IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

The disjoint multipath challenge: multiple disjoint paths guaranteeing scalability

DIEGO LOPEZ-PAJARES[1,2], ELISA ROJAS[1], JUAN A. CARRAL[1], ISAIAS MARTINEZ-YELMO[1] and JOAQUIN ALVAREZ-HORCAJO[1]

[1]Universidad de Alcalá, Departamento de Automática, 28805, Madrid, España
[2]Universidad Politécnica de Madrid, Departamento de Ingeniería de Sistemas Telemáticos, 28040, Madrid, España

Corresponding author: Elisa Rojas (e-mail: elisa.rojas@uah.es).

This work was funded by grants from Comunidad de Madrid through project TAPIR-CM (S2018/TCS-4496) and project IRIS-CM (CM/JIN/2019-039), by Junta de Comunidades de Castilla-La Mancha through project IRIS-JCCM (SBPLY/19/180501/000324), and from University of Alcalá through "Programa de Formación del Profesorado Universitario (FPU)"

ABSTRACT The multipath challenge is a research line in continuous development because of its multiple benefits, however, these benefits are overshadowed by scalability, which goes down considerably when the paths are multiple and disjoint. The disjointness aggregates an extra value to the multiple paths, but it also implies more complex mathematical operations that increase the computational cost. In fact, diverse proposals exist that try to increase scalability by limiting the number of paths obtained to the minimum possible (two-disjoint paths), which is enough for backup applications but not for other purposes. This paper presents an algorithm that solves these drawbacks by discovering multiple disjoint paths among multiple nodes in an efficient way, while keeping bounded the computational cost and ensuring scalability. The proposed algorithm has been validated thoroughly by performing a theoretical analysis, bolstered afterwards by an exhaustive experimental evaluation. The collected results are promising, our algorithm reduces the time spent to obtain the disjoint paths regarding its competitors between one and three orders of magnitude, at the cost of a slight decrease in the number of paths discovered.

INDEX TERMS Algorithms, Disjoint, Multipath, Graph theory, Dijkstra's algorithm, Routing

I. INTRODUCTION

The shortest path search is a well-known topic in graph theory, being Dijkstra's algorithm [1] the most renowned solution. It has been extensively used as the basis for routing protocols in data networks over the last 60 years. Not only it is still in use in current telecommunications systems to route data flows through the network [2]–[4], but is also used in newer applications such as digital Google Maps services [5], robot planning routes [6], or even in medical problems [7].

The shortest path problem evolved into an even more intricated challenge: to obtain the shortest path plus one (or more) additional disjoint paths to the previous one, being Suurballe and Tarjan [8] the first ones to achieve it, based, precisely, on Dijkstra's algorithm. In fact, just in the last five years several authors have worked on the basis of Dijkstra's algorithm to obtain multiple disjoint paths, either by modifying it or by the application of heuristics, hence adapting the original algorithm to the new era [9]–[16]. The design objectives of these research works are diverse. Numerous proposals restrict the disjoint path search to only a pair of nodes, while others also limit the search to two-disjoint paths exclusively, which is not enough for the latest application requirements. Besides, scalability remains a pending issue despite the performance improvements introduced by the newest advances in multipath computing methods, such as new heuristics algorithms or the Integral Linear Programming (ILP) method. Thereby, the multiple disjoint paths problem still constitutes an intriguing area of research for society due to its numerous benefits (such as enhanced robustness against failures, enhanced computation in machine learning applications, improved resource allocation or increased security), whose main application field is usually computer network routing, both in traditional networks and Software-Defined Networking (SDN).

Our proposal, Multiple Disjoint Path algorithm (MDPAlg), is able to obtain multiple disjoint paths among a given node and the remaining nodes in a graph. It works as a two-phase

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

process. The first phase performs a cost analysis following the same principle as Dijkstra's shortest path algorithm. However, unlike Dijkstra's algorithm, which focuses on collecting information about the minimum-cost tree, MDPAlg gathers cost information about the entire graph (including cross-links) with a similar computational effort. Thereafter, the second phase leverages this extra information to build multiple disjoint paths, thus avoiding iterative executions when more than one path is required as in Dijkstra and other alike algorithms. Moreover, this methodology keeps the number of mathematical operations bounded, guaranteeing the scalability of the algorithm in large graphs.

In summary, the main contributions of this paper are as follows:

- We describe MDPAlg, a novel algorithm to create multiple disjoint paths among a given node and the remaining nodes in a graph with a single full graph search.
- We analyze, implement and comprehensively evaluate MDPAlg against its direct competitor: Dijkstra's algorithm (as well as some of its enhanced versions), obtaining a drastic reduction (up to three orders of magnitude) of the computational complexity, independently of the graph type. To the best of our knowledge, no other state-of-the-art algorithm yields similar performance results.
- Thanks to a comprehensive evaluation, we prove that MDPAlg can be applied to a multitude of classic engineering problems (including computing, networking, transport, etc.).

The paper is structured as follows: Section II establishes the background of the paper and its justification, while Section III formally defines the proposal and describes its behavior with an example. Afterward, Section IV studies the computational complexity of the algorithm and confronts it to widely-known solutions, followed by Section V, which comprehensively evaluates the algorithm in different scenarios. Finally, Section VI provides the main conclusions.

II. BACKGROUND

A graph is composed by a set of objects called nodes (or vertices) connected by a set of links (or edges) that characterizes relations between such elements, whose study dates back to the century XVIII (Könisgberg problem). According to the directionality of the links, graphs can be classified as directed graphs (each link has associated one or more directions to be traversed), and non-directed graphs (the direction of the links are not defined and they can be used in both directions). Moreover, depending on the cost of traversing those links, the graphs can be classified as weighted graphs (each link has its own weight cost) or unweighted graphs (links have no weight).

There are two main search algorithms for non-directed and unweighted graphs, Depth First Search (DFS) and Breadth First Search (BFS) [17], which obtain as a result a tree rooted at a given node that spans all the nodes in the graph. DFS performs a graph search from top to bottom, exploring a branch of the tree as far as possible until the node processed is the node searched for or it has no children, backtracking then to explore other branches; while BFS performs a breadth search, visiting all the nodes at the same hop distance before advancing to the next depth level. Based on BFS, Dijkstra's algorithm [1] introduces, as a novelty, a search process on non-directed weighted graphs that replaces the breadth search of hop-levels by a cost-level breadth search, resulting in a minimum-cost predecessor matrix for the root node. Moreover, from this information, it is straightforward to derive the minimum-cost path from the root node towards the remaining nodes. However, this solution only provides one minimum-cost path for each pair of nodes. Thus, to obtain multiple paths between the same pair of nodes, the process should be repeated after deleting the previous discovered path (set of links) from the graph.

Regarding the number of paths discovered, algorithms can be classified as single-path (they only provide a path between two nodes), and multiple-path (they provide more than one path between two nodes). Multiple paths not only can be used for routing purposes (as alternative routes to send data), but also for back-up (just in case of failure of the main path), security and so on. Within the multiple-path group, path disjointness is an interesting feature that clearly improves the benefits of multiple-path approaches. According to this property, multiple-path algorithms can be classified as link-disjoint (the paths obtained cannot share any link) and node-disjoint (the paths cannot share any node, and hence any link, except for the end nodes). Additionally, we can group multiple-path algorithms depending on whether they provide two or more than two paths. For example, the following proposals only provide two disjoint paths: [8] (for directed weighted graphs), [18], [19] (for non-directed weighted graphs), [20]–[22] (for non-directed unweighted graphs).

Whereas having two disjoint paths may be enough to ensure reliability or for back-up purposes in some applications, it might remain insufficient in other scenarios, such as to increase the robustness of the connections against *Man-In-the-Middle* attacks through data diversification across disjoint routes [23], or to improve the bandwidth speed by using disjoint paths with Quality of Service (QoS) policies [15]. However, the calculation of multiple disjoint paths is usually a greedy process and its computational complexity –defined as the amount of resources required to run the algorithm– grows exponentially with the graph size (number of nodes and links), which limits its scalability. Aiming to reduce this high computational cost, some approaches rely on distributing the computation among all the nodes in a graph by exchanging local information between neighbors nodes to build the disjoint paths, namely: [24]–[26] (for non-directed and unweighted graphs), and [27]–[29] (for non-directed and weighted graphs).

Also based on this distributed approach, the One-Shot Multiple Disjoint Paths (1S-MDP) [30] network protocol obtains multiple link- or node-disjoint paths among a given node and any other nodes in the network, boosting the path search process efficiency and guaranteeing its scalability

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

in large networks. However, due to its distributed nature, 1S-MDP cannot be applied in many fields such as vehicular traffic routing, emergency evacuation systems and others, where its impact could be significant. Thus, we propose the transformation of 1S-MDP into MDPAlg, a centralized version of the former, designed to adapt it to new challenges and encompassing it for as many application fields as possible. MDPAlg works in non-directed weighted graphs and provides multiple link and node-disjoint paths between a target node and other nodes in the graph.

III. ONE-SHOT MULTIPLE DISJOINT PATH ALGORITHM

This section describes in detail the procedure followed by MDPAlg to obtain multiple link- or node-disjoint paths among multiples nodes. First, it provides a high-level overview of the algorithm and the reasons for its development, focusing later on the in-depth description of its multipath discovery procedure.

A. ALGORITHM DESCRIPTION

MDPAlg is conceived as a centralized and enhanced version of 1S-MDP [30], which already showed good results in terms of number of disjoint paths discovered, and low convergence time to obtain them, in distributed environments. The rationale behind this decision is that 1S-MDP is potentially applicable to many scenarios of diverse nature, but its distributed essence hinders its practical implementation on some of them. In particular, MDPAlg is designed as a centralized algorithm that, instead of exchanging topological information in a distributed manner, operates over a graph that symbolically represents the underlying topology. Like 1S-MDP, MDPAlg is able to obtain multiple disjoint paths among a given node and the remaining nodes in a graph, following a two-phase process, with just a single full graph search.

During the first phase, MDPAlg carries out a cost analysis on the graph starting from the given source node (*s*), much like Dijkstra's algorithm search strategy but storing some extra information. For each target node, it stores the aggregated cost from *s* computed through all its neighbours (not only the minimum cost as in Dijkstra) within a cost matrix (see Section III-B). The second phase derives multiple link- or node-disjoint paths among *s* and any other node in the graph from the information stored in the cost matrix (see Section III-C). Additionally, the set of target nodes can be configured by the user, including the possibility to mark as target one single node or up to all nodes of the graph except *s*.

B. FIRST PHASE: ANALYSIS OF COSTS

During this phase, the algorithm performs an analysis to obtain the aggregated cost incurred to go from *s* to any other node in the graph. This analysis is a modified version of Dijkstra's algorithm in which a graph node not only collects information from the minimum-cost tree, but from all its neighbors (including cross-links). By using this extra information, MDPAlg is able to build multiple disjoint paths, hence avoiding iterative executions when more than one path is required, as in Dijkstra's algorithm.

Algorithms 1 and 2, shown in parallel, present the pseudo-code of Dijkstra and MDPAlg, respectively. They are based on the high-level description of Dijkstra's algorithm from [17] so that it is easier to spot the main differences between them. Dijkstra's algorithm relies on two vectors, *C* and *P*, to store the minimum cost from *s* to any other node, and the parent node through which the minimum cost was computed, respectively. From this information, obtaining the paths included in the minimum-cost spanning tree for a given node is straightforward, but still requires a second phase to build them, and only provides a single path for each *s*-*X* pair in the graph. On the other hand, MDPAlg relies on a more complex data structure, named the Cost Matrix (CM) for node *s*, to store the aggregated cost from *s* to any other node, computed through all its neighbors (not only the minimum cost as in Dijkstra). CM is a non-complete NxN cell matrix (being N the number of nodes in the graph); an empty cell meaning that those particular two nodes are not neighbors. For example, for every node *X*, neighbor of node *Y*, row ***X*** column ***Y*** shows the accumulated cost to go from *s* to *Y* through the minimum cost path from *s* to *X* plus the cost from *X* to *Y*; hence, row ***X*** column ***X*** shows the cost associated to go from *s* to node *X* (plus the cost from *X* to *X*, which is zero), through the minimum cost path. From this information, the algorithm builds multiple disjoint paths between *s* and any other graph node in the second phase.

It is important to note that MDPAlg obtains multiple disjoint paths between *s* and any other/s selected node/s with a single full graph search, while Dijkstra's algorithm only obtains one. A new execution of Dijkstra's algorithm is required to obtain one additional disjoint path between *s* and another node (after removing the previous path from the graph). For example, in a graph of N nodes, if the objective is to compute p disjoint paths between s and every other node, we have to execute Dijkstra´s algorithm $p \cdot (N-1)$ times, while MDPAlg just needs to be executed once. Hence, several iterative executions of Dijkstra's algorithm are required to yield similar results than MDPAlg.

Comparing Dijkstra's algorithm and the analysis of costs phase of MDPAlg, both functions receive a graph (*G*) and a given source node (*s*) as input parameters, which provide a representation of the underlying topology (composed by nodes, links and the cost of traversing each link), and the point where the algorithm procedure starts, respectively. First, both functions initialize their variables, *S*, *Q*, *P* and *C* vectors in Dijkstra's algorithm function; *CM* and *Q* in MDPAlg. The vectors *S* and *Q* contain the nodes analyzed by the algorithm and the remaining nodes to be analyzed, respectively, while *P* stores the parent of each node in the minimum-cost tree. Moreover, both algorithms initialize the vector *C* and the matrix *CM* with an infinite value in all of their entries, except for the one associated with *s*, set up with zero cost, which ensures that the algorithms start the

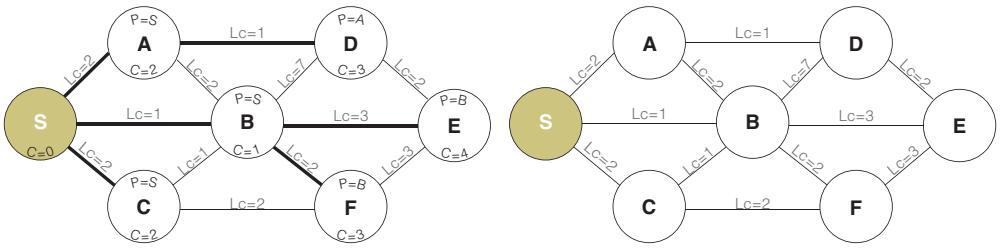
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

Algorithm 1 Dijkstra's algorithm

```
1:  function DIJKSTA'SALGORITHM(G, s)
2:      Initialize(S, P)
3:      C = Initialize_single_source(G, s)
4:      Q = Get nodes from G
5:      while Q ≠ ∅ do
6:          u = extract_min(Q)
7:          Insert u in S
8:          for each vertex v neighbour of u do
                R { if S ∩ v = ∅ then
                      w = L_c from u to v
9:                    if C(u) + w < C(v) then
                          C(v) = C(u) + w
                          P(v) = u
10:     return C, P
```

Algorithm 2 Analysis of costs of MDPAlg

```
1:  function ANALISISOFCOSTS(G, s)
2:
3:      CM = Initialize_single_source(G, s)
4:      Q = Get nodes from G
5:      while Q ≠ ∅ do
6:          u = extract_min(Q)
7:
8:          for each vertex v neighbour of u do
                R { if v ≠ s then
                      w = L_c from u to v
9:                    if CM[u][u] + w < CM[v][v] then
                          CM[v][v] = CM[u][u] + w
                      CM[u][v] = CM[u][u] + w
10:     return CM
```



Node	S	A	B	C	D	E	F
P	-	S	S	S	A	B	B
C	0	2	1	2	3	4	3

a) Dijkstra's algorithm result

	S	A	B	C	D	E	F
S		2	1	2			
A		2	4		3		
B		3	1	2	8	4	3
C			3	2			4
D		4	10		3	5	
E			7		6	4	7
F			5	5		6	3

b) MDPAlg result

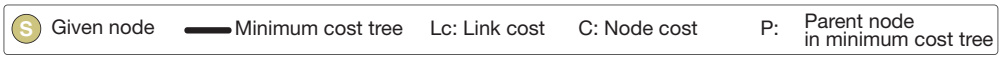


FIGURE 1: Comparision of the search process in Dijkstra's algorithm and MDPAlg

analysis procedure at this node. Later, both proposals analyze each node of the graph once, following a selection criteria by cost, that extracts, in each iteration, the node with the lowest cost from Q. Subsequently, Dijkstra´s algorithm applies the *RELAX* function, as originally described in [17], to compute the cumulative cost incurred to go from s to the target node through all of its neighbors and selects the neighbor that provides the lowest cost.

MDPAlg modifies the *RELAX* function to obtain extra information about the cross-links that Dijkstra's algorithm does not consider relevant. With this modification and the *CM* structure, MDPAlg obtains a complete characterization of the graph according to the accumulated cost incurred from s, also including the minimum-cost path in the solution. The changes in the *RELAX* function consist of computing the accumulated cost from the given node to the rest of the nodes through all the links, so that it stores the cost incurred to get to the target node through all its neighbors, not only through the best one. Therefore, although applying a similar analysis procedure, MDPAlg stores more information than Dikjstra´s algorithm about the graph.

Figure 1 provides a graphical example illustrating the operation of Dijkstra's algorithm and MDPAlg in a simple graph comprised of 7 nodes. It is vertically divided into two parts, the top half shows the graph composed of nodes, links, and the associated cost to each link, together with additional data related to both algorithms; while the bottom half shows the data structures returned by each algorithm. In particular, Figure 1a displays the result of applying Dijkstra's algorithm. As previously anticipated, the top half shows, for each node, its name, the minimum cost from s and the neighbour (parent) used to compute this minimum cost. From this information we can compute the minimum-cost tree spanning from s, which is also highlighted in the figure in bold black color

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

lines. The bottom half depicts the Cost vector (*C*) and the Parent vector (*P*) returned by the algorithm.

Similarly, Figure 1b shows the result of executing MDPAlg in the example graph, representing the returned Cost Matrix (*CM*) returned in the bottom half. The information highlighted in gray in the matrix represents information not strictly relevant or necessary for MDPAlg, as the first column symbolizes the cost to travel from *s* to itself, and the diagonal values contain the information about the minimum-cost tree, just as vector *C* in Dijkstra's algorithm; as it can be observed, these diagonal values are duplicated and equivalent to the lower-cost cell of each column. As a proof of this, in Figure 1b, the diagonal of *CM* is equal to vector *C* shown in Figure 1a. For further elucidation, row ***E*** column ***E*** contains the same cost value than ***B*** column ***E***, which directly means that the minimum-cost path from *s* to *E* is reached through *B*. This can be extrapolated to the rest of the columns of *CM*.

C. SECOND PHASE: DISJOINT PATH CONSTRUCTION

As already indicated in the previous section, unlike Dijkstra's algorithm, MDPAlg obtains many disjoint paths with a single execution. All of these paths are computed from the information stored in *CM*. This translation of the *CM* into specific paths is called the *construction phase*, which generates the final set of disjoint paths based on a configurable group of target nodes (from one to all nodes in the graph).

This section explains the path construction process among the given node and the set of predefined target nodes following the same structure as in Section III-B. In particular, Algorithm 3 provides the pseudo-code for the path construction phase. The path construction process function has, as input parameters, the graph *G*, *s*, and *CM* obtained in the first phase. This function starts by initializing the main variables used during the disjoint path construction processes, *previous*, *next_hop*, *new_path* and *Paths*. The first two are used to build each disjoint path in a sequential way, while the last two save the path under construction and the set of paths generated, respectively. Moreover, variable *D* obtains the target nodes from the graph. For each target node stored in *D*, the procedure starts a new path construction process towards *s* through each one of its neighbors, selecting them in increasing cost order with the function *GetMinCostNeighbour*. Moreover, due to the centralized nature of the algorithm, this procedure is sequential, so a path construction process is not started until the previous one has finalized. The path construction process between a target node and *s* requires some intermediate steps in which, the procedure, through the function *GetMinCostNeighbour* and the variables *previous* and *next_hop*, consecutively selects the next available neighbour node with the lowest cost from *CM*, until *s* is reached. Additionally, the function *ApplyDisjointnessRestrictions* performs the operations to ensure disjointness among paths belonging to the same *s-target node* tuple. More specifically, after selecting a neighbor node to continue the path construction towards *s*, the two entries representing the link between them in *CM* are disabled (since the paths obtained are bidirectional). Alternatively, if the operation mode is node-disjoint, all the entries in *CM* associated to the selected node are disabled to guaranty this node can not be selected for other paths. Once *s* is reached, the path obtained is saved in the variable *Paths*, and a new disjoint path construction process for another neighbor or target node is started. Finally, in the case that function *GetMinCostNeighbour* does not return a node, meaning there is no way to continue the path construction towards *s*, a back track mechanism is invoked by calling *GoBackFunction*. Finally, when all the available neighbors of the target node have been assessed, a new target node is chosen to continue the construction phase.

To better understand the whole procedure and the concepts described above, Figures 2 and 3 illustrate a specific example of the path construction process in link- and node-disjoint modes, respectively, between three pair of nodes: *S-E* (Figures 2a and 3a), *S-D* (Figures 2b and 3b), and *S-F* (Figures 2c and 3c). Both figures present the corresponding graph (top

Algorithm 3 Disjoint path construction process

```
1:  function DisjointPathConstruction(G, s, CM)
2:      Initialize_variables(previous, next_hop, new_path, Paths)
3:      D = GetTargetNodes(G)
4:      while D ≠ ∅ do
5:          dst = ExtractNode(D)
6:          for each neighbour of dst do
7:              next_hop = GetMinCostNeighbour(CM, dst)
8:              while next_hop ≠ s  do
9:                  Advance : previous = next_hop; next_hop = GetMinCostNeighbour(CM, previous)
10:                 Insert(next_hop, new_path)
11:                 ApplyDisjointnessRestrictions(CM, previous, next_hop)
12:                 if next_hop == s then
13:                     Insert(new_path, Paths)
14:                 if next_hop == ∅ then
15:                     GoBack(CM, previous, next_hop)
        return Paths
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

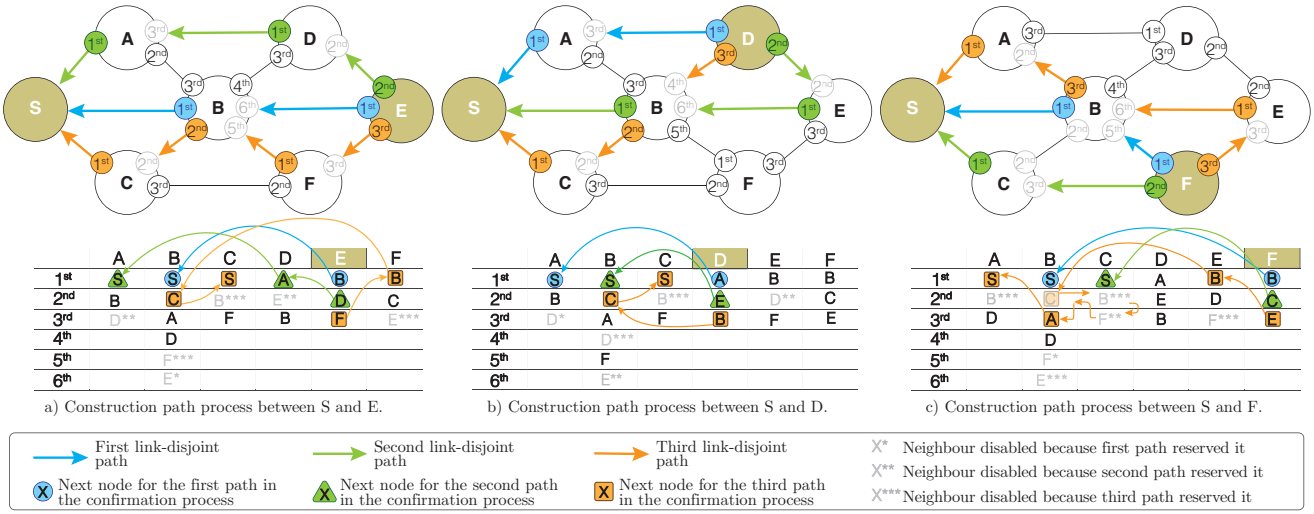


FIGURE 2: Link-disjoint path construction process in MDPAlg

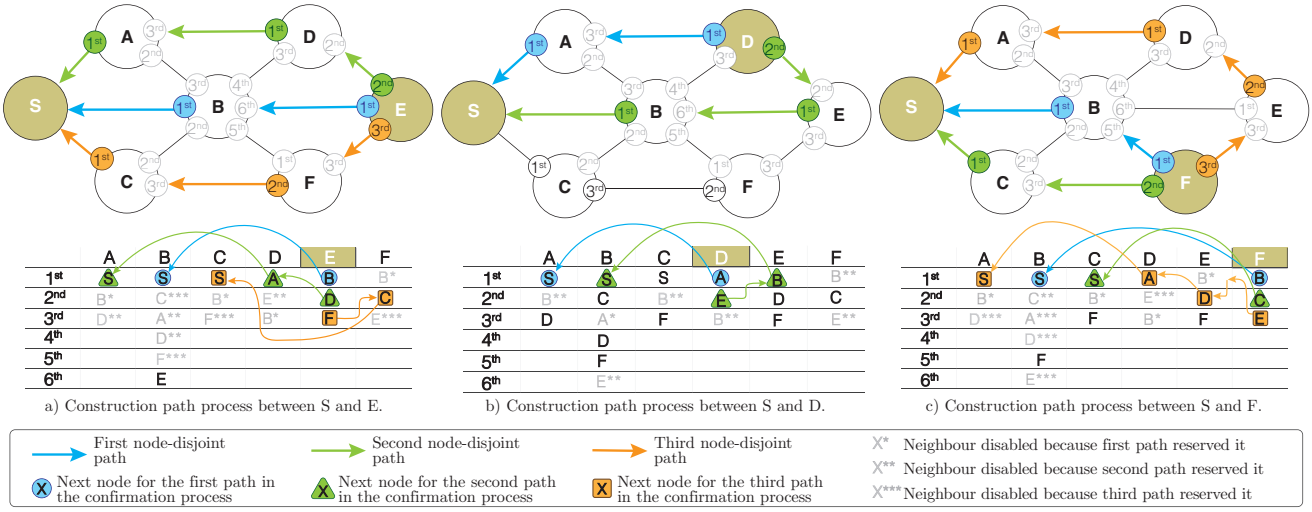


FIGURE 3: Node-disjoint path construction process in MDPAlg

half), together with a different view of the information in *CM* (bottom half). This view shows the order in which the different neighbors of a node are selected by the *GetMinCostNeighbour* function during the path construction phase, so that, for each column, the different rows are ordered by increasing cost to *s*, and the corresponding neighbor node names are listed instead of the costs.

Figure 2 exemplifies the path construction process in link-disjoint mode. In the example, node *S* acts as source node *s*, while the set of target nodes is comprised by three nodes, namely *E*, *D* and *F*. Let us first focus on Figure 2a, which depicts the process between nodes *S* and *E*, which is the first target node in the list. The link-disjoint path construction process is started at node *E* by selecting, as next hop, its lowest-cost neighbor, which is *B*. Following the same method, *B* will then select its lowest-cost neighbor too, node *S*, hence concluding the first link-disjoint path construction process. This path is marked in blue both in the graph and in the *CM*, where the path is represented in column *E*, row *B*, with an arrow that goes directly from column ***E*** to column ***B***, where *s* is found. During this process, node *E* in column *B* is also disabled to satisfy the link-disjoint policies because it represents the reverse path from *E* to *B*. This is represented with a gray color and one asterisk after *E*.

When the construction process of the first path ends, the second one repeats the same procedure, selecting in each step the lowest-cost free neighbor and disabling the reverse link in *CM*. More specifically, the second path construction process results in a link-disjoint path composed by nodes *E*, *D* (which is the second lowest cost at *E*, just after *B*), *A* (which is the first option at *D*) and *S*; this second disjoint paths is depicted with green arrows. Similarly, the procedure also disables from *CM* the entries that describe the reverse links, marked in the figure with gray and two asterisks. Finally, the process is repeated a third time, hence yielding the third link-disjoint path, which is shown with orange arrows in the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

figure. When the third path construction ends, as there are no more options to build new disjoint paths between *E* and *S*, the algorithm initiates another path building process for the next target node, i.e., node *D*, whose path construction process is illustrated in Figure 2b. Finally, the process is repeated for the last target node, *F*, which is shown in Figure 2c.

In node-disjoint mode, the procedure is similar, each node selects the lowest-cost available neighbor until *s* is reached. However, the policy to ensure node-disjoint paths differs from the link-disjoint mode. In this mode, each node selected for a disjoint path is disabled from the entire *CM*. Figure 3a depicts how node *B* is disabled in all columns after being selected in the first node-disjoint path.

Up to this point, the paths were easily built, by simply selecting the next available neighbor. However, as already anticipated in the algorithm description, this search might find dead ends, hence having the need to *go back* to look for alternative routes. This additional mechanism implemented by MDPAlg can be observed both in Figure 2c and Figure 3b, which illustrate a successful and unsuccessful attempt to avoid a dead end, respectively. In the case of Figure 2c, belonging to the link-disjoint mode, it progresses from *F* to *E* and *B*, where the *GetMinCostNeighbour* function returns *C* as first option, as it is the next available lower-cost neighbor. However, *C* has no valid neighbors because they were disabled by previous steps of the link-disjoint path construction process. Thus, the procedure goes back to node *B*, which executes again the *GetMinCostNeighbour* function hence obtaining *A* as the next valid lowest-cost neighbor. This alternative route completes the third link-disjoint path construction process. On the other hand, the third attempt to build a disjoint path between *D* and *S* fails as shown in Figure 3b. When the construction phase reaches *B*, all its entries are disabled (because *B* was already selected for the second path), hence it activates the hop back mechanism and backtracks to *D*. Unfortunately, *D* does not have any other available neighbor and, besides, cannot hop back again because it is the target node, thus, finishing the construction phase with only two disjoint paths built for this pair.

IV. THEORETICAL STUDY OF COMPUTATIONAL COMPLEXITY

Computational complexity is a term that, in computer science, usually relates to the efficiency of algorithms to solve a problem in relation to the amount of physical or temporal resources required, which constitutes a good estimator to quantify the quality of algorithms. In this section, we study the computational complexity of MDPAlg by comparing it to that of a well known and proven competitor solution, namely the primitive (naive) implementation of Dijkstra's algorithm, in a worst-case scenario (a full mesh graph where each node is connected to each other). We quantify the number of calculations needed by both proposals to obtain a certain number of disjoint paths. Dijkstra's algorithm was selected for comparison because it is a good benchmark to position MDPAlg in the disjoint path ecosystem. First, the section formally defines the mathematical problem and, afterwards, it analyzes the computational complexity of Dijkstra's algorithm versus MDPAlg in depth. Finally, the study is extended to take into account several enhanced implementations of Dijkstra's algorithm, as well as other related algorithms. In this regard, Table 1 illustrates the differences and the computational complexity values, without delving into details to avoid extending the article excessively.

A. MATHEMATICAL FORMULATION

The mathematical formulation of the disjoint path problem can be defined as follows. Given a graph $G = (\mathcal{N}, \mathcal{L})$ (consisting of a set $\mathcal{N}$ of N nodes and a set $\mathcal{L}$ of L links) and two nodes $\{s, t\} \in \mathcal{N}$. For $k > 0$, find k paths $P_1, P_2, ..., P_k$ from s to t that do not share any links or nodes. The problem is NP-complete in both cases, that is, link- and node-disjoint modes [34]. Regarding the worst-case scenario, in which each node is connected to every other node in the graph, the *WorstCaseGraph* can be defined as $WC_G = (\mathcal{N}, \mathcal{L})$, composed by a set $\mathcal{N}$ of N nodes and a set $\mathcal{L}$ of L links, in which each $n_i \in \mathcal{N}$ is connected with all $n_j \in \mathcal{N}$, $\forall$ $n_i \neq n_j$. Additionally, the worst case scenario also implies that all the disjoint paths between all possible pair of nodes

TABLE 1: Computational complexity analysis

Protocol/Algorithm	**Worst case scenario (full mesh graph)**			
	One/two path(s) per pair $\{s,t\} \in \mathcal{N} \; \forall \; s \neq t$	All paths per pair $\{s,t\} \in \mathcal{N} \; \forall \; s \neq t$	All paths between $s \; and \; all \; t_j \in \mathcal{N} \; \forall \; s \neq t_j$	All paths between $s_i \; and \; all \; t_j \in \mathcal{N} \; \forall \; s_i \neq t_j$
MDPAlg protocol (node disjoint mode)	-	$O(N^2)$	$O(N^2)$	$O(N^3)$
MDPAlg protocol (link disjoint mode)	-	$O(N^2)$	$O(N^3)$	$O(N^4)$
Naive Dijkstra (node and link disjoint)	$O(N^2)$ [17]	$O(N^3)$	$O(N^4)$	$O(N^5)$
Naive Dijkstra min-heap (node and link disjoint)	$O(\frac{N^2}{2} \cdot log(N))$ [17]	$O(N^3)$	$O(N^4)$	$O(N^5)$
Naive Dijkstra Fibonnaci-heap (node and link disjoint)	$O(N \cdot log(N) + \frac{N^2}{2})$ [17]	$O(N^3)$	$O(N^4)$	$O(N^5)$
Robertson's algorithm (node disjoint)	-	$O(N^3)$ [31]	$O(N^4)$	$O(N^5)$
Kawarabayashi's algorithm (node disjoint)	-	$O(N^2)$ [32]	$O(N^3)$	$O(N^4)$
Eppstein's algorithm (without disjointness)	-	-	$O(\frac{N^2}{2} + N \cdot log(N) + \frac{N^2}{2} \cdot N)$ [33]	$O(N^4)$
Karaata's algorithm (node-disjoint)	$O(N^2)$ [22]	$O(N^3)$	$O(N^4)$	$O(N^5)$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

should be discovered, so the mathematical description is as follows: Given the $WC_G = (\mathcal{N}, \mathcal{L})$, a set of given nodes $\mathcal{S} \in \mathcal{N}$ / $s_j \in \mathcal{S}$, $0 < j \leqslant N$, a set of target nodes $\mathcal{T} \in \mathcal{N}$, for each s_j / $t_i \in \mathcal{T}$, $s_j \neq t_i$, $0 < i \leqslant N$, find k paths $(P_{1(s_j,t_i)}, P_{2(s_j,t_i)}, ...P_{k(s_j,t_i)}, 0 < k \leqslant N-1)$ for each pair of nodes $\{s_j, t_i\}$ that do not share any common link or node.

B. DIJKSTRA'S ALGORITHM

Dijkstra's algorithm determines the minimum-cost tree from a given node towards the remaining nodes in a graph. Therefore, to obtain multiple disjoint paths between a pair of nodes, Dijkstra's algorithm must be executed several times after removing from the graph the path discovered in the previous iteration. In this way, each newly found minimum-cost path guarantees the disjointness condition with the previous ones. After several runs, all the disjoint paths for a given pair of nodes will be discovered in an increasing cost order, since the lowest-cost paths were erased previously from the graph. Finally, to discover all the disjoint paths between every pair of nodes in the graph, this process must be repeated as many times as pairs of nodes in the network.

Theorem 1. *To obtain the minimum-cost path between a pair of nodes, the computational complexity of Dijkstra's algorithm is $O(N^2)$ [17]. Thus, in WC_G, in which all the nodes are connected between them, Dijkstra's algorithm requires $O(N^3)$ time to find all the disjoint paths between a pair of nodes $\{s, t\} \in \mathcal{N}$, $s \neq t$.*

Proof. To discover all the disjoint paths between a pair of nodes, Dijkstra's algorithm must be executed as many times as disjoint paths exist between that pair of nodes. In a WC_G all nodes are interconnected among them, hence each node has $N-1$ neighbours through which is possible to obtain $N-1$ disjoint paths with each target node. Therefore, to obtain all the disjoint paths between a pair of nodes $\{s, t\}$, $s \in \mathcal{S}$, $t \in \mathcal{T}$, $s \neq t$, Dijkstra's algorithm must be executed as many times as possible disjoint paths exist between that pair of nodes $\{s, t\}$, i.e., $N-1$ times. Accordingly, as the computational complexity of each Dijkstra's run is $O(N^2)$, the complexity to discover all the disjoint paths between a pair of nodes is $O(N^2) \cdot (N-1) \simeq O(N^3)$. □

Theorem 2. *To obtain all the disjoint paths between a given node $s \in \mathcal{S}$ and all the target nodes $t_i \in \mathcal{T}$, $\forall\, t_i \neq s$, $0 < i \leqslant N$ in WC_G, the computational complexity of Dijkstra's algorithm is $O(N^4)$.*

Proof. Theorem 1 defines that the computational complexity to discover all the disjoint paths between a pair of nodes is $O(N^3)$. To obtain all the disjoint paths between a given node $s \in \mathcal{S}$ and all the target nodes $t_i \in \mathcal{T}$, $\forall\, t_i \neq s$, $0 < i \leqslant N$, the process described in Theorem 1 must be repeated $N-1$ times, once per each $t_i \in \mathcal{T}$, $t_i \neq s_j$. Hence, this complexity is $O(N^3) \cdot (N-1) \simeq O(N^4)$. □

Theorem 3. *In WC_G, using Dijkstra's algorithm, the resulting computational complexity to obtain all the disjoint paths between all $s_j \in S$, $0 < j \leqslant N$ and all $t_i \in T$, $0 < i \leqslant N$, is $O(N^5)$.*

Proof. On the basis of Theorem 2, the computational complexity of Dijkstra's algorithm is $O(N^4)$ to discover the disjoint paths between a given node $s \in \mathcal{S}$ and all $t_i \in \mathcal{T}$, $\forall\, t_i \neq s$, $0 < i \leqslant N$. Therefore, to discover the paths between all $s_j \in \mathcal{S}$, $0 < j \leqslant N$ and all $t_i \in \mathcal{T}$, $0 < i \leqslant N$, the Theorem 2 must be applied as many times as the number of s_j nodes are (N times). Accordingly, the resulting is $O(N^4) \cdot N = O(N^5)$ □

As a conclusion, to obtain all the disjoint paths in the worst-case scenario (a hyper-connected mesh graph), Dijkstra's algorithm computational complexity is related to the number of nodes in the graph in a fifth-order exponential function.

C. MDPALG

MDPAlg looks for disjoint paths between a given node and a set of target nodes in an efficient way, aiming to lessen the computational complexity. As described in Section III, MDPAlg provides link-disjoint or node-disjoint paths by performing a search process in two steps: an initial analysis of cost phase, followed by a multi-node path selection phase. Therefore, to calculate MDPAlg's computational complexity, these two phases must be analyzed. The total computational complexity is the result of adding the computational complexity of each phase. Since the exploration phase is common in both modes (link- and node-disjoint) it will be analyzed first. The study is subsequently performed for the link-disjoint confirmation phase and for the node-disjoint confirmation phase, respectively.

Theorem 4. *In WC_G, to perform an analysis of the accumulated cost incurred from a given node $s \in \mathcal{S}$ with MDPAlg, the computational complexity cost is $O(N^2)$. If the analysis is extended to obtain the accumulated cost from each node $s_j \in \mathcal{S}$, $0 < j \leqslant N$, the resulting computational complexity is $O(N^3)$.*

Proof. The analysis of costs of MDPAlg uses the same search structure as Dijkstra's algorithm to characterize the graph according to the accumulated cost incurred from a given node. In particular, MDPAlg does not add new operations implying the evaluation of new nodes or links, hence the resulting computational complexity is the same as the one obtained by executing Dijkstra's algorithm ($O(N^2)$). Additionally, considering that a single analysis of costs allows for the computation of all disjoint paths between a given node $s \in \mathcal{S}$ and all the target nodes $t_i \in \mathcal{T}$, $\forall\, t_i \neq s$, $0 < i \leqslant N$, this process must be repeated as many times as nodes exist in WC_G to obtain all the disjoint paths between $\{s_j, t_i\}$, $s_j \in \mathcal{S}$, $t_i \in \mathcal{T}$, $\forall\, t_i \neq s_j$, $0 < \{i, j\} \leqslant N$. Therefore, as $\mathcal{S}$ contains all the nodes of WC_G, the cost analysis phase must be repeated N times, yielding a computational complexity of $O(N^2) \cdot N = O(N^3)$. □

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

Theorem 5. *In WC_G, to discover all link-disjoint paths between all $s_j \in \mathcal{S}$, $0 < j \leqslant N$ and all $t_i \in \mathcal{T}$, $0 < i \leqslant N$, the computational complexity of MDPAlg's construction phase is $O(N^4)$.*

Proof. In link-disjoint mode, the worst-case scenario in terms of computational complexity occurs when the disjoint paths obtained between a pair of nodes $\{s,t\} \in N$ use all the links $l_i \in L$ available in WC_G, since it involves processing all links in the graph. In WC_G, as all nodes are interconnected, each node is connected with the rest of the nodes in the graph ($N-1$ nodes), which provides $N \cdot (N-1) = N^2 - N$ links in total. However, as links are bidirectional, in practice, WC_G has half of the links, $\frac{N^2-N}{2}$, because of the previous calculation assumed both directions of each link. Therefore, to obtain all the disjoint paths between a pair of nodes $\{s,t\}$, $s \in \mathcal{S}$, $t \in \mathcal{T}$, $s \neq t$, the computational complexity is proportional to the number of links, which involves a computational complexity $O(N^2)$. To obtain all the disjoint paths between a given node $s \in \mathcal{S}$ and all the target nodes $t_i \in \mathcal{T}$, $\forall\ t_i \neq s$, $0 < i \leqslant N$, the previous process must be repeated as many times as target nodes t_i are in $\mathcal{T}$ $(N-1)$, whose resulting computational complexity is $O(N^2) \cdot (N-1) = O(N^3)$. Finally, to discover the paths between all $s_j \in \mathcal{S}$, $0 < j \leqslant N$ and all $t_i \in \mathcal{T}$, $0 < j \leqslant N$, the previous steps must be executed as many times as s_j are in $\mathcal{S}$ (N nodes), giving a final computational complexity of $O(N^3) \cdot N = O(N^4)$. □

Theorem 6. *In WC_G, to discover all the node-disjoint paths between all $s_j \in \mathcal{S}$, $0 < j \leqslant N$ and all $t_i \in \mathcal{T}$, $0 < i \leqslant N$, the computational complexity of MDPAlg's construction phase is $O(N^3)$.*

Proof. In node-disjoint mode, the worst-case scenario in terms of computational complexity occurs when the disjoint paths obtained between a pair of nodes $\{s,t\} \in N$ use all the nodes $n_i \in N$ available in WC_G, since it involves processing all nodes of the graph. The number of nodes in a WC_G graph is N, therefore, to obtain *k* node-disjoint paths between $\{s,t\}$, $s \in \mathcal{S}$, $t \in \mathcal{T}$, $s \neq t$, the computational complexity is $O(N)$. To obtain all the disjoint paths between a given node $s \in \mathcal{S}$ and all the target nodes $t_i \in \mathcal{T}$, $\forall\, t_i \neq s$, $0 < i \leqslant N$, the previous process must be repeated as many times as target nodes t_i are in $\mathcal{T}$ $(N-1)$, whose resulting computational complexity is $O(N) \cdot (N-1) = O(N^2)$. Finally, to discover the paths between all $s_j \in \mathcal{S}$, $0 < j \leqslant N$ and all $t_i \in \mathcal{T}$, $0 < j \leqslant N$, the previous steps must be executed as many times as s_j are in $\mathcal{S}$ (N nodes), giving a final computational complexity of $O(N^2) \cdot N = O(N^3)$. □

The resulting computational complexity of MDPAlg is given by the sum of the computational complexity of its phases (analysis of cost and paths construction). In link-disjoint mode the final result is $O(N^3) + O(N^4) \simeq O(N^4)$, while in node-disjoint mode is $O(N^3) + O(N^3) \simeq O(N^3)$, which decreases up to two orders of magnitude the computational complexity regarding Dijkstra's algorithm.

D. COMPARATIVE RESULTS

In this section, we provide a quick overview about how the theoretical study performed in the previous section has been extended to analyze other algorithm approaches, under the same restrictions (i.e., to obtain all the disjoint paths between all $s_j \in \mathcal{S}$, $0 < j \leqslant N$ and all $t_i \in \mathcal{T}$, $0 < j \leqslant N$ in a WC_G). This extended analysis firstly includes the implementation improvements of Dijkstra's algorithm (min-priority queue with min-heap and Fibbonaci-heap), and afterwards it focuses on other relevant works found in the literature. All the results are collected in Table 1, omitting the in-detail analytical study because of its length.

In view of the results of Table 1, none of the improvements of Dijkstra's algorithm outperforms MDPAlg. The same occurs with Robertson's, Kawarabayasi's and Karaata's, algorithms, whose computational complexity is at least one order of magnitude higher than MDPAlg for the same operation mode (node-disjoint). Eppstein's algorithm could be considered the closest competitor to MDPAlg, as it is the only one providing a similar computational complexity than MDPAlg's link-disjoint ($O(N^4)$) but it does not guarantee path disjointness.

V. EVALUATION

This section aims to evaluate the implementation of MDPAlg in terms of: (1) computational complexity, with a particular focus on its scalability considering the network size, (2) the number of disjoint paths discovered, and (3) the time needed to obtain all disjoint paths (convergence time), in direct comparison to Dijkstra's algorithm. First, this section presents the selected testbed and the implementation, it then explains the experimental setup and, finally, it collects, represents, and analyzes the obtained results.

A. TESTBED AND IMPLEMENTATION

To perform the evaluation of MDPAlg, we selected the MATLAB software tool [35] because of its versatility, its coding speed and debugging properties, its friendly interface, and its powerful toolboxes for graphical representation.

For the comparison, we chose Dijkstra's algorithm because it is a well-proven solution, if not the most popular. Furthermore, as its computational complexity was studied theoretically in Section IV, this evaluation will serve to validate it. More specifically, we leveraged Xiaodong Wang's library [36], available for MATLAB, because it implements Dijkstra's algorithm based on a matrix of costs, which simplifies the coding process of MDPAlg. This is particularly relevant since the first phase of MDPAlg is inspired by Dijkstra's algorithm search process, and besides, it computes the cumulative cost from the given node in a cost matrix. However, as Dijkstra's algorithm does not provide disjoint paths by definition, we must execute Dijkstra's algorithm iteratively to obtain the disjoint paths between a given pair of nodes. After each run, we remove from the graph the minimum-cost path obtained, launching Dijkstra's algorithm with the modified graph again to calculate a new disjoint

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

path. The process ends when no more paths are discovered between the pair of nodes. In this way, the disjointness among paths is guaranteed. To obtain all the disjoint paths available in a graph this process is repeated as many times as pairs of nodes are in the graph.

MDPAlg was implemented in MATLAB platform modifying Xiaodong Wang's library to develop the first phase of the algorithm. Moreover, the second phase of MDPAlg was implemented from scratch trying to optimize the generated code.

Finally, regarding the hardware platform, all the experiments were executed in a Intel(R) Core(TM) i7-8700K CPU computer with 32 GB RAM.

B. EXPERIMENTAL SETUP

To fully characterize MDPAlg, the experimental setup includes different scenarios in which it was comprehensively evaluated. Our intention was to validate that the performance of MDPAlg remained excellent compared to Dijkstra's, from the most complex to the simplest scenario.

To this purpose, we first evaluated both protocols (MDPAlg and Dijkstra's algorithm) in the *WorstCaseGraph* to validate the theoretical analysis performed in Section IV. Afterward, the tests were repeated in 2-dimension (2D) square mesh graphs ranging from 4 to 36 nodes (2x2 to 6x6), and in random graphs ranging from 20 to 100 nodes to complete the analysis. On the one hand, square mesh graphs provide a regular structure that maintains a medium-high connectivity ratio among nodes, hence keeping the multipath choice while simplifying the structure of the *WorstCaseGraph*. On the other hand, random graphs aim to synthesize real scenarios consisting of heterogeneous connections among nodes, such as Metropolitan Area Network (MAN) with hyper-connected urban nodes or peripheral not-so-well-connected nodes. Furthermore, to maintain the multipath choice, the random graphs generated have an average of two, four and six links per node. However, for the sake of simplicity, the paper only shows the results of four links per node for two reasons: (1) an average of four paths between a pair of nodes is enough for multipath application requirements, such as in networking [37] or evacuation route planning [38] scenarios, and additionally (2) the results for two and six neighbors per node are very similar and do not provide further insights. In summary, this set of three types of graphs was selected to illustrate the evolution of both algorithms when executed both in regular (hyper-connected) and irregular (non so well-connected) graphs, including an intermediate transition scenario (represented by the 2D square meshes).

The random graph generator tool chosen was Boston university Representative Topology gEnerator (BRITE) [39] as it has also been used in the evaluation phase of the original 1S-MDP [30]. This tool provides two random connection models, Waxman model [40] and Barabási–Albert model [41], which covers a wide range of heterogeneous graphs; Waxman connects nodes randomly based on Euclidean distance, while Barabási–Albert follows a power-law model.

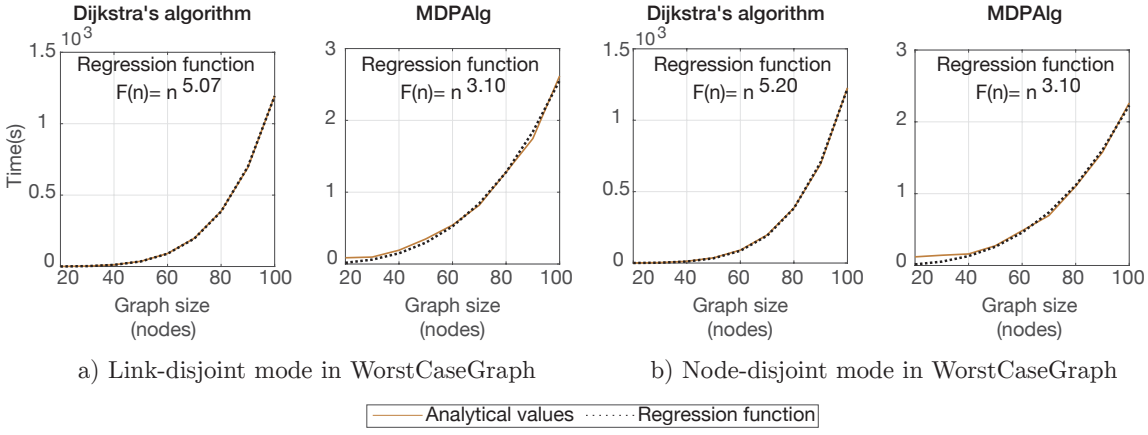
Each test was repeated 30 times to compute 95% confidence intervals, ensuring that the deviation of the confidence intervals regarding the average values did not exceed the 10%. The cost per link was set randomly except for the *WorstCaseGraph*, in which the cost of all links were the same and the test was repeated just once, in order to check the hypotheses raised in Section IV.

C. RESULTS

This section presents the results and analyzes them classified according to the parameters previously defined: (1) computational complexity, (2) number of disjoint paths discovered, and (3) convergence time. All values obtained for MDPAlg are compared to Dijkstra's algorithm. Moreover, this section also serves to experimentally validate the theoretical study performed in Section IV.

1) Computational complexity

This first stage of the evaluation aimed to validate the hypotheses raised in Section IV, while quantifying and comparing the computation complexity of both algorithms, Di-



a) Link-disjoint mode in WorstCaseGraph b) Node-disjoint mode in WorstCaseGraph

FIGURE 4: Computational complexity in *WorstCaseGraph*

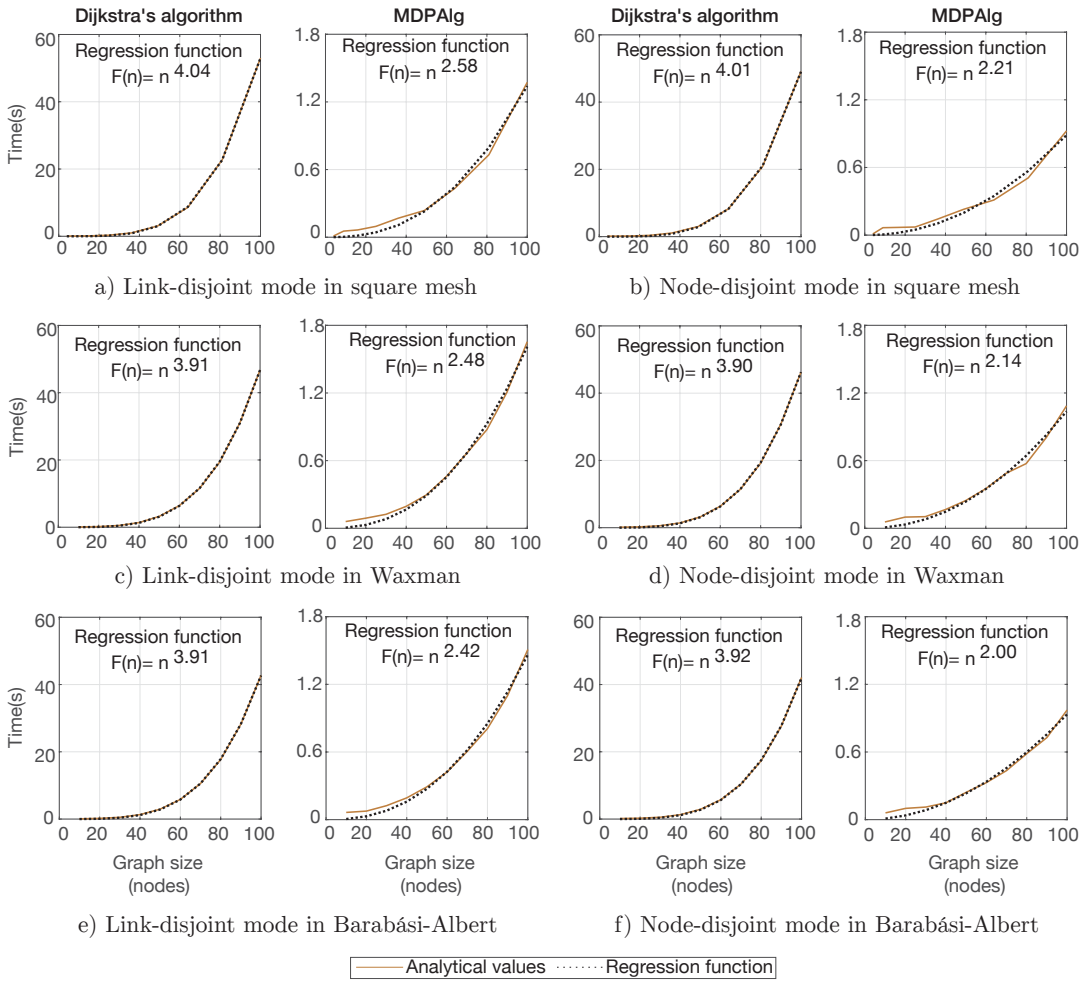
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

jkstra's and MDPAlg. The computational complexity is an estimator related to the amount of physical or temporal resources consumed by the algorithms to obtain the solution. Given that the amount of memory required is negligible in both cases, we decided to measure the time invested by both algorithms to obtain the disjoint paths, as a function of the number of nodes in the graph. Then, we computed the regression function that fits best those results and compared it to the results of the theoretical study of Section IV.

Figure 4 depicts the evolution of the computational complexity for Dijkstra's algorithm and MDPAlg, as the number of nodes increases in the *WorstCaseGraph*. Please note that, for the sake of simplicity, the y-axis is different for Dijkstra's and MDPAlg. So even if the graphs look similar, the values differ in several orders of magnitude, indeed. As there are two modes to generate the disjoint paths, Figure 4a shows the results for link-disjoint mode, while Figure 4b displays those for node-disjoint mode. Moreover, solid lines represent the experimental data measurements, while dotted lines depict the regression function associated to those experimental results, whose mathematical function (the computational complexity), is located at the top of each figure.

As depicted in Figure 4, the results validate the computational complexity study of Dijkstra's algorithm elaborated in Section IV, since the computational complexity function evolves in both cases (theoretically and experimentally) as a power of 5 function of the graph size.

A similar conclusion can be reached for MDPAlg, as it follows a cubic function in both modes (node- and link-disjoint), which validates the theoretical study for node-disjoint mode. However, it slightly differ for link-disjoint mode, since the measured values further reduce by one the exponent of the computational complexity function (from $O(N^4)$ to $O(N^3)$). This is due to the combination of the high-connectivity of *WorstCaseGraph* together with the MDPAlg's path construction phase algorithm, which chooses the lowest-cost available option for each new disjoint path. The combination of these two features causes the path construction process to obtain disjoint paths with only one or two hops, since the high connectivity provides alternative paths with a very short length. Therefore, only a small subgroup of links of the graph are eventually used, which drastically reduces the computation complexity compared to the worst case of the link-disjoint problem, in which all the links of the graph are used. As



a) Link-disjoint mode in square mesh
b) Node-disjoint mode in square mesh
c) Link-disjoint mode in Waxman
d) Node-disjoint mode in Waxman
e) Link-disjoint mode in Barabási-Albert
f) Node-disjoint mode in Barabási-Albert

FIGURE 5: Computational complexity in the rest of graphs (mesh and random)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

a matter of fact, the solution obtained is closer to a typical node-disjoint solution, as the procedure is finding paths that rarely share any node.

Before checking the rest of graph scenarios, we would like to provide an additional remark about Figure 4 related to the regression function. As it can be observed, the values are slightly higher than the ones from the theoretical analysis (e.g. 5.07 instead of a 5, 3.10 instead of 3, etc.). This is mainly caused by concurrent processes running in the operating system in the computer, which slows down the overall procedure. Nevertheless, the values are still consistent and close to the theoretical ones, which validates the behavior of both algorithms.

Once the *WorstCaseGraph* has been analyzed, Figure 5 displays the computational complexity for the rest of the evaluated graphs (square mesh and random models). We can observe a similar behaviour in all cases: MDPAlg decreases approximately by a half the computational complexity compared to Dijkstra's algorithm. This effect occurs because MDPAlg optimizes the analysis of cost phase by gathering more information in a single execution, which, as a result, reduces the computational load and improves the scalability of the algorithm.

2) Number of paths and convergence time

In this second stage of the evaluation, we examined both the number of disjoint paths obtained by each proposal and the time invested to obtain them in the set of graphs under study. Figures 6 and 7 depict the number of paths discovered for *WorstCaseGraph* and the rest of the evaluated graphs, respec-

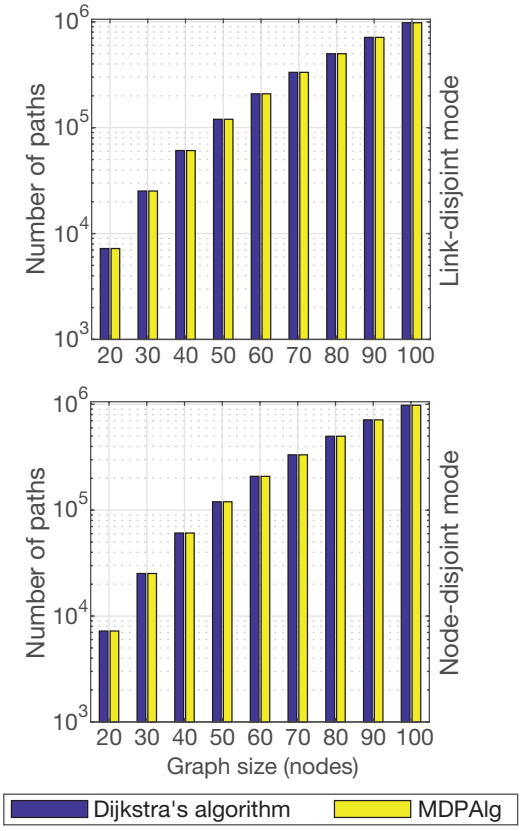


FIGURE 6: Number of paths discovered in *WorstCaseGraph*

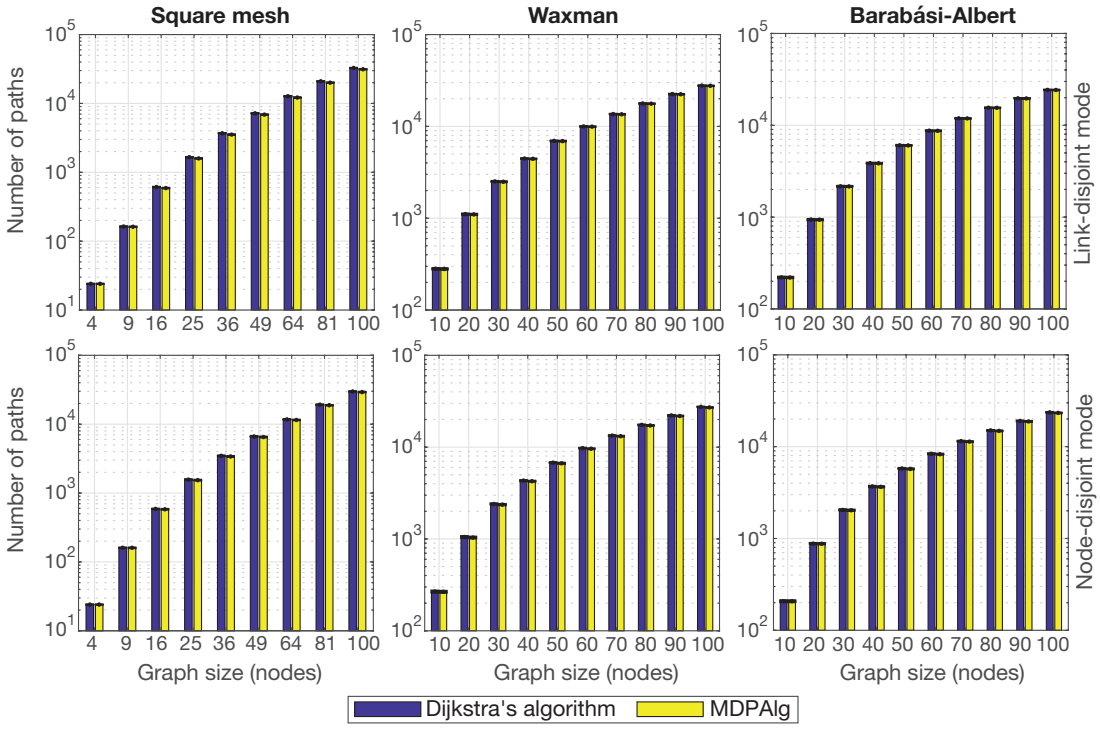


FIGURE 7: Number of paths discovered: link-disjoint mode (top), node-disjoint mode (bottom)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3080931, IEEE Access

tively, while Figures 8 and 9 focus on the convergence time. All figures present at the top half the results for link-disjoint mode, and at the bottom half the results for node-disjoint mode. Moreover, Figures 7 and 9 displays from left to right the results obtained in square mesh, and the two random graphs, respectively. As seen in Figure 6, *WorstCaseGraph*, the number of paths discovered by both algorithms is the same due to the high-connectivity degree of the graph, which causes that, independently of the number of cost analyses performed, both solutions always found the very same paths.

However, the number of paths discovered by MDPAlg, compared to Dijkstra's algorithm, slightly decreases in the rest of the evaluated graphs, as seen in Figure 7.

This effect is caused by the simplified cost analysis implemented by MDPAlg, which only collects information once and uses it to obtain the whole set of paths for a given source node, while Dijkstra's algorithm does it several times, once per computed path. This gap ranges from 0% up to 10-12%, depending on the type and size of the graph. Nevertheless, this slight decrease of performance is overcompensated by the gain obtained in the convergence time, which is reduced by two to three orders of magnitude compared to Dijkstra's algorithm. Indeed, MDPAlg drastically reduces the number of mathematical calculations needed to obtain the disjoint paths.

The great reduction in convergence time yielded by MDPAlg in all graphs is particularly higher in large graphs, independently of their type. Again, this is due to the single cost analysis phase of MDPAlg, which successfully deals

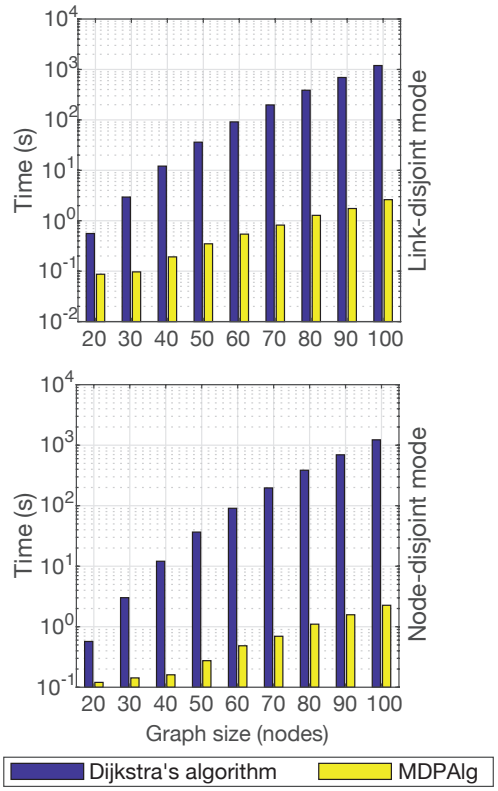


FIGURE 8: Convergence time in *WorstCaseMesh*

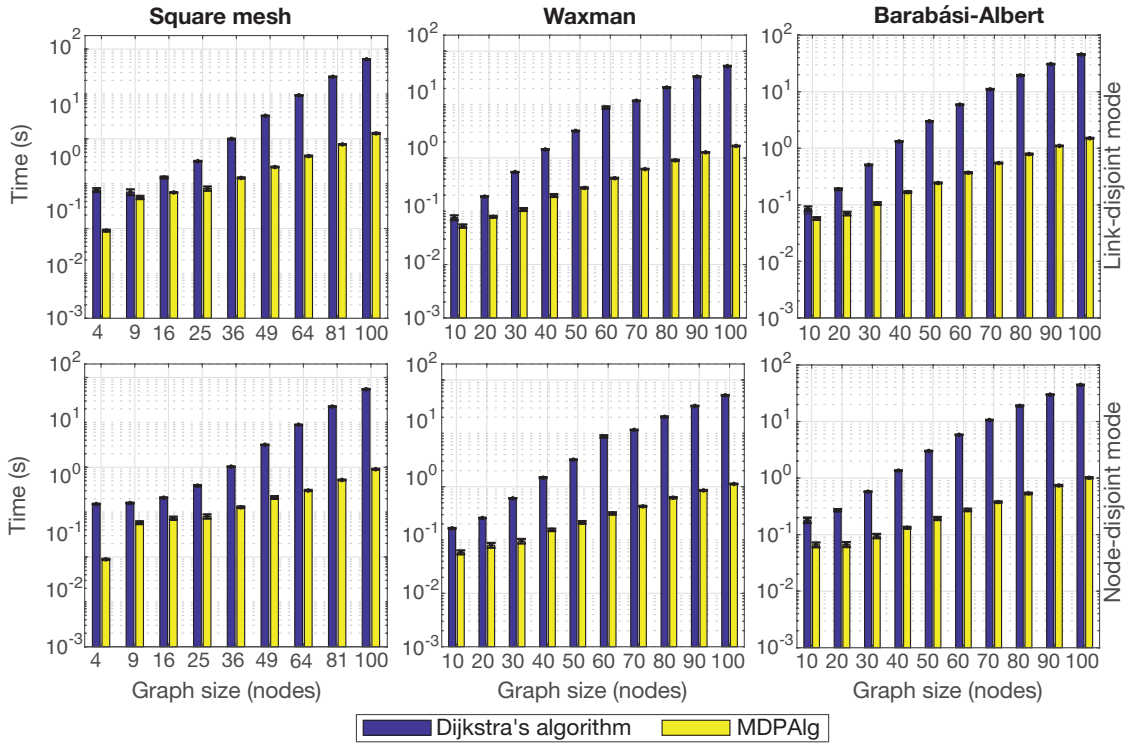


FIGURE 9: Convergence time: link-disjoint mode (top), node-disjoint mode (bottom)

with the increment in the number of links with the graph size. However, it produces an exponential growth of mathematical operations in the case of Dijkstra´s algorithm, which is responsible for the exponential growth in convergence time.

Therefore, the small reduction in the number of paths obtained by MDPAlg is overcompensated by the drastic convergence time reduction, which exhibits exceptional scalability in this regard.

VI. CONCLUSION

In this paper we have studied MDPAlg, an algorithm that searches multiple (link- or node-) disjoint paths among a given source node and a set of target nodes, in two phases. The first phase analyzes the graph according to the accumulated cost incurred from the source node, while the second phase leverages the information of the previous phase to build the disjoint paths in an increasing cost order. Moreover, we have studied the computational complexity of MDPAlg and compared the results to other solutions, and concluded that MDPAlg drastically decreases the computational complexity compared to its opponents.

MDPAlg has been tested in graphs of different nature (from structured meshes up to random models) to experimentally validate the theoretical study of the computational complexity carried out in the paper, and to study its behaviour in different scenarios. The results obtained are promising, as MDPAlg reduces the computational complexity up to three orders of magnitude while keeping the number of paths discovered close to its rivals, independently of the graph type.

As future work, we want to study the quality of the disjoint paths obtained in terms of load balancing or failure resilience, as well as other scenarios in which MDPAlg could fit, such as vehicular networks, path planning problems, biomedical applications, or evacuation tasks.

REFERENCES

[1] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," Numerische Mathematik, vol. 1, no. 1, pp. 269–271, June 1959.
[2] J. Moy, "OSPF Version 2," RFC 2328, p. 244, April 1998.
[3] V. P. Kompella, J. C. Pasquale, and G. C. Polyzos, "Multicast routing for multimedia communication," IEEE/ACM transactions on Networking, vol. 1, no. 3, pp. 286–292, June 1993.
[4] J.-R. Jiang, H.-W. Huang, J.-H. Liao, and S.-Y. Chen, "Extending Dijkstra's shortest path algorithm for software defined networking," in The 16th Asia-Pacific Network Operations and Management Symposium. IEEE, September 2014, pp. 1–4.
[5] D. R. Lanning, G. K. Harrell, and J. Wang, "Dijkstra's algorithm and Google maps," in Proceedings of the 2014 ACM Southeast Regional Conference, March 2014, pp. 1–3.
[6] H. Wang, Y. Yu, and Q. Yuan, "Application of Dijkstra algorithm in robot path-planning," in 2011 Second International Conference on Mechanic Automation and Control Engineering, July 2011, pp. 1067–1069.
[7] D. Hong, "Medical Image Segmentation Based on Accelerated Dijkstra Algorithm," in Advances in Intelligent Systems. Springer, March 2012, pp. 341–348.
[8] J. W. Suurballe and R. E. Tarjan, "A Quick Method for Finding Shortest Pairs of Disjoint Paths," Networks, vol. 14, no. 2, pp. 325–336, June 1984.
[9] J. O. Abe, H. A. Mantar, and A. G. Yayimli, "K-Maximally Disjoint Path Routing Algorithms for SDN," in Proceedings - 2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2015. IEEE, October 2015, pp. 499–508.
[10] A. Tolba, "Organizing Multipath Routing in Cloud Computing Environments," International Journal of Advanced Computer Science and Applications, vol. 8, no. 1, pp. 455–462, January 2017.
[11] M. Doshi and A. Kamdar, "Multi-constraint QoS disjoint multipath routing in SDN," in Moscow Workshop on Electronic and Networking Technologies, MWENT 2018 - Proceedings. Institute of Electrical and Electronics Engineers Inc., March 2018, pp. 1–5.
[12] A. Prakash, R. Kannan, and G. BAFNA, "Disjoint path computation systems and methods in optical networks," Ciena Corp. US patent US10003867B2, June 2019.
[13] J. Tapolcai, G. Retvari, P. Babarczi, and E. R. Berczi-Kovacs, "Scalable and Efficient Multipath Routing via Redundant Trees," IEEE Journal on Selected Areas in Communications, vol. 37, no. 5, pp. 982–996, March 2019.
[14] O. Lemeshko, O. Yeremenko, M. Yevdokymenko, and B. Sleiman, "Enhanced Solution of the Disjoint Paths Set Calculation for Secure QoS Routing," in 2019 IEEE International Conference on Advanced Trends in Information Theory, ATIT 2019, December 2019, pp. 210–213.
[15] L. Zuo, M. Zhu, C. Wu, A. Hou, and L. Cao, "Bandwidth reservation for data transfers through multiple disjoint paths of dynamic HPNs," Proceedings - 21st IEEE International Conference on High Performance Computing and Communications, 17th IEEE International Conference on Smart City and 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019, pp. 2455–2460, August 2019.
[16] B. Martín, Á. Sánchez, C. Beltran-Royo, and A. Duarte, "Solving the edge-disjoint paths problem using a two-stage method," International Transactions in Operational Research, vol. 27, no. 1, pp. 435–457, January 2020.
[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to algorithms, 2nd ed. MIT Press, 2001.
[18] S. Shaikh, "Span-disjoint paths for physical diversity in networks," in Proceedings IEEE Symposium on Computers and Communications, July 1995, pp. 127–133.
[19] M. Gottschau, M. Kaiser, and C. Waldmann, "The undirected two disjoint shortest paths problem," Operations Research Letters, vol. 47, no. 1, pp. 70–75, January 2019.
[20] R. Bhandari, "Optimal diverse routing in telecommunication fiber networks," Proceedings of INFOCOM '94 Conference on Computer Communications, vol. 3, pp. 1498–1508, June 1994.
[21] A. Björklund and T. Husfeldt, "Shortest two disjoint paths in polynomial time," SIAM Journal on Computing, vol. 48, no. 6, pp. 1698–1710, 2019.
[22] M. H. Karaata, "An algorithm for finding two node-disjoint paths in arbitrary graphs," Journal of Computing and Information Technology, vol. 27, no. 3, pp. 1–14, September 2019.
[23] H. Nguyen, C. Phung, S. Secci, B. Felix, and M. Nogueira, "Can MPTCP secure Internet communications from man-in-the-middle attacks?" in 2017 13th International Conference on Network and Service Management, CNSM 2017, November 2017, pp. 1–7.
[24] A. Itai and M. Rodeh, "The Multi-Tree Approach To Reliability In Distributed Networks," Information and Computation, vol. 79, no. 1, pp. 43–59, January 1988.
[25] R. Ogier and N. Shacham, "A distributed algorithm for finding shortest pairs of disjoint paths," in IEEE INFOCOM '89, Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies, April 1989, pp. 173–182.
[26] R. Hadid, M. H. Karaata, and V. Villain, "A Stabilizing Algorithm for Finding Two Node-Disjoint Paths in Arbitrary Networks," International Journal of Foundations of Computer Science, vol. 28, no. 4, pp. 411–435, January 2017.
[27] D. Sidhu, R. Nair, and S. Abdallah, "Finding disjoint paths in networks," Conference on Communications architecture & protocols (SIGCOMM '91), pp. 43–51, September 1991.
[28] J. Tanaka, "Path Generating method, relay device, and computer product," Fujitsu Ltd. US patent US8665754B2, March 2014.
[29] D. Lopez-Pajares, J. Alvarez-Horcajo, E. Rojas, J. A. Carral, and G. Ibanez, "Iterative Discovery of Multiple Disjoint Paths in Switched Networks with Multicast Frames," in 2018 IEEE 43rd Conference on Local Computer Networks (LCN), October 2018, pp. 409–412.
[30] D. Lopez-Pajares, J. Alvarez-Horcajo, E. Rojas, J. A. Carral, and I. Martinez-Yelmo, "One-Shot Multiple Disjoint Path Discovery Protocol (1S-MDP)," IEEE Communications Letters, vol. 24, no. 8, pp. 1660–1663, April 2020.

[31] N. Robertson and P. Seymour, "Graph Minors.XIII. The Disjoint Paths Problem," Journal of Combinatorial Theory, Series B, vol. 63, no. 1, pp. 65 – 110, January 1995.
[32] K.-i. Kawarabayashi, Y. Kobayashi, and B. Reed, "The disjoint paths problem in quadratic time," Journal of Combinatorial Theory, Series B, vol. 102, no. 2, pp. 424–435, March 2012.
[33] D. Eppstein, "Finding the k shortest paths," SIAM Journal on computing, vol. 28, no. 2, pp. 652–673, July 1998.
[34] M. Middendorf and F. Pfeiffer, "On the complexity of the disjoint paths problem," Combinatorica, vol. 13, no. 1, pp. 97–107, March 1993.
[35] The MathWorks, Inc., "Matlab (R2018b)," 2018.
[36] X. Wang, "Dijkstra Shortest Path Routing Library," in MATLAB Central File Exchange. https://www.mathworks.com/matlabcentral/fileexchange/5550-dijkstra-shortest-path-routing, 2004, Accesed: July 2020.
[37] S. Nelakuditi and Z.-L. Zhang, "On Selection of Paths for Multipath Routing," in Proceedings of the 9th International Workshop on Quality of Service, ser. IWQoS '01. Berlin, Heidelberg: Springer-Verlag, June 2001, pp. 170–186.
[38] V. Campos, R. Bandeira, and A. Bandeira, "A method for evacuation route planning in disaster situations," Procedia-Social and Behavioral Sciences, vol. 54, pp. 503–512, October 2012.
[39] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: An approach to universal topology generation," in MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. IEEE, 2001, pp. 346–353.
[40] B. M. Waxman, "Routing of multipoint connections," IEEE journal on selected areas in communications, vol. 6, no. 9, pp. 1617–1622, December 1988.
[41] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," Science, vol. 286, no. 5439, pp. 509–512, October 1999.

DIEGO LOPEZ-PAJARES (M'16) obtained a Master's Degree in Telecommunications Engineering in 2016. He worked as a researcher with the NetIS-NETSERV research group of the University of Alcala for five years, since 2015. He is focused on topics related to delay-tolerant and SDN networks, specifically low-latency routing solutions and the multipath problem, which are the main topics of his Ph.D. In addition, he has participated in several GIST-NETSERV research projects, such as TIGRE5-CM, TAPIR-CM, EsPECIE and SIMPSONS. He is currently working in the Department of Telematics System Engineering at the Polytechnic University of Madrid (UPM), as a teaching assistant.

ELISA ROJAS (PhD'13) received her PhD in Information and Communication Technologies engineering from the University of Alcala, Spain, in 2013. As a postdoc, she worked in IMDEA Networks and, later on, as CTO of Telcaria Ideas S.L., an SME dedicated to both research and development of virtualized network services. She has participated in diverse projects funded by the EC, such as FP7-NetIDE or H2020-SUPERFLUIDITY. She currently works as an Assistant Professor in the University of Alcala where her current research interests encompass SDN, NFV, high performance and scalable Ethernet, IoT and data center networks. She has worked in three different areas in relation with SDN: as a researcher in several EU projects, as a designer and developer in an SDN project for a network operator, and as a professor.

J.A. CARRAL (PhD'13) graduated in Telecommunications Engineering from the Polytechnic University of Madrid (UPM) in 1993. There, he worked as a research assistant in the Telematics Engineering Department (UPM) and was granted a 4-year FPU-MEC (1994-1998) by the Spanish Ministry of Science and Education. He joined the Polytechnic School of the University of Alcala as a teaching assistant in 1998 and obtained a teaching position in 2001. He received his PhD degree from the University of Alcala in 2013 in the field of advanced Ethernet switches and was promoted to the position of associate professor. He has participated in several regional, national, and international (EU) research projects both at the UPM and the University of Alcala. He has published more than a dozen articles in selected international journals in the field of telecommunications and has participated in several international and national conferences. Currently, he also collaborates with the Madri+d Foundation as a member of the VERIFICA panel of experts.

ISAIAS MARTINEZ-YELMO (PhD'10) obtained a PhD in Telematics from the Carlos III University of Madrid in Spain in 2010. After working as a postdoctoral assistant at the Carlos III University of Madrid, he became and remains a teaching assistant in the Automatics Department at the University of Alcala in Spain. His research interests include Peer-to-Peer Networks, Content Distribution Networks, Vehicular Networks, NGN, and Internet protocols. Nowadays, he is especially interested in advanced switching architectures and Software Defined Networks (SDN). He has participated in various competitive research projects funded by the Madrid regional government (Medianet, TIGRE5-CM), National projects (CIVTRAFF), and European projects (CONTENT, CARMEN, etc.). His research papers have been published in high impact JCR indexed research journals such as Communications Magazine, Computer Communications, and Computer Networks, among others. In addition, he has been a reviewer for high quality conferences (i.e. IEEE INFOCOM) and scientific journals (IEEE Transactions on Vehicular Technology, Computer Communications, ACM Transactions on Multimedia Computing, etc.) and was a Technical Program Committee member for IEEE ICON from 2011-2013.

JOAQUIN ALVAREZ-HORCAJO (PhD'20) obtained his PhD in Information and Communication Technologies engineering from the University of Alcala in 2020. He started his engineering career having worked at Telefonica as a test engineer for COFRE and RIMA networks. Currently, he is working as a researcher with the NetIS-NETSERV research group of the University of Alcala (from 2015 to current). During this period, he was awarded the university professor training grant (FPU) and the postdoctoral grant at the University of Alcala. The areas of research where he has worked include Software Defined Networks (SDN), Internet protocols, and new generation protocols. At present, he is especially interested in topics related to advanced switches and SDN networks. He has participated in various competitive projects funded through the Community of Madrid plan, such as TIGRE5-CM and TAPIR-CM.