An efficient heuristic for calculating a protected path with specified nodes

Lúcia Martins *†, Teresa Gomes*†, David Tipper §
*Department of Electrical and Computer Engineering, University of Coimbra, 3030-290 Coimbra, Portugal
Email: lucia@deec.uc.pt, teresa@deec.uc.pt
†INESC Coimbra, DEEC, Rua Sílvio Lima, Pólo II, 3030-290 Coimbra, Portugal
§Graduate Telecommunications and Networking Program
University of Pittsburgh, Pittsburgh, PA, USA
E-mail: dtipper@pitt.edu

***Abstract*—The problem of determining a path between two nodes in a network that must visit specific intermediate nodes arises in a number of contexts. For example, one might require traffic to visit nodes where it can be monitored by deep packet inspection for security reasons. In this paper a new recursive heuristic is proposed for finding the shortest loopless path, from a source node to a target node, that visits a specified set of nodes in a network. In order 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. The performance of the heuristic, calculating a path with and without protection, is evaluated by comparing with an integer linear programming (ILP) formulation for each of the considered problems. The ILP solver may fail to obtain a solution in a reasonable amount of time, especially in large networks, which justifies the need for effective, computationally efficient heuristics for solving these problems. Our numerical results are also compared with previous heuristics in the literature.**

***Keywords*—Resilient routing, visiting a given set of nodes.**

I. Introduction

Communication networks are a key infrastructure of everyday life, and they should provide uninterrupted service in the presence of various challenges [1]. An architectural framework for resilience and survivability in communication networks can be found in [2]. Network service providers, depending on service level agreements, may need to ensure distinct levels of resiliency per service, which led to the introduction of the concept of Quality of Resiliency classes [3]–[5]. 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). Sometimes it is necessary to establish a path with specified nodes. These nodes may have been chosen due to inter-operators agreements, or to other network management constraints, such as transiting nodes where traffic can be inspected for security.

Very few works address the problem of calculating the shortest path from a source node to a target node that visits a given set of nodes. The earliest attempt to solve this problem is due to Saksena and Kumar [6], where the authors sought to develop an exact algorithm, using Bellmans's optimality principle, for calculating the 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 in [7] questioned their approach because it did not necessarily obtain the minimum cost solution.

In [8] the authors evaluated the GeoDivRP routing protocol with minimum-cost and the delay-skew requirement. The geo-diverse routes required by this protocol are calculated by the heuristic iWPSP. These routes are a set of geodiverse k paths, where each path is separated by a distance d, with a limit on the additional number of hops 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, for ensuring geodiversity.

Ibaraki in [9] considered separately the problem of calculating the shortest loopless path that visits a given set of nodes and the shortest path (possibly with cycles) that visits a set of specified nodes. Two approaches for obtaining the 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 [9]. Computational results in [9] suggest that the algorithm based on dynamic programming is less efficient than the algorithm based on branch and bound principle.

Andrade in [10] 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. The numerical results presented in [10], show that the new primal-dual based mixed integer formulation, designated Q3, is the the most effective.

In [7], Dreyfus proposes an approach for obtaining the 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 [11] 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 [11]. 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 [11] was proposed, where ASK is used followed by BSK if necessary.

In [12] the algorithm VSN was proposed for visiting 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 the shortest paths visiting all nodes in the graph can be obtained using a k-shortest path enumeration algorithm like Yen's [13]. A solution can only be found in a short time if the the number of nodes of S if small (less than 9), otherwise generating paths until finding paths visiting all nodes cant 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 n arcs have been deleted, or no paths from s to t can be obtained in the auxiliary graph.

The remainder fo the paper is structured as follows. In Section II the notation is introduced and the routing problem is formalized. In Section III a new and effective heuristic is proposed for the computation of a loopless path with specified nodes. The algorithm is then modified to take into account the constraint that the obtained path must be protected by node-disjoint path. Computational results are presented in Section IV. Section V concludes the paper.

II. Notation and problems formulation

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

A. Notation

The heuristics proposed in section III use 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 $p = \langle s \equiv v_1, v_2, \ldots, v_k \equiv t \rangle$, where $(v_i, v_{i+1}) \in A, \forall i \in \{1, \ldots, k-1\}$, 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 of path p will be represented by V_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 of a path, D_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 path is designated by S.

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 the shortest path from s to the nodes in S and the 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 $\mathcal{P}_1$. The rest of the notation, closely related with algorithms will be defined as needed.

B. Problems formulation

Problem $\mathcal{P}_1$, finding the shortest loopless path, from s to v, visiting a set of nodes S, can be stated as follows:

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

$$\text{subject to:} \quad V_{p_1} \cap S = S \quad (2)$$

The Integer Linear Programming formulation Q3, for obtaining p_1^*, can be found in [10] and is used in the numerical results section for comparative evaluation of the heuristics.

Problem $\mathcal{P}_2$, seeks to obtain the shortest path, visiting the set of nodes S, such that it can be protected by a node-disjoint path, and can be written as:

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

$$\text{subject to:} \quad V_{p_1} \cap S = S, \quad V_{p_1} \cap V_{p_2} = \{s, t\} \quad (4)$$

An Integer Linear Programming formulation for obtaining (p_1^*, p_2^*) can be found in [11], which is an adaptation of the

formulation Q3 in [10] with the additional constraint that the obtained path can be protected by a node-disjoint path. The exact results obtained using the formulation in [11] will be used to evaluate the performance of the heuristics.

III. 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 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 the shortest path is calculated.

Algorithm 1 (PSN) first calculates P_p. Specifically, the first for loop in Algorithm 1 (lines 3-7) determines the shortest routes between nodes in S. The second outer for loop (lines 8-13) finds the 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 26, is the concatenation of the initial chosen segment with the output of 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 in the solution of $p'(r)$; P'_p is the set containing sets $P'_p(r)$ of available segments not used in the concatenation of r segments of $p'(r)$.

The stopping conditions of Algorithm 2 are in lines 2-4: 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 6-16) 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 15) is evaluated by functions *existsCycle* and *goodSeg*. These functions are described by Algorithms 3 and 4, respectively.

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 $p_{v_i v_j} \leftarrow$ 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**
10 $p_{sv_i} \leftarrow$ shortestPath$(s, v_i, V \setminus S \setminus \{t\} \cup \{v_i\})$
11 $p_{v_i t} \leftarrow$ shortestPath$(v_i, t, V \setminus S \setminus \{s\} \cup \{v_i\})$
12 $P_{v_i} \leftarrow P_{v_i} \cup \{p_{v_i t}\}$
13 $P_s \leftarrow P_s \cup \{p_{sv_i}\}$
14 **if** $P_s = \emptyset \vee \forall v_i \in S \ \nexists p_{v_i t} \in P_{v_i}$ **then**
15 **return**
16 $P_p \leftarrow \cup_{v_i \in S \cup \{s\}} P_{v_i}$
17 $P'_p(1) \leftarrow P_p$
18 **while** $P_p \neq \emptyset \vee counter <$ UpperBound **do**
19 $p_{v_i v_j} \leftarrow \arg\min_{p_{v_i v_j} \in P_p} \sum_{(v_m, v_n) \in p_{v_i v_j}} w(v_m, v_n)$
20 $p'(1) \leftarrow p_{v_i v_j}$
21 $P_p \leftarrow P_p \setminus \{p_{v_i v_j}\}$
22 $V'_S \leftarrow S \cup \{s,t\} \setminus \{v_i, v_j\}$
23 $P'_p(1) \leftarrow P'_p(1) \setminus \{p_{v_i v_j}\}$
24 $FP(r) \leftarrow \emptyset, \quad r = 1, 2, \ldots, |S| - 1$
25 $P'_p(r) \leftarrow \emptyset, \quad r = 2, \ldots, |S| - 1$
26 $p' \leftarrow p_{v_i v_j} \circledcirc$ Pcompute$(p'(1), s, t, V'_S, P'_p, FP)$
27 $P'_p(1) \leftarrow P'_p(1) \cup \{p_{v_i v_j}\}$
28 **if** p' is a valid path from s to t **then**
29 $P_{st} \leftarrow P_{st} \cup \{p'\}$
30 $counter \leftarrow counter + 1$
31 **if** $P_{st} \neq \emptyset$ **then**
32 $p^*_{st} \leftarrow \arg\min_{p_{st} \in P_{st}} \sum_{(v_m, v_n) \in p_{st}} w(v_m, v_n)$
33 **return**

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 22-27 – removing the last added 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 be updated.

In Algorithm 3 (function *existCycle*), given a candidate path segment, $p_{v_i v_j}$, to be concatenated with $p'(r)$, 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 $P'_p(r)$

Algorithm 2: 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 till p' becomes: a path starting at node s, passing through all nodes in S without cycles, and ending at node t with a disjoint path for protection (if required); or an incomplete path which could not be ended.

begin
  if $V'_S = \emptyset$ **then return** $\emptyset$
  $p_{v_l v_k} \leftarrow p'(r)$
  if $\nexists\ p_{v_i v_l} \vee p_{v_k v_j} \in P'_p(1) \setminus FP(1)$ **then return** $\emptyset$
  $cycles \leftarrow true$
  while $cycles$ `/* search feasible segment */`
  do
    if $v_l \neq s \wedge v_k \neq t$ **then**
      $P''_p \leftarrow \big\{p_{v_i v_j} \in P'_p(r) \setminus FP(r) :$
        $(v_i = v_k \wedge v_j \in V'_S) \vee (v_l = v_j \wedge v_i \in V'_S)\big\}$
    if $v_l = s$ **then**
      $P''_p \leftarrow \big\{p_{v_k v_j} \in P'_p(r) \setminus FP(r) : v_j \in V'_S\big\}$
    if $v_k = t$ **then**
      $P''_p \leftarrow \big\{p_{v_i v_l} \in P'_p(r) \setminus FP(r) : v_i \in V'_S\big\}$
    $p_{v_i v_j} \leftarrow \arg\min_{P''_p} \sum_{(v_m, v_n) \in p_{v_i v_j}} w(v_m, v_n)$
    if $p_{v_i v_j} = \emptyset \vee$
    $\big(\neg\text{existCycle}(p_{v_i v_j}, p'(r), s, t, V'_S, P'_p, FP) \wedge$
    $\text{goodSeg}(p_{v_i v_j}, p'(r), s, t, V'_S, P'_p, FP)\big)$ **then**
      $cycles \leftarrow false$
  if $p_{v_i v_j} = \emptyset \wedge r = 1$ **then return** $\emptyset$
  if $p_{v_i v_j} = \emptyset$ `/* backtracking */`
  then
    $p_{v_l v_k} \leftarrow p'(r) \setminus p'(r-1)$
    $FP(r-1) \leftarrow FP(r-1) \cup \{p_{v_l v_k}\}$
    $FP(r) \leftarrow \emptyset$
    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
    $p_{v_i v_j} \circledcirc \text{Pcompute}\big(p'(r-1), s, t, V'_S, P'_p, FP\big)$
  else
    if $v_i \in V'_S$ **then** $V'_S \leftarrow V'_S \setminus \{v_i\}$
    else $V'_S \leftarrow V'_S \setminus \{v_j\}$
    $P'_p(r+1) \leftarrow P'_p(r) \setminus \{p_{v_i v_j}\}$
    $p'(r+1) \leftarrow p_{v_i v_j} \circledcirc p'(r)$ `/* add segment */`
    return
    $p_{v_i v_j} \circledcirc \text{Pcompute}\big(p'(r+1), s, t, V'_S, P'_p, FP\big)$

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 $\mathcal{P}_2$ is the problem being solved, this function

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

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 later 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).

begin
  if $\exists v_k \neq v_i, v_j : v_k \in p_{v_i v_j} \wedge v_k \in p'(r)$ **then**
    $p'_{v_i v_j} \leftarrow$
    $\text{shortestPath}(v_i, v_j, V \setminus V'_S \setminus V_{p_{vivj}} \cup \{v_i, vj\})$
    $P'_p(r) \leftarrow P'_p(r) \setminus \{p_{v_i v_j}\}$
    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 */`
  else return false `/* no cycle */`

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

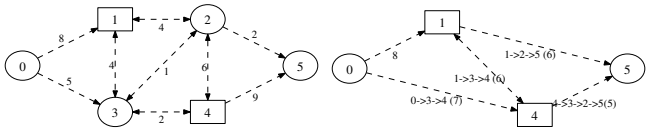
Data: $G = (V, A)$; s and t; $p'(r)$, path with r segments; V'_S, set of mandatory nodes to be included; P'_p, set of sets $P'_p(r)$; FP, set of sets $FP(r)$; $p_{v_i v_j}$, to be evaluated

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.

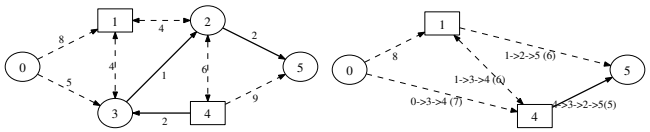
begin
  $p_{v_l v_k} \leftarrow p_{v_i v_j} \circledcirc p'(r)$
  if $(v_l = s \wedge v_k = t \wedge |V'_S| - 1 \neq 0)$
  $\vee (v_l \neq s \wedge$
  $\text{shortestPath}(s, v_l, V \setminus V'_S \setminus V_{p_{v_l v_k}} \cup \{v_l, s\}) = \emptyset)$
  $\vee (v_k \neq t \wedge$
  $\text{shortestPath}(v_k, t, V \setminus V'_S \setminus V_{p_{v_l v_k}} \cup \{v_k, t\}) = \emptyset)$
  $\vee$ (path requires protection $\wedge$
  $\text{shortestPath}(s, t, V \setminus V'_S \setminus V_{p_{v_l v_k}} \cup \{s, t\}) = \emptyset)$
  then
    $P'_p(r) \leftarrow P'_p(r) \setminus \{p_{v_i v_j}\}$
    $FP(r) \leftarrow FP(r) \cup \{p_{v_i v_j}\}$
    return false `/* not good */`
  else
    return true `/* possibly good */`

also evaluates if the segment resulting from the concatenation of $p_{v_i v_j}$ and $p'(r)$, allows 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 the shortest path node-disjoint with the path constructed by PPSN.

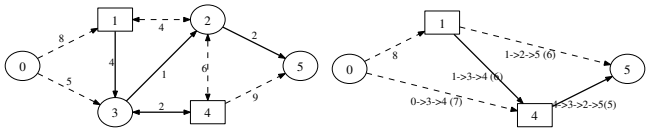
In Figure 1 an illustrative example of the algorithm is



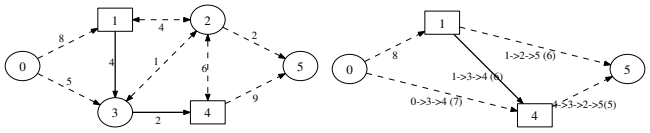
(a) Original network and subgraph



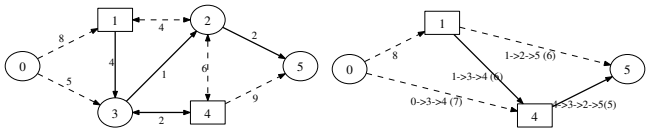
(b) First segment selected



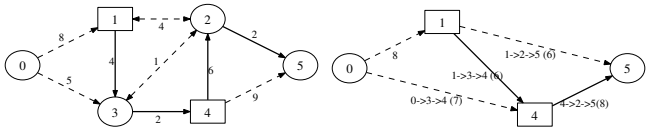
(c) First cycle is detected



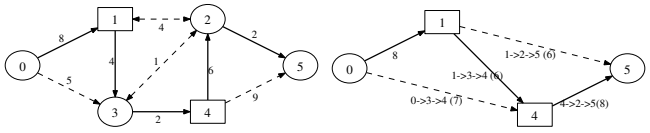
(d) New beginning



(e) Next selected segment



(f) Cycle resolution



(g) Final segment is concatenated

Fig. 1: Illustrative path construction example

presented, with $s = 0$, $t = 5$ and $S = \{1, 4\}$. On the left and right side of each figure is the 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. 1a each arc of the subgraph corresponds to the paths in P_p (see line 16 of Algorithm 1). Then, according to line 19 of Algorithm 1), the first selected arc in the subgraph is the one of minimum cost, arc $(4, 5)$ – see Fig. 1b. Then function *Pcompute* is called to build the remaining path. This function seeks for the next minimum cost arc that can be concatenated (at left or right) with the existing sub-path. The first candidate arc would result in a path with a cycle – see Fig. 1c. Function *existCycle* (Algorithm 3) tries to find and alternative path from node 1 to node 4, avoiding nodes 0, 2, 3, 5, but no such path exists; arc $(1, 4)$ is marked as leading to no solution and *Pcompute* returns an empty path, because the remaining arc entering node 4, arc $(0, 4)$, prevents node 1 from being in the solution (as can be seen in line 3 of function *goodSeg*). The second iteration of the while cycle (see lines 18-30 of Algorithm 1) starts, using the second shortest arc in the subgraph, arc $(1, 4)$ – see Fig. 1d. Next *Pcompute* selects arc $(4, 5)$, the next minimum cost arc that can be concatenated (at left or right) with the existing sub-path; this results in path with a cycle – see Fig. 1e. Function *existCycle* (Algorithm 3) recalculates the path from node 4 to 5 avoiding nodes 0, 1, 3, and obtains the path $\langle 4, 2, 5\rangle$ with cost 8, which is still minimum – see Fig. 1f. Then *Pcompute* is called once again to finish the path, obtaining the solution shown in Fig.1g, which in this case is the optimal solution, $\langle 0, 1, 3, 4, 2, 5\rangle$ of cost 22.

IV. Results

Networks newyork, norway, india35, pioro40 and germany50, from the SNDlib [14] repository, were used to evaluate the heuristics; the cost of each edge was the first module cost as given in SNDlib. A second set of networks was generated with the Doar-Leslie model [15] using Georgia Tech Internetwork Topology Models software (GT-ITM) [16] (http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html). Five networks with 500 nodes, arc cost between 1 and 100 and with an average degree of around 7 (sum of the in and out degree) were generated. If the generated networks contained spurs, they were removed before solving problems $\mathcal{P}_1$ and $\mathcal{P}_2$ (this resulted in removing between 4 and 8 nodes), thus ensuring the networks studied were biconnected.

The number of specified nodes was considered to be equal to 2, 4, 10 and 20, corresponding to a small, medium and large size of S. The elements in each set S where 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 and germany50 the maximum value of $|S|$ was 10 and only for the 500 nodes 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. Although in optical networks the size of S will in general be small in other contexts (for example wireless sensor networks) the number of given nodes may be larger.

Twenty samples were obtained for each network, and 95% confidence intervals around the estimated mean were calculated, appearing in the graphs as error bars. First, the new proposed heuristic (algorithm PSN) and algorithm SK [11] are compared, regarding their efficiency solving problem $\mathcal{P}_1$. Then results are presented regarding algorithm PPSN and ABSK [11]; this last algorithm was selected, because in [11] it was considered the best compromise heuristic for solving Problem $\mathcal{P}_2$.

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 are presented regarding the percentage of problems solved (optimally or sub-optimally) with respect to the (optimal or sub-optimal) solutions obtained by the CPLEX solver. In order to evaluate the quality of the solutions (of the heuristics), the relative error of the cost of the (active) path obtained by the heuristics, for which CPLEX also found an optimal solution, was calculated.

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 [17]. In a some problem instances, CPLEX could take a very long time. For example, in network india35, for a node pair (with $|S| = 2$), after 8 hours in a Laptop (i7-3520M, 2.9 GH) no solution was found for problem $\mathcal{P}_1$. Hence, it was decided, in order obtain solutions in a reasonable time, to establish a limit of 5 minutes per node pair for the CPLEX solver.

For some networks and a given $|S|$ no results are shown for algorithms SK and ABSK because, in at least one of the 20 instances of specified nodes, the corresponding algorithm was unable to find a solution (for any of the 100 node pairs).

A. Results regarding solving Problem $\mathcal{P}_1$

The results obtained solving Problem $\mathcal{P}_1$ are shown in Figs. 2 for the networks from SNDlib [14].

The number of feasible solutions found by solving problem $\mathcal{P}_1$ for the five SNDlib networks is very close to 100% for PSN – see Fig. 2a. In contrast, Algorithm SK has a poor performance, namely for $|S| = 10$. SK is a very greedy algorithm: it only considers shortest paths as possible building segments of the final path, therefore reducing the possibilities of finding a solution.

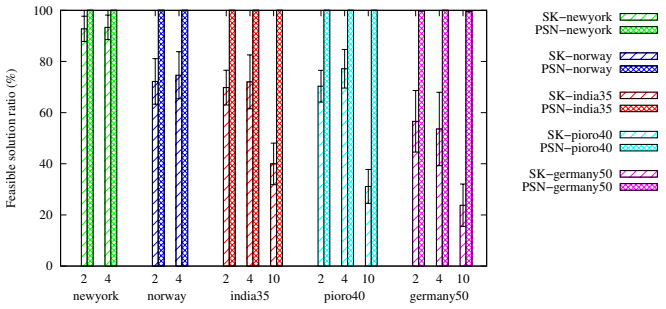
Regarding the accuracy of the obtained solutions, the relative error (with respect to the solution found by the ILP solver) is shown in Fig. 2b, where PSN and SK have less than 5% relative error for $|S| = 2, 4$; for $S = 10$ the error grows for both heuristics: it is always below 10% for PSN but it it reaches 20% for SK in pioro40 network.

The CPU time of both heuristics is similar (with some advantage for SK) and is always, at least, one order of magnitude smaller than the CPU time required by CPLEX – see Fig. 2c.

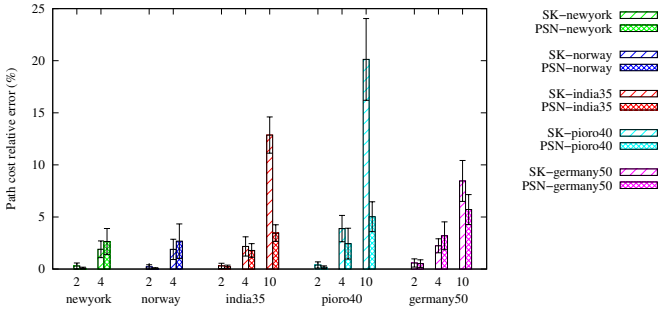
In Fig. 3 results obtained using the 500 node networks, labeled 500_i with $i = 0, 1, 2, 3, 4$, can be found for PSN and SK. Note that when $|S| = 10$ and $|S| = 20$, for one network (500_4) and two networks (500_0 and 500_4), respectively, no results are shown for algorithm SK.

Regarding the feasible solution ratio of PSN it is very close to 100% (in fact it is 100% in most tested cases). The solution ratio for SK is now between 40% and 80% in average, and its performance degrades for $|S| = 10, 20$.

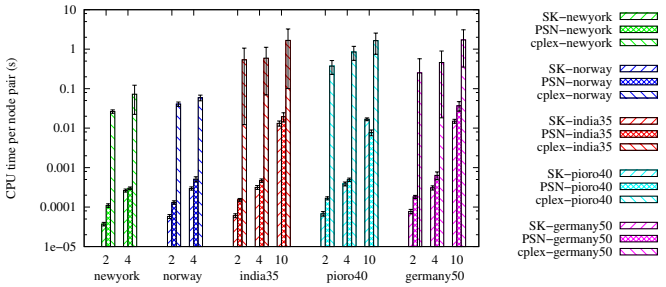
The relative error of the solutions found is (in general) below 5% for $|S| = 2, 4$ but when $|S| = 10, 20$ both heuristics



(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

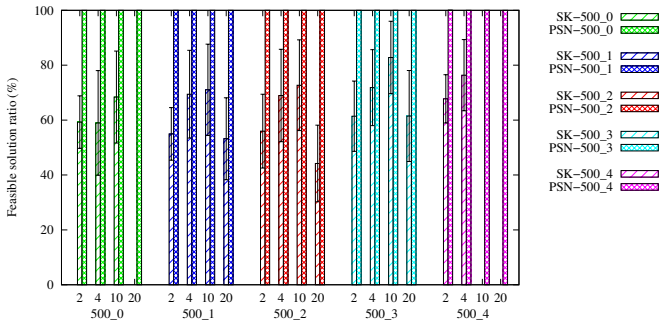
Fig. 2: Loopless path with specified nodes, results for five SNDlib [14] networks

have an increase in the relative error, but it is much more pronounced for SK. Regarding the CPU time both heuristics perform quite well with respect to the CPLEX solver, being at least an order of magnitude faster. For $|S| = 20$ they both require less than a second of CPU time in average, while CPLEX may require tens of seconds. Note that, in a few instances, CPLEX ended due to the imposed CPU time limit of 5 minutes, while both SK and PSN were able to find a solution.

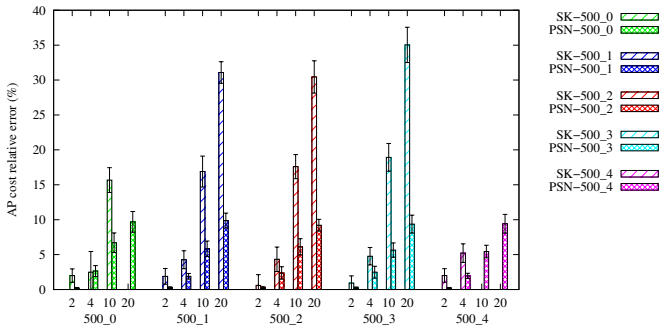
In conclusion, PSN has a larger feasible solution ratio, smaller relative error (for $|S| = 10, 20$) than SK, and it requires a CPU time similar to SK, therefore one can conclude that it outperforms SK.

B. Results regarding solving Problem $\mathcal{P}_2$

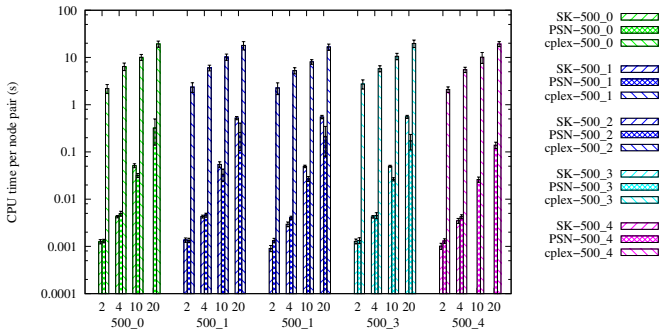
The results obtained solving Problem $\mathcal{P}_2$ for the networks from SNDlib [14] with the proposed heuristics PPSN and ABSK are shown in Figs. 4. ABSK has a much better relative



(a) Feasible solutions ratio with respect to CPLEX

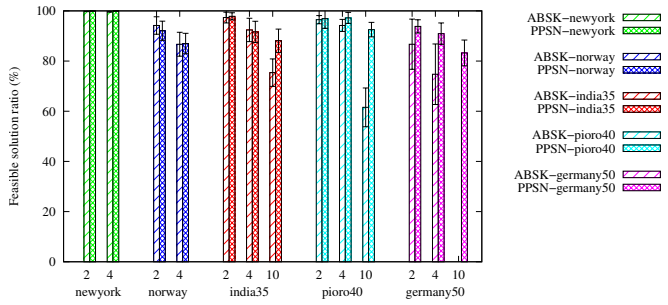


(b) Relative error of the feasible solutions found by the heuristics

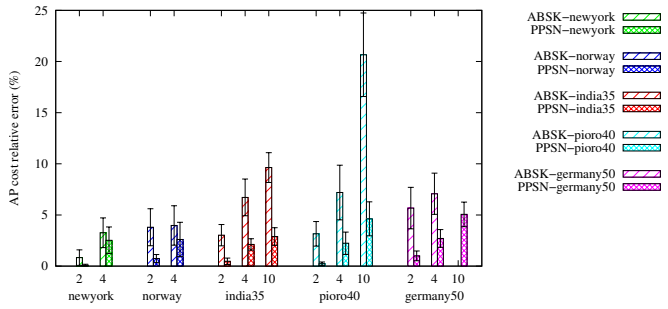


(c) CPU time per node pair

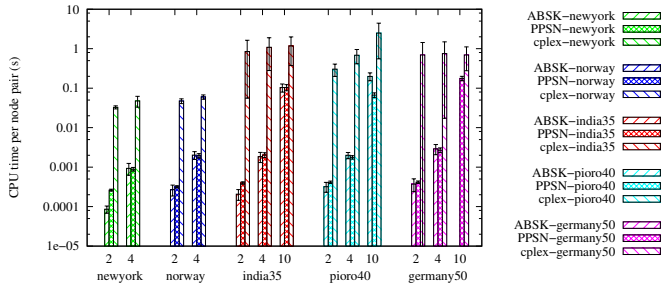
Fig. 3: Loopless path with specified nodes, results for five 500 nodes networks



(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

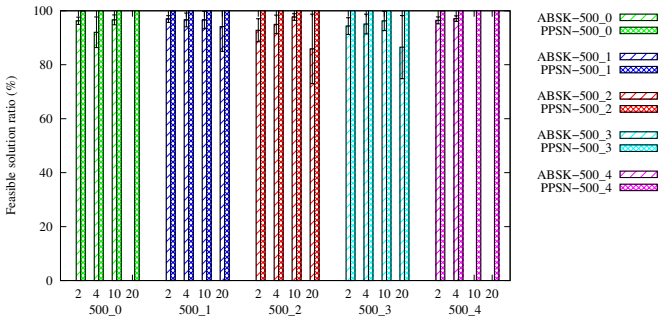
Fig. 4: Protected loopless path with specified nodes, results for five SNDlib [14] networks

performance than SK because ABSK is a combination of two algorithms and, when no solution is found, the initial and last segments of the final candidate active paths are recalculated – no equivalent efforts are done in SK.

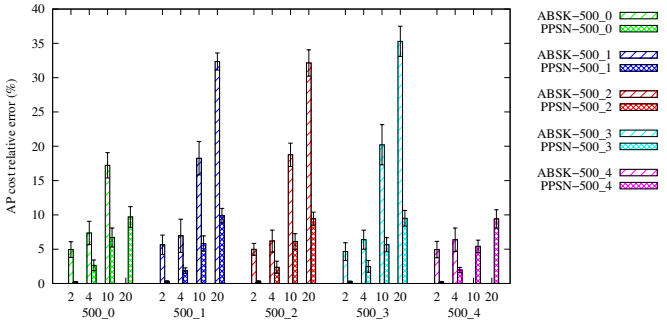
In Fig. 4a, for $|S| = 2, 4$ the two heuristics presents (in most cases) similar results regarding the feasible solution ratio. However, for $|S| = 10$ the advantage of PPSN over ABSK becomes clear. In Fig. 4b, it can be observed that the error of the calculated solutions is below 10% on average, when $|S| = 2, 4$, but when $|S| = 10$, the error of PPSN remains below 10%, while for pioro40 the error of ABSK gets to 20%. In Fig. 4c, presenting the CPU time, the observed values (for the heuristics) are only slightly larger than in Fig. 2c, with the exception of germany50 when $|S| = 10$, due to the relatively larger number of unsolved problems in this network (the most sparse network in the considered set).

The results for the 500 node networks created using the GT-ITM software are given in Fig. 5. For the 500 node networks, the feasible solution ratio is very close to 100% for PPSN and in general above 80% for ABSK – see Fig. 5a. Note that ABSK (like SK) can't find a solution for all cases (e.g., $|S| = 10$, $|S| = 20$ for network 500_4). The relative error of the feasible solutions increases with the number of specified nodes to visit, as can be seen in Figure 5b, being more marked in the case of ABSK. 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. 5c. Note that SK, for $|S| = 20$ requires more CPU time than CPLEX. For $|S| = 2, 4$ both heuristics have small CPU times, with a clear advantage to PPSN. For $|S| = 10, 20$ the CPU time of PPSN is below 0.1 and 1 second, respectively, while the CPU time of ABSK is significantly larger.

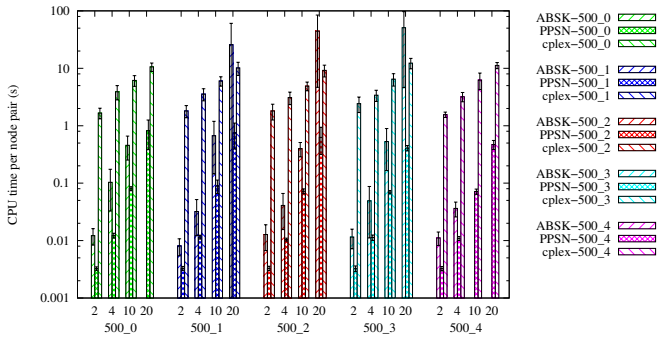
In summary, the performance of PPSN is superior to the



(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

Fig. 5: Protected loopless path with specified nodes, results for five 500 nodes networks

performance of ABSK, particularly for larger values of $|S|$.

V. Conclusions and Future work

The problem of calculating the shortest path that visits a given set of nodes is a difficult problem. In this paper a new recursive heuristic (PSN) is proposed for finding the shortest loopless path, that visits a specified set of nodes in a directed graph. Further the extension of PSN to provide a node disjoint backup path PPSN is given.

Extensive numerical results clearly show the better performance of PSN and PPSN with respect to SK and ABSK heuristics from the literature. The accuracy of the solutions, evaluated using an optimization problem solution, diminished with increasing $|S|$. Nevertheless algorithm PSN presents a very high number of feasible solutions with reasonable relative error and algorithm PPSN is also quite effective. The CPU time with respect to the optimization problem solver was always almost an order of magnitude smaller. 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 a node-disjoint path pair, each with a different set of specified nodes.

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.

References

[1] J. Rak, *Resilient Routing in Communication Networks*. Springer, 2015.

[2] 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," *Computer Networks*, vol. 54, no. 8, pp. 1245 – 1265, 2010. Resilient and Survivable networks.

[3] P. Cholda, A. Mykkeltveit, B. Helvik, O. Wittner, and A. Jajszczyk, "A survey of resilience differentiation frameworks in communication networks," *Communications Surveys Tutorials, IEEE*, vol. 9, pp. 32–55, Fourth 2007.

[4] P. Cholda, J. Tapolcai, T. Cinkler, K. Wajda, and A. Jajszczyk, "Quality of resilience as a network reliability characterization tool," *Network, IEEE*, vol. 23, pp. 11–19, March 2009.

[5] R. Stankiewicz, P. Cholda, and A. Jajszczyk, "Qox: What is it really?," *Communications Magazine, IEEE*, vol. 49, pp. 148–158, April 2011.

[6] J. P. Saksena and S. Kumar, "The routing problem with 'k' specified nodes," *Operations Research*, vol. 14, no. 5, pp. 909–913, 1966.

[7] S. E. Dreyfus, "An appraisal of some shortest-path algorithms," *Operations Research*, vol. 17, no. 3, pp. 395–412, 1969.

[8] Y. Cheng, D. Medhi, and J. P. G. Sterbenz, "Geodiverse routing with path delay and skew requirement under area-based challenges," *Networks*, vol. 66, no. 4, pp. 335–346, 2015.

[9] T. Ibaraki, "Algorithms for obtaining shortest paths visiting specified nodes," *SIAM Review*, vol. 15, no. 2, pp. 309–317, 1973.

[10] R. C. de Andrade, "Elementary shortest-paths visiting a given set of nodes," in *Simpósio Brasileiro de Pesquisa Operacional*, pp. 16–19, Setember 2013.

[11] T. Gomes, S. Marques, L. Martins, M. Pascoal, and D. Tipper, "Protected shortest path visiting specified nodes," in *7th International Workshop on Reliable Networks Design and Modeling (RNDM 2015)*, pp. 120–127, Oct 2015.

[12] T. Gomes, L. Martins, S. Ferreira, M. Pascoal, and D. Tipper, "Algorithms for determining a node-disjoint path pair visiting specified nodes," *Optical Switching and Networking*, pp. –, 2016.

[13] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Management Science*, vol. 17, pp. 712–716, July 1971.

[14] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski, "SNDlib 1.0–Survivable Network Design library," *Networks*, vol. 55, no. 3, pp. 276–286, 2010. http://sndlib.zib.de.

[15] M. Doar and I. Leslie, "How bad is naive multicast routing?," in *INFOCOM '93. Proceedings.Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future, IEEE*, pp. 82–89 vol.1, 1993.

[16] K. Calvert and E. Zegura, "GTInternetworkTopologyModels (GT-ITM)." College of Computing, Georgia Institute of Technology, 1996.

[17] *IBM ILOG CPLEX Optimization Studio V12.6*. IBM, 2013.