Available online at www.sciencedirect.com
ScienceDirect
European Journal of Operational Research 181 (2007) 1055–1064
EUROPEAN JOURNAL OF OPERATIONAL RESEARCH
www.elsevier.com/locate/ejor

Efficient calculation of the most reliable pair of link disjoint paths in telecommunication networks

Teresa Gomes *, José Craveirinha

Department of Electrical and Computer Engineering, Faculty of Sciences and Technology of the University of Coimbra, Pinhal de Marrocos, 3030-290 Coimbra, Portugal
INESC-Coimbra, Rua Antero de Quental, 199, 3000-033 Coimbra, Portugal

Available online 19 April 2006

Abstract

In transmission networks an important routing problem is to find a pair of link disjoint paths which optimises some performance measure. In this paper the problem of obtaining the most reliable pair of link disjoint paths, assuming the reliability of the links are known, is considered. This is a non-linear optimisation problem. It is further introduced the constraint that the length of the paths should not exceed a certain number of links, which makes the efficient resolution of the problem more complex.

In this paper two variants of a novel exact algorithm for finding the most reliable pair of link disjoint paths with lengths constraints, are presented. Also a computational study on the performance of the variants, is described.

Keywords: Reliability; OR in telecommunications; Routing

1. Introduction

Due to the extensive use of optical fibers, links in transmission networks generally carry a large amount of traffic. In wavelength-routed WDM (Wavelength Division Multiplexing) optical networks it is desirable to provide some degree of protection against link and/or node failures [19]. A common approach, in this and in other types of telecommunication networks, is to establish two link disjoint paths for every connection request (the working path and the protection path) hence preventing the failure of both paths in single failure scenarios corresponding to the failure of any link common to both paths. MPLS networks also require a fast path recovery process in the event of failures, and therefore LSPs (Label-Switched Paths) can similarly be set in disjoint pairs (the working and the protection paths) [16].

Suurballe [17] proposed an algorithm for obtaining k-shortest node (or arc) disjoint paths, with polynomial computation complexity. Suurballe and Tarjan [18] proposed an algorithm for obtaining shortest pairs of disjoint paths. Some other improved versions of algorithms for diverse routing

* Corresponding author. Address: Department of Electrical and Computer Engineering, Faculty of Sciences and Technology of the University of Coimbra, Pinhal de Marrocos, 3030-290 Coimbra, Portugal.
E-mail addresses: teresa@deec.uc.pt (T. Gomes), jcrav@deec.uc.pt (J. Craveirinha).

0377-2217/$ - see front matter
doi:10.1016/j.ejor.2006.03.005

(i.e. finding physically disjoint paths between a pair of nodes) can be found in [2].

In [10] an algorithm for obtaining k disjoint paths between two different nodes, s and t, in a network with k different costs on every edge, is proposed, such that the total cost of the paths is minimised (where the jth edge-cost is associated with the jth path). The selection of the optimal pair of disjoint paths for solving dynamic routing problems requires the use of adequate algorithms for diverse routing. Several objective functions to be optimised may be considered in this context. One such objective can be the most reliable pair of paths. In telecommunication networks constraints are often imposed in terms of number of links. Hence an algorithm for efficiently obtaining the most reliable pair of paths with a maximum number of links (arcs of the network graph) is proposed in this paper. The problem of finding the set of paths which maximises the two-terminal reliability metric has not received much attention according to Papadimitratos et al. [14]. These authors propose a heuristic (with polynomial worst case complexity) for obtaining a set of disjoint paths (without length constraints) with high two-terminal reliability applied in the context of Mobile Ad Hoc Networks (MANET).

In [8] it is shown that the problem of finding a maximum number of disjoint paths with at most D arcs is NP-Hard for $D \geqslant 5$. The problem under analysis requires only two such paths, but although the costs (defined for the arcs) are additive regarding path cost calculation, the function to be optimised (union probability) is not linear in the path costs. Therefore the problem cannot be reduced to a minimal-cost feasible solution with at most D arcs in each path.

In this paper we present two variants of an exact algorithm for finding the most reliable pair of link disjoint paths (in undirected or directed networks) with a maximal number D of arcs per path.

The resolution approach proposed for this problem uses an algorithm for sequentially obtaining k-shortest length constrained paths (that is with a maximum number of arcs per path) developed by the authors, designated KD [7]. A first version of the resolution procedure was considered [5] where the algorithm chosen for obtaining the disjoint path for each candidate working path was based on an algorithm for sequential enumeration of disjoint length constrained paths by decreasing reliability order [6]. This auxiliary algorithm (sometimes) has difficulties in detecting that no disjoint path, with at most D arcs, exists for the present candidate working path. Once the Bellman–Ford algorithm obtains the shortest paths from every node to a given destination node, by increasing order of the number of arcs per path, this was used in a second variant of the proposed algorithm. The resolution approach has similarities with the enhancement of the Two-Step-Approach [9] and with the Iterative Two-Step-Approach (ITSA) algorithm [13] for optimal diverse routing with shared protection in connection-oriented networks, where the arc costs of the protection path depend on the selected working path. Note that, in our case, this condition does not apply and the function to be optimised is not a linear combination of the costs of the working and protection paths, unlike the problem in [13]; this will be shown in Section 2.

Also an experimental study on the performance of the two versions of the algorithm, in a set of randomly generated networks, is presented.

The paper is organised as follows. Firstly (in Sections 2 and 3) the problem under analysis is described, the algorithm is presented and its complexity is discussed for the two variants mentioned above. In Section 4 experimental results for various sets of randomly generated networks with different connectivities are analysed and some conclusions are presented. Finally, in Appendix, KD algorithm is reviewed and the auxiliary algorithm for obtaining the disjoint path for each candidate working path, is described.

2. Problem formulation

Let $G = (N, L)$ be a directed graph where $N = \{v_1, v_2, \ldots, v_n\}$ is the node set and L the arc (or link) set, composed of ordered pairs of elements in N, where n represents the cardinality of set N. Let $l = (i,j)$ be an arc where j is the head of l and i its tail. A path from s to t ($s, t \in N$) in this graph will be specified by the sequence $p = \langle s, (s, v_1), v_1, \ldots, (v_w, t), t\rangle$, where all $l = (v,u) \in p$ belong to L. If all nodes in p are different it is called a loopless path. Although up till now only the term path was used, the loopless condition is implicitly assumed. The word "loopless" will continue to be omitted until explicit reference is needed.

Each link $l \in L$ has a probability $p_L(l)$ of being operational. Nodes are assumed not to fail. In a network where links fail (independently) and one seeks link disjoint paths from s to t, a *cost matrix* $[c_{ij}]$ of

dimension $n \times n$ is defined such that the cost of an arc is the additional cost of introducing that arc in a path:

$$c_{ij} = \begin{cases} -\ln p_L(l) & \text{if } l = (i,j) \in L \\ +\infty & \text{if } l = (i,j) \notin L \end{cases} \quad (1)$$

The cost of a path $p = \langle s, (s, v_1), v_1, \ldots, (v_w, t), t \rangle$ is $\mathscr{C}(p) = \sum_{(v_i,v_j)\in p} c_{v_i v_j}$, and its reliability is

$$\Pr(p) = e^{-\mathscr{C}(p)} \quad (2)$$

where "$\Pr(p)$" represents the probability of path p being operational. Eq. (2) establishes a relation between the cost of a path and its reliability. Using the cost matrix $[c_{ij}]$, the enumeration of the k shortest paths with a maximum of D arcs (per path) is equivalent to the enumeration of the k most reliable paths with at most D arcs by decreasing order of their reliability.

The most reliable pair of link disjoint paths (p_w, p_v) has a reliability given by

$$\max_{p_w,p_v} \Pr(p_w \cup p_v) = \Pr(p_w) + (1 - \Pr(p_w))\Pr(p_v) \quad (3)$$

where p_w and p_v are the working and protection paths, respectively. As can be seen from (3) $\Pr(p_w \cup p_v)$ cannot be written as a linear function of the costs of p_w and p_v. Two disjoint paths may have minimum $\mathscr{C}(p_w) + \mathscr{C}(p_v)$ but they may not be the paths with maximum $\Pr(p_w \cup p_v)$.

The sequential generation of paths p_i (selected by decreasing reliability order) with a maximum number of arcs per path can be made by using the algorithm KD proposed by the authors in [7]. For each i-shortest path p_i (where i represents the order of a selected path satisfying the length constraints—p_i is a candidate working path) there may exist more than one link disjoint path (p_j, a candidate protection path for p_i). The path p_j which maximises $\Pr(p_i \cup p_j)$ (with p_i fixed) will be the one with highest reliability among all of the feasible paths. Therefore a sub-algorithm is needed for efficiently obtaining the most reliable path disjoint with p_i (with at most D arcs). Such a sub-algorithm can be obtained with a slight alteration of the KD-ld algorithm proposed by the authors in [6], which will be designated hereafter as KD-ld1 algorithm, or using the Bellman–Ford algorithm (hereafter designated as B–F algorithm).

3. Algorithm description

Both variants of the proposed algorithm require a condition for detecting that the optimal pair of disjoint paths was obtained. Suppose that for each path p_w (sequentially generated by a k-shortest path sub-algorithm) the most reliable link disjoint path p_v was obtained, such that at any given step of the algorithm the only recorded pair of paths is the one with the highest $\Pr(p_w \cup p_v)$. Considering that the next (most reliable) path, generated by the k-shortest path sub-algorithm, to be selected in the main algorithm is p_i ($i > w$) such that

$$\Pr(p_i) + (1 - \Pr(p_i))\Pr(p_i) \leqslant \Pr(p_w) + (1 - \Pr(p_w))\Pr(p_v) \quad (4)$$

then (p_w, p_v) is the pair of paths with maximal reliability. The verification of this statement is straightforward. Let p_j be the most reliable path link disjoint with p_i, if $\Pr(p_j) \leqslant \Pr(p_i)$ then any other pair of paths obtained from this point onwards will always have reliability less than $\Pr(p_i) + (1 - \Pr(p_i))\Pr(p_i)$ therefore lower than $\Pr(p_w \cup p_v)$.

Having established the optimal stopping rule of the algorithm, its flowchart is presented in Fig. 1. The first variant was designated as RLDPC (Reliable Link Disjoint Pair of paths with length Constraints) and the second variant, using the B–F algorithm, will be designated as RLDPC-BF.

The main structure of the algorithm has two phases: to obtain the first pair of link disjoint paths with at most D arcs and to find and/or detect the optimal pair of paths. In the first phase the algorithm may terminate without finding a solution: no single link disjoint pair of paths with at most D arcs was identified. To check that no solution exists, the algorithm has (in phase 1) to generate all paths of length up to D and, for every such path, KD-ld1 or B–F will have to seek a disjoint path without success. Having completed phase 1 and having recorded a pair of link disjoint paths, the second phase of the algorithm consists of improving this solution (whenever possible) until either the recorded pair of paths is detected to be optimal, or no more (working) paths can be found. This last condition implies that the best recorded pair is in fact the optimal one.

The efficiency of RLDPC lies in the fact that KD-ld1 is a sub-algorithm that is based on KD, therefore capable of using efficiently the information generated by KD, as it will be explained in the next sub-section.

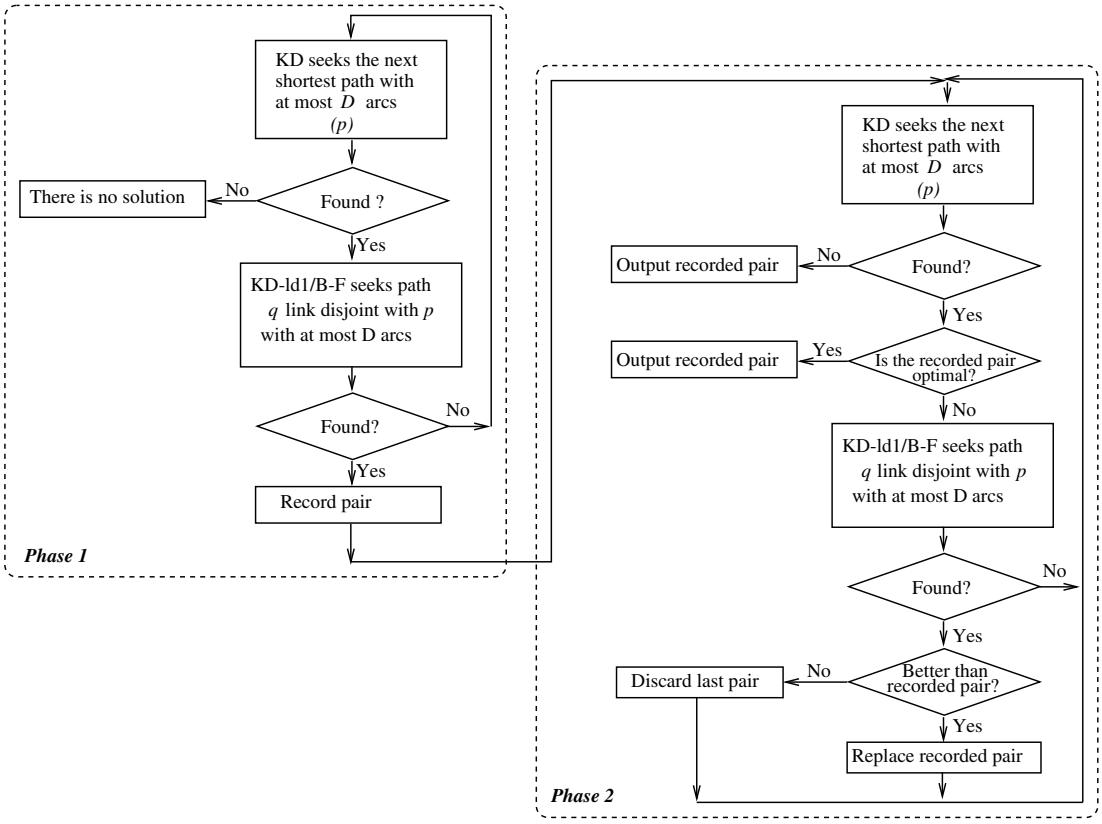

Fig. 1. Flowchart of proposed variants of the algorithm (RLDPC and RLDPC-BF).

3.1. Notation and definitions

Note that algorithm KD in [7] (which was derived from algorithm MPS [12]) and MPS are both 'deviation' algorithms. Each time a path p is chosen from a set of candidate paths, X, new paths may be added to X. Now some notation will be introduced that will be used for a brief review of KD.

In the context of the algorithm the node v_k of path p, from which a new candidate path is generated, is the *deviation node* of that new path (which coincides with p up to v_k). In a path the link the tail of which is the deviation node, is called the *deviation arc* of that path [12]. By definition s is the deviation node of p_1 (the shortest path from s to t). The *concatenation* of path p, from i to j, with path q, from j to l, is the path $p \Diamond q$, from i to l, which coincides with p from i to j and with q from j to l.

Let $\mathscr{T}_t$ designate a tree where there is a unique path from any node i to t (tree rooted at t as defined in [12]) and $\pi_i(\mathscr{T}_t)$ denote the cost of the path p, from i to t, in $\mathscr{T}_t$; the *reduced cost* $\bar{c}_{ij}$ of arc $(i,j) \in L$ associated with $\mathscr{T}_t$ is $\bar{c}_{ij} = \pi_j(\mathscr{T}_t) - \pi_i(\mathscr{T}_t) + c_{ij}$. So all arcs in $\mathscr{T}_t$ have a null reduced cost. The advantage of using reduced costs was first noted by Eppstein [4] and they are shown by Theorems 8 and 9 in [12] and by Theorem 2.1 in [11] (in the context of the MPS algorithm) to lead to less arithmetic operations and to sub-path generation simplification.

Let $\mathscr{T}_t^*$ be the tree of the shortest paths from all nodes to t and $p_{v_j t}^*$ the shortest path from v_j to t in $\mathscr{T}_t^*$. The sub-path from v_k to t in p is represented by $p_{v_k t}$, and the sub-path from s to v_k by p_{sv_k}. The set of arcs of L of $G = (N, L)$ is arranged in the *sorted forward star form*—for details see [3]. That is, the set L is sorted in such a way that, for any two arcs $(i,j),(k,l) \in L, (i,j) < (k,l)$ if $i < k$ or $(i = k$ and $\bar{c}_{ij} \leqslant \bar{c}_{kl})$.

Let $|p|$ denote the number of arcs in path p, from s to t, or *path length*. Let p be a path that contains nodes v_i and v_j. The *distance from v_i to v_j in p* will be given by the number of arcs in p, from v_i to v_j; if v_i, v_j are extreme nodes of an arc then the distance is 1; the distance of a node to itself is zero. Let p be a

path from s to t which contains v_i; the *depth of v_i in p* is given by the distance from s to v_i in p and will be designated by $d_p(v_i)$.

3.2. The main idea behind KD and KD-ld1 algorithms

In KD (and KD-ld1) every new path p' (from s to t), with deviation node v_i, added to X, deviates from a previously selected path p. The paths p and p' coincide from s up to v_i; let v_j be the head of the deviation arc of p', then $p'_{v_jt} = p^*_{v_jt}$. This is why one of the first steps of KD is to obtain $\mathscr{T}^*_t$. Also note that the ordering of L makes it easy to locate the deviation arc of a new path.

The basic idea of the KD algorithm is as follows. Let p be the shortest path in X and v its deviation node. Paths with deviation node at a depth at most $D-1$ will be obtained from p only if their deviation node v_i, $v_i \neq t$ and $v_i \in p_{vt}$ are such that $|p_{sv_i}| < D-1$ or, $|p_{sv_i}| = D-1$ and $p_{sv_i} \diamondsuit \langle v_i, (v_i,t), t\rangle \neq p$. Therefore paths of length greater than D will only be placed in X if their deviation node is at a depth less than D. The efficiency of KD results precisely from not generating paths of length greater than D which are of no interest because, when selected, they would be discarded and any path obtained from them would also certainly be discarded.

Let p_i be the ith path selected by KD and let X_i be a copy of the set of candidates paths X. Given $p_i, X_i, \mathscr{T}^*_t$ and L, KD-ld1 will start by marking all arcs in p_i as *used* (this simulates the removal of arcs of p_i from the network graph). This is followed by the selection of the shortest path from X_i. If the path selected from X_i is not the solution seeked by KD-ld1, it will possibly lead to the generation of new paths to be added to X_i. A path will only be added to the set X_i of candidate paths, firstly if it satisfies all rules of KD and, secondly, if at the time of the insertion in X_i none of its arcs is marked as *used*, or if from that path a new path disjoint with p_i may possibly be obtained. KD-ld1 efficiency results from not generating useless candidate paths to be added to X_i.

KD-ld1 ends when the selected shortest path in X_i is a feasible path disjoint with p_i or when X_i becomes empty. The *used* marks of the arcs of p_i are removed before KD-ld1 terminates (successfully or not). X_i will initially contain some inadequate paths (paths which are not disjoint with p_i and/or which will not originate any paths disjoint with p_i) but new paths added to X_i by KD-ld1 will not have that feature. The duplication of X_i, that may seem a heavy operation, is really a very simple procedure. Considering that the set X is represented by a heap of pointers to the candidate paths, duplicating X is simply the duplication of an array of dimension equal to the number of candidate paths.

In Appendix algorithms KD and KD-ld1 are described.

3.3. Using KD with the Bellman–Ford algorithm

The Bellman–Ford algorithm seemed a good alternative to KD-ld1, especially because, with Bellman–Ford, the effort of finding (for each candidate working path) a protection path is similar to the effort of detecting that no such path exists, whenever that is the case, and is approximately constant for networks with the same number of arcs, for D fixed [1].

After KD has found a candidate working path p_w, from s to t, all arcs in p_w will be marked as *used* (this simulates the removal of those arcs from the network graph). Then B–F is used to find a path with at most D arcs from s to t. If such a path cannot be found its cost will be ∞; the *used* marks in all arcs in p_w will have to be removed after running B–F. This new version of RLDPC will be designated by RLDPC-BF.

Each time KD generates a candidate working path p_i, KD-ld1, which is basically a k-shortest path enumeration algorithm, takes advantage of the information generated by KD (see Appendix B) to obtain p_v, the shortest path disjoint with candidate working path p_i. If a disjoint path with p_i exists, KD-ld1 may find it very rapidly (it is one of the first paths already in X_i) or if p_v is a path with very low reliability (it is at the bottom of X_i or still has to be generated), then it may take some time to obtain p_v. Therefore the CPU effort of KD-ld1 can vary significantly depending on the working path p_i; this effort can be especially high when no disjoint path p_v with p_i exists and a significant number of paths are already in X_i or will be added to X_i because they may potentially generate disjoint paths with p_i, with at most D arcs.

Having all this in mind, it will be expectable that RLDPC exhibits a higher variability, in CPU time, than RLDPC-BF.

3.4. Directed and undirected networks

If the graph $G' = (N, L)$ that represents a telecommunication network structure is undirected

RLDPC can still be used. Each undirected arc is replaced by two directed arcs in opposite directions and with the same cost as the original undirected arc. In the algorithm when marking all (undirected) arcs of p as *used* will now imply that, if $(v_i, v_j) \in p$ then both directed arcs (v_i, v_j) and (v_j, v_i), will be marked as *used*.

3.5. Complexity

Algorithm KD-ld worst case complexity (all paths with deviation node with depth at most $D-1$ are generated and picked from the set of candidate paths) is equal to $\mathcal{O}(D\delta^{D+2} + m\log n)$ [6], where m is the cardinal of set L and δ is the node maximum degree; the term $m\log n$ is due to obtaining $\mathscr{T}_t^*$, changing the costs and rearranging the links in the sorted forward star form. In KD-ld1 this term does not apply because the algorithm uses $\mathscr{T}_t^*$ and the sorted forward star form created by KD. Therefore KD-ld1 worst case complexity is simply $\mathcal{O}(D\delta^{D+2})$. The B–F algorithm implementation used in this context has worst case complexity $\mathcal{O}(Dm)$ [1].

The KD algorithm is a variant of MPS for loopless paths with length constraints. According to the authors of MPS the complexity of this algorithm is hard to calculate and its worst case complexity (only for particular values of the arcs costs and network topology) is $\mathcal{O}(n!)$, where n is the cardinality of N [15]. This complexity results from considering that all loopless paths from the initial node to any other node (except a particular one) have to be generated and added to X before obtaining the second shortest loopless path. As KD only generates paths with deviation with depth at most $D-1$, in a worst case scenario its complexity is equal to KD-ld complexity.

Therefore the total complexity of the algorithm is,

$$\mathcal{O}((D\delta^{D+2} + m\log n)^2\delta^{D+1}) \quad \text{or} \quad \mathcal{O}((D\delta^{D+2} + m\log n)Dm\delta^{D+1}) \tag{5}$$

where the second expression applies to the variant using B–F and δ^{D+1} is an upper bound to the total number of paths with at most D arcs that can be generated by KD in the search for the optimal stopping condition and/or stopping rule. In fact, in the worst case all paths with deviation node with depth at most $D-1$ will be computed, and these are $\sum_{i=0}^{D-1}\delta(\delta-1)^i \leqslant \delta^{D+1}$.

4. Experimental results

Results are presented for undirected networks, with low connectivity, as indicated in Table 1. This type of features is common in WDM optical networks. For each number of nodes n, 10 different networks were randomly generated[1] with the same number of arcs and nodes; the arc reliabilities were randomly generated in $[1-5\times10^{-4}, 1-10^{-6}]$. Two different network densities were used: $m = 2n$ and $m = 3n$. The maximum number of arcs considered, per path, was $D = 8$, 9 for the networks with $m = 3n$ and $D = 12$, 13 for networks with $m = 2n$. The values of D were chosen such that they were equal (or greater) than $\max d(G)$ ($d(G)$ is the network diameter of graph G) for each of the two types of networks ($m = 2n, 3n$).

Due to the low network connectivity great variation in CPU time used by RLDPC (RLDPC-BF) was observed depending on the s–t pair. Therefore for each network a pair of disjoint paths was seeked for all $(n\times(n-1))$ node pairs[2] and the average CPU time obtained per pair of disjoint paths for each node pair was calculated. Algorithm KD starts by building $\mathscr{T}_t^*$ (the tree of the shortest paths from all nodes to t); this structure can be re-used for different node pairs with the same end node t; therefore node pairs were generated by changing s for each t.

There are situations where no solution will be found (no feasible disjoint protection path exists for each feasible working path); also, after the identification of a pair of disjoint paths, the optimality condition may be hard to check (many new pairs have to be seeked). Both these situations may lead to excessive CPU time usage. Therefore a limiting mechanism must be provided. For both variants of the algorithm two control mechanisms were implemented to prevent it from using too much CPU time. Considering that phase 1 was successful and that the first disjoint pair was obtained, pairs of paths will be generated until the stopping condition is satisfied or a maximum CPU time (maxCPU) is used. Taking into account that exiting phase 1 without finding a single pair (because the permitted CPU time was exceeded) is more disruptive than exiting without satisfying the optimal stopping condition,

[1] The used program for network generation was kindly borrowed from José Luis Santos.

[2] Due to the nature of KD and KD-ld1 the cost of obtaining the optimal disjoint pair from s to t and from t to s is not identical.

Table 1
Test networks, where n is the number of nodes, m the number of arcs, $d(G)$ is the network diameter and $\bar{d}(u,v)$ the average node distance

n		50	100	150	200	250	300	350	400	450	500
$m=2n$	$d(G)$	5–8	7–10	8–9	8–10	9–11	9–11	9–11	9–11	10–11	10–12
	$\bar{d}(u,v)$	4.2	4.7	5.1	5.2	5.4	5.5	5.5	5.7	5.9	5.9
$m=3n$	$d(G)$	4–6	5–6	6–7	6–7	6–7	6–8	7–8	7–8	7–8	7–8
	$\bar{d}(u,v)$	3.5	3.9	4.1	4.3	4.4	4.5	4.6	4.7	4.7	4.8

a CPU time per pair of paths equal to $2 \times \text{maxCPU}$ was allowed for phase 1.

The previous conditions imply that if the first pair selection is successful in phase 1 but uses more than maxCPU time of CPU, then phase 2 will not be executed and the algorithm will exit with a single (possibly) non-optimal path. A value of 125 milliseconds was used for maxCPU in a Bi-Pentium III at 833 MHz with 512 MB of memory, under Linux. In Fig. 2 the CPU time per pair of disjoint paths (non-existing, optimal and possibly sub-optimal) is presented. These average values are significantly below the upper bound of 125 milliseconds which is attained only for node pairs where only sub-optimal solutions were found or for some node pairs for which no solution could be found. The behaviour of the algorithms concerning the frequency of exits due to CPU time usage is also shown in Fig. 4. In Figs. 2–4 an error bar was added, centred on the average value (μ) of the collected sample of values (one sample per network) such that the bar goes from $\max(0, \mu - s)$ to $\mu + s$, where s is the sample standard deviation. The purpose of this bar is to show that using B–F leads to less variability in the results (% of terminations due to CPU time usage and average CPU time per node pair). This variability pattern should be expected: the B–F algorithm requires a number of iterations equal to D while the number of paths generated by KD-ld1, although it depends on D, it also depends on the network topology and the node pair.

In the case of networks with $m = 2n$ and maxCPU = 125 milliseconds we obtained a worst case value of node pairs for which a sub-optimal solution (or no solution) was found, equal to 7.7% (7.5%) for RLDPC (RLDPC-BF). Nevertheless RLDPC-BF exits in average only in about 0.2% of the cases which is much less than 1.8%, the value obtained for RLDPC. Also the average values of frequency of terminations due to lack of CPU time, for each

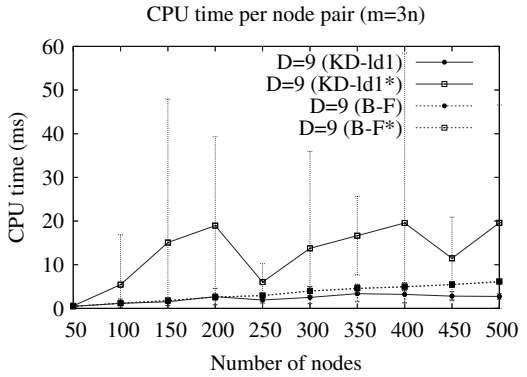

Fig. 3. Average CPU time per node pair in RLDPC with long run time (KD-ld1*) and RLDPC-BF with long run time (B–F*).

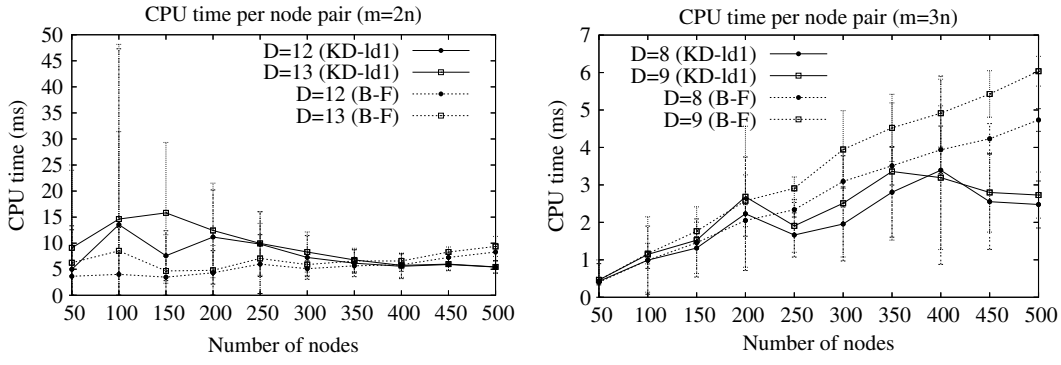

Fig. 2. Average CPU time per pair of nodes in the networks.

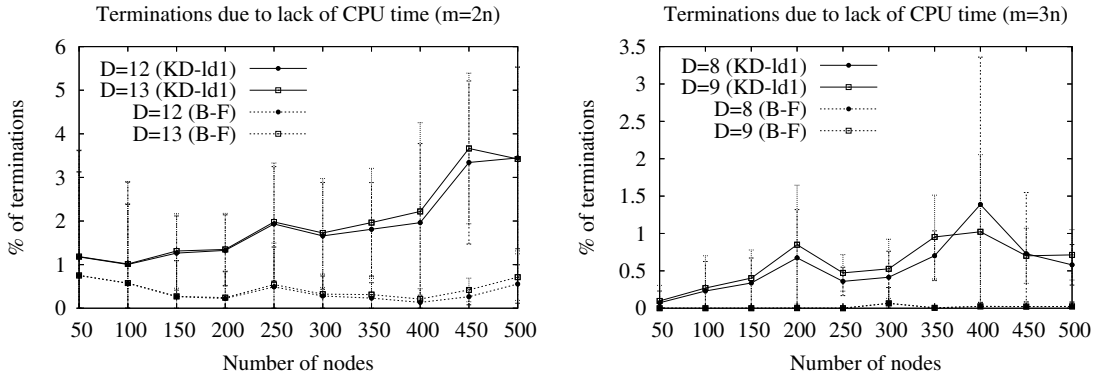

Fig. 4. Percentage of node pairs for which the algorithms RLDPC (KD-ld1) and RLDPC-BF (B–F) exit due to lack of CPU time.

type of network presented in Fig. 4 have a maximum of 3.7% and 0.75% for RLDPC and RLDPC-BF, respectively.

Regarding CPU time per node pair, apparently RLDPC performs better than RLDPC-BF, for more dense networks ($m = 3n$), and its relative performance apparently also improves with the dimension of the network, as shown in Fig. 2. In less dense networks ($m = 2n$) RLDPC-BF seems to be more efficient than RLDPC, for smaller values of n, but as the number of nodes in the network increases it seems to become less efficient. We have said "apparently" because these results are in fact misleading: RLDPC-BF uses more CPU time per node pair but it also obtains more optimal solutions than RLDPC.

Looking at the CPU time per node pair for both algorithms (in individual networks) we see that in fact RLDPC, for those node pairs for which an optimal solution was found, used in average less time than RLDPC-BF. However the number of node pairs for which RLDPC could not find a solution is significantly high compared to RLDPC-BF. For such pairs, if the allowed CPU time was substantially increased the CPU times required by RLDPC would be very high and this would invert the average relative performance of both variants of the algorithm. As an example consider the case $n = 350$, $m = 3n$, $D = 9$, where RLDPC obtained in average, for the 10 tested networks, 1083 sub-optimal solutions and found no solution for 81 node pairs (due to CPU time exhaustion). RLDPC-BF, for the same set of 10 networks, found solutions for all node pairs and exited with an average of 10 sub-optimal solutions per network.

Therefore, although RLDPC is more efficient than RLDPC-BF for most node pairs of a network, there is a small number of node pairs for which it performs rather poorly. To verify this conclusion long runs of RLDPC and RLDPC-BF were executed: the two versions of the algorithm were allowed to run without CPU time limit for $D = 9$ and $m = 3n$. Observing Fig. 3 the relative behaviour of the two versions was inverted. As an example consider $n = 500$, $m = 3n$, $D = 9$: RLDPC found 1644 sub-optimal solutions and failed to find a solution for 138 node pairs (in average per network) with a CPU time per node pair of 2.8 milliseconds and RLDPC-BF did not find an optimal solution in average for only 121 node pairs and failed for 1 node pair using in average 6.0 milliseconds per node pair, when maxCPU was 125 milliseconds; with unlimited CPU time, RLDPC required 19.6 milliseconds and RLDPC-BF 6.2 milliseconds per node pair to obtain all optimal solutions.

Although RLDPC/RLDPC-BF sometimes need a significant number of pairs of paths to be obtained in order to check the optimality condition, a frequency counter of the sequential order of the selected (optimal) pair of paths indicates that this pair is one of the first four pairs in almost all cases. Also, in a great percentage of the cases, the first identified pair was the optimal one.

5. Conclusions

Two variants of an efficient algorithm for obtaining the most reliable pair of link disjoint paths with length constraints, have been presented. Also a worst case complexity analysis of the algorithm was performed. The performance of the proposed versions of the algorithm was evaluated through numerous experiments for randomly generated networks with low connectivity. This experimental environment is of the type encountered in WDM

optical transmission networks. These tests enabled to put in evidence the efficiency of the algorithmic approach as well as its limitations which stem from the nature of the addressed problem. In particular if between end nodes s and t no feasible pair of disjoint paths exists, the algorithm will have to generate all paths with at most D arcs form s to t and, for each of these paths, it will seek to obtain a disjoint pair. This procedure may become too heavy in terms of CPU time and therefore a limiting mechanism was introduced.

The comparison of the two proposed implementations indicates that the version which uses Bellman–Ford algorithm for finding the disjoint path appears to be advantageous when compared to the version which uses KD-ld1, because it has a more stable performance and obtains a higher number of optimal solutions with no significant increase in CPU time.

Acknowledgements

Work partially supported by programme POSI of the III EC programme cosponsored by FEDER and national funds.

Appendix A. KD Algorithm

1. Input: the representation of the graph (N, L) and arc costs c_{ij}.
2. By using Dijkstra inverse algorithm (including the node heights calculations), obtain the shortest paths from every node to t, hence constructing the *shortest tree rooted at* $t, \mathscr{T}_t^*$.
3. Calculate the reduced cost $\bar{c}_{ij}$ for every $(i,j) \in L$.
4. Rearrange the arcs of (N, L) in the sorted forward star form (for the computed $\bar{c}_{ij}$, leading to $L = \{a_1, \ldots, a_m\}$ such that for any $h \in \{1, \ldots, m-1\}, \bar{c}_{a_h} \leqslant \bar{c}_{a_{h+1}}$ if $v = \text{tail}(a_h) = \text{tail}(a_{a_{h+1}})$)
5. $p \leftarrow$ shortest path from s to t ($p \in \mathscr{T}_t^*$).
6. $k \leftarrow 0$
7. $X \leftarrow \{p\}$ (X is the set of paths that are candidates to shortest path).
8. `While` ($k < K$) and ($X \neq \emptyset$) `do`
   (a) $p \leftarrow$ shortest path in X
   (b) $X \leftarrow X - \{p\}$
   (c) `If` $|p| \leqslant D$ and p has no loops `then`
      i. $k \leftarrow k + 1$
      ii. $p_k \leftarrow p$ (the kth path which has at most D arcs, was found)
   `EndIf`
   (d) Let v_i be the deviation node of p
   (e) `If` The length p and depth of v_i are adequate `then` (possibly new paths will be placed in X)
      i. $l \leftarrow$ maximum depth of deviation nodes of possible new paths
      ii. `Repeat`
         A. $a_h \leftarrow$ the arc of p the tail of which is v_i
         B. $p_{sv_i} \leftarrow$ sub-path of p from s to v_i
         C. `While` (v_i is the tail of a_{h+1}) and (a_{h+1} forms a loop with p_{sv_i}) `do`
            $h \leftarrow h + 1$
            `EndWhile`
         D. `If` v_i is the tail of a_{h+1} `then`
            - $v_j \leftarrow$ head of a_{h+1}
            - $X \leftarrow X \cup \{p_{sv_i} \diamondsuit \langle v_i, a_{h+1}, v_j \rangle \diamondsuit p_{v_j t}^*\}$

            `EndIf`
         E. `If` $d_p(v_i) = l$
            `then` $l \leftarrow -1$ (no more paths will be obtained from p)
            `else` $v_i \leftarrow$ following node in ${}^l p$ (truncated path p after l arcs)
         `EndIf`
      `Until` (p_{sv_i} has a loop) or ($l = -1$)
      `EndIf`

`EndWhile`

Appendix B. KD-ld1 Algorithm

The input of KD-ld1 is: the representation of the graph (N, L) and arc costs $c_{ij}, \mathscr{T}_t^*$. $\bar{c}_{ij}$ for every $(i,j) \in L$, the arcs of (N, L) in the sorted forward star form (for the computed $\bar{c}_{ij}$), the path p_y and X_y. p_y is the yth path obtained by KD for which KD-ld1 seeks a disjoint path; X_y is a copy of X after p_y was selected (all this input is provided by KD—see Section 3.2).

The first action of KD-ld1 is marking *used* all the arcs in p_y. If a path p_x selected from X_y is disjoint with p_y and has at most D arcs the algorithm terminates returning p_x (after unmarking all arc of p_y).

For each selected path p from X_y which is not the solution seeked by KD-ld1, new paths, deviating from p will only be added to X_i if p_{sv} has no arc marked *used* (where v is the deviation node of p).

The maximum depth of deviation nodes (l) in KD-ld1 is calculated using the rules of KD, and also ensuring that no arc in ${}^l p$ is marked *used*. Having said this, the generation of paths to be added to X_y in KD-ld1 is similar to the cycle in step 8(e)ii in KD, with the condition in step 8(e)iiC rewritten:

"(v_i is the tail of a_{h+1}) and [(a_{h+1} is *used*) or (a_{h+1} forms a loop with p_{sv_i})]".

References

[1] D. Bertsekas, R. Gallager, Data Networks, Prentice-Hall, 1992.

[2] R. Bhandari, Survivable Networks, Algorithms for Diverse Routing, Kluwer Academic Publishers, 1999.

[3] R. Dial, F. Glover, D. Karney, D. Klingman, A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees, Networks 9 (1979) 215–348.

[4] D. Eppstein, Finding the k shortest paths, SIAM Journal on Computing 28 (1998) 652–673.

[5] T. Gomes, J. Craveirinha, An algorithm for obtaining the most reliable pair of link diskoint paths with length constraints, in: W. Ben-Ameur, A. Petrowski (Eds.), International Network Optimization Conference (INOC'2003), INT & get, 2003, pp. 248–253.

[6] T. Gomes, J. Craveirinha, L. Martins, M. Pascoal, An algorithm for calculating the k most reliable link disjoint paths with a maximum number of arcs in each path, in: T. Cinkler (Ed.), Proceedings of Design of Reliable Communication Networks (DCRN 2001), 2001, pp. 205–212.

[7] T. Gomes, L. Martins, J. Craveirinha, An algorithm for calculating the k shortest paths with a maximum number of arcs, Investigação Operacional 21 (2) (2001) 235–244.

[8] A. Itai, Y. Pearl, Y. Shiloach, The complexity of finding maximum disjoint paths with length constraints, Networks (1982) 277–286.

[9] P. Laborczi, J. Tapolcai, P.-H. Ho, T. Cinkler, A. Recski, H.T. Mouftah, Algorithms for asymmetrically weighted pair of disjoint paths in survivable networks, in: T. Cinkler (Ed.), Proceedings of Design of Reliable Communication Networks (DCRN 2001), 2001, pp. 213–219.

[10] C.-L. Li, S.T. McCormick, S.-L. David, Finding disjoint paths with different path costs: Complexity and algorithms, Networks 22 (1992) 653–667.

[11] E. Martins, M. Pascoal, J. Santos, An algorithm for ranking loopless paths, Technical Report 99/007, CISUC, 1999. Available from: <http://www.mat.uc.pt/~marta/Publicacoes/mps2.ps>.

[12] E. Martins, M. Pascoal, J. Santos, Deviation algorithms for ranking shortest paths, International Journal of Foundations of Computer Science 10 (3) (1999) 247–263.

[13] H.T. Mouftah, P.-H. Ho, Optical networks—architecture and survivability, Kluwer Academic Publishers, 2003.

[14] P. Papadimitratos, Z.J. Hass, E.G. Sirer, Path selection in mobile ad doc networks, in: Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking and Computing, ACM Press, 2002, pp. 1–11.

[15] M.M.B. Pascoal, J.L.E. Santos, An algorithm for ranking loopless paths, 2000. Available from: <www.mat.uc.pt/~marta/Publicacoes/mps3.ps.gz>.

[16] V. Sharma, F. Hellstrand, B. Mack-Crane, S. Makam, K. Owens, C. Huang, J. Weil, B. Cain, L. Anderson, B. Jamoussi, A. Chiu, S. Civanlar, Framework for multi-protocol label switching (MPLS)-based recovery, IETF RFC 3469 (2003).

[17] J.W. Suurballe, Disjoint paths in networks, Networks 4 (1974) 125–145.

[18] J.W. Suurballe, R.E. Tarjan, A quick method for finding shortest pairs of disjoint paths, Networks 14 (2) (1984) 325–336.

[19] H. Zang, J.P. Pue, B. Mukherjee, A review of routing and wavelength assignment approaches for wavelength routed optical WDM networks, Optical Networks Magazine 1 (1) (2000) 47–60.