}$ or (exclusive) $\hat{p} \diamond p$. Moreover $p \circledcirc \emptyset$ (or $\emptyset \circledcirc p$) results in p. Given a path p such that $p = \hat{p} \circledcirc \dot{p}$, the operation of removing $\hat{p}$ from p, resulting in path $\dot{p}$, will be represented by $p \oslash \hat{p}$ or by $\hat{p} \oslash p$.

A pair of paths from s to t is represented by (p, q). The paths are node-disjoint if and only if $V_p \cap V_q = \{s, t\}$. Two paths that can be concatenated, like $p_{v_i v_j}$ and $p_{v_k v_i}$, are node-disjoint if $V_{p_{v_i v_j}} \cap V_{p_{v_k v_i}} = \{v_i\}$, that is if they only share the possible concatenation node. Each arc $(v_i, v_j) \in A$ is associated with a strictly positive cost $w(v_i, v_j)$, and the cost D_p of a path p is the sum of the costs of the arcs constituting the path: $D_p = \sum_{(v_i,v_j) \in A_p} w(v_i, v_j)$.

Let $\mathcal{P}_{st}$ represent the set of all paths from s to t in the network. The set of nodes that must be visited by the active or working path is designated by S. In the case the backup path must also visit a set of specified nodes this set is denoted by B.

The algorithms require the following additional notation. Let P_p designate the set of shortest paths between each distinct pair of nodes in S, excluding all other nodes in S and in $\{s, t\}$. Also, P_p contains a shortest path from s to the nodes in S and a shortest paths from the nodes in S to t, in both cases calculated excluding all other nodes in S together with t and s, respectively. The elements of P_p are, potentially, segments of the solution of the problem $\mathcal{P}_0$. The rest of the notation, closely related with algorithms, will be defined as needed.

2.2 Formulation of Problems

Problem $\mathcal{P}_0$, which finds a shortest loopless path p_1^*, from s to t, visiting a set of specific nodes S, can be stated as follows:

$$p_1^* = \arg\min_{p_1 \in \mathcal{P}_{st}} D_{p_1} \tag{1}$$

$$\text{s.t.} \quad V_{p_1} \cap S = S \tag{2}$$

An ILP formulation to obtain p_1^* can be found in [8] and is used in the numerical results section for comparative evaluation of the heuristics.

Problem $\mathcal{P}_1$ seeks to obtain a shortest path p_1^*, visiting the set of nodes S, such that it can be protected by a node-disjoint path p_2^*, and can be written as:

$$(p_1^*, p_2^*) = \arg\min_{p_1, p_2 \in \mathcal{P}_{st}} D_{p_1} \tag{3}$$

$$\text{s.t.} \quad V_{p_1} \cap S = S, \quad V_{p_1} \cap V_{p_2} = \{s, t\} \tag{4}$$

In [13] we proposed an ILP formulation for obtaining (p_1^*, p_2^*), which is an adaptation of the formulation in [7,8] with the additional constraint that the obtained path can be protected by a node-disjoint path.

Problem $\mathcal{P}_2$ seeks to obtain a pair of node-disjoint simple paths, each visiting a specified set of nodes S and B respectively, such that the sum of the costs of the paths is minimum.

$$(p_1^*, p_2^*) = \arg\min_{p_1, p_2 \in \mathcal{P}_{st}} D_{p_1} + D_{p_2} \tag{5}$$

$$\text{s.t.} \quad V_{p_1} \cap S = S, V_{p_2} \cap B = B$$

$$V_{p_1} \cap V_{p_2} = \{s, t\} \tag{6}$$

An ILP formulation to obtain (p_1^*, p_2^*) in equations (5)-(6) can be found in our previous work [14], which is also an adaptation of the formulation in [7, 8].

The formulations of optimization problems $\mathcal{P}_1$ in [13] and $\mathcal{P}_2$ in [14] are arc-flow models. However, one can formulate a path-based optimization model for the problems. Here we present the formulation for $\mathcal{P}_1$ to illustrate the complexity. Let $\mathcal{P}_{st}^S$ represent the set of all paths from s to t that visit the specified set of nodes S and let $\mathcal{P}_{st}^{\bar{S}}$ be the set of paths from s to t that *do not* visit the nodes in S. Let $d_{s,t}$ denote the volume of traffic demand from s to t and let c_e be the unit cost of capacity on edge (link) $e \in A$. The variable t_e is the amount of traffic demand routed on edge $e \in A$. The decision variable $y_{s,t,k}$ is equal to the demand routed on the kth path in $\mathcal{P}_{st}^S$. Similarly, $z_{s,t,k}$ is equal to the demand on the kth backup path in $\mathcal{P}_{st}^{\bar{S}}$. We define binary flow variables $w_{s,t,k}$ which are equal to 1 if the kth path of $\mathcal{P}_{st}^S$ is selected for the working path, otherwise it is 0. Similarly, $b_{s,t,k}$ is set to 1 if the kth path of $\mathcal{P}_{st}^{\bar{S}}$ is chosen for the backup path and is 0 otherwise. Also, $\delta_{e,k}$ is an indicator parameter that is equal to 1 if the kth path of $\mathcal{P}_{st}^S$ uses link e, otherwise it is zero. Correspondingly, $\beta_{e,k}$ is an indicator parameter that is equal to 1 if the kth path of $\mathcal{P}_{st}^{\bar{S}}$ uses link e, otherwise it is zero. In like fashion, $\psi_{v,k}$ is an indicator parameter which is equal to 1 if the kth path of $\mathcal{P}_{st}^{\bar{S}}$ uses node v and $\eta_{v,k}$ is an indicator parameter which is equal to 1 if the kth path of $\mathcal{P}_{st}^{\bar{S}}$ uses node v.

$$\min \quad \sum_{e \in A} c_e t_e \tag{7}$$

s.t.

$$\sum_{k \in \mathcal{P}_{st}^{S}} w_{s,t,k} = 1 \tag{8}$$

$$y_{s,t,k} = w_{s,t,k} d_{s,t} \quad \forall k \in \mathcal{P}_{st}^{S} \tag{9}$$

$$\sum_{k \in \mathcal{P}_{st}^{\bar{S}}} b_{s,t,k} = 1 \tag{10}$$

$$z_{s,t,k} = b_{s,t,k} d_{s,t} \quad \forall k \in \mathcal{P}_{st}^{\bar{S}} \tag{11}$$

$$\sum_{k \in \mathcal{P}_{st}^{S}} \delta_{e,k} y_{s,t,k} + \sum_{k \in \mathcal{P}_{st}^{\bar{S}}} \beta_{e,k} z_{s,t,k} \le t_e \quad \forall e \in A \tag{12}$$

$$\sum_{k \in \mathcal{P}_{st}^{S}} \psi_{v,k} w_{s,t,k} + \sum_{k \in \mathcal{P}_{st}^{\bar{S}}} \eta_{v,k} b_{s,t,k} \le 1 \quad \forall v \in V \setminus S \setminus \{s,t\} \tag{13}$$

$$y_{s,t,k} \ge 0, \qquad z_{s,t,k} \ge 0$$

$$w_{s,t,k}, \;\; b_{s,t,k}, \;\; \psi_{v,k}, \;\; \eta_{v,k} \quad \text{binary variables.}$$

In the problem above, the objective function is to minimize the cost of routing the traffic demand. The constraints (8) - (9) ensure that the demand is routed through the specified nodes and only one working path is selected. Constraints (10) - (11) route the demand on the disjoint backup path along a single path. Constraint (12) relates the working and backup traffic routed on a link to the total traffic on a link. The set of constraints given by (13) ensure the paths selected are node disjoint. Lastly there are constraints to ensure the decision variables are nonnegative and binary as appropriate. Note that preprocessing for the optimization problem consists of using a k-shortest path algorithm to determine $\mathcal{P}_{s,t}$ then processing the paths into the sets $\mathcal{P}_{st}^{S}$ and $\mathcal{P}_{st}^{\bar{S}}$ depending on the set S. Typically, given the preprocessing, the path formulations of ILP problems are more computationally efficient than the arc formulations. However the computation time grows quickly with the path set size, thereby limiting the size of network that can be studied. One can also try to scale the model by solving the optimization problem over a reduced search space by limiting the size of $\mathcal{P}_{st}^{S}$ and $\mathcal{P}_{st}^{\bar{S}}$, but the results will be not be guaranteed to be optimal. For example one can simply restrict the size of the two sets to a maximum size L or limit them based on network structural properties such as a hop count limit.

Here we provide some illustrative numerical results showing how the accuracy and computation time changes with a hop count limit on the path sets. We consider the Polska network topology from SNDlib, which consists of $|V| = 12$ nodes and $|A| = 18$ edges. We conducted numerical experiments by randomly selecting a set of specific nodes S and then considering every (s,t) pair not containing nodes in S and solving the optimization problem above using Matlab. This was repeated for 30 random selections of S and the results averaged over all cases. The demand $d_{s,t}$ and link cost c_e were both set to 1. Fig. 1 shows the percent of feasible solutions found as the hop count limit for $\mathcal{P}_{st}^{S}$ and $\mathcal{P}_{st}^{\bar{S}}$ increases for different sizes of $|S|$. Note that for $|S| = 1$ at a hop count of 8 all the solutions can be found. However as the number of specified nodes S increases the hop count limit must increase to find all of the feasible solutions. In fact for this network, the hop count limit must be the maximum of 11 to find all of the feasible solutions for $|S| = 2, 3$. The computational time is largely determined by the hop count limit. Fig. 1 shows the computational time of solution of the optimization problem (k-shortest path computation is not included) for $|S| = 1$ on a MacBook Pro 2.6GHz Intel Core I7. The effect of increasing the size of S is to slightly reduce the computational time as the number of infeasible solutions increases with $|S|$. For example with the maximum hop count of 11, the computational times corresponding to $|S| = 1, 2, 3$ are $3.592, 3.534, 3.261$ seconds respectively. Note the attempts to solve the optimization problem trading off optimality by using the hop count limit to control the computation time for large networks were unsuccessful as the sizes of $\mathcal{P}_{st}^{S}$ and $\mathcal{P}_{st}^{\bar{S}}$ increase rapidly with the size of the network. For example, we were unable to get results for the Italia network from [27] for hop counts greater than 8 within 24 hours. Clearly, this approach can not be applied for large networks.

In the remainder of the paper we use the formulations of the problems $\mathcal{P}_1$ and $\mathcal{P}_2$ above given in [14] to evaluate the performance of the heuristics proposed here. For problem $\mathcal{P}_0$, as in [14], we used the $Q3$ formulation [8] to evaluate the accuracy of the heuristics.

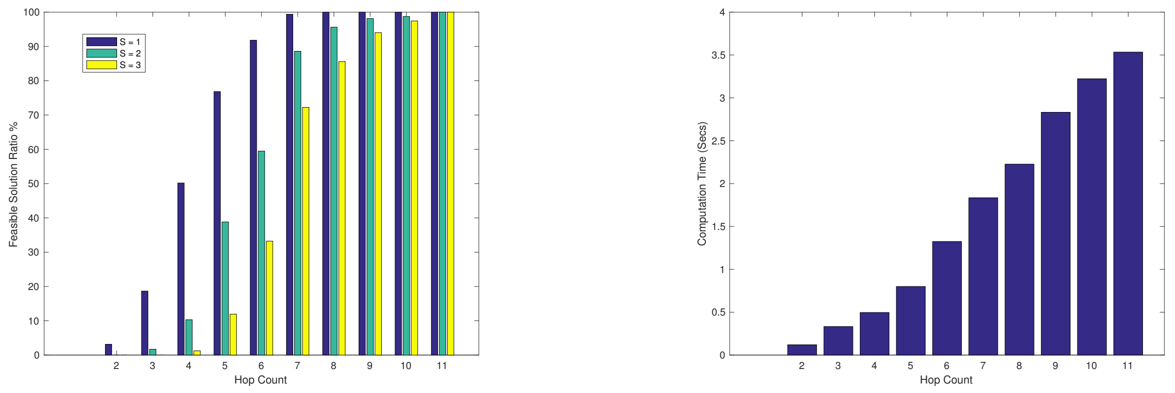


Figure 1: Polska network results

3 Heuristic for the computation of a Path with Specified Nodes, with or without protection

The main idea of the heuristic is to recursively construct the path p', which starts with a segment which is the minimal cost path among the elements of P_p (set of paths between the nodes in S or between these nodes and s and t). Then the heuristic builds the rest of the path by successive concatenation of shortest paths in P'_p (initially a copy of P_p), such that, in each iteration, the new added segment is the one with the lowest cost among those that can be concatenated with the current segment of p'. This approach may fail because the path under construction can not lead to a valid solution. In this case the algorithm *backtracks*, removing the last added segment to the path under construction. This segment will be forbidden, from this point onwards, for the rest of the construction of the path. As this strategy does not ensure a good solution, several elements in P_p are tried out (up to a chosen value *Upperbound*) to be the starting segment of p', as can be seen in Algorithm 1. Notice that Algorithm 1 makes extensive use of a shortest path calculation denoted *shortestPath* and we utilize Dijkstra's algorithm for the calculation. In function *shortestPath* the first two arguments are the source node and target node, and the third argument is the set of nodes that induces the subgraph of G where a shortest path is calculated.

Algorithm 1 (PSN) first calculates P_p. Specifically, the first for loop in Algorithm 1 (lines 3-7) determines a shortest paths between all node pairs in S. The second outer for loop (lines 8-13) finds shortest paths from s and t to the nodes in S. These paths are added to the set of shortest routes between nodes in S to form the set P_p. The following while loop in Algorithm 1 (lines 18-30) determines the set of all valid paths P_{st}, by selecting the starting segment of p', and initializes the input parameters of the recursive function *Pcompute* defined by Algorithm 2. Note that the calculated path p', obtained in line 25, is the concatenation of the initial chosen segment with the output of the recursive function *Pcompute*. The algorithm ends (line 32) by selecting the lowest cost path in set P_{st}.

As already mentioned, Algorithm 2 (function *Pcompute*), builds the path from source node s to destination node t, by successive concatenation of segments, or ends with an incomplete path which can be the empty path. Let $p'(r)$ represent a segment of the final path obtained through the concatenation of r segments; V'_S is the set of mandatory nodes not in $p'(r)$, including s and t nodes; FP is the set containing sets $FP(r)$ of segments that are forbidden to be concatenated with $p'(r)$; P'_p is the set containing sets $P'_p(r)$ of candidate segments that may be concatenated with $p'(r)$.

The stopping conditions of Algorithm 2 are in lines 2-3: all the specified nodes are in the path or it is no longer possible to find a valid path. Then the algorithm enters in a cycle (lines 4-14) to determine the next segment to be concatenated with $p'(r)$. The selected segment in each iteration, the one of minimum cost among possible candidates (see line 13) is evaluated by functions *existsCycle* and *goodSeg*. These functions are described by

Algorithm 1: Heuristic for the computation of a Path with Specified Nodes (PSN) or a Protected Path with Specified Nodes (PPSN)

```
    Data: $G = (V, A)$, $s$, $t$, $S \subset V$.
    Result: $p^*_{st}$, lowest cost path in $P_{st}$ visiting $S$.
1   begin
2       $P_{st} \leftarrow \emptyset$, $P_s \leftarrow \emptyset$, $counter \leftarrow 0$, $p^*_{st} \leftarrow \emptyset$
3       for $v_i \in S$ do
4           $P_{v_i} \leftarrow \emptyset$
5           for $v_j \in S \wedge v_j \neq v_i$ do
6               $p_{v_i v_j} \leftarrow$
                $\text{shortestPath}(v_i, v_j, V \setminus S \setminus \{s, t\} \cup \{v_i, v_j\})$
7               $P_{v_i} \leftarrow P_{v_i} \cup \{p_{v_i v_j}\}$
8       for $v_i \in S$ do
9           if $P_{v_i} = \emptyset$ then return
            $p_{s v_i} \leftarrow \text{shortestPath}(s, v_i, V \setminus S \setminus \{t\} \cup \{v_i\})$
10          $p_{v_i t} \leftarrow \text{shortestPath}(v_i, t, V \setminus S \setminus \{s\} \cup \{v_i\})$
11          $P_{v_i} \leftarrow P_{v_i} \cup \{p_{v_i t}\}$
12          $P_s \leftarrow P_s \cup \{p_{s v_i}\}$
13      if $P_s = \emptyset \vee \forall v_i \in S \ \nexists p_{v_i t} \in P_{v_i}$ then
14          return
15      $P_p \leftarrow \cup_{v_i \in S \cup \{s\}} P_{v_i}$
16      $P'_p(1) \leftarrow P_p$
17      while $P_p \neq \emptyset \vee counter <$ UpperBound do
18          $p_{v_i v_j} \leftarrow$
            $\arg\min_{p_{v_i v_j} \in P_p} \sum_{(v_m, v_n) \in p_{v_i v_j}} w(v_m, v_n)$
19          $p'(1) \leftarrow p_{v_i v_j}$
20          $P_p \leftarrow P_p \setminus \{p_{v_i v_j}\}$
21          $V'_S \leftarrow S \cup \{s, t\} \setminus \{v_i, v_j\}$
22          $P'_p(1) \leftarrow P'_p(1) \setminus \{p_{v_i v_j}\}$
23          $FP\,(r) \leftarrow \emptyset, \quad r = 1, 2, \ldots, |S|$
24          $P'_p\,(r) \leftarrow \emptyset, \quad r = 2, \ldots, |S|$
25          $p' \leftarrow p_{v_i v_j} \odiamond \text{Pcompute}(p'(1), s, t, V'_S, P'_p, FP)$
26          $P'_p(1) \leftarrow P'_p(1) \cup \{p_{v_i v_j}\}$
27          if $p'$ is a valid path from $s$ to $t$ then
28              $P_{st} \leftarrow P_{st} \cup \{p'\}$
29          $counter \leftarrow counter + 1$
30      if $P_{st} \neq \emptyset$ then
31          $p^*_{st} \leftarrow \arg\min_{p_{st} \in P_{st}} \sum_{(v_m, v_n) \in p_{st}} w(v_m, v_n)$
32      return
```

Algorithm 2: $\text{Pcompute}\big(p'(r), s, t, V'_S, P'_p(r), FP\big)$

```
    Data: $G = (V, A)$; $s$ and $t$; $p'(r)$ which is the
          concatenation of $r$ segments; $V'_S$; $FP$, set of
          sets $FP(r)$ of forbidden concatenation
          segments for $p'(r)$; $P'_p$, set of sets $P'_p(r)$ of
          candidate concatenation segments of $p'(r)$.
    Result: The concatenation of each segment of $p'$
            with a first selected segment until $p'$ is: a
            loopless path starting at $s$, passing through
            all nodes in $S$, and ending at $t$ with a
            disjoint protection path (if required); or an
            incomplete path.
1   begin
2       if $V'_S = \emptyset$ then return $\emptyset$ $p_{v_l v_k} \leftarrow p'(r)$
3       if $\nexists\ p_{v_i v_l} \vee p_{v_k v_j} \in P'_p(1) \setminus FP(1)$ then return
        $\emptyset$ $cycles \leftarrow true$
4       while $cycles$        /* search feasible segment */
5       do
6           if $v_l \neq s \wedge v_k \neq t$ then
7               $P''_p \leftarrow \big\{p_{v_i v_j} \in P'_p(r) \setminus FP(r) :$
8                   $(v_i = v_k \wedge v_j \in V'_S) \vee (v_l = v_j \wedge v_i \in V'_S)\big\}$
9           if $v_l = s$ then
10              $P''_p \leftarrow \big\{p_{v_k v_j} \in P'_p(r) \setminus FP(r) : v_j \in V'_S\big\}$
11          if $v_k = t$ then
12              $P''_p \leftarrow \big\{p_{v_i v_l} \in P'_p(r) \setminus FP(r) : v_i \in V'_S\big\}$
13          $p_{v_i v_j} \leftarrow \arg\min_{P''_p} \sum_{(v_m, v_n) \in p_{v_i v_j}} w(v_m, v_n)$
14          if $p_{v_i v_j} = \emptyset \vee$
            $\big(\neg\text{existCycle}(p_{v_i v_j}, p'(r), s, t, V'_S, P'_p, FP) \wedge$
15          $\text{goodSeg}(p_{v_i v_j}, p'(r), s, t, V'_S, P'_p, FP)\big)$ then
16              $cycles \leftarrow false$
17      if $p_{v_i v_j} = \emptyset \wedge r = 1$ then return $\emptyset$ if $p_{v_i v_j} = \emptyset$
        /* backtracking */
18      then
19          $p_{v_l v_k} \leftarrow p'(r) \setminus p'(r-1)$
20          $FP(r-1) \leftarrow FP(r-1) \cup \{p_{v_l v_k}\}$
21          $FP(r) \leftarrow \emptyset$
22          if $v_l \in V_{p'(r-1)}$ then $V'_S \leftarrow V'_S \cup \{v_k\}$ else
            $V'_S \leftarrow V'_S \cup \{v_l\}$ return
            $p_{v_l v_k} \not\odiamond \text{Pcompute}\,\big(p'(r-1), s, t, V'_S, P'_p, FP\big)$
23      else
24          if $v_i \in V'_S$ then $V'_S \leftarrow V'_S \setminus \{v_i\}$ else
            $V'_S \leftarrow V'_S \setminus \{v_j\}$ $P'_p(r+1) \leftarrow P'_p(r) \setminus \{p_{v_i v_j}\}$
25          $p'(r+1) \leftarrow p_{v_i v_j} \odiamond p'(r)$     /* add segment */
26          return
            $p_{v_i v_j} \odiamond \text{Pcompute}\,\big(p'(r+1), s, t, V'_S, P'_p, FP\big)$
```

Algorithms 3 and 4, respectively.

If the while cycle ends with an empty path, *i.e.*, no segment was found to concatenate with $p'(r)$, Algorithm 2 *backtracks* – see lines 19-22 – removing the last added

Algorithm 3: existCycle $\left(p_{v_i v_j}, p'(r), s, t, V'_S, P'_p, FP\right)$

```
Data: $G = (V, A)$; $s$ and $t$; $p'(r)$, path with $r$ segments; $V'_S$; $P'_p$, set of sets $P'_p(r)$; $FP$, set of sets $FP(r)$; $p_{v_i v_j}$, candidate segment under evaluation.
Result: False, if the new segment $p_{v_i v_j}$ is node-disjoint with $p'(r)$, otherwise is true. In the latter case, if a new segment from $v_i$ to $v_j$ can be computed then $P'_p(r)$ is updated, otherwise $p_{v_i v_j}$ becomes forbidden (both $P'_p(r)$ and $FP(r)$ are updated).
1 begin
2     if $\exists v_k \neq v_i, v_j : v_k \in p_{v_i v_j} \wedge v_k \in p'(r)$ then
3         $p'_{v_i v_j} \leftarrow$
          shortestPath$(v_i, v_j, V \setminus V'_S \setminus V_{p'(r)} \cup \{v_i, vj\})$
4         $P'_p(r) \leftarrow P'_p(r) \setminus \{p_{v_i v_j}\}$
5         if $p'_{v_i v_i} = \emptyset$ then $FP(r) \leftarrow FP(r) \cup \{p_{v_i v_j}\}$
          else $P'_p(r) \leftarrow P'_p(r) \cup \{p'_{v_i v_j}\}$ return true
          /* there was a cycle */
6     else return false                              /* no cycle */
```

Algorithm 4: goodSeg $\left(p_{v_i v_j}, p'(r), s, t, V'_S, P'_p, FP\right)$

```
Data: $G = (V, A)$; $s$ and $t$; $p'(r)$, path with $r$ segments; $V'_S$, set of mandatory nodes to be included; $P'_p$, set of sets $P'_p(r)$; $FP$, set of sets $FP(r)$; $p_{v_i v_j}$, to be evaluated
Result: True, if the new segment $p_{v_i v_j}$ concatenated with $p'(r)$ may possibly lead to a solution, otherwise is false and $P'_p(r)$, $FP(r)$ are updated.
1  begin
2      $p_{v_l v_k} \leftarrow p_{v_i v_j} \circledcirc p'(r)$
3      if $(v_l = s \wedge v_k = t \wedge |V'_S| - 1 \neq 0)$
4      $\vee(v_l \neq s\wedge$
5      shortestPath$(s, v_l, V \setminus V'_S \setminus V_{p_{v_l v_k}} \cup \{v_l, s\}) = \emptyset)$
6      $\vee$ $(v_k \neq t\wedge$
7      shortestPath$(v_k, t, V \setminus V'_S \setminus V_{p_{v_l v_k}} \cup \{v_k, t\}) = \emptyset)$
8      $\vee$ (path requires protection$\wedge$
9      shortestPath$(s, t, V \setminus V'_S \setminus V_{p_{v_l v_k}} \cup \{s, t\}) = \emptyset)$
10     then
11         $P'_p(r) \leftarrow P'_p(r) \setminus \{p_{v_i v_j}\}$
12         $FP(r) \leftarrow FP(r) \cup \{p_{v_i v_j}\}$
13         return false                              /* not good */
14     else
15         return true                          /* possibly good */
```

segment from the solution being built, not before adding the segment to the set $FP(r-1)$ of forbidden segments for $p'(r-1)$. Otherwise the obtained segment is concatenated with $p'(r)$, creating $p'(r+1)$ and the relevant sets are updated.

In Algorithm 3 (function *existCycle*), given a candidate path segment $p_{v_i v_j}$ to be concatenated with $p'(r)$, the routine returns false if the resulting path does not contains a cycle; otherwise, if a new segment from v_i to v_j, node-disjoint with $p'(r)$, is successfully calculated then it replaces the previous segment in $P'_p(r)$; if no such path could be obtained, the segment from v_i to v_j becomes a forbidden segment (is moved from $P'_p(r)$ to $FP(r)$).

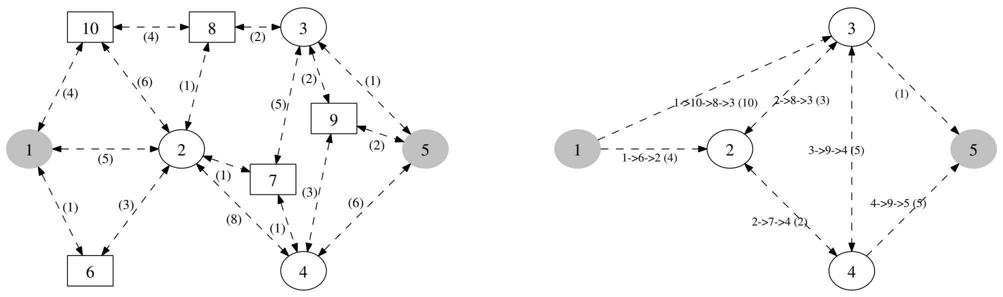
Algorithm 4 (function *goodSeg*) evaluates if the concatenation of $p_{v_i v_j}$ with $p'(r)$ will prevent a path from s to t to be obtained. If that is the case $p_{v_i v_j}$ is moved from P'_p to $FP(r)$. Furthermore, if $\mathcal{P}_1$ is the problem being solved, this function also evaluates if the segment resulting from the concatenation of $p_{v_i v_j}$ and $p'(r)$ allows one to obtain a path from s to t which is node-disjoint with the path being built. The inclusion of this additional test converts algorithm PSN into the algorithm that calculates Protected Path with Specified Nodes (PPSN). In this case, at the end of the algorithm the backup path can be calculated as a shortest path node-disjoint with the path constructed by PPSN.

If the objective is to solve problem $\mathcal{P}_2$ then two actions are required:

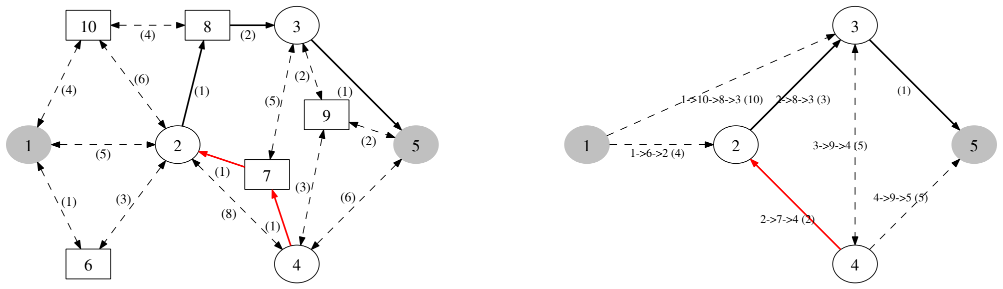
- the active path is calculated in the network where the nodes in set B have previously been removed;
- a shortest path calculation in line 9 of Algorithm 4 (function *goodSeg*) is replaced by PSN, in a network where the nodes in set B have been added back and the nodes in set $|S|$ have been removed.

This allows one to verify if a node-disjoint path (visiting the second set of nodes) with the one under construction exists, resulting in the algorithm MSPSN.

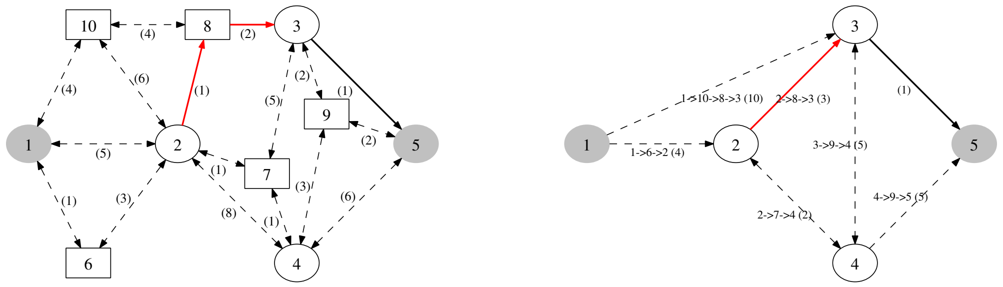
In Fig. 2 an illustrative example of Algorithm PSN is presented, with $s = 0$, $t = 5$ and $S = \{2, 3, 4\}$. On the left and right sides of each figure are the original graph and the subgraph, respectively. The arcs selected to be in the path are chosen in the subgraph and marked by a full line in both graphs. In Fig. 2(a) each arc of the subgraph corresponds to the paths in P_p (see line 15 of Algorithm 1). Then, according to line 18 of Algorithm



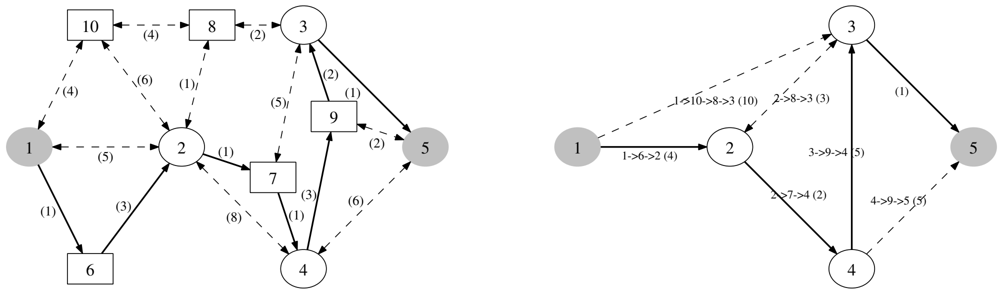
(a) Original network and subgraph



(b) Third candidate segment rejected



(c) Backtracking required



(d) Final segment is concatenated

Figure 2: Illustrative path construction example

1), the first selected arc in the subgraph is the one of minimum cost, arc $(3,5)$ which corresponds to arc (3,5) in the original network. Then function *Pcompute* is called to build the remaining path. This function looks for the next minimum cost arc that can be concatenated (at left or right) with the existing sub-path. The second selected arc in the subgraph is (2,3) concatenated to candidate path – arc (2,3) corresponds to sub-path $\langle 2,8,3\rangle$ in the original network.

The third candidate arc to be considered is (4,2) – see Fig. 2(b). However from node 4 it is not possible to reach node 1 (the source); this problem is detected by function *goodSeg*) – see line 5 of Algorithm 4. Therefore arc (4,2) in the subgraph can not be added to the solution under construction and must be discarded. As there is no alternative to continue building the path, the algorithm needs to backtrack.

At line 19 of Algorithm 2, the last added segment (path $p_{v_l v_k} = \langle 2,8,3\rangle$), represented by arc (2,3) in the subgraph, is identified for removal from the candidate path – see Fig. 2(c). After updating the relevant sets, the segment $p_{v_l v_k}$ is removed from the present solution, using backtracking, at line 22 of Algorithm 2.

Next, the selected candidate arc is (4,3) which is concatenated to the present solution; this is followed by arc (2,4), also concatenated to the path by function *Pcompute*. Then *Pcompute* is called once more to link the source node to the candidate path, obtaining the solution shown in Fig. 2(d), which in this case is the optimal solution $\langle 1,6,2,7,4,9,3,5\rangle$ of cost 12. In [17] an additional example can be found, where cycles are detected and solved.

3.1 Algorithms obtained using the trap avoidance approach

The trap avoidance approach proposed in [14], which combined with algorithm VSN resulted in algorithms VTA and MSPTA for solving problems $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively, was also used in the present work.

In the case of problem $\mathcal{P}_1$, once a candidate active path is calculated, the nodes in S are removed but the arcs incident on intermediate nodes (not in S) of the candidate active path have their cost increased by a sufficient large amount before the backup path is calculated (using Dijkstra's algorithm [9] in this modified graph). If the active and backup path share a node (the conflicting node), then an arc incident to the shared node is selected for removal before attempting again to obtain a candidate active path (for more details see [14]). This process is repeated until no active path can be obtained or a node-disjoint path is calculated. This algorithm will be designated PSNTA and is detailed in Algorithm 5.

If the problem to be solved is $\mathcal{P}_2$, the process is similar, but the nodes in set B are removed before calculating the active path and the backup path is calculated using algorithm PSN with B being the set of nodes to visit in the modified network. This variant will be designated MSPTA. The corresponding algorithm can be obtained modifying Algorithm 5 as follows: remove nodes B from G in line 7 and replace a shortest path calculation by PSN, with specified node set B, in line 15.

Note that the introduction of the trap avoidance approach requires also the modification of Algorithm 4, where lines 8-9 need to be eliminated.

4 Results

In this work we will compare VSN, VTA and MSVTA (proposed in [14]) with PSN, PPSN/PSNTA and MSPSN/MSPTA respectively.

Five networks (newyork, norway, india35, pioro40 and germany50) from the SNDlib [19] repository, were used to evaluate the heuristics; the cost of each edge was the first module cost as given in SNDlib. An additional set of networks, with 500 nodes, arc cost between 1 and 100 and with an average degree of around 7 (sum of the in and out degrees), was generated with the Doar-Leslie model [10] using Georgia Tech Internetwork Topology Models software (GT-ITM) [2] (http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html). If the generated networks contained spurs, they were removed before solving problems $\mathcal{P}_0$, $\mathcal{P}_1$ and $\mathcal{P}_2$ (this resulted in removing between 4 and 8 nodes), thus ensuring the networks studied were biconnected. Although these networks now have a number of nodes between 492 and 496, they will be referred in the text as the 500 node networks. To evaluate the performance of the proposed heuristics in physical networks, like the ones considered in [3], two more networks were considered: the CORONET CONUS network with 75 nodes and 99 edges (`http://www.monarchna.com/topology.html`) and the TeliaSonera network [25] with 21 nodes and 25 edges; the cost of each edge was set to the distance in km (rounded to an integer) between the GPS coordinates of end nodes The CORONET CONUS will be simply designated CORONET here. In Table 1 are the main characteristics of the used networks, where the average node betweenness is unweighted.

Algorithm 5: Heuristic for the computation of a protected Path with Specified Nodes using Trap Avoidance based approach (PSNTA)

```
Data: $G = (V, A)$, $s$, $t$, $S \subset V$.
Result: $p^*_{st}$, lowest cost path in $P_{st}$ visiting $S$, that
        can be protected by a node-disjoint path
        ($b_{st}$).
1  begin
2      $FA \leftarrow \emptyset$                         /* forbidden arcs */
3      $try \leftarrow true$
4      $p^*_{st} \leftarrow \emptyset$                   /* initially no solution */
5      $b_{st} \leftarrow \emptyset$
6      while $try = true$ do
7          $G' \leftarrow G(V, A \setminus FA)$          /* subgraph */
8          $p \leftarrow PSN(G', s, t, S)$
9          if $p = \emptyset$ then
10             $try \leftarrow false$                    /* failed */
11         else
               /* Trap avoidance procedure */
12             $G' \leftarrow G$                         /* saves $G$ */
13             Removes from $G$ arcs of $p$ incident in $S$
14             Increases cost of all arcs (in $G$) incident in
               $v \in V_p \setminus S \setminus \{s, t\}$
15             $b \leftarrow$ shortest path from $s$ to $t$ in $G$
16             if $V_p \cap V_b = \{s, t\}$ then
                   /* $p$ and $b$ are node-disjoint */
17                 $p^*_{st} \leftarrow p$               /* feasible solution */
18                 $b_{st} \leftarrow b$                 /* backup path */
19                 $try \leftarrow false$                /* to end cycle */
20             else
21                 $a \leftarrow$ arc incident in conflicting node
22                 $FA \leftarrow FA \cup a$
23             $G \leftarrow G'$                         /* Restores $G$ */
24     return $(p^*_{st}, b_{st})$
```

The number of specified nodes was considered to be equal to 2, 4, 8, 10 and 20, corresponding to a small, medium and large size of S. The elements in each set S were randomly generated, considering 20 different seeds. For each set S of given nodes, 100 node pairs were randomly generated for each network. However, for newyork and norway, only $|S| = 2, 4$ was considered, due the smaller number of nodes in these networks; also for india35, pioro40, germany50 and CORONET the maximum value of $|S|$ was 10, for TeliaSonera was 8, and only for the 500 node networks was the value $|S| = 20$ considered. Note that since the nodes in S are randomly generated, if $|S| + |\{s, t\}|$ is a significant percentage of the total number of nodes, many of the problems will have no solution; moreover if the average node degree is low, for many randomly generated problems, no solutions can be found. The seeds were selected (by trial with the ILP solver) to ensure that solutions exist for a large number of the 100 randomly generated node pairs, namely in the case of the SNDlib networks. Although in optical networks the size of S will in general be smaller in other contexts (for example wireless sensor networks) the number of given nodes may be larger.

Name	$\vert V\vert$	$\vert A\vert$	Average node degree	diam.	Average node betweenness
newyork	16	49	6.13	3	5.38
norway	27	51	3.78	7	27.70
india35	35	80	4.57	7	33.03
pioro40	40	89	4.45	7	45.13
germany50	50	88	3.52	9	74.68
500_0	492	1760	7.15	6	579.47
500_1	496	1675	6.75	6	606.79
500_2	494	1667	6.75	6	603.19
500_3	494	1703	6.89	6	598.36
500_4	492	1791	7.28	6	573.54
CORONET	75	99	2.64	17	201.81
TeliaSonera	21	25	2.38	9	30.57

Table 1: Topological characteristics of the networks used

Twenty samples were obtained for each network, and 95% confidence intervals around the estimated mean were calculated, appearing in the graphs as error bars.

The computational platform was a Desktop with 16 GB of RAM and an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz processor, with Kubuntu 14.04 and the CPLEX solver, version 12.6 [32]. In order to obtain solutions in a reasonable time, a limit of 5 minutes per node pair for the CPLEX solver was established. In PSN and PPSN, the recursive function *PCompute* was implemented in iterative form, and the backtracking was limited, depending on $|S|$ and network size, seeking to attain compromise between the resolution ratio and accuracy of the solutions.

Results will be presented sequentially for Problems $\mathcal{P}_0$, $\mathcal{P}_1$, $\mathcal{P}_2$, for the SNDlib networks followed by the 500 node networks in Sections 4.1, 4.2 and 4.3, respectively. Then in Section 4.4, results obtained with PSN for CORONET and TeliaSonera networks are presented, to illustrate the potential of the proposed heuristic to solve problem $\mathcal{P}_0$.

4.1 Results regarding solving Problem $\mathcal{P}_0$

The results obtained when solving Problem $\mathcal{P}_0$ are shown in Fig. 3 and Fig. 4 for the networks from SNDlib [19] and for the 500 node networks, respectively.

The number of feasible solutions found by solving problem $\mathcal{P}_0$ for the five SNDlib networks is very close to 100% for PSN – see Fig. 3(a). Algorithm VSN has a slightly lower performance than PSN, which is more visible for germany50 network.

Regarding the accuracy of the obtained solutions, the relative error (with respect to the solution found by the ILP solver) is shown in Fig. 3(b), where VSN has less than 3% average relative error for $|S| = 2, 4, 8, 10$; PSN for $|S| = 2$ has consistently the smallest relative error, however its error grows with $|S|$ and is usually larger than the relative error of VSN for $|S| = 4, 8, 10$; for PSN the highest relative error of the path cost, on average, is around 6% for the germany50 network, when $|S| = 10$.

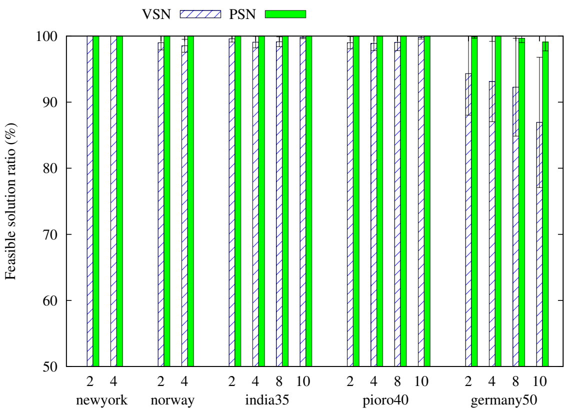
The CPU time of both heuristics is similar for $|S| = 2, 4$; for $|S| = 8$ the CPU time of PSN is slightly lower than the CPU time of VSN, and both are much smaller than the CPU time of CPLEX, as can be seen in Fig. 3(c). The advantage of the new approach (PSN), regarding CPU time, is clear for $|S| = 10$ where the CPU of PSN is at least one order of magnitude smaller than the CPU time required by CPLEX, while VSN for the germany50 network requires only slightly less CPU time than the solver – see Fig. 3(c).

For all randomly generated S and node pairs, the percentage of feasible solutions is 100% for the 500 networks for PSN and was very close to 100% for VSN (only 8, of the 10^4 node pairs of tested cases, were not solved by the VSN heuristic), hence the corresponding figure is omitted.

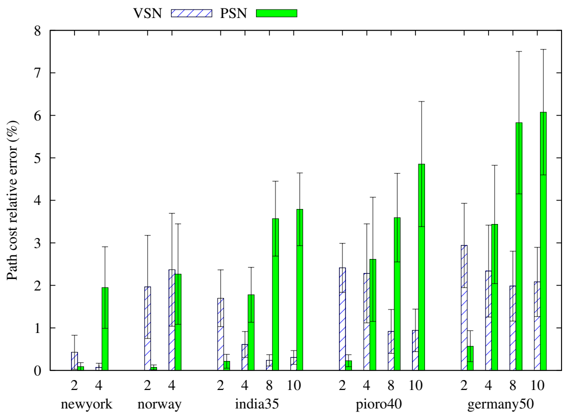
The relative error of the solutions found by VSN is (in general) below 5% for $|S| = 2, 4, 8, 10$ – see Fig. 4(a). In the case of PSN and for $|S| = 2$ the error is very small (and much smaller than the one observed for VSN); PSN and VSN present similar errors (below 3%) for $S = 4$; the PSN error is below 6%, 7% and 10%, for $|S| = 8$, $|S| = 10$ and $|S| = 20$ respectively.

For $|S| = 20$ no results are shown for VSN in Fig. 4(a) and Fig. 4(b) because, as can be seen in Fig. 4(b) for $|S| = 10$, the CPU time of VSN is on average larger than CPLEX.

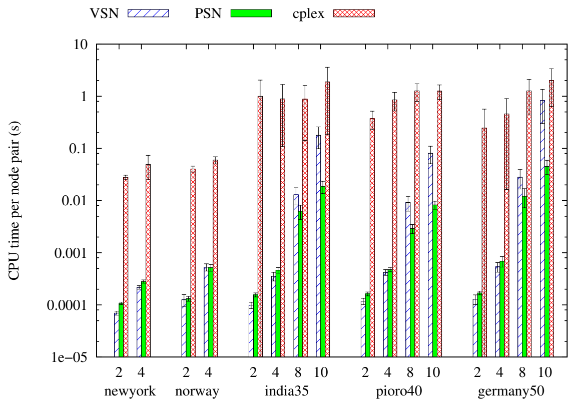
Regarding the CPU time both heuristics perform quite well with respect to the CPLEX solver, being at least two orders of magnitude faster for $|S| = 2, 4$ For $|S| = 8$ VSN starts to present a CPU time only slightly lower than CPLEX, and for $|S| = 10$ it uses too much CPU



(a) Feasible solutions ratio with respect to CPLEX



(b) Relative error of the feasible solutions found by the heuristics



(c) CPU time per node pair

Figure 3: Loopless path with specified nodes, results for five SNDlib [19] networks

time. In contrast, algorithm PSN requires less than a 0.5 seconds of CPU time in average ($|S| = 20$), while CPLEX may require an average of tens of seconds. Note that, in some of the few instances where CPLEX ended without a solution, due to the imposed CPU time limit of 5 minutes, PSN was able to find a solution.

In conclusion, for the SNDlib networks VSN solves slightly fewer problems than PSN, but presents solutions with smaller relative error for $|S| = 4, 8, 10$; however for $|S| = 10$ it can use a CPU time very close to CPLEX. In the case of the 500 node networks, PSN has the best feasible solution ratio, requires less CPU time than VSN (and CPLEX), but presents a larger relative error for $|S| = 8, 10$. Moreover PSN is still effective when $|S|$ is equal to 20, when VSN is no longer a viable approach.

4.2 Results regarding solving Problem $\mathcal{P}_1$

The results obtained solving Problem $\mathcal{P}_1$ for the networks from SNDlib [19] are shown in Fig. 5 and for the 500 node networks results are in Fig. 6.

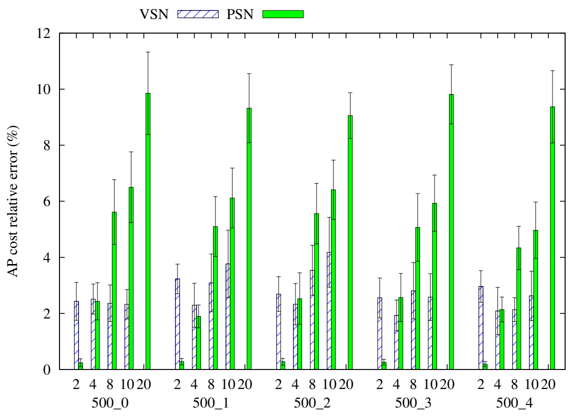
In Fig. 5(a), for $|S| = 2, 4$ the heuristics present in most cases similar results (the confidence intervals overlap) regarding the feasible solution ratio. However, the variant resulting from combining PSN with the trap avoidance approach (PSNTA) is the one with best performance (more feasible solutions), especially in the case of the germany50 network and for $|S| = 10$.

The relative error of the feasible solutions found by PSNTA and PPSN is below 1.1% for $|S| = 2$, and is on average below 7% for all heuristics. In Fig. 5(b), it can be observed that PSNTA has consistently the largest relative error among the three heuristics for $|S| = 4, 8, 10$, although the confidence intervals of PSNTA and PPSN usually overlap.

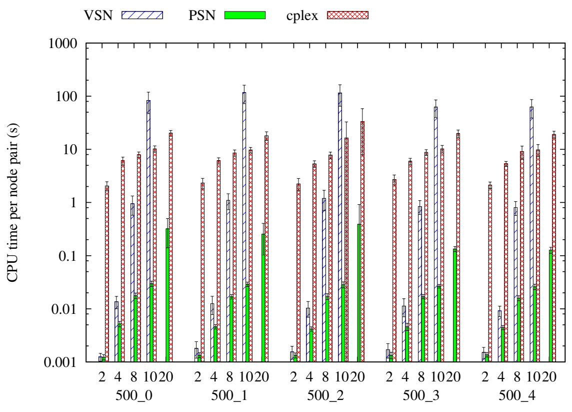
Regarding CPU time, Fig. 5(c) shows that PPSN presents smaller CPU times than PSNTA and VTA, especially for $|S| = 8, 10$ and for the pioro40 and germany50 networks.

The results for the 500 node networks created using the GT-ITM software are given in Fig. 6. As in the case of Problem $\mathcal{P}_0$, the feasible resolution ratio is very close to 100% for all heuristics, and once again no figure is presented.

The relative error of the feasible solutions is below 4% for VTA and $|S| = 4, 8, 10$. For PPSN and PSNTA the relative error of the cost of the active path (AP) of the feasible solutions increases with the number of specified nodes to visit, as can be seen in Fig. 6(a). Note that for $|S| = 2$, PPSN and PSNTA present a very small error,



(a) Relative error of the feasible solutions found by the heuristics

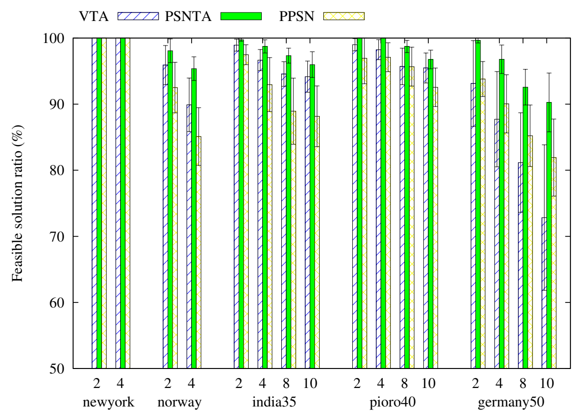


(b) CPU time per node pair

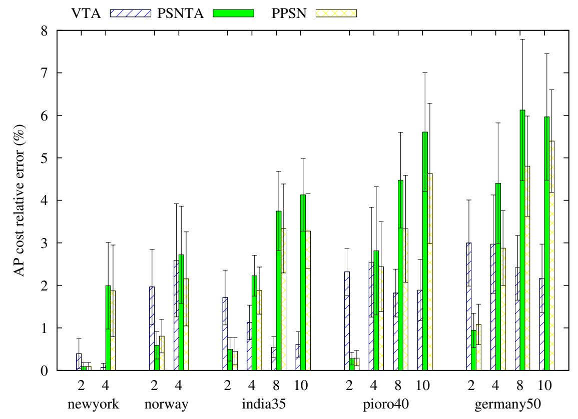
Figure 4: Loopless path with specified nodes, results for five 500 node networks

while VTA presents an average relative error of around 3%. However, for $|S| = 4, 8, 10, 20$, the relative error is below 3%, 6%, 7% and 10%, respectively, for both PPSN and PSNTA. This increase of the relative error of PPSN and PSNTA with $|S|$, while VTA experiences little fluctuation, is consistent with the behavior of the underlaying heuristics, PSN (for PPSN and PSNTA) and VSN for VTA one.

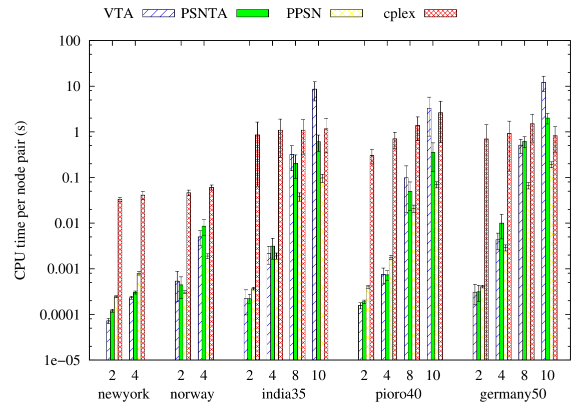
The CPU time per node pair required by CPLEX in the 500 node networks is on average around 10 seconds for $|S| = 20$ – see Fig. 6(b). Note that VTA, for $|S| = 8$ it requires a CPU time close to the CPU time for CPLEX and for $|S| = 10$ requires significantly more CPU time



(a) Feasible solutions ratio with respect to CPLEX

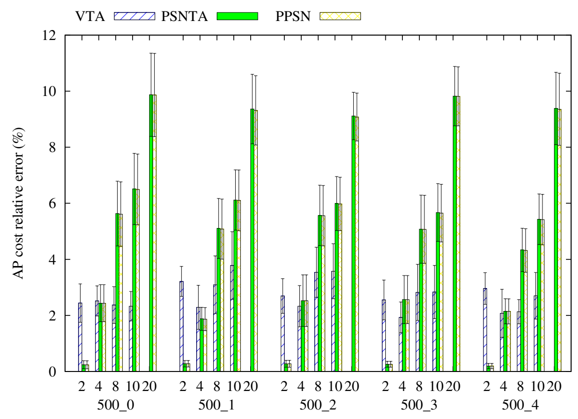


(b) Relative error of the feasible solutions found by the heuristics

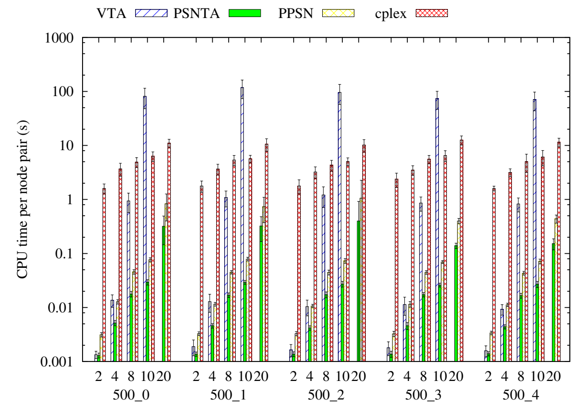


(c) CPU time per node pair

Figure 5: Protected loopless path with specified nodes, results for five SNDlib [19] networks



(a) Relative error of the feasible solutions found by the heuristics



(b) CPU time per node pair

Figure 6: Protected loopless path with specified nodes, results for five 500 node networks

than CPLEX. This results from the fact these networks have both more nodes and a larger average node degree than the SNDlib networks, which makes it possible to have much more attempts to recalculate paths with $|S|+1$ arcs in the auxiliary sub-graphs with $|S|+2$ nodes.

For $|S| = 2, 4$ all heuristics have small CPU times, with a slight advantage to PSNTA. For $|S| = 8, 10$ the CPU time of PSNTA and PPSN is below 0.1 while the CPU time of CPLEX is significantly larger (around 5 seconds). For $|S| = 20$ the CPU time of PSNTA and PPSN is below 1 second while CPLEX requires a little over 10 seconds, per node pair.

In summary, PSNTA requires less CPU time for the same error and resolution ratio as PPSN, particularly for larger values of $|S|$; VTA can be the best choice for $S = 4$ because its solutions have smaller relative error than PSNTA or PPSN; for large networks and $|S| \geq 8$, VTA should be avoided because it uses too much CPU time.

4.3 Results regarding solving Problem $\mathcal{P}_2$

Results illustrating the relative behavior of the heuristics proposed for solving problem $\mathcal{P}_2$ can be found in Figs. 7–8.

In the case of the networks from the SNDlib [19], MSPTA is the heuristic with highest feasible solution ratio – see Fig. 7(a). For $|S| = 2$ it presents a percentage of feasible solutions larger than 90%; for $|S| = 4$ the ratio of feasible solutions is between 78.5% (pioro40) and 96.4% (newyork).

Regarding the accuracy of the obtained solutions, the error is on average below 6% for $S = 2$ and is less than 4% for $|S| = 4$. For MSPTA, the heuristic with a larger number of feasible solutions, the error is less than 4.5% and 3% for $|S| = 2$ and $|S| = 4$, respectively.

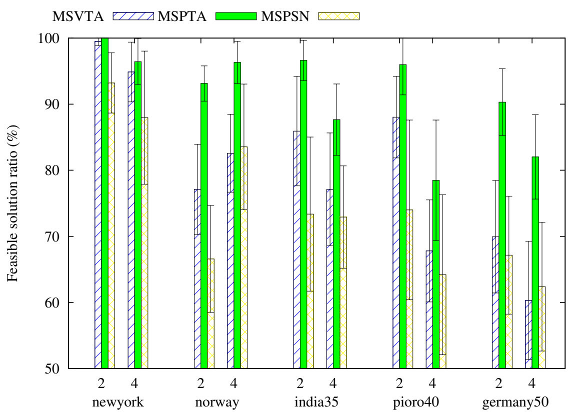
The largest average CPU time (15 seconds) was observed for CPLEX in the pioro40 network, for $|S| = 2$: for some node pairs the 5 minute per CPU pair limit only allowed us to obtain a sub-optimal solution and in other (less frequent cases) no information was obtained. In this case the heuristics required in average less than 0.01 seconds. The average CPU time of the heuristics grows with $|S|$ – see Fig. 7(c) – but that does not happen with CPLEX. For $|S| = 4$, there are a much larger number of node pairs for which the problem in infeasible, when compared to $|S| = 2$, and the ILP is quite effective identifying most of those infeasible problems.

For problem $\mathcal{P}_2$ and the 500 node networks (as in the case of problems $\mathcal{P}_0$ and $\mathcal{P}_1$) the percentage of feasible solutions found by the heuristics is very close to 100%, and hence no figure is used to present the feasible resolution ratio in Fig. 8.

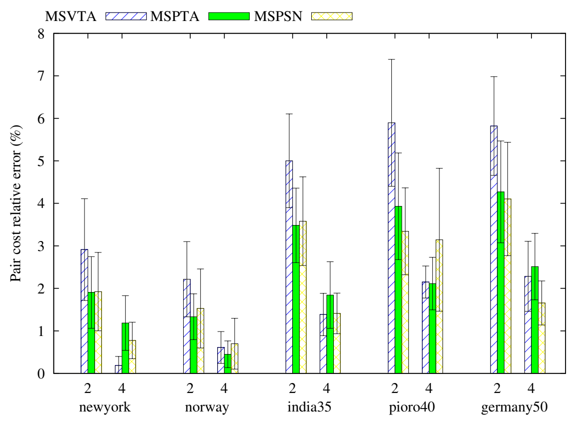
For several node pairs only a sub-optimal solution of $\mathcal{P}_2$ was calculated by CPLEX, and for a few node pairs – at least one in each of the networks for each value of $|S|$ – CPLEX was unable to find any solution (optimal or sub-optimal) under the 5 minute CPU time limit.

The heuristic MSPTA, which requires less CPU time – see Fig. 8(b) – solved all problems, except for three node pairs (one for each value of $|S|$) in one of the five tested networks.

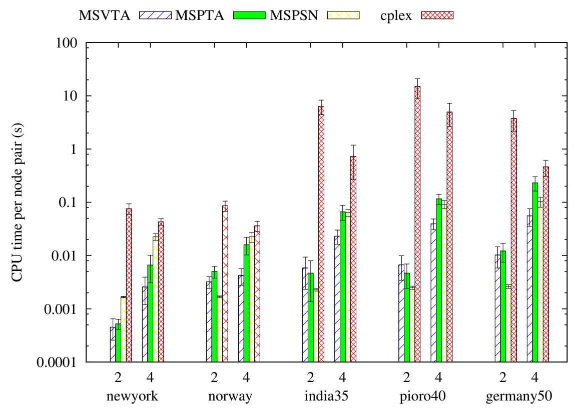
The heuristic with a more stable behavior regarding the error of the cost of the obtained feasible solutions is



(a) Feasible solutions ratio with respect to CPLEX

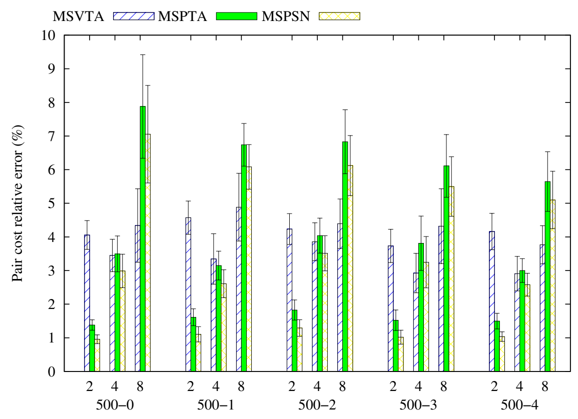


(b) Relative error of the feasible solutions found by the heuristics

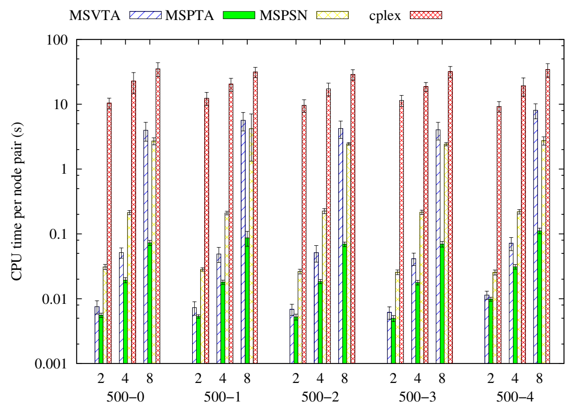


(c) CPU time per node pair

Figure 7: Min-sum cost disjoint path pair, each with specified nodes, results for five SNDlib [19] networks



(a) Relative error of the feasible solutions found by the heuristics



(b) CPU time per node pair

Figure 8: Min-sum cost disjoint path pair, each with specified nodes, results for five 500 node networks

MSVTA, with an error always below 5% – see Fig. 8(a). However for $|S| = 2$ both MSPTA and MSPSN present a relatively smaller error: less than 2%.

Regarding CPU time, CPLEX takes an average of tens of seconds to solve problem P_2 while the most computationally efficient heuristic MSPTA needs less than 0.1 second. For $|S| = 4$ and $|S| = 8$ this heuristic presents an average error of less than 4.5% and 8%, respectively.

4.4 Results for CORONET and TeliaSonora

Networks with nodes of degree two can be simplified, as explained in [3]. In case of mandatory nodes with degree two, a pair of adjacent arcs (or of adjacent edges, for undirected networks) must be in the solution. Both CORONET and TeliaSonera networks could have been highly simplified as can be deduced from their average node degree. However as this simplification may lead to parallel arcs, we opted to change the edge arcs (edges) cost to zero before executing PSN, without reducing these networks. Finally the cost of the obtained path is adequately corrected.

Fig. 9 presents results for PSN, considering different backtracking limits and rules for the *UpperBound* in the algorithm. In Fig. 9 the labels have following meaning:

- Original Param.: backtracking was limited as in the previously presented results; in the case of these networks it corresponds to the total number of arcs in the auxiliary subgraph. The *UpperBound* value, as in the previous results, was determined using the following rule: the algorithm would stop as soon as $2 + |S|$ solutions were obtained, or after having tried $4|S|$ initial arcs.

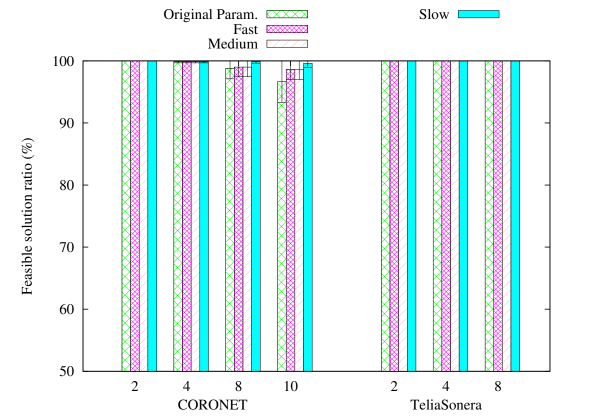
- Fast: backtracking was limited to the $(|S| + 2)^2$ arcs in the auxiliary subgraph. The *UpperBound* value was determined using the following rule: the algorithm would stop as soon as the first solution was obtained or after having tried the total number of arcs in the auxiliary subgraph.

- Medium: backtracking was limited to the same value as in Fast. The *UpperBound* value was determined using the following rule: the algorithm would stop as soon as the first five solutions were obtained or after having tried the total number of arcs in the auxiliary subgraph.

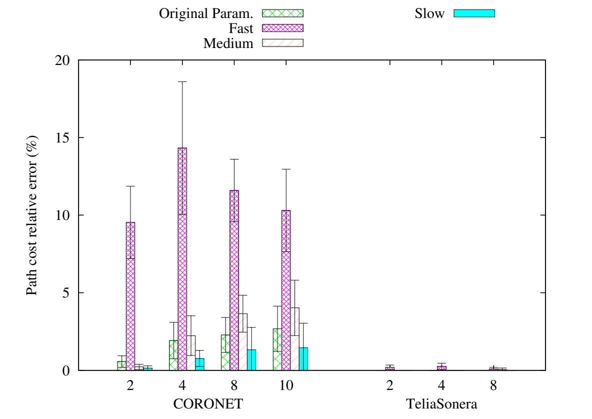
- Slow: backtracking was limited to the square value of the total number of arcs in the auxiliary subgraph. The *UpperBound* is the total number of arcs in the auxiliary subgraph.

As can be seen from the results, the heuristic was well tuned for the SNDLib and 500 node networks, because it presented good compromise solutions regarding execution time, resolution rate and the error for the obtained solutions.

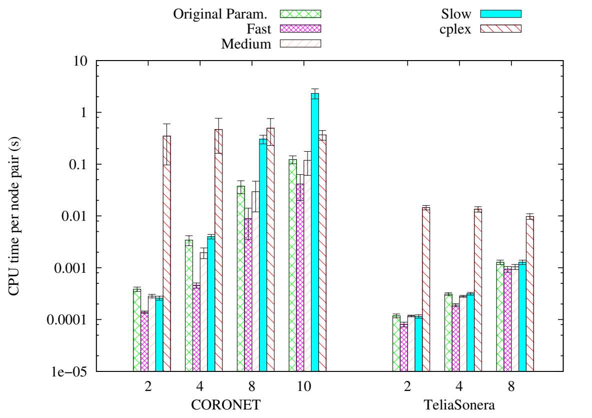
The results are extremely good for the TeliaSonera network, as can be observed in Fig. 9, because it is a small network of very low node degree. The results for the CORONET network show that the backtracking value can be important for solving some problems, but with



(a) Feasible solutions ratio of PSN with respect to CPLEX



(b) Relative error of the feasible solutions found by PSN



(c) CPU time per node pair

Figure 9: Loopless path with specified nodes, results for CORONET and TeliaSonera networks for PSN heuristic

very high cost in terms of CPU time. The *UpperBound* limit is not as relevant to solve the most difficult problems.

Finally PSN can obtain a valid solution in a very short time, possibly with some error. Hence it has potential to be applied in a distributed manner in order to find a path visiting a set of specified nodes.

5 Conclusions and Future work

Three problems were addressed in this work:

$\mathcal{P}_0$: to find a shortest simple path, from a source node to a target node, visiting a set of specific nodes;

$\mathcal{P}_1$: to obtain a shortest simple path, visiting a set of specified nodes, such that it can be protected by a node-disjoint path;

$\mathcal{P}_2$: to obtain a pair of node-disjoint simple paths, each visiting a specified set of nodes, such that the sum of the cost of the paths is minimum.

A path-based formulation of optimization problem $\mathcal{P}_1$ is given, along with numerical results showing the difficulty in solving the problem, thus providing motivation for the heuristics.

A new recursive heuristic (PSN) is proposed to solve $\mathcal{P}_0$ [17]. The extension of PSN to provide a node-disjoint backup path, designated PPSN, which conditions the adding of each node to visit, to the existence of a node-disjoint backup path, was presented; alternatively, the use of a trap avoidance approach, combined with PSN to solve problem $\mathcal{P}_1$, was also proposed and the resulting algorithm was designated PSNTA. Algorithms MSPSN and MSPTA were then developed to solve problem $\mathcal{P}_2$, the first one conditions the addition of each node to visit on the successful calculation of the backup disjoint path, and the second utilizes a trap avoidance approach seeking to obtain a pair of node disjoint paths.

The performance of the proposed heuristics was evaluated with respect to algorithms VSN, VTA and MSVTA, introduced in [14] for solving problems $\mathcal{P}_0$, $\mathcal{P}_1$, $\mathcal{P}_2$, respectively.

Algorithm PSN seems to be adequate for finding solutions for a set S with larger dimension than VSN, and also in networks of greater dimension. The accuracy of the solutions, evaluated using an optimization problem solution, diminishes with increasing $|S|$ for PSN. A similar but less pronounced effect is observed in VSN, but it solves fewer problems than PSN (and can not be used for large values of $|S|$).

For solving problem $\mathcal{P}_1$, PSNTA is the best compromise heuristic. It solves more problems and requires similar or less CPU time than the other two studied heuristics, although it has a slightly larger error in the cost of AP, especially when compared with VTA. However the latter requires too much CPU time when $|S| = 10$.

In larger networks algorithm MSPTA requires significantly less CPU time than MSPSN, and is slightly less accurate than the former. MSPTA also solves more problems in the SNDlib networks than MSPSN or MSVTA. In the case of the 500 node networks, the three heuristics are effective, however $|S| = 8$ seems to be the maximum value of $|S|$ that can be used with MSVTA. Hence one may conclude that MSPTA is the best compromise heuristic for solving problem $\mathcal{P}_2$.

The CPU time of all the heuristics, with respect to the optimization problem solver, was (almost) always at least an order of magnitude smaller, and often two and even three times (namely for smaller values of $|S|$). The heuristics can obtain solutions for instances that may require too much time to solve by an integer linear programming optimization problem solver.

Future work will be extending the heuristic for calculating maximally node-disjoint path pairs, each with a different set of specified nodes. Furthermore this algorithm has potential for being applied in a distributed manner throughout a network, with appropriate signaling, until a valid solution is obtained. This subject however needs further investigation.

Acknowledgment

Lúcia Martins and Teresa Gomes acknowledge financial support by Fundação para a Ciência e a Tecnologia (FCT) under project grant UID/MULTI/00308/2013 of INESC Coimbra. This article is based upon work from COST Action CA15127 ("Resilient communication services protecting end-user applications from disaster-based failures - RECODIS") supported by COST (European Cooperation in Science and Technology).

COST EUROPEAN COOPERATION IN SCIENCE AND TECHNOLOGY

RECODIS Resilient communication services protecting end-user applications from disaster-based failures

References

[1] C.P. Bajaj, Some constrained shortest-route problems, Math Methods Oper Res 15 (1971), 287–301.

[2] K. Calvert and E. Zegura, GT Internetwork Topology Models (GT-ITM), available at: http://www.cc.gatech.edu/projects/gtitm.

[3] E.K. Çetinkaya, M.J.F. Alenazi, Y. Cheng, A.M. Peck, and J.P.G. Sterbenz, On the fitness of geographic graph generators for modelling physical level topologies, Proc 5th Int Congress Ultra Modern Telecommunications Control Syst Workshops (ICUMT), Almaty, Kazakhstan, 2013, pp. 38–45.

[4] Y. Cheng, D. Medhi, and J.P.G. Sterbenz, Geodiverse routing with path delay and skew requirement under area-based challenges, Networks 66 (2015), 335–346.

[5] P. Cholda, A. Mykkeltveit, B. Helvik, O. Wittner, and A. Jajszczyk, A survey of resilience differentiation frameworks in communication networks, IEEE Commun Surveys & Tutorials 9 (2007), 32–55.

[6] P. Cholda, J. Tapolcai, T. Cinkler, K. Wajda, and A. Jajszczyk, Quality of resilience as a network reliability characterization tool, IEEE Netwo 23 (2009), 11–19.

[7] R.C. de Andrade, Elementary shortest-paths visiting a given set of nodes, Simpósio Brasileiro de Pesquisa Operacional, Natal, Brasil, 2013, pp. 16–19.

[8] R.C. de Andrade, New formulations for the elementary shortest-path problem visiting a given set of nodes, Eur J Oper Res 254 (2016), 755 – 768.

[9] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer Mathematik 1 (1959), 269 – 271.

[10] M. Doar and I. Leslie, How bad is naive multicast routing?, Proc Twelfth Ann Joint Conference IEEE Comput Commun Societies. Networking: Foundation Future (INFOCOM '93), vol. 1, San Francisco, CA, 1993, pp. 82–89.

[11] S.E. Dreyfus, An appraisal of some shortest-path algorithms, Oper Res 17 (1969), 395–412.

[12] M. Furdek, L. Wosinska, R. Goścień, K. Manousakis, M. Aibin, K. Walkowiak, S. Ristov, M. Gushev, and J.L. Marzo, An overview of security challenges

in communication networks, Proc 8th Int Workshop Resilient Networks Design Modeling (RNDM 2016), Halmstad, Sweden, 2016, pp. 43–50.

[13] T. Gomes, S. Marques, L. Martins, M. Pascoal, and D. Tipper, Protected shortest path visiting specified nodes, Proc 7th Int Workshop Reliable Networks Design Modeling (RNDM 2015), Munich, Germany, 2015, pp. 120–127.

[14] T. Gomes, L. Martins, S. Ferreira, M. Pascoal, and D. Tipper, Algorithms for determining a node-disjoint path pair visiting specified nodes, Optical Switching Networking 23, Part 2 (2017), 189 – 204.

[15] T. Ibaraki, Algorithms for obtaining shortest paths visiting specified nodes, SIAM Revi 15 (1973), 309–317.

[16] R. Kalaba, On some communication network problems, Technical report P-1325, Engineering Division, The RAND Corporation, 1959.

[17] L. Martins, T. Gomes, and D. Tipper, An efficient heuristic for calculating a protected path with specified nodes, Proc 8th Int Workshop Resilient Networks Design Modeling (RNDM 2016), Halmstad, Sweden, 2016, pp. 150–157.

[18] A. Mauthe, D. Hutchison, E.K. Çetinkaya, I. Ganchev, J. Rak, J.P.G. Sterbenz, M. Gunkelk, P. Smith, and T. Gomes, Disaster-resilient communication networks: Principles and best practices, Proc 8th Int Workshop Resilient Networks Design Modeling (RNDM 2016), Halmstad, Sweden, 2016, pp. 1–10.

[19] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski, SNDlib 1.0–Survivable Network Design library, Networks 55 (2010), 276–286.

[20] J. Rak, Resilient routing in communication networks, Computer Communication and Networks, Springer, 2015.

[21] J.P. Rohrer, A. Jabbar, and J.P.G. Sterbenz, Path diversification for future internet end-to-end resilience and survivability, Telecommunication Syst 56 (2014), 49–67.

[22] J.P. Saksena and S. Kumar, The routing problem with "K" specified nodes, Oper Res 14 (1966), 909–913.

[23] R. Stankiewicz, P. Cholda, and A. Jajszczyk, QoX: What is it really?, IEEE Commun Magazine 49 (2011), 148–158.

[24] J.P. Sterbenz, D. Hutchison, E.K. Çetinkaya, A. Jabbar, J.P. Rohrer, M. Schöller, and P. Smith, Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines, Comput Networks 54 (2010), 1245 – 1265.

[25] J.P. Sterbenz, J.P. Rohrer, E.K. Cetinkaya, M.J. Alenazi, A. Cosner, and J. Rolfe, KU-Topview network topology tool, available at: http://ittc.ku.edu/resilinets/maps.

[26] J.W. Suurballe and R.E. Tarjan, A quick method for finding shortest pairs of disjoint paths, Networks 14 (1984), 325–336.

[27] M. Tornatore, G. Maier, and A. Pattavina, Availability design of optical transport networks, IEEE J Selected Areas in Commun 23 (2005), 1520–1532.

[28] H. Vardhan, S. Billenahalli, W. Huang, M. Razo, A. Sivasankaran, L. Tang, P. Monti, M. Tacca, and A. Fumagalli, Finding a simple path with multiple must-include nodes, Proc IEEE Int Symp Modeling, Anal Simulation Comput Telecommunication Syst, London, 2009, pp. 1–3.

[29] Y. Wu, W. Chen, X. Zhang, and G. Liao, Improving the performance of arrival on time in stochastic shortest path problem, Proc IEEE 19th Int Conference Intelligent Transportation Syst (ITSC), Rio de Janeiro, Brazil, 2016, pp. 2346–2353.

[30] D. Xu, Y. Xiong, C. Qiao, and G. Li, Trap avoidance and protection schemes in networks with shared risk link groups, J Lightwave Technology 21 (2003), 2683–2693.

[31] J.Y. Yen, Finding the k shortest loopless paths in a network, Manage Sci 17 (1971), 712–716.

[32] IBM ILOG CPLEX optimization studio v12.6, IBM, 2013.