An Efficient Algorithm for Enumerating all Minimal Paths of a Graph

Khalid Housni
MISC Laboratory
Department of Computer Sciences
Faculty of Sciences, Ibn Tofail University
University Campus, BP 133 Kenitra, 14000, Morocco

***Abstract*—The enumeration of all minimal paths between a terminal pair of a given graph is widely used in a lot of applications such as network reliability assessment. In this paper, we present a new and efficient algorithm to generate all minimal paths in a graph *G(V, E)*. The algorithm proposed builds the set of minimal paths gradually, starting from the source node *s*. We present two versions of our algorithm; the first version determines all feasible paths between a pair of terminals in a directed graph without cycle, and this version runs in linear time $O(|V| + |E|)$. The second version determines all minimal paths in a general graph (directed and undirected graph). In order to show the process and the effectiveness of our method, an illustrative example is presented for each case.**

***Keywords*—Minimal path; network reliability; linked path structure; recursive algorithm**

I. Introduction

The evaluation of the reliability of a system that can be modeled as a network can be made in terms of either minimal cuts (MCs) or minimal paths (MPs) [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. In [11], [10], [12], [13], [14] it was shown that the set of all minimal paths can be used for generating all the minimal cut-sets of a graph and vice-versa.

The use of MPs and MCs for reliability assessment is well documented in [15], [16], and for details on the use of MPs and MCs in reliability evaluation, we refer to these papers. In this paper, we especially focused on determination of all minimal paths in graph. In the literature, there exists several algorithms related to minimal paths' problem [1], [5], [10], [17], [18], [19], [20], [21], [9].

In [17], Al-Ghanim presents, to generate all minimal paths, an algorithm based on heuristic programming. The algorithm proposed produces redundant paths, and to remove them, the author uses an extensive comparison. Al-Ghanim's algorithm has been improved by Yeh [18] through eliminating the possibility of generating duplicate MPs. The last two approaches, [17] and [18], and others like[22], [19], belong to the category of search algorithms based on augmentation. The general principle of these approaches consists of adding arc by arc, starting from the source until completing the target network. After each increase, the MPs thus constructed are collected.

Another family of algorithms for MPs enumeration is called, according to Chen [21], *direct search-based algorithms* [21], [23], [24], [25], [20], [10]. These methods are based on depth-first search (DFS). In [10], Shen introduced an algorithm to enumerate all minimal paths between specified single terminal pair of arbitrary graphs. The proposed algorithm is based on elementary concept of graph theory and dual principle. To improve Shen's algorithm, Kobayashi [20] adds some additional processes based on the level set of nodes. In [21], Chen uses a backtracking process to generate all MPs. Backtracking is a family of algorithms that consists of going back on decisions made to get out of a deadlock. This process, which is a characteristic of the descriptive software languages, makes it possible to abandon each partial candidate which cannot lead to a valid solution. Bai further improved Chen's algorithm by adding conditions for backtracking to reduce the number of search branches [9]. To the author's Knowledge, currently, Bai, Tian, and Zuo's algorithm [9] is the best known DFS algorithm.

In this work, we present a new method to enumerate all minimal paths in an oriented graph with no cycles. We also give a more general algorithm which determines all the minimal paths of a general graph.

This paper is organized as follows: In the next section we present the basic definitions and terminology. In Section 3, we introduce a new algorithm to find all minimal paths in an oriented graph with no cycle. For this, we will, first, introduce a directed graph reduction algorithm to eliminate nodes that cannot appear in the set of minimal paths. In Section 4, we introduce the enumeration algorithm of all minimal paths in a general graph (oriented or not). For that, we will, first, introduce an algorithm for the reduction of undirected graphs allowing the elimination of the nodes which cannot appear in any path of the set of minimal paths. In Section 5, we provide an analysis of the complexity of our algorithms. We also present a comparative study of our method with the recent method developed in 2016 by Bai [9]. In Section 6, we present all the tests we made and the results obtained. We also compare our work to recent works. An extension to the multi-terminal case is presented in Section 7. Finally, we will conclude with some suggestions for future research in the field of minimal paths' enumeration.

II. Notation and nomenclature

A. Graph Representation

There are two classical ways to represent a garph: an adjacency matrix, or a set of adjacency lists. The choice of the type of representation depends on the operations performed on this structure:

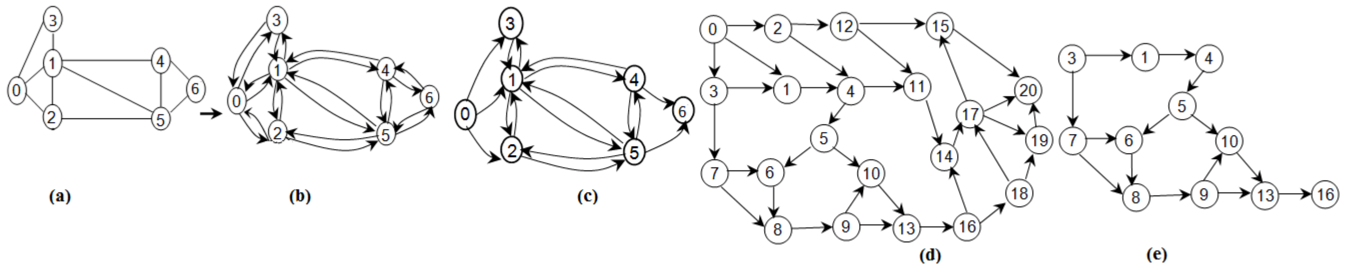

Fig. 1. Example of networks indexed by nodes: (a) undirected network and its oriented representation (b).(c) reduction result of the graph b. (d) Oriented network without cycle. (e) reduction result of the graph d.

- for the representation by the adjacency matrix, the verification of the existence of an arc between two given vertices is in $O(1)$, whereas the search for the neighbors of a given vertex is in $O(n)$.
- for the representation by the adjacency list, the verification of the existence of an arc between two given vertices is in $O(n)$, whereas the search for the neighbors of a given vertex is in $O(1)$.

In our case, we have opted for the adjacency list representation because the most common operation is the traversal of the list of neighbors. The graph (a) of Fig. 1 and its oriented representation (b) will be represented by the following list: L ={(1, 2, 3), (0, 2, 3, 4, 5), (0, 1, 5), (0, 1), (1, 5, 6), (1, 2, 4, 6), (4, 5)}. L[0]=(1, 2, 3) is outgoing nodes list of node 0. L[1]= (0, 2, 3, 4, 5) is outgoing nodes list of node 1, etc.

In this paper, we opted for using the oriented representation of the graphs. Thus, in the case of an undirected graph, it is necessary to convert the non-oriented edges into oriented edges. For this, we used the traditional Hagstrom technique that involves replacing each undirected edge by two directed edges with the opposite direction [26].

B. Notations

Let *G= (V, E)* be a graph with $n = |V|$ vertices and $m = |E|$ edges. One refers to source vertex by *s* and to sink vertex by *t*. For a vertex *v*∈*V*, we denote by $\Gamma(v)$, the set of all vertices in *V* that are adjacent to *v*. In the case of directed graph, for any vertex *v*∈*V*, the incoming neighborhood (also called predecessors) of *v* is defined as $\Gamma^{-}(v) = \{u \in V/(u,v) \in E\}$ and the outgoing neighborhood (also called successors) of *v* is defined as $\Gamma^{+}(v) = \{u \in V/(v,u) \in E\}$. We denote by $\Gamma^{+*}(v) = \{u \in \Gamma^{+}(v)/\forall x \in \Gamma^{-}(u),\ State[x] =' examined'\}$ all the outgoing neighbors of which all the incoming neighborhood are already processed. One refers to the set of all incoming neighbors who are already processed by Γ^{-*} and by Γ^{+nt} the set of outgoing neighbors who are not yet processed. *G/u* is the vertex-induced sub graph $G(V\backslash\{v\},\ E')$ for *v*∈*V* where *E'={(p,q)*∈*E / p≠v and q≠v}*. Likewise, for edge *e*∈*E, G/e= (V, E*$\backslash\{e\}$*)* is the edge-induced sub graph. We refer to a path π=*(a*$_l$→*a*$_2$→*a*$_3$ ... →*a*$_k$*)* by its natural sequence of vertices *(a*$_l$ *,a*$_2$*, a*$_3$*, ... , a*$_k$*)* such that any two consecutive vertices x_i and x_{i+1} are connected by an arc of G: $\forall i\ ,\ 0 \leq i \leq n-1\ ,\ (x_i\ ,\ x_{i+1}) \in E$. A path π from *s* to *t* is denoted by *st-path*. We noted by P_{st} the set of all st-minimal paths in *G*. A node or edge is called invalid if and only if it cannot appear in any path in the P_{st} set. An active node is defined as any processed node having at least one of its outgoing neighbors not yet processed.

A path π in *G* is called minimal if no vertex occurs more than once (also called elementary path). In a directed graph without cycle, all paths are minimal paths. A path can have a single vertex and be of length 0.

C. Functions and Abbreviations

In this sub-section, we describe the functions used in our algorithms:
push: this function adds an item to a collection.
pop: this function returns and removes an item from a stack.
AMG: this function extends a path to other nodes.
LsOC: Lines of Code.
LOC: Line of Code.

III. Enumerating all Minimal Paths in an Oriented Graph with no Cycles

In this section, we introduce a new minimal path enumeration algorithm in a directed graph with no cycles. First, we will introduce a graph reduction algorithm to remove edges and nodes that cannot appear in the paths of the set of minimal paths P_{st}. The aim of this operation is to reduce the execution time that is in $O(|V|+|E|)$ while eliminating the invalid arcs and nodes. In a second step, we will give our algorithm for enumeration of minimal paths.

A. Graph Reduction

We call reduction of the graph, the operation of of removing, from the graph, all nodes and arcs that can not appear in any path in the minimum path set P_{st}.

In the next two sub-sections, we detail the different steps which will enable us to reduce the graph. Thereafter, the complete algorithm is given in algorithm1.

1) Delete Edges from a Graph: At the beginning, we remove all incoming arcs to the source *s*, $\{(x,s) \in E\}$, because the starting node is *s* and the searched paths are minimal paths, so no arc in the set $\{(x,\ s) \in E\}$ cannot be appeared in the paths of $P_{s,t}$ (a path cannot contain the node *s* twice). The same applies to the arcs $\{(t,\ y) \in E\}$ because the destination node is *t*.

2) Delete Vertices from a Graph: All nodes in the set $V \setminus$ {s, t} cannot be starting nodes, because all paths in the set P_{st} start from *s*. Similarly, these nodes cannot be destination nodes. From this remark, all the nodes of $V \setminus$ {s, t} must have the following two characteristics: $\{x \in V\setminus\{s\} / \, card(\Gamma^{-}(x)) \neq 0\}$ and $\{x \in V\setminus\{t\} / \, card(\Gamma^{+}(y)) \neq 0\}$. Consequently the nodes of $V\setminus$\{s, t\} whose $\{x \in V \setminus \{s\} \; / \; card \; (\Gamma^{-} \; (x)) \; = \; 0\}$ or $\{x \in V \setminus \{t\} \; / \; card \; (\Gamma^{+} \; (x \;)) \; = \; 0\}$ cannot be intermediate nodes between *s* and *t* and should be deleted.

Note: When a node is deleted, the arcs linking this node to the graph will also be deleted. So we have $G\setminus v = G(V\setminus\{v\}, \; E')$ where $E' = \{(p,q) \in \; E \; / \; p \neq v \; and \; q \neq v\}$.

After each deletion of a node, the nodes' validity testing process will be iterated on these predecessors in the case of the nodes' checking $\{x \in V \setminus \{t\} \; / \; card \; (\Gamma^{+} \; (y)) \; = \; 0\}$, and on the successors for the nodes checking $\{x \in V \setminus \{s\} \; / \; card \; (\Gamma^{-} \; (x)) \; = \; 0\}$.

Algorithm 1 $Oriented\_graph\_reduction(G, s, t)$

```
1: for all $y \in \Gamma^{-}(s)$ do
2:     $E \leftarrow E\setminus(y, s)$
3: end for
4: for all $y \in \Gamma^{+}(t)$ do
5:     $E \leftarrow E\setminus(t, y)$
6: end for
7: for all $x \in V\setminus\{s\}$ do
8:     if $card(\Gamma^{-}(x)) = 0$ then
9:         $To\_delete \leftarrow To\_delete \cup x$
10:    end if
11: end for
12: while $To\_delete \; is \; not \; empty$ do
13:    $x \leftarrow element \, from \, To\_delete$
14:    $To\_delete \leftarrow To\_delete - \{x\}$
15:    for all $y \in \Gamma^{+}(x)$ do
16:        if $card(\Gamma^{-}(x)) = 1$ then
17:            $To\_delete \leftarrow To\_delete \cup \{x\}$
18:        end if
19:    end for
20:    $G \leftarrow G\setminus x$
21: end while
22: for all $x \in V\setminus\{t\}$ do
23:    if $card(\Gamma^{+}(x)) = 0$ then
24:        $To\_delete \leftarrow To\_delete \cup x$
25:    end if
26: end for
27: while $To\_delete \, is \, not \, empty$ do
28:    $x \leftarrow element \, from \, To\_delete$
29:    $To\_delete \leftarrow To\_delete - \{x\}$
30:    for all $y \in \Gamma^{-}(x)$ do
31:        if $card(\Gamma^{+}(x)) = 1$ then
32:            $To\_delete \leftarrow To\_delete \cup \{x\}$
33:        end if
34:    end for
35:    $G \leftarrow G\setminus x$
36: end while
```

3) Illustration on an Example: Consider a graph shown in Fig. 1(d) where the source node *s=3* and sink node *t=16*.

Delete edges
LsOC 1,2 and 3: the arc (0,3) will be removed.
LsOC 4,5 and 6: the arcs (16,14) and (16,18) will be removed.
Delete vertices
LsOC 7, 8, 9, 10 and 11: these LsOC initialize the stack of vertices to be deleted by all vertices that have card(Γ^{-}) = 0, in our case by 0, 18.
LsOC from 12 to 21: this loop treats the nodes of the stack of vertices to delete as follows:

- a node is retired from the stack; it is noted by x.
- any element y of $\Gamma^{+}(x)$ whose $card(\Gamma^{-}(y)) = 1$ is added to the stack.
- delete vertices *x* from a graph.

Table I below gives the various iterations of the loop. A line represents one iteration of the loop. Column 1 shows the contents of the 'To_delete' stack at the beginning of the loop. Column 2 shows the element removed from the stack. Column 3 shows the nodes to be added to the stack of nodes to be removed. The content of the stack after the addition of these nodes is given in column 4. Column 5 shows the accumulation of the deleted nodes. Deleting a node implies deleting all the arcs connect to that node. Column 6 shows the accumulation of deleted arcs.

LsOC from 22 to 26: these LsOC initializes the stack of vertices to delete by all vertices that have card($\Gamma^{+}(x)$) = 0, in our case by {20}.

LsOC 27 to 36: these LsOC treats the nodes of the stack of vertices to delete. Table II below presents the various iterations of the loop.

The result of the reduction process is shown in Figure 1.e.

B. The Enumeration of all Minimal Paths in Directed Graph without Cycle

1) The principle of the algorithm: Starting from the fact that the minimal paths linking a source node *s* to another node *x* can be obtained from the lists of minimal paths linking *s* to the predecessors of *x* by applying a simple increase of the paths *eq. 1*, the main idea of our method is to build, little by little, all minimal paths P_{st}. The algorithm starts with an initialization of P_{ss} by the set {*(s)*}. Afterwards, the algorithm constructs all the possible paths of the outgoing neighbors of *s* who's all the predecessors are already processed. Let's note by Γ^{+*} all the outgoing neighbors of which all the predecessors are already processed. The process is thus repeated passing each time to the outgoing neighbors whose predecessors are already processed until the processing of node *t*.

$$P_{SX} = \bigcup_{Y \in \Gamma^{-}(X)} \bigcup_{\pi \in P_{SY}} AGM(\pi, x) \qquad (1)$$

The construction of the possible paths for a node *x* from the paths of its predecessors is made by a simple increase of these paths (eq.2).

$$P_{SX} = \{AGM(\pi, x) \; for \; each \; \pi \in P_{SY} \\ for \; each \; y \; in \; \Gamma^{-}(X)\} \qquad (2)$$

TABLE I. Illustration of the graph reduction algorithm (algorithm 1). Lines of Code between 12 and 21.

Stack of vertices to delete	The current element	Vertices to add to to_delete	Stack of vertices to delete	Accumulation of deleted nodes	Accumulation of deleted arcs
{0, 18}	0	2	{2, 18}	0	(0,3)
{2, 18}	2	12	{12, 18}	{0, 2}	(2,4), (2,12), (0,3)
{12, 18}	12	-	{18}	{0, 2, 12}	(12,11), (12,15), (2,4), (2,12), (0,3)
18	18	-	{∅ }	{0, 2, 12, 18}	(18,17), (18,19), (12,11), (12,15), (2,4), (2,12), (0,3)

TABLE II. Illustration of the graph reduction algorithm (algorithm 1). Code lines between 27 and 36.

Stack of vertices to delete	The current element	Vertices to add to to_delete	Stack of vertices to delete	Accumulation of deleted nodes	Accumulation of deleted arcs
{20}	20	15,19	{15,19}	20	(15,20), (17,20), (19,20)
{15, 19}	15	-	{19}	{15, 20}	(17,15), (15,20), (17,20), (19,20)
{19}	19	17	{17}	19,15,20	(17,19), (17,15), (15,20), (17,20), (19,20)
{17}	17	14	{14}	17,19,15,20	(14,17), (17,19), (17,15), (15,20), (17,20), (19,20)
{14}	14	11	{11}	14,17,19,15,20	(16,14), (11,14), (14,17), (17,19), (17,15), (15,20), (17,20), (19,20)
{11}	11	-	{∅ }	11,14,17,19,15,20	(4,11), (16,14), (11,14), (14,17), (17,19), (17,15), (15,20), (17,20), (19,20)

AGM is defined as follows:

$$\begin{array}{l} if\ \pi = (x_1 \rightarrow x_2 \ldots x_n)\ then \\ AGM(\pi, y) = (x_1 \rightarrow x_2 \ldots x_n \rightarrow y) \end{array} \quad (3)$$

2) The proof of correction and termination: The graph is without any cycle, which implies $\exists\, x \in V \,/\, \Gamma^{-}(x) = 0$. Knowing that the graph is reduced then:
$card(\{\, x \in V \,/\, \Gamma^{-}\,(x) = 0\}) = card(\{s\}) = 1$.
The graph is without any cycle so the vertex-induced subgraph G\s is also without cycle; this implies $\exists y \in G\backslash s$ such that $\Gamma^{-}(y) = 0$ and $card(\{\, y \in V\backslash s \,/\, \Gamma^{-}(y) = 0\}) >= 1$. Since $\Gamma^{-}\,(y) \neq 0$ in *G* so $y \in \Gamma^{+}\,(s)$ ($\Gamma^{-}(y) = 0$ is the result of the deletion of *s* from *G*). Thus the nodes of the graph will be explored from the source *s* until the last node of the graph *t* whose $\Gamma^{+}(t) = 0$.

The fact that the size of the graph is reduced from one iteration to another, implies the termination of the algorithm.

To optimize the algorithm in terms of memory space, we proceed as follows: let x be the current node (LOC 21: the element extracted from To_Treat) and let $y \in \Gamma^{-*}(x)$. At the level of LsOC 22 to 26, if the current node x is the last node of the set $\Gamma^{+}(y)$ which is not yet processed, so we move all the minimal paths of P_{sy}, after having increased them, to P_{sx}. This decision is made because all these successors are already processed, so P_{sy} will never be used in the process of building paths for nodes that have not yet been processed.

Algorithm 2 $MPs\_DirectedGraph(G, s, t)$

```
 1: input data:
 2: G=(V,E): a graph oriented without cycle;
 3: s , t: node; // source and terminal node
 4: local variables:
 5: x,y: node;
 6: $To\_Treat$: list of nodes;
 7: Begin
 8: $Oriented\_graph\_reduction(G, s, t)$
 9: $To\_Treat \leftarrow \{\emptyset\}$
10: for all $x \in V\backslash\{s\}$ do
11:     $P_{SX} \leftarrow \emptyset$
12:     $State(x) \leftarrow$ "$not\ reached$"
13: end for
14: $P_{SS} \leftarrow \{(s)\}$
15: $State(s) \leftarrow treated$
16: for all $x \in \Gamma^{+*}(s)$ do
17:     $push(x, To\_Treat)$
18:     $State(x) \leftarrow$ "$reached$"
19: end for
20: while $To\_Treat\ is\ not\, empty$ do
21:     $x \leftarrow pop(To\_Treat)$;
22:     for all $y \in \Gamma^{-}(x)$ do
23:         for all $\pi \in P_{Sy}$ do
24:             $P_{sx} \leftarrow P_{sx} \cup AGM(, \pi, x)$
25:         end for
26:     end for
27:     $State(x) \leftarrow$ "$examined$"
28:     for all $y \in \Gamma^{+*}(x)\ and\ State(y) =$ "$not\ eached$" do
29:         $push(y, To\_Treat)$
30:         $State(y) \leftarrow$ "$reached$"
31:     end for
32: end while
```

C. Illustration on an Example

Considering the reduced graph in Fig. 1(e) where the source node *s=3* and sink node *t=16*.

Firstly, we initialize the set P_{ss} by $\{(s)\}$, in our case $P_{33} = \{(3)\}$ and the list of vertices to treats by all elements of $\Gamma^{+*}(s)$, in our case $\{1, 7\}$. Table III presents the various iterations of the loop (iterations from 20 to 32);

- Column 1 shows the contents of the list To_Treat at the beginning of the loop.
- Column 2 shows the the node that is retrieved from To_Treat (LOC 21).
- Column 3 shows the sets P_{sx} (LsOC from 22 to 26).
- Column 4 indicates the predecessors of the current node that are not yet reached. These nodes will be added to the To_Treat set (LsOC from 28 to 31).

IV. Minimal paths algorithm for general graphs

The algorithm presented in the previous section cannot be applied to graphs containing cycles, which also implies non-oriented graphs. This is due to the fact that the condition that a node must satisfy to be added to the list of nodes

TABLE III. Illustration of the MPs_DirectedGraph algorithm (algorithm 2). Code lines between 20 and 32.

Contents of To_Treat	Current node (x)	$P_{sy}/y \in \Gamma^{+}(x)$		Paths set	Vertices to add to To_Treat
		P_{sy} to duplicate	P_{sy} to move		
$\{1, 7\}$	1	$P_{3,3}$	-	$P_{3,3}=\{(3)\}$; $P_{3,1}=\{(3,1)\}$	4
$\{4, 7\}$	4		$P_{3,1}$	$P_{3,3}=\{(3)\}$; $P_{3,1}=\{\}$; $P_{3,4}=\{(3,1,4)\}$	5
$\{5,7\}$	5		$P_{3,4}$	$P_{3,3}=\{(3)\}$; $P_{3,1}=\{\}$; $P_{3,4}=\{\}$; $P_{3,5}=\{(3,1,4,5\}$	-
$\{7\}$	7		$P_{3,3}$	$P_{3,3}=\{\}$; $P_{3,1}=\{\}$; $P_{3,4}=\{\}$; $P_{3,5}=\{(3,1,4,5\}$, $P_{3,7}=\{(3,7)\}$	6
$\{6\}$	6	$P_{3,5}$, $P_{3,7}$		$P_{3,3}=\{\}$; $P_{3,1}=\{\}$; $P_{3,4}=\{\}$; $P_{3,5}=\{(3,1,4,5\}$; $P_{3,7}=\{(3,7)\}$; $P_{3,6}=\{(3,7,6),(3,1,4,5,6)\}$	8
$\{8\}$	8		$P_{3,7}$, $P_{3,6}$	$P_{3,3}=\{\}$; $P_{3,1}=\{\}$; $P_{3,4}=\{\}$; $P_{3,5}=\{(3,1,4,5)\}$; $P_{3,7}=\{\}$; $P_{3,6}=\{\}$; $P_{3,8}=\{(3,7,6,8),(3,1,4,5,6,8),(3,7,8)\}$	9
$\{9\}$	9		$P_{3,8}$	$P_{3,3}=\{\}$; $P_{3,1}=\{\}$; $P_{3,4}=\{\}$; $P_{3,5}=\{(3,1,4,5)\}$; $P_{3,7}=\{\}$; $P_{3,6}=\{\}$; $P_{3,8}=\{\}$; $P_{3,9}=\{(3,7,6,8,9),(3,1,4,5,6,8,9),(3,7,8,9)\}$	10
$\{10\}$	10	$P_{3,9}$	$P_{3,5}$	$P_{3,3}=\{\}$; $P_{3,1}=\{\}$; $P_{3,4}=\{\}$; $P_{3,5}=\{\}$; $P_{3,7}=\{\}$; $P_{3,6}=\{\}$; $P_{3,8}=\{\}$; $P_{3,9}=\{(3,7,6,8,9), (3,1,4,5,6,8,9), (3,7,8,9)\}$; $P_{3,10}=\{(3,1,4,5,10), (3,7,6,8,9,10), (3,1,4,5,6,8,9,10), (3,7,8,9,10) \}$	13
$\{13\}$	13		$P_{3,9}$, $P_{3,10}$	$P_{3,3}=\{\}$; $P_{3,1}=\{\}$; $P_{3,4}=\{\}$; $P_{3,5}=\{\}$; $P_{3,7}=\{\}$; $P_{3,6}=\{\}$; $P_{3,8}=\{\}$; $P_{3,9}=\{\}$; $P_{3,10}=\{\}$; $P_{3,13}=\{(3,1,4,5,10,13), (3,7,6,8,9,10,13), (3,1,4,5,6,8,9,10,13), (3,7,8,9,10,13) , (3,7,6,8,9,13), (3,1,4,5,6,8,9,13), (3,7,8,9,13) \}$	16
$\{16\}$	16		$P_{3,13}$	$P_{3,3}=\{\}$; $P_{3,1}=\{\}$; $P_{3,4}=\{\}$; $P_{3,5}=\{\}$; $P_{3,7}=\{\}$; $P_{3,6}=\{\}$; $P_{3,8}=\{\}$; $P_{3,9}=\{\}$; $P_{3,13}=\{\}$, $P_{3,16}=\{(3,1,4,5,10,13,16), (3,7,6,8,9,10,13,16), (3,1,4,5,6,8,9,10,13,16), (3,7,8,9,10,13,16) , (3,7,6,8,9,13,16), (3,1,4,5,6,8,9,13,16), (3,7,8,9,13,16) \}$	-

to be processed (the nodes whose predecessors are already processed) is not necessarily verified at each iteration. To overcome this problem, we propose to keep the same principle in its general philosophy; that is, the construction of the set of minimal paths between two given nodes *s* and *t* is done starting from nodes *s*, then these neighbors, then the neighbors of the neighbors, until the processing of the final node *t*. During the process of treatment, we pass from a node to its successors even that all their predecessors are not yet processed.
The problem: knowing that the construction of the set of paths P_{sx} for a given node *x*, is made from the sets of paths of these predecessors, when creating P_{sx} for a given node $x \in V \backslash \{s, t\}$, it is possible that there are nodes in $\Gamma^{-}(x)$ that are not yet processed; therefore, P_{sx} will not be complete (it does not contain all minimal paths from *s* to *x*).
Suggestion: to complete the set of minimal paths of a node x, we propose to update it as soon as we process a node that belongs to $\Gamma^{-}(x)$. The details of update procedure are given in the next section. It is a backtracking to update the already processed nodes.

To optimize the memory space, we propose to adopt the principle used in the algorithm introduced in the previous section; if the current node *x* is the last node of the set $\Gamma^{+}(y)$ which is not yet processed, then we move all the minimal paths of P_{sy}, after increasing them, to P_{sx}, but for a given node *x*, if all its successors are already processed, the set P_{sx} will never be displaced from this node to one of these successors (because according to the hypothesis, they are already processed). To illustrate this, consider the graph given in Fig. 1(c), if the order of treatment of the nodes is 1, 2, 3, 4, 5 then 6, then the set of minimal paths P_{S3} cannot be moved from P_{S3} to P_{S1} because node *1* is processed before *3*. This implies a problem of optimization of the memory space. To solve this problem, at each iteration we will select from the set To_Treat (LOC 21) the node that has the largest distance to *t*. The algorithm we used for the determination of distances to the sink is given in subsection 4.B.

Algorithm 3 $MPs\_GeneralGraph(G, s, t)$

```
1:  input data:
2:  G=(V,E): a graph oriented without cycle;
3:  s , t: node; // source and terminal node
4:  local variables:
5:  x,y: node;
6:  $To\_Treat$: list of nodes;
7:  Begin
8:  $GeneralGraphReduction(G, s, t)$
9:  $To\_Treat \leftarrow \{\emptyset\}$
10: for all $x \in V \backslash \{s\}$ do
11:     $P_{SX} \leftarrow \emptyset$
12:     $State(x) \leftarrow$ "not reached"
13: end for
14: $P_{SS} \leftarrow \{(s)\}$
15: $State(s) \leftarrow treated$
16: for all $x \in \Gamma^{+}(s)$ do
17:     $push(x, To\_Treat)$
18:     $State(x) \leftarrow$ "reached"
19: end for
20: while $To\_Treat$ is not empty do
21:     $x \leftarrow pop(To\_Treat)$;// the element whose distance
        from t is greater
22:     for all $y \in \Gamma^{-*}(x)$ do
23:         for all $\pi \in P_{Sy}$ do
24:             $P_{sx} \leftarrow P_{sx} \cup AGM(, \pi, x)$
25:         end for
26:     end for
27:     $State(x) \leftarrow$ "examined"
28:     for all $y \in \Gamma^{+}(x)$ and $State(y) =$ "not reached" do
29:         $push(y, To\_Treat)$
30:         $State(y) \leftarrow$ "reached"
31:     end for
32:     for all $y \in \Gamma^{+}(x)$and $State(y) \neq$ "not reached" do
33:         $update(y, x, card(P_{sx}), \{y\}, \{x\})$
34:     end for
35: end while
```

A. Update Process

At the moment of processing a given node *x*, from LOC *22* to LOC *26*, it is possible that there exist among its predecessors nodes that are not yet reached. In this case, the treatment of *x* will be made only based on the elements of $\Gamma^{-*}(x)$. The updating process is the operation that completes all the minimal paths of a node x as soon as a node of its predecessors has just been processed or updated. In the worst case, the case where the graph is strongly connected, the overall number of execution of the update process is equal to $1+2+...+(|V|-2) = (|V|-3)(|V|-2)/2$ (no update after the first iteration, only one update after the second iteration, ..., $(|V|-2)$ update after the iteration $(V-1)$).

The nodes of the sub-graph already processed will be updated from the list of the current node paths using a list of suffixes representing the different possibilities of reaching these nodes from current node. These suffixes will be used in the phase of increasing paths. To determine these suffixes, we will explore the sub-graph already processed starting from the current node. In the beginning, the suffix is initialized by null. Before the update process is executed on a given node x, the suffix is first incremented by x.

In order to optimize the update algorithm, we consider the following assertions:

1) The updates of the successors of a given node x will be made only based on the new paths added to P_{SX}.
2) During the update process (the exploration of sub-graph already processed), we do not pass from a node, denoted by x, to these successors only if P_{SX} was powered by new paths.
3) In the update process, if all the successors of a node to update are already processed, it is obvious that the update of this node is useless, since it will not contribute any more in the process of determination of the P_{ST} because all their successors are already processed. It is also obvious that the exploration of the already processed sub-graph must continue in order to explore all the active nodes; The purpose of the update process is to update the active nodes only.

Proof

Assertion 1: let c be the processed node, x a successor of c, and y a successor of x, where $x \in \Gamma^{-*}(c)$ and $y \in \Gamma^{-*}(x)$. Node x will be updated from P_{SC} paths which do not pass through x. The y node will be updated from P_{SC} paths which do not pass through x and y, i.e. the paths that have already been added to P_{SX} and do not pass through y.

Assertion 2: if the node x has not been fed by other paths, forcing these successors will not be powered too (result of assertion 1). It is therefore useless to continue the update process.

Assertion 3: already justified.

In the last assertion, the finalized nodes will not be updated, which will allow us to optimize the memory space, but in this case, the exploration of sub-graph already processed will be maximum, that on is the one hand. On the other hand, the updating of the active nodes, in most cases, will be done directly from the list of current node paths (the node from which we started the exploration of the already processed sub-graph), and this is very expensive. To avoid useless explorations, during the process of updating the list of y-node paths from the list of x-node paths, if the node y is active, then we update P_{SY} from paths of P_{SX}. Otherwise, we do a simple swap of the paths of P_{SX} while moving the paths that can be added to P_{SY} (P_{SX} paths which do not pass through the node y) at the beginning of P_{SX}. In the latter case, the updating process is continued if at least one permutation operation takes place. For more details, see update procedure given in Algorithm 4.

Algorithm 4 $update(x,\ y,\ n,\ suffix,\ TMA)$

```
1: description of input data:
   This procedure will update the $P_{sx}$ from the n first paths of $P_{sy}$.
   $suffix$ is a sequence of nodes used in the path augmentation phase.
   $TMA$ is the set of nodes already traversed from the beginning of update process
2: local variables:
3: nbre: Integer;
4: Begin
5: if $Psx \neq \{\emptyset\}$ then
6:   $nbre \leftarrow 0$
7:   for all $\pi$ $in\ the\ first\ n\ paths\ of\ Psy$ do
8:     if $no\ element\ of\ \{suffix \cup \{x\}\}\ does\ not\ appear\ in\ \pi$ then
9:       $P_{sx} \leftarrow P_{sx} \cup AGM(\pi,\ \{suffix \cup \{x\}\})$
10:      $nbre++$
11:    end if
12:  end for
13:  if $n \neq 0$ then
14:    for all $zin\Gamma^{+}(x)$ do
15:      if $State(z) = "reached"\ and\ z \notin TMA$ then
16:        $update(z, x, nbre, \{x\}, TMA \cup \{x\})$
17:      end if
18:    end for
19:  end if
20: else
21:  $nbre \leftarrow 0$
22:  for all $\pi$ $in\ the\ first\ n\ paths\ of\ Psy$ do
23:    if $no\ element\ of\ \{suffix \cup \{x\}\}\ does\ not\ appear\ in\ \pi$ then
24:      $MovePathAtBeginning(\pi,\ Psy)$
25:      $nbre++$
26:    end if
27:  end for
28:  if $n \neq 0$ then
29:    for all $z\ in\ \Gamma^{+}(x)$ do
30:      if $State(z) = "reached"\ and\ z \notin TMA$ then
31:        $update(z, x, nbre, \{x\}, TMA \cup \{x\})$
32:      end if
33:    end for
34:  end if
35: end if
```

B. The Distance between Node Pair

In graph theory, the distance between two nodes of a graph is the length (in number of edges) of a shorter path between these two nodes. The calculation of this distance can be done by a simple BFS (Breadth First Search) algorithm proceeding as follows: the starting node will be initialized by *0*. During the process “graph traversal”, each node x reached from a given node y will have the distance of y plus one. Another technique for calculating node distances from t can be found in [9]. In this work, we used the *BFS* algorithm to determine the distance of each node from the sink.

C. Undirected Graph Reduction

The reduction of the graphs used in this part consists only of eliminating the arcs outgoing from sink t $\{(t,\ y)\ \in E\}$ and the incoming arcs to the source s $\{(x,s)\ \in E\}$. The justification for this treatment is already given in section 1.3.3.

D. A Numerical Illustration of Minimal Paths Algorithm for General Network

Consider the reduced graph of Figure 1.c with s = 0 and t = 6.

Initialization
$P_{sx} = \{\emptyset\}$ for x=1, 2, 3, 4, 5, and 6.
$P_{SS} = P_{00} = \{(0)\}$
LsOC from 16 to 19:
$To\_Treat \leftarrow \{3,1,2\}, State[3] \leftarrow reached,$
$State[1] \leftarrow reached,\ State[2] \leftarrow reached$
Graph exploration
Iteration N1:
LOC 21: we retrieve node 3 from To_Treat (element whose distance to the sink is the largest). To_Treat becomes {1, 2}.
LsOC 22 to 26: feeding of P_{S3} by the elements P_{SX} where x in $\Gamma^{-}(3)$ which are already processed and which are in our case $\{P_{00}\}$. P_{S3} becomes $\{(0,3)\}$.
LOC 27: $State[3] \leftarrow examined.$
LsOC from 28 to 31: all successors of 3 are already reached, so no item will be added to $To\_Treat.$
LsOC from 32 to 34: since $\Gamma^{-*}(3) = \{\emptyset\}$, there is no update to make.
Iteration N2:
LOC 21: we retrieve node 1 from $To\_Treat.$ To_Treat becomes $\{2\}$.
LsOC from 22 to 26: feeding of P_{S1} by the elements of $\Gamma^{-}(1)$ which are already processed and which are, in our case:

- node 3: since $\Gamma^{+nt}(3) = \{1\}$, P_{S3} will be moved after increase to P_{S1}. P_{S1} becomes {(0, 3, 1)} and P_{S3} becomes empty. After this operation, the state of node 3 will be changed to finalized ($State[3] \leftarrow Finalized$).
- node 0: P_{S1} becomes {(0,1), (0, 3, 1)}.

LOC 27: $State[1] \leftarrow Examined$
LsOC from 28 to 31: feeding of To_Treat by the successors of node 1 whose state is not reached, in our case node 4 and node 5. To_tarit becomes {2, 4, 5}.
LsOC from 32 to 34: call to the update procedure for all element in $\Gamma^{-*}(1) = \{3\}$.
Call N1: update(3, 1, 2, {3},{1})
Since node 3 is finalized, we will move the paths which do not pass through node 3 at the beginning of the P_{S1} list; Moving of (0, 1) at the beginning of P_{S1}. P_{S1} becomes {(0,1), (0, 3, 1)}.
Since the set $\{x \in \Gamma^{+}(3)/(State[x] = Examined$ or $State[x] = finalized)$ and $x \notin \{1,3\}\} = \{\emptyset\}$, there is no recursive call.
Iteration N3
LOC 21: we retrieve node 2 from $To\_Treat.$ To_Treat becomes {4, 5}.
LsOC from 22 to 26: feeding of P_{S2} by the elements of $\Gamma^{-}(2)$ which are already processed and which are, in our case:

- node 0: since $\Gamma^{+nt}(0) = 2$, P_{S0} will be moved after increase to P_{S2}. P_{S2} becomes {(0, 2)} and P_{S0} becomes empty. After this operation, the state of node 0 will be changed to finalized ($State[0] = Finalized$).
- node 1: P_{S2} becomes {(0,1,2), (0,3,1,2), (0, 2)}.

LOC 27: $State[2] \leftarrow Examined$
LsOC from 28 to 31: all successors of 2 are already reached, so no item will be added to $To\_Treat.$
LsOC from 32 to 34: call to the update procedure for all element in $\Gamma^{-*}(2) = 1$.
Call N 1: Update(1, 2, 3, {1},{2})
Since node 1 is active, P_{S1} will be updated from $PS2$ paths. P_{S1} becomes {(0, 2, 1), (0, 3, 1), (0,1)} (number of paths added is 1).
Since the set $\{x \in \Gamma^{+}(1)/(State[x] = Examined$ or $State[x] = finalized)$ and $x \notin \{1,2\}\} = \{3\}$, we'll make the following recursive call: [call N 1.1:Update (3, 1, 1, {3}, {2,1})].
Call N 1.1: Update(3, 1, 1, {3}, {2, 1})
Since node 3 is finalized, we will move the paths of P_{S1} (only the first path that was added to PS1 during the last update is affected by this operation) which do not pass through the node 3 at the beginning.
Moving of (0, 2, 1) to the beginning. P_{S1} becomes {(0, 2, 1), (0,1), (0, 3, 1)}.
Since the set $\{x \in \Gamma^{+}(3)/(State[x] = Examined$ or $State[x] = finalized)$ and $x \notin \{1,2,3\}\}$ is empty, then there is no recursive call.
Iteration N4
LOC 21: we retrieve node 4 from $To\_Treat.$ To_Treat becomes $\{5\}$.
LsOC from 22 to 26: feeding of PS4 by the elements of $\Gamma^{-}(4)$ which are already processed and which are, in our case:

- node 1: P_{S4} becomes {(0,2,1,4), (0, 1, 4), (0,3,1,4)}.

LOC 27: $State[4] \leftarrow Examined$
LsOC form 28 to 31: feeding of To_Treat by the successors of node 4 whose state is not reached, in our case, node 6. To_tarit becomes {5, 6}.
LsOC from 32 to 34: call to the update procedure for all element in $\Gamma^{-*}(4) = \{1\}$.
Call N1: Update(1, 4, 3, {1}, {4})
Since node 1 is active, P_{S1} will be updated from P_{S4} paths.All P_{S4} paths go through node 1, so no P_{S4} path can be added to P_{S1} and therefore no recursive update call.
Iteration N5
LOC 21: we retrieve node 5 from $To\_Treat.$ To_Treat becomes $\{6\}$.
LsOC from 22 to 26: feeding of P_{S5} by the elements of $\Gamma^{-}(5)$ which are already processed and which are, in our case:

- node 2: since $\Gamma^{+nt}(2) = 5$, P_{S2} will be moved after increase to P_{S5}. P_{S5} becomes {(0,1,2,5), (0,3,1,2,5), (0,2,5)} and P_{S2} becomes empty. After this operation, the state of node 2 will be changed to finalized ($State[2] \leftarrow Finalized$).

- node 4: P_{S5} becomes {(0,2,1,4,5), (0, 1, 4,5), (0,3,1,4,5), (0,1,2,5), (0,3,1,2,5), (0,2,5)}.

LOC 27: $State[5] = Examined$
LsOC from 28 to 31: all successors of 5 are already reached, so no item will be added to To_Treat.
LsOC from 32 to 34: call to the update procedure for all element in $\Gamma^{-*}(5) = 2,4$ [call1:Update(2,5,6,{2}, {5}), call2:Update(4, 5,6,{4},{5})].
Call N1: Update(2, 5, 6, {2},{5})
Since node 2 is finalized, we will move the paths of P_{S5} which do not pass through the node 3 at the beginning.
Moving of (0, 1, 4,5) to the beginning of P_{S5}. P_{S5} becomes {(0, 1, 4,5), (0,2,1,4,5), (0,3,1,4,5), (0,1,2,5), (0,3,1,2,5), (0,2,5)}
Moving of (0,3,1,4,5) to the beginning of P_{S5}. P_{S5} becomes {(0,3,1,4,5), (0, 1, 4,5), (0,2,1,4,5), (0,1,2,5), (0,3,1,2,5), (0,2,5)} .
Since the number of displacements is different from zero (equal to 2) and since the set $\{x \in \Gamma^{+}(2)/(State[x] = Examined$ or $State[x] = finalized)$ and $x \notin \{5, 2\}\} = \{1\}$, then we will make the following recursive call: [Update (1, 5, 2, {2,1}, {5,2})].
Call N1.1: Update(1, 5, 2, {2,1},{5,2}).
Since node 1 is finalized, we will move, among the first two paths of P_{S5}, the paths that do not contain node 1 to the beginning of P_{S5}.
The first two paths of P_{S5} go through node 1, so none of them can be added to P_{S1} and therefore no recursive sub call.
Call N2: Update(4, 5, 6, {4},{5})
Since node 4 is active, P_{S4} will be updated from P_{S5} P_{S4} becomes {(0,2,5,4), (0,3,1,2,5,4), (0,1,2,5,4), (0,2,1,4), (0, 1, 4), (0,3,1,4)} (number of paths added is 3)
Since the set $\{x \in \Gamma^{+}(4)/(State[x] = Examined$ or $State[x] = finalized)$ and $x \notin \{5, 4\}\} = \{1\}$, then we will make the following recursive call 2.1: [Update (1, 4, 3 , {1}, {5,4})].
Call N2.1: Update(1, 4, 3, {1},{5,4}).
Since node 1 is finalized, we will move, among the three paths of P_{S4}, the paths that do not contain node 1 to the beginning of P_{S4}.
Moving of (0,2,5,4) to the beginning of P_{S4}. P_{S4} becomes {(0,2,5,4), (0,3,1,2,5,4), (0,1,2,5,4), (0,2,1,4), (0, 1, 4), (0,3,1,4)}
Since the displacement number is different from zero (equal to 1) and since the set $\{x \in \Gamma^{+}(1)/\ (State[x] = Examined$ or $State[x] = finalized)$ and $x \notin \{5, 4, 1\}\} = \{3, 2\}$, then we will make the following recursive calls: [Update (3, 4, 1, {1,3}, {5,4,1}) and Update (2, 4, 1, {1,2}, {5,4,1})].
Call N2.1.1: Update(3, 4, 1, {1,3},{5,4,1})
Since node 3 is finalized, we will move, among the first path of P_{S4}, the paths that do not contain node 3 to the beginning of P_{S4}. Moving of (0,2,5,4) to the beginning of P_{S4}. P_{S4} becomes {(0,2,5,4), (0,3,1,2,5,4), (0,1,2,5,4), (0,2,1,4), (0,1,4), (0,3,1,4)} Since the set $\{x \in \Gamma^{+}(3)/(State[x] = Examined$ or $State[x] = finalized)$ and $x \notin \{5, 4, 1\}\}$ is empty, then there is no recursive call.
Call N2.1.2: Update(2, 4, 1, {1,2},{5,4,1})
Since node 2 is finalized, we will move, among the first path of P_{S4}, the paths that do not contain node 2 to the beginning of P_{S4}.
The first path of P_{S4} goes through 2 so no movement will be made. Therefore no recursive sub call.
Iteration N6
LOC 21: we retrieve node 6 from To_Treat. To_Treat becomes $\{\emptyset\}$.
LsOC from 22 to 26: feeding of P_{S6} by the elements of $\Gamma^{-}(6)$ which are already processed and which are, in our case:

- node 4: since $\Gamma^{+nt}(4) = \{6\}$, P_{S4} will be moved after increase to P_{S9}. P_{S9} becomes {(0,2,5,4,6), (0,3,1,2,5,4,6), (0,1,2,5,4,6), (0,2,1,4,6), (0,1,4,6), (0,3,1,4,6)} and P_{S4} becomes empty. After this operation, the state of node 4 will be changed to finalized (State [4] = Finalized).

- node 5: since $\Gamma^{+nt}(5) = \{6\}$, P_{S5} will be moved after increase to P_{S9}. P_{S9} becomes {(0,3,1,4,5,6), (0, 1,4,5,6), (0,2,1,4,5,6), (0,1,2,5,6), (0,3,1,2,5,6), (0,2,5,6), (0,2,5,4,6), (0,3,1,2,5,4,6), (0,1,2,5,4,6), (0,2,1,4,6), (0,1,4,6), (0,3,1,4,6)} and P_{S5} becomes empty. After this operation, the state of node 5 will be changed to finalized (State [5] = Finalized).

LOC 27: $State[6] = Examined$
LsOC from 28 to 31: all successors of 6 are already reached, so no items will be added to To_Treat.
LsOC from 32 to 34: since the destination node has been processed, the program stops at this point.
Paths found: {(0, 3, 1, 4, 5, 6), (0, 1, 4, 5, 6), (0, 2, 1, 4, 5, 6), (0, 1, 2, 5, 6), (0, 3, 1, 2, 5, 6), (0, 2, 5, 6), (0, 2, 5, 4, 6), (0, 3, 1, 2, 5, 4, 6), (0, 1, 2, 5, 4, 6), (0, 2, 1, 4, 6), (0, 1, 4, 6), (0, 3, 1, 4, 6)}

V. Complexity Analysis

A. Enumeration of Minimal Paths in an Oriented Graph without Cycles (Algorithm 1 and 2)

For the graph reduction algorithm (Algorithm 1), the execution time depends on the number of nodes to be deleted; in the worst case $O(|V| - 2)$. The memory space used by the algorithm is in the order of $O(2 * (|V| - 1))$ with respect to one input list $O(V - 1)$, one list of nodes to delete $O(|V| - 2)$.

For the path enumeration algorithm in an oriented graph without cycles (Algorithm 2), the execution time complexity is in the order $O(|V| + |E|)$; each node is visited once $O(|V|)$. At each node, the search for successors whose predecessors are already processed runs in $O(deg(node))$, which gives $O(|E|\ in\ the\ worst\ case)$. Let λ denote the average number of nodes in a MPs, and π denotes the total number of MPs. The memory space required for the execution of the algorithm is $O(3(|V| - 1) + \lambda * \pi)$, with respect to the buffer memory used for storing paths $O(\lambda * \pi)$, one input list $O(|V| - 1)$, one list used for storing the items to be processed $O(|V| - 1)$, and one list that contains the state of each node $O(|V| - 1)$.

B. Enumeration of Minimal Paths in the General Graph (Algorithm 3 and 4)

The maximum time complexity, in number of tests, for the minimal paths enumeration algorithms based on BFS/DFS (Breadth First Search / Depth First Search) is $O(\lambda * \pi + C)$

where λ indicates the average number of links for each minimal path, and π denotes the total number of minimal paths. This complexity is the result of the exhaustive traversal of all possible branches starting from the source node. This is the case in [21] where the author has used a exhaustive traversal and principle of backtracking which consists of going back as soon as a cycle is detected or when there are no more outgoing neighborhoods. In [9], the same principle is used with the addition of a condition on backtracking. This condition has made it possible to reduce the number of cycles visited, which implies the reduction of number of tests to $O(\lambda * \pi + \overline{C})$. In our algorithm, no test is performed when building the P_{SX} sets from the predecessors. The tests are performed at the update level. In this case, a large number of paths will be built without doing any test. Let $\overline{\pi}$ be the number of paths built at the update level. Consequently, the time complexity of our algorithm can be written as follows: $O(\overline{\lambda} * \overline{\pi} + C)$ where $\overline{\lambda}$ indicates the average number of links for each $\overline{\pi}$. The advantage of our algorithm can be seen as follows, $O(\overline{\lambda} * \overline{\pi}) \ll O(\lambda * \pi)$.

The memory space required by the minimal paths enumeration algorithm (Algorithm 3 and 4) $O(4|v| + (\lambda * \pi) - 6)$ with respect to one input list $O(V - 1\ in\ length)$, one list used for storing the items to be processed $O(|V| - 1\ in\ the\ worst\ case)$, the buffer memory used for storing paths $O(\lambda * \pi\ in\ length)$, one list of distances to the terminal $O(V - 1\ in\ length)$, and one list used for storing the suffix $O(V - 3\ in\ the\ worst\ case)$, where the suffix is an ordered subset of nodes used in the path-augmentation phase at the update level.

The storage complexity of G. Bai et al's algorithm [9] is $O(4(|V| - 1))$, with respect to one input list $L(|V| - 1\ in\ length)$, one path buffer $P(|V|\ in\ the\ worst\ case)$, one distance list $Q(|V| - 1\ in\ the\ worst\ case)$, and one distance checking list $S(|V| - 1\ in\ the\ worst\ case)$. The time complexity of its algorithm is $O(\eta * (\pi + c))$, where c denotes the number of cycles in the network who are visited by its algorithm.

The storage complexity of G. Bai et al's algorithm is less than ours. But in order to compare our algorithm to that of Bai, and not to count the execution time necessary for output immediately each path found (screen display or storage in a file or others), when an minimal path is found, it will be kept in memory until the end of the execution. This makes the memory space necessary for the execution of the Bai algorithm $O(3(|V| - 1) + \lambda * \pi)$.

VI. Benchmarks and Test

A. Benchmarks

To test our algorithm, we have used a set of networks taken in the literature. The network presented in Fig. 2 is taken from [1]. A classical grid network, used in [9], [21], is shown in Fig. 3. We have implemented and tested our algorithm using C language. All tests were performed on a personal computer equipped with CPU being an Intel Core i3 1.7 GHz, and with 4Gb RAM.

In order to compare our method to that of Bai which allows to enumerate all the minimal paths in the general graphs, we used our second algorithm (Algorithm 3) in all the tests we performed.

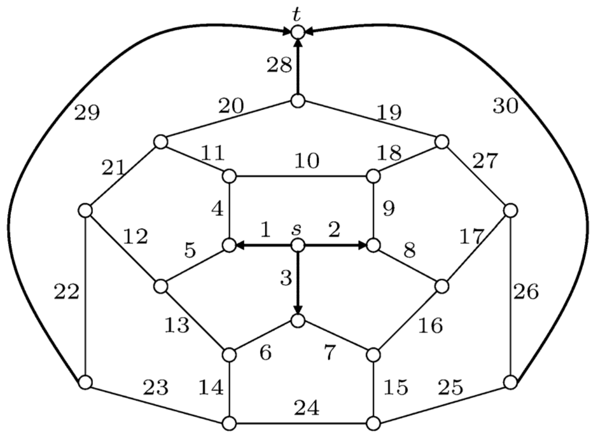

Fig. 2. The benchmark network used in [21], [9], [27]

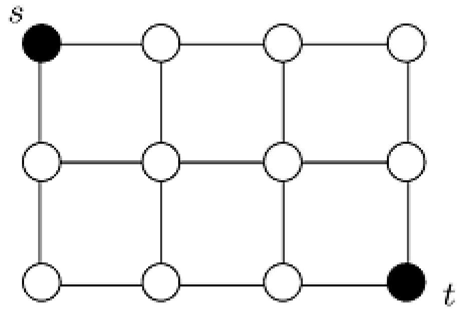

Fig. 3. The benchmark network used in [21], [9]

B. Comparison with G. Bai et al's Algorithm

In order to show the effectiveness of our algorithm, we compared it with that of G. Bai [9]. For that, we are interested in the required execution time with respect to different networks. The first test presented in this study is done using the benchmark network given by Luo and Trivedi [27], as shown in Fig. 2. Using the proposed algorithm, the number of minimal paths found is 780, which agrees the result in [9] and in [21]. The execution time for the proposed algorithm and the Bai's algorithm are 0.4320ms and 2.668ms, respectively. The ratio, which is defined as the ratio of the CPU time of the Bai's algorithm to the proposed algorithm, is about 6.1759, indicating that the proposed algorithm is 6.1759 times faster than Bai's algorithm in finding all the minimal paths of the benchmark network.

In the second test, we used the classical grid networks. A typical example of 12 nodes and 17 edges is shown in Fig. 3. On all the networks tested in this experiment, the two algorithms generate the same sets of minimal paths. Fig. 4 shows a comparison of the average CPU times (in milliseconds) of 14 grid networks for the Bai's algorithm, and our second algorithm. In order to illustrate the difference between these two algorithms according to the size of the network, the ratio of the CPU time of the Bai's algorithm over the proposed algorithm is given in Fig. 5. As we can see, as the network size increases, the efficiency of our algorithm increases accordingly.

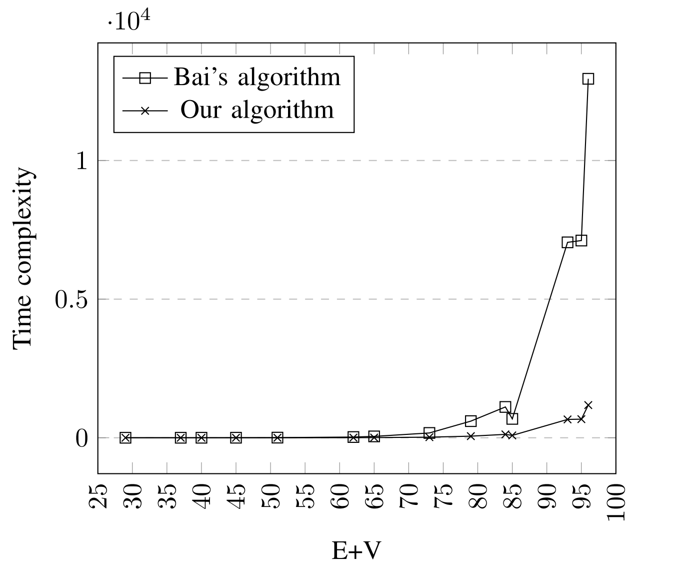

Fig. 4. Comparison of the CPU time of the Bai's algorithm to the proposed algorithm.

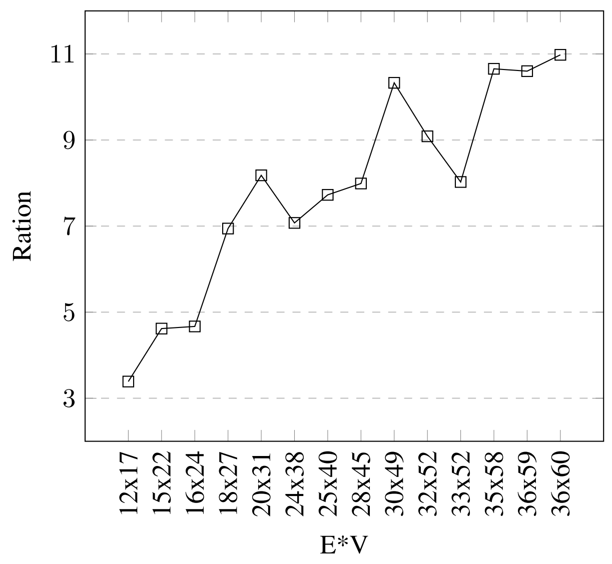

Fig. 5. Ratio of the CPU time of the Bai's algorithm to the proposed algorithm.

VII. Enumeration of MPs for networks with multiple source/sink nodes

As introduced in [28], a multi-terminals network is said to be operative if there exist operating paths between each pair of node (s, t) such that s belongs to the set of source nodes and t belongs to the set of sink nodes.

The simple way to extend the minimal paths enumeration algorithm introduced previously to the multi-terminals' case is to convert the multi-terminals network into a simple binary network. To do this, we can use the classical technique introduced in [28]. The technique is as follows: first, we add two nodes that will play the role of the two terminals in the new network, naming these nodes artificial source and artificial sink. The second step is the addition of artificial direct links from artificial source node to the source nodes. The latest step

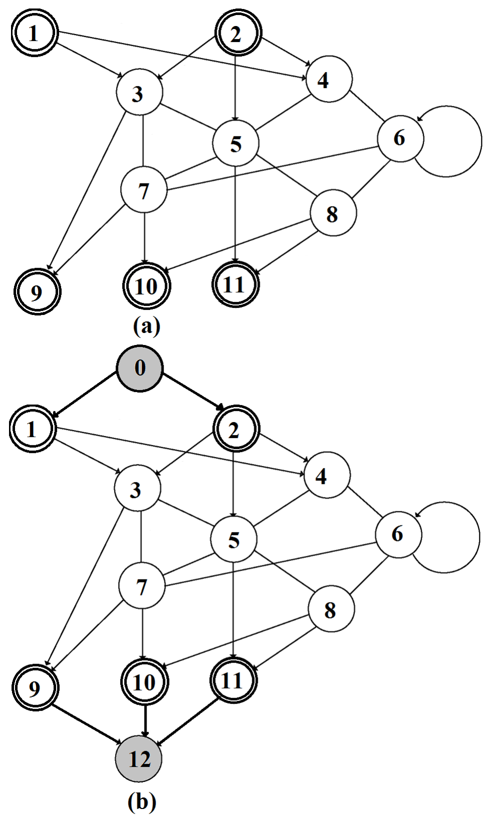

Fig. 6. (a): Multi-terminal network (sources s={1,2} sinks t={9, 10, 11}) from [21] and its binary form (b).

consists of adding the artificial direct links from sink nodes to the artificial sink node. Fig. 6 shows the benchmark network from [21] and its transformation. This technique is also used by Bai [9].

To test this technique, we used the network shown in Fig. 6. Using the proposed algorithm, the number of minimal paths found is 145, which agrees with the results in [21] and the same number is obtained using Bai's algorithm. The execution time for the proposed algorithm and the Bai's algorithm are 0.052ms and 0.135ms, respectively, and the ratio is about 2.5961.

VIII. Conclusion

In this paper, we have proposed a new method that finds all the minimal paths in the graph. We started by presenting a version for graphs without cycles. This version was subsequently extended to the general case (oriented, undirected and mixed graphs). We also presented an algorithm for graph reduction. For the case of graphs with multi-terminals, we adopted, as in [21], [9] and others, the method introduced in [28].

The analysis of the complexity of our algorithm and the comparison with that of Bai's algorithm show that the proposed method herein is very efficient.

Another advantage of our algorithm is the possibility to implement a large part of the algorithm, such as update operations, using parallel programming. This will allow us to further improve the effectiveness of our approach.

Acknowledgment

The author would like to thank Youssef Laaraj (ELT Researcher at Mohammed V University, Rabat, Morocco) for its linguistic revision of the manuscript.

References

[1] B. Roberts, "Estimating the Number of s-t Paths in a Graph," *Journal of Graph Algorithms and Applications*, vol. 11, pp. 195–214, 2007.

[2] Y.-K.Lin and P. C. Chang, "Maintenance reliability estimation for a cloud computing network with nodes failure," *Expert Systems with Applications*, vol. 38, pp. 14–185, 2011.

[3] S. G. C. Lin, "On performance evaluation of ERP systems with fuzzy mathematics," *Expert Systems with Applications*, vol. 36, pp. 6362–6367, 2009.

[4] S. G. Chen, "Optimal device planning and performance evaluation in AMS," in *APARM 2010*, Wellington, New Zealand,2010, 2010, pp. 113–120.

[5] Y.-K.Lin and C. T. Yeh, "Determine the optimal double-component assignment for a stochastic computer network," *Omega*, vol. 40, no. 1, pp. 120–130, 2012.

[6] S.Rai and S. Soh, "A computer approach for reliability evaluation of telecommunication networks with heterogeneous link—capacities," *IEEE Trans. Reliability*, vol. 40, pp. 441–451, 1991.

[7] Y. K. Lin, "A novel algorithm to evaluate the performance of stochastic transportation systems, Expert Systems with Applications," *Expert Systems with Applications*, vol. 37, no. 2, pp. 968–973, 2010.

[8] P.Doulliez and J. Jamoulle, "Transportation networks with random arc capacities," *Recherche Operationnelle*, vol. 3, pp. 45–60, 1972.

[9] G. H. Bai, Z. G. Tian, and M. J. Zuo, "An improved algorithm for finding all minimal paths in a network," *Reliability Engineering and System Safety*, vol. 150, pp. 1–10, 2016.

[10] Y. Shen, "A New Simple Algorithm for Enumerating all Minimal Paths and Cuts of a Graph," *Microelectronics and Reliability*, vol. 35, no. 6, pp. 973–976, 1995.

[11] S. Rai and K. K. Aggarwal, "On complementation of pathsets and cutsets," *IEEE Trans. Reliab*, vol. 29, pp. 139–140, 1980.

[12] D. R. Shier and A. E. Whited, "Algorithms for generating Minimal Cutsets by Inversion," *IEEE Transactions on Reliability*, vol. R-34, no. 4, pp. 314–319, 1985.

[13] S. S. Elias, N. Mokhles, and S. A. N. Ibrahim, "A New Technique in a Cutset Evaluation," *Microelecfironics & Realiability*, vol. 33, no. 9, pp. 1351–1355, 1993.

[14] H. schabe, "An Improved Algorithm For Cutset Evaluation From Paths," *Microelecfironics & Realiability*, vol. 35, no. 5, pp. 783–787, 1995.

[15] W. Yeh, "A new approach to evaluate reliability of multistate networks under the cost constraint," *Omega*, vol. 33, pp. 203–209, 2005.

[16] ——, "Multistate network reliability evaluation under the maintenance cost constraint," *Int J. Production Economics*, vol. 88, pp. 73–83, 2004.

[17] A. M. Al-Ghanim, "A heuristic technique for generating path and cutsets of a general net-work," *Computers & Industrial Engineering*, vol. 36, pp. 45–55, 1999.

[18] W. C. Yeh, "A simple heuristic algorithm for generating all minimal paths," *IEEE TransReliab*, vol. 56, no. 3, pp. 488–494, 2007.

[19] W. Yeh, "Search for minimal paths in modified networks ," *Reliability Engineering & System Safety*, vol. 75, pp. 389–395, 2002.

[20] K. Kobayashi and H. Yamamoto, "A new algorithm enumerating all minimal paths in aspars enetwork," *Reliability Engineering and System Safety*, vol. 65, no. 1, pp. 11–15, 1999.

[21] S. Chen and Y. K. Lin, "Search for all minimal paths in a general large flow network," *IEEE TransReliab*, vol. 61, no. 4, pp. 949–956, 2012.

[22] J. Nahman, "Enumeration of mps of modified networks," *Microelectronics and Reliability*, vol. 34, pp. 475–484, 1994.

[23] S. Chen, Y. C. Guo, and W. Z. Zhou, "Search for All Minimal Paths With Backtracking," in *The 16th ISSAT International Conference on Reliability and Quality in Design. Washington DC USA*, 2010, pp. 425–429.

[24] S. Chen, "Search for all minimal paths in a general directed flow network with unreliable nodes," *International Journal of Reliability and Quality Performance*, vol. 2, no. 2, pp. 63–70, 2011.

[25] C. Colbourn, *The combinatorics of network reliability*. New York, NY: Oxford University Press, 1987.

[26] J. Hagstrom, "Note on independence of arcs in antiparallel for network flow problems," *Networks*, vol. 14, pp. 567–570, 1984.

[27] T. Luo and K. S. Trivedi, "A improved algorithm for coherent-system reliability," *IEEE Trans. Reliability*, vol. 47, no. 1, pp. 73–78, 1998.

[28] M. Ball, C. Colbourn, and J. Provan, "Network reliability," in *Handbooks in operations research and management science*, vol. 7, 1995, p. 673–762.