Trends in Computational and Applied Mathematics, **24**, N. 1 (2023), 19-34
Sociedade Brasileira de Matemática Aplicada e Computacional
Online version ISSN 2676-0029
www.scielo.br/tcam
doi: 10.5540/tcam.2022.024.01.00019

An Improved Vectorization Algorithm to Solve the *d-MP* Problem

M. FORGHANI-ELAHABAD[1*] and E. FRANCESQUINI[2]

Received on December 23, 2021 / Accepted on July 1, 2022

ABSTRACT. The *d*-minimal path (*d-MP*) problem is to find all the system state vectors (SSV) under which *d* units of data can be transmitted from a source node to a destination node in a stochastic-flow network (SFN). This problem has been very attractive in the last decades as one can compute the exact amount of the network's reliability through the *d-MP*s. Although several algorithms have been proposed in the literature to address the problem, the research continues because it is NP-hard. Since the number of *d-MP*s grows exponentially with the size of the network, the available algorithms in the literature are not so practical. Hence, we employ the vectorization techniques for proposing an improved algorithm to address the problem. We conduct many experimental results on the known benchmarks and two hundred randomly generated SFNs in the sense of performance profile introduced by Dolan and Moré. The experimental results show the vectorization algorithm to be considerably more efficient than the non-vectorization ones.

Keywords: vectorization techniques, network reliability, *d-MP* problem, stochastic-flow network.

1 INTRODUCTION

A stochastic-flow network (SFN) is a flow network whose components, and consequently itself, can have more than two different states [14, 20, 28]. The components' states in an SFN are represented by a vector, the so-called system state vector (SSV), in which each component represents the state of a corresponding component of the SFN. There are three types of data transmission in an SFN: (1) *two*-terminal, in which the data is transmitted from a source node to a sink node, (2) *k*-terminal, in which the data is transmitted among *k* components, and (3) *all*-terminal, in which the data is transmitted among all the components in the network. The focus of this work is on the first case, where the network reliability at demand level d, denoted by R_d, is the probability of transmitting at least d units of data from a source node to a destination node in the network [15, 19, 31].

*Corresponding author: Majid Forghani-elahabad – E-mail: m.forghani@ufabc.edu.br
[1]Center of Mathematics, Computing and Cognition–CMCC, Federal University of ABC–UFABC, Santo André, SP, Brazil – E-mail: m.forghani@ufabc.edu.br https://orcid.org/0000-0003-1691-7633
[2]Center of Mathematics, Computing and Cognition–CMCC, Federal University of ABC–UFABC, Santo André, SP, Brazil – E-mail: e.francesquini@ufabc.edu.br https://orcid.org/0000-0002-5374-2521

System reliability is indeed one of the most essential and widely-used indices for measuring the performance of real-world systems. One can model many real-world systems such as transportation networks, computer and communication networks, distribution systems, and power transmission systems as SFNs [4, 13, 22, 38]. Therefore, the reliability evaluation of SFNs has been a very attractive problem in the last decades, and several direct and indirect algorithms have been proposed in the literature to address it [1, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 22, 23, 25, 26, 27, 28, 29, 30, 31, 34, 35, 36, 37, 39].

The network reliability, R_d, can be assessed indirectly in terms of minimal paths (MPs) [4, 9, 11, 12, 17, 23, 25, 30, 31, 34, 36, 39] or minimal cuts (MCs) [1, 5, 7, 10, 13, 14, 16, 22, 27, 28, 29, 35] through a three-stage process; (1) calculating all the MPs or MCs, (2) determining all the *d-MP*s in the case of MPs or d-minimal cuts (*d-MC*) in the case of MCs, and (3) calculating a union probability as the network reliability.

We focus on the second stage and *d-MP* problem which aims to find all the *d-MP*s.

A path is a sequence of adjacent arcs through which the data can be sent from a source node to a sink node, and an MP is a path with no proper subset being a path from the source node to the sink node.

Several authors have worked to improve the solution of the *d-MP* problem. Forghani-elahabad and Bonani [9] proposed an improved algorithm to address the problem and showed its efficiency in comparison with others in the literature by providing the complexity and numerical results. Lin and Chen [26] proposed a new algorithm to the problem by using the maximal flow evaluation and fast enumeration technique. Based on a breadth-first search technique, Chen et al. [4] proposed a recursive algorithm for the determination of all the *d-MP*s for all the demand values d. Considering a budget constraint, Forghani-elahabad and Kagan [11] proposed an improved algorithm for the determination of all the *d-MP*s within a limited budget. The authors showed their proposed algorithm's efficiency and explained how it could be adopted to assess the reliability of a multi-source multi-sink communication smart grid network. Balan and Traldi [2] first showed that listing the MPs by their sizes is not indeed an optimal choice for the sum of disjoint products algorithms and then proposed a more efficient strategy to be used. By employing heuristic and recursive techniques and using the state-space decomposition technique, Bai et al. [1] proposed an MP-based algorithm to address the problem. In [34], a novel technique was presented to identify the duplicate solutions by using which an improved MP-based algorithm was proposed for reliability evaluation of SFNs. Pointing out some of the obstacles in the available algorithms in the literature, Yeh [37] proposed an improved addition-based algorithm to overcome those obstacles. Yeh and Zuo [39] proposed a subtraction-based algorithm to determine all the *d-MP*s for all the d values and showed its practical efficiency to the available algorithms in the literature. Forghani-elahabad et al. [12] developed an approximation algorithm based on the exact algorithms to address the problem. By presenting a simple method for checking the candidates and a new technique to remove the duplicates, Niu et al. [31] proposed an improved algorithm to address the problem. The authors also conducted a sensitivity analysis to explore the most effective arc in improving network reliability. Lamalem et al. [23] proposed an addition-based

algorithm to solve the problem for all the d values that recognized which MPs can lead to the valid $(i+1)$-MPs starting from *i-MP*s.

However, no vectorization algorithms have been published in the literature on this problem to the best of our knowledge. Hence, in this work, we first state a slightly improved version of an available algorithm in the literature, based on which we propose a vectorization algorithm to address the problem efficiently. These are the main contributions of this work. We also show its practical efficiency by conducting several experimental results on the known benchmarks and one thousand randomly generated SFNs in the sense of the performance profile introduced by Dolan and Moré [6].

The remaining of the paper is organized as follows. Section 2 states the required notations, acronyms and assumptions and also provides some preliminaries on the problem. Section 3 presents a background on the vectorization process. The experimental results are provided in Section 4, and the concluding remarks are given in Section 5.

2 THE *d-MP* PROBLEM

2.1 Acronyms

MP	Minimal path
SFN	Stochastic-flow network
SSV	System state vector
FFV	Feasible flow vector

2.2 Nomenclature

Following the literature [14, 23, 25, 26, 27, 35], we use the following nomenclature.

- For two system state vectors (SSV) $X = (x_1, x_2, \cdots, x_m)$ and $Y = (y_1, y_2, \cdots, y_m)$, it is said that $X \preceq Y$ if $x_i \leq y_i$, for any $i = 1, 2, \cdots, m$. And $X \prec Y$ when $X \preceq Y$ and there exists at least one $1 \leq j \leq m$ with $x_j < y_j$. For instance, setting $X = (2,2,2)$, $Y = (2,2,1)$ and $Z = (2,1,2)$, we have $Y \prec X$, $Y \not\prec Z$ and $Z \not\prec Y$.
- Letting Ψ be an arbitrary set of vectors, we say $X \in \Psi$ is a minimal vector if there is no any $Y \in \Psi$ such that $Y < X$. For instance, every vector in $\{(1, 2, 3), (2, 3, 1), (3, 2, 1)\}$ is a minimal vector.

2.3 Assumptions

We consider the same assumptions as many other related works in the literature [1, 4, 9, 14, 25, 26, 28, 30, 31, 35, 39]. We note that assumptions 1 to 4 are required to have an integer SFN with reliable nodes, and assumption five is meant to highlight that finding the minimal paths is not considered a part of our algorithm.

1. The capacity of arc $a_i \in A$ takes random integer values from $\{0,1,\cdots,M_i\}$ according to a given probability distribution function, for $i = 1,2,\cdots,m$.
2. Every node is perfectly reliable.
3. The arcs' capacities are statistically independent one from the other.
4. Flow in the network satisfies the flow conservation law.
5. All the minimal paths of the network are given in advance.

2.4 The algorithm

Let $G = G(N,A,M)$ be a stochastic-flow network (SFN), where $N = \{1,2,\cdots,n\}$ is the set of nodes with nodes 1 and n being the source and destination nodes, respectively, $A = \{a_i |\ 1 \leq i \leq m\}$ is the set of arcs, and $M = (M_1, M_2, \cdots,\ M_m)$ is a vector in which M_i gives the maximum capacity of arc a_i, for $i = 1,2,\cdots,m$. The vector $X = (x_1, x_2, \cdots, x_m)$ is a state vector in which x_i represents the current capacity of a_i, for $i = 1,2,\cdots,m$. Let $V(X)$ be the maximum flow of the network from nodes 1 to n under SSV X, $Z(X) = \{a_i \in A | x_i > 0\}$ be the set of arcs with positive capacity, and $e_i = 0(a_i)$ be a system vector in which the capacity level is 1 for a_i and 0 for the other arcs. Let also $P_1,\ P_2,\ \cdots,\ P_h$ be all the MPs in the network (so h is the number of all MPs) and $K_j = \min\{M_r | a_r \in P_j\}$ be the capacity of the jth MP, for $j = 1,2,\cdots,h$.

For instance, in the given network in Fig. 1, we have $N = \{1,2,3,4\}$ and $A = \{a_1, a_2, a_3, a_4, a_5\}$. Assuming $M = (3,1,1,2,2)$, any non-negative integer-valued vector $(x_1, x_2, x_3, x_4, x_5) \leq M$ can be considered as an SSV for the network. We also have four MPs, $P_1 = \{a_1, a_4\}$, $P_2 = \{a_1, a_3, a_5\}$, $P_3 = \{a_2, a_5\}$, and $P_4 = \{a_2, a_3, a_4\}$ in this network. So, considering $M = (3,1,1,2,2)$, it is easy to see that $K_1 = \min\{M_1, M_4\} = \min\{3,2\} = 2$, $K_2 = \min\{M_1, M_3, M_5\} = 1$, $K_3 = \min\{M_2, M_5\} = 1$ and $K_4 = \min\{M_2, M_3, M_4\} = 1$. Next, we have three definitions from the literature [12, 31].

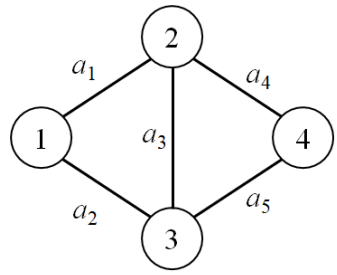

Figure 1: A simple benchmark network example taken from the literature [9].

Definition 2.1. *Assuming f_j as the amount of flow on MP P_j, for $j = 1, \cdots, h$, the vector $F = (f_1, f_2, \cdots, f_h)$ is called a feasible flow vector (FFV) at demand level d, denoted by d-FFV here, when it satisfies the following system.*

$$\begin{array}{ll} (i)\ f_1 + f_2 + \cdots + f_h = d, & \\ (ii)\ 0 \le f_j \le \min\{K_j, d\}, & j = 1, 2, \cdots, h, \\ (iii)\ \sum_{j:\ a_i \in P_j} f_j \le \min\{M_i, d\}, & i = 1, 2, \cdots, m. \end{array} \tag{2.1}$$

For example, it can be calculated that $f_1 = f_2 = f_3 = 1$ and $f_4 = 0$ satisfy the system (2.1) for $d = 3$ in the given network in Fig. 1. Therefore, $F = (1, 1, 1, 0)$ is a 3-*FFV* in this network.

Definition 2.2. *An SSV $X = (x_1, x_2, \cdots, x_m)$ is a d-MP if and only if the following system is satisfied.*

$$\begin{array}{l} (i)\ V(X) = d, \\ (ii)\ V(X - e_i) = d - 1, \quad \text{for each } i \text{ that } a_i \in Z(X). \end{array} \tag{2.2}$$

For example, considering $X = (2, 1, 1, 1, 2)$ in the network given in Fig. 1, we have $V(X) = 3$, $V(X - e_1) = V(1, 1, 1, 1, 2) = 1$, $V(X - e_2) = 1$, $V(X - e_3) = 1$, $V(X - e_4) = 1$, and $V(X - e_5) = 1$. As a result, X is a 3-*MP* for this network.

Definition 2.3. *An SSV $X = (x_1, x_2, \cdots, x_m)$ is called a d-MP candidate when there is a d-FFV, say $F = (f_1, \cdots, f_h)$, that satisfies the following equation:*

$$x_i = \sum_{j:\ a_i \in P_j} f_j, \qquad \forall i = 1, 2, \cdots, m. \tag{2.3}$$

The obtained *d-MP* candidate from a *d-FFV* through Eq. (2.3) is called the associated *d-MP* candidate with that *d-FFV*. For instance, we know that $F = (1, 1, 1, 0)$ is a 3-*FFV* in Fig. 1. Now, we use Eq. (2.3) to calculate its associated SSV. We have $x_1 = \sum_{j:\ a_1 \in P_j} f_j = f_1 + f_2 = 1 + 1 = 2$, $x_2 = \sum_{j:\ a_2 \in P_j} f_j = f_3 + f_4 = 1 + 0 = 1$, $x_3 = \sum_{j:\ a_3 \in P_j} f_j = f_2 + f_4 = 1 + 0 = 1$, $x_4 = \sum_{j:\ a_4 \in P_j} f_j = f_1 + f_4 = 1 + 0 = 1$, $x_5 = \sum_{j:\ a_5 \in P_j} f_j = f_2 + f_3 = 1 + 1 = 2$. As a result, $X = (2, 1, 1, 1, 2)$ is a 3-*MP* candidate associated with $F = (1, 1, 1, 0)$. Notably, this vector is also a (real) 3-*MP*. The following result, which is proven in [25], shows the relation between the (real) *d-MP*s and the candidates.

Theorem 2.1. *Every d-MP is a d-MP candidate.*

Although one can use the given condition in Definition 2.2 to check each candidate for being a *d-MP*, Lin and his co-workers [25] proposed the following theorem which can be employed to determine all the *d-MP*s among the candidates.

Theorem 2.2. *Let Ψ_d be the set of all the d-MP candidates. Then, $\Psi_{d,min} = \{X |\ X \text{ is a minimal vector in } \Psi\}$ is the set of all the d-MPs.*

We remind that a vector $X \in \Psi$ is a minimal vector if there is no other vector $Y \in \Psi$ such that $Y < X$, namely, X is not necessarily less than all the vectors in Ψ; however, there is no another vector in Ψ less than it. Therefore, one can: (1) find all the *d-FFV*s by solving the Diophantine system (2.1); (2) then calculate the associated *d-MP* candidates with the *d-FFV*s by using Eq. (2.3); and finally (3) remove the possible duplicates and the non-minimal vectors to determine the set of all the *d-MP*s.

The proposed algorithm in [25] does not remove the duplicate solutions first, and in fact, it removes the duplicates and non-minimal vectors simultaneously. Hence, in the worst case, it needs to compare all the vectors. However, suppose one first removes the duplicates, which can be done efficiently using vectorization techniques, and subsequently removes the non-minimal vectors. In that case, we have less work to do in the later stage, and, in practice, keeping a reduced number of items in memory could lead to better performance. As a result, Algorithm 1, shown below, which first removes the duplicates and then the non-minimal vectors, is a slightly improved version of the proposed algorithm by Lin et al. [25].

Algorithm 1
Step 1. Solve the system (2.1) for determination of all the *d-FFV*s.
Step 2. Use Eq. (2.3) to calculate the associate *d-MP* candidate with each *d-FFV* obtained in Step 1.
Step 3. Remove the duplicate *d-MP* candidates.
Step 4. Remove the non-minimal *d-MP* candidates.

Although some algorithms in the literature are more efficient than Algorithm 1, they are either not efficient enough for using in real-world systems or not directly optimizable by vectorization, parallelism, or cache-friendly approaches. On the other hand, the focus of this paper is to show how the usage of the vectorization techniques can be effective on the *d-MP* problem algorithms. Therefore, in this case, the choice of algorithm is arbitrary as any of them would be adequate (though some are more convenient than others) to generate the experimental evaluation results shown in Section 4.

3 VECTORIZATION

Vectorization can be described as the process of converting the application of an operation to a single value (a scalar) to the application of the same operation to multiple values (a vector) at once [32]. It is, therefore, a parallelization technique that can improve the performance of many kinds of computer loads. It should not be confused with the application of standard parallelization techniques, which use multiple processors to achieve similar results since vector operations happen in the context of a single processor and a single instruction [24].

Vector processors have existed in one form or another since the ’70s, and virtually all modern processors provide some form of SIMD (Single Instruction Multiple Data) capabilities. Intel x86,

for example, offers multiple versions of their SSE and AVX extensions. Other processor vendors such as ARM also provide extensions such as Neon. More recently, GPUs have also begun to include vector instructions [21].

Invariably all these approaches work using long registers that are usually several times the size of regular registers used for scalar operations. While 128 and 256-bit registers are commonplace, 4096-bit registers are also available in some specialized architectures. For example, consider 128-bit vector registers. Typical integers are stored in 32 bits which means that the time it takes to, for instance, sum one pair of integers using scalar operations is the same as summing four pairs of integers when using vector instructions.

Most optimizing compilers and numeric computing suites such as Mathematica, MATLAB, and NumPy offer some support for SIMD operations. However, effectively and automatically vectorizing code that was not explicitly written to profit from these SIMD instructions is still a very active research problem in Computer Science.

In this work, we leverage vectorization explicitly, using 128-bit vectors to improve the algorithm's performance. In particular, for increased compatibility we focus on Intel's x86 architecture with SSE2 extensions. While some more recent extensions such as SSE4.3 and AVX offer larger vector registers, these extensions are only available in most recent processors[1].

To exemplify how vector instructions can be used, let us examine a simple example in which we want do compare if two vectors are equal, element-wise. A traditional implementation might look like this:

```
1 int vequals (int n, const uint8_t  *v1, const uint8_t  *v2) {
2         for (int i = 0; i < n; ++i)
3             if (v1[i] != v2[i])
4                 return 0;
5         return 1;
6 }
```

We compare each element of `v1` to its respective element in `v2` (lines 2-4). If any of these comparisons determines that a pair of elements are not equal, we stop the execution and return 0 (false, *i.e.*, vectors are not equal) in line 4. If, after all the elements have been compared, we have not found a pair which was different, we conclude the vectors are the same and return 1 (true) in line 5.

[1]Intel Intrinsics Guide, Version 3.6.1. Last accessed in December 2021. Available at: `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`.

This same idea is used in the vectorized version of the algorithm:

```
1  int vequals2 (int n, const uint8_t *v1, const uint8_t *v2) {
2          // Each char is 8 bits, thus we can perform 16 comparisons at
3          // once using a 128-bit register. This code assumes n is
4          // multiple of 16.
5          for (int i = 0; i < n; i += 16) {
6                  // The following lines load 16 integers from each vector
7                  // starting at index into 128-bit registers
8                  __m128i d1 = _mm_loadu_si128((__m128i *)&v1[i]);
9                  __m128i d2 = _mm_loadu_si128((__m128i *)&v2[i]);
10                 // Using a single machine instruction, compares the 16
11                 // integers
12                 __m128i eq = _mm_cmpeq_epi8(d1, d2);
13                 // Since each byte only contains a boolean value, movemask
14                 // gathers all booleans into a single 32-bit integer.
15                 int ieq = _mm_movemask_epi8(eq) == 0xffff;
16                 // one-bit difference is enough to determine inequality
17                 if (!ieq)
18                     return 0;
19         }
20         return 1;
21 }
```

The difference is that, in this case, we perform multiple comparisons per step. The code starts with a loop (line 5-19) that scans both vectors. Since we are performing 16 comparisons at once, we step 16 positions at a time. Then, using special compiler intrinsic functions we load 16 integers into 128-bit registers (lines 8-9). The comparison of these 16-element vectors is done in a single instruction (line 12). The results are placed in a vector, also with 16 elements. To avoid comparing each of these 16 elements one at a time (and thus doing exactly the same thing we did with the traditional approach) we use another intrinsic function (line 15) which is able to, in a single instruction, gather the results of all comparisons into a single 32 bit integer (although we use only the least significant 16 bits). If any of these bits is 0, it means that at least a pair of elements of the vectors is different and thus we can stop the execution and return 0 (line 17-18). If no such element is found, we conclude the vectors are equal (line 20).

We use vector instructions to compare, sum, determine minimality, and sort vectors (to remove duplicates) as explained in Section 2.4.

4 EXPERIMENTAL RESULTS

We first compare three versions of Algorithm 1 on known benchmarks. Then, we conduct numerical results on two hundred randomly generated test problems in the sense of the performance

profile introduced by Dolan and Moré [6]. These results show clearly the practical efficiency of using vectorization techniques in solving the *d-MP* problem.

4.1 Comparing three versions on known benchmarks

Here, we make several numerical comparisons between the three versions of Algorithm 1. The first one (*A1*) is Algorithm 1. The second one (*MA*) is an improved version of Algorithm 1 in which steps 2 and 3 are merged so that no duplicate candidate is generated. The third version (*VMA*) is the vectorized version of the second one.

The experiments were done on a computer Intel(R) Core(TM) i7-7500U CPU 2.7 GHz, with 16 GB of RAM. We employ three benchmark networks taken from the literature given in Figs. 2, 3 and 4. The maximum capacities of arcs in all of these benchmarks are set to be 3, i.e., $M_i = 3$.

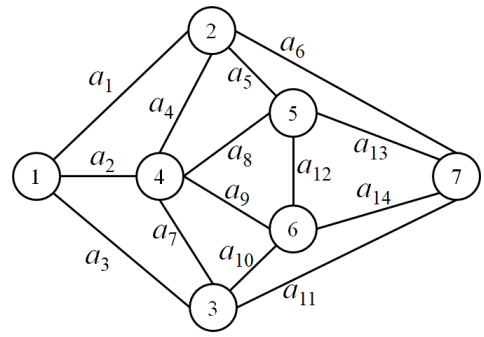

Figure 2: A benchmark network example with 14 arcs taken from the literature [9].

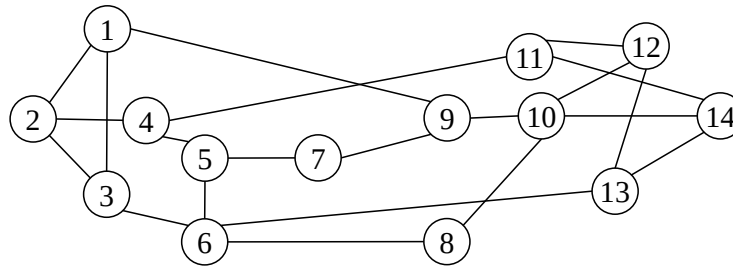

Figure 3: The NSF benchmark network with 14 nodes and 101 MPs taken from [12].

All the generated results are summarized in Tables 1 and 2. The columns in the tables are d, the demand value, n_{can}, the number of candidates, n_{d-MP}, the number of *d-MP*s. Also, t_{A1}, t_{MA}, t_{VMA} are respectively the running times of A1, MA and VMA. We note that Table 2 provides the related results to the general network in Fig. 4 whose size varies with u, and the first column in this table, u, lists the parameter used to generate the corresponding network. Comparing the running times of the three algorithms given in Table 1 clearly shows the practical efficiency of MA and VMA

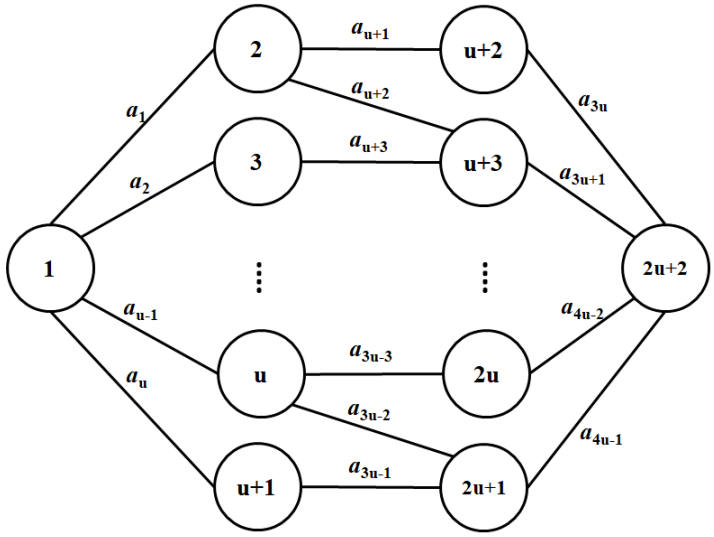

Figure 4: The general network topology taken from the literature [9].

compared to A1. Hence, we did not run A1 on the general network topology resulting in the absence of column t_{A1} in Table 2. The presented results in the tables are an average of at least ten executions. The standard deviation of the samples is shown alongside the average.

Table 1: The final results on the benchmarks given in Figs. 2 and 3.

Benchmarks	d	n_{can}	n_{d-MP}	t_{A1} (s)	t_{MA} (s)	t_{VMA} (s)	t_{MA}/t_{VMA}
Fig. 2	4	76530	8071	474,18 ± 0,25	3.52 ±0.02	1.63 ± 0.00	2.16
Fig. 2	5	180266	12292	14,15 ± 0,07	16.31 ±0.15	5.84 ± 0.01	2.79
Fig. 2	6	220762	12639	197,47 ± 0,07	21.54 ±0.13	7.99 ± 0.04	2.70
Fig. 2	7	149408	7350	958,25 ± 0,38	9.76 ±0.08	4.95 ± 0.02	1.97
Fig. 2	8	62748	3011	1354,26 ± 4,93	4.58 ±0.04	3.98 ± 0.04	1.15
						Geo. Mean	2.06
Fig. 3	4	616152	18950	717,18 ± 1,13	132.95 ±0.33	38.80 ± 0.07	3.43
Fig. 3	5	1069403	20153	3413,77 ± 1,24	229.83 ±0.34	70.49 ± 0.10	3.26
Fig. 3	6	689297	12387	5006,83 ± 2,45	100.61 ±0.21	39.40 ± 0.06	2.55
Fig. 3	7	193204	3109	1371,52 ± 0,27	25.69 ±0.05	21.85 ± 0.21	1.18
Fig. 3	8	24170	438	78,41 ± 0,05	25.25 ±0.10	25.31 ± 0.20	1.00
						Geo. Mean	2.02
						Full Geo. Mean	2.04

Table 2: The final results on the general network topology given in Fig. 4.

u	d	n_{can}	n_{d-MP}	t_{MA} (s)	t_{VMA} (s)	t_{MA}/t_{VMA}
	10	93511	6485	4,42 $\pm$ 0.04	2,38 $\pm$ 0.02	1,86
	11	44292	2702	0,83 $\pm$ 0.01	0,61 $\pm$ 0.00	1,36
5	12	13495	805	0,19 $\pm$ 0.00	0,18 $\pm$ 0.00	1,05
	13	2834	155	0,13 $\pm$ 0.00	0,12 $\pm$ 0.00	1,03
	14	304	19	0,12 $\pm$ 0.00	0,12 $\pm$ 0.00	1,01
					Geo. Mean	1.22
	13	870069	25554	252,67 $\pm$ 0.75	70,97 $\pm$ 0.22	3,56
	14	304361	8170	32,13 $\pm$ 0.39	12,24 $\pm$ 0.07	2,63
6	15	72792	1876	5,80 $\pm$ 0.04	5,04 $\pm$ 0.05	1,15
	16	10648	281	4,56 $\pm$ 0.05	4,57 $\pm$ 0.04	1,00
	17	832	26	4,52 $\pm$ 0.04	4,56 $\pm$ 0.05	1,00
					Geo. Mean	1.60
	16	6439087	82614	6853.59 $\pm$ 10.77	1784.97 $\pm$ 2.96	3.84
	17	1749225	21198	669.61 $\pm$ 0.53	287.60 $\pm$ 1.13	2.33
7	18	324272	3906	186.45 $\pm$ 0.55	174.31 $\pm$ 0.95	1.07
	19	36512	469	170.31 $\pm$ 1.34	169.85 $\pm$ 0.36	1.00
	20	2176	34	171.71 $\pm$ 1.55	169.54 $\pm$ 0.40	1.01
					Geo. Mean	1.58
					Full Geo. Mean	1.45

If Eq. (2.3) is naively applied during Step 2, duplicate candidates might be generated. Therefore MA employs a radix-tree with a branching factor of M to merge steps 2 and 3 in A1. During the generation of the candidates, MA performs a synchronized walk of the radix-tree containing all candidates already generated, and only generates values for each element of the under-construction candidate vector which have not yet been explored. Therefore, in $O(log_M n_{can})$ steps, it can determine if a newly generated candidate was already known. Computationally, this is more efficient than generating all the candidates and then removing the duplicates, not due to lower algorithmic complexity, but because of the savings in memory usage for the temporary storage of duplicates, and thus a more efficient use of the processor caches. The use of the radix-tree, although beneficial for the reduction of total execution time, comes at a cost. Some of the operations of the merged procedure are not directly suitable for vectorization. In fact, in any code in which control flow is present (which MA needs to perform the walk on the radix-tree) the application of regular vectorization techniques can be severily compromised [33]. Therefore, our code is a carefully chosen mix of regular and vector instructions intended to minimize execution time.

The effect of this approach can be seen in the number of candidates, n_{can}, and execution times. For instance, in the network given in Fig. 2, considering $d = 4$, n_{can} is lower than the respective value when $d = 5$ (76530 *vs.* 180266). The results provided in tables 1 and 2 clearly show the practical

efficiency of VMA compared to the other ones. One notes that as the number of candidates or *d-MP*s increases, so does VMA's performance due to more efficient use of vectorization.

The performance bottleneck of the current implementation is in the fourth step of Algorithm 1, *i.e.*, the removal of the non-minimal candidates. More than 95% and 75% of the execution times for MA and VMA are spent in this step, respectively. Currently, the complexity of this step is quadratic. However, one may improve it by using multidimensional divide-and-conquer approaches such as the one proposed by Bentley [3] which we aim to investigate in our future works.

4.2 Comparing the two best versions on random networks

In addition to tables 1 and 2, to have a more intuitive comparison between the two best versions of Algorithm 1, we solve two hundred randomly generated test problems by these two versions. To not have very complex test problems, we consider $n = 6, 7$, and 8 as the number of nodes in the randomly generated SFNs. For the number of arcs in each network, a random integer number between or equal to $u = \lceil \frac{(n-1)(n+2)}{4} \rceil$ and $l = u - 4$ is considered. We note that u is the average of the minimum and maximum possible numbers of arcs in a connected graph, i.e., $n-1$ and $\frac{n(n-1)}{2}$, respectively. The capacity of each arc in the randomly generated test problems is also set to $M_i = 3$ as for the benchmarks.

This way, we consider the running times of the two best versions on these two hundred random test problems for producing the performance profile introduced by Dolan and Moré [6], in which the proportion of the executing times of the desired algorithms versus the best ones are considered. Assuming $t_{i,\mathrm{MA}}$ and $t_{i,\mathrm{VMA}}$, respectively, as the running times of the versions MA and VMA, for $i = 1, 2, \cdots, 200$, the performance ratios are $r_{i,s} = \frac{t_{i,s}}{\min\{t_{i,s}:\ s = \mathrm{MA},\ \mathrm{VMA}\}}$, for $s = \mathrm{MA},\ \mathrm{VMA}$ [6]. For each algorithm, the performance is calculated by $Pr_s(T) = \frac{N_s}{200}$, where N_s is the number of SFNs for which $r_{i,s} \leq T$, $i = 1, 2, \cdots, 200$.

Fig. 5 shows the final result of the Dolan and Moré performance profile for the desired versions of Algorithm 1. In this figure, at any τ on the horizontal axis, the difference between the diagrams shows the percentage of the test problems solved τ times faster by the algorithm whose diagram lies above the other one. Hence, the figure expresses that VMA solves 95% of the test problems faster than MA (see the difference between the diagrams at $\tau = 1$). Or observing at $\tau = 2$ in the figure, one can see that VMA solves around 25% of the test problems at least two times faster than MA. It also shows that VMA solves some test problems more than four times more quickly than MA. In this profile, the algorithm whose performance diagram lies above the other is preferred [6].

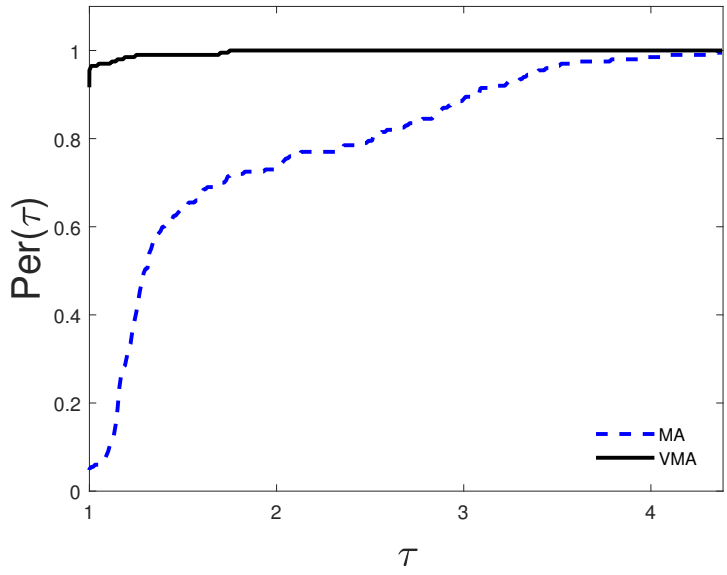

Figure 5: CPU time performance profiles for versions MA and VMA.

All the provided numerical results, including Fig. 5 and Tables 1 and 2 show the superiority of VMA to MA, and that as the network's size grows, this superiority increases.

5 CONCLUDING REMARKS

The *d-MP* problem has been very attractive in recent decades as the reliability of stochastic-flow networks can be computed indirectly through *d-MP*s. This work proposed a vectorization algorithm based on the literature's available regular (non-vectorization) algorithms. We first improved the first version of the algorithm so that no duplicate candidate is produced (see Step 3 in Algorithm 1) and then enhanced its practical efficiency using vectorization techniques. Three known benchmarks were employed to generate the numerical results in which our proposed vectorization algorithm outperformed the others considerably. Moreover, we compared the two best versions on the two hundred randomly generated test problems in the sense of Dolan and Moré's performance profile to have a more intuitive comparison. This performance showed that the vectorization algorithm solves some cases at least four times faster than the other versions. We plan to improve candidates' checking process, use longer vector registers, and use parallelization techniques for future works.

Acknowledgments

The authors thank CNPq (grant 306940/2020-5) for supporting this work.

REFERENCES

[1] G. Bai, Z. Tian & M.J. Zuo. Reliability evaluation of multistate networks: An improved algorithm using state-space decomposition and experimental comparison. *IISE Transactions*, **50**(5) (2018), 407–418.

[2] A.O. Balan & L. Traldi. Preprocessing minpaths for sum of disjoint products. *IEEE Transactions on Reliability*, **52**(3) (2003), 289–295.

[3] J.L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, **23**(4) (1980), 214–229.

[4] X. Chen, J. Tao, G. Bai & Y. Zhang. Search for d-MPs without duplications in multistate two-terminal networks. In "2017 Second International Conference on Reliability Systems Engineering (ICRSE)". IEEE (2017), p. 1–7.

[5] E. Datta & N.K. Goyal. Evaluation of stochastic flow networks susceptible to demand requirements between multiple sources and multiple destinations. *International Journal of System Assurance Engineering and Management*, **10**(5) (2019), 1302–1327.

[6] E.D. Dolan & J.J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, **91**(2) (2002), 201–213.

[7] M. Forghani-elahabad. 1 Exact reliability evaluation of multistate flow networks. In "Systems Reliability Engineering". De Gruyter (2021), p. 1–24.

[8] M. Forghani-Elahabad. 3 The Disjoint Minimal Paths Reliability Problem. In "Operations Research". CRC Press (2022), p. 35–66.

[9] M. Forghani-elahabad & L.H. Bonani. Finding all the lower boundary points in a multistate two-terminal network. *IEEE Transactions on Reliability*, **66**(3) (2017), 677–688.

[10] M. Forghani-elahabad & N. Kagan. An approximate approach for reliability evaluation of a multistate flow network in terms of minimal cuts. *Journal of Computational Science*, **33** (2019), 61–67.

[11] M. Forghani-elahabad & N. Kagan. Reliability evaluation of a stochastic-flow network in terms of minimal paths with budget constraint. *IISE Transactions*, **51**(5) (2019), 547–558.

[12] M. Forghani-elahabad, N. Kagan & N. Mahdavi-Amiri. An MP-based approximation algorithm on reliability evaluation of multistate flow networks. *Reliability Engineering & System Safety*, **191** (2019), 106566.

[13] M. Forghani-elahabad & N. Mahdavi-Amiri. On Search for all d-MCs in a Network Flow. *Iranian Journal of Operations Research*, **4**(2) (2013), 108–126.

[14] M. Forghani-elahabad & N. Mahdavi-Amiri. A new efficient approach to search for all multi-state minimal cuts. *IEEE Transactions on Reliability*, **63**(1) (2014), 154–166.

[15] M. Forghani-elahabad & N. Mahdavi-Amiri. An efficient algorithm for the multi-state two separate minimal paths reliability problem with budget constraint. *Reliability Engineering & System Safety*, **142** (2015), 472–481.

[16] M. Forghani-elahabad & N. Mahdavi-Amiri. An improved algorithm for finding all upper boundary points in a stochastic-flow network. *Applied Mathematical Modelling*, **40**(4) (2016), 3221–3229.

[17] M. Forghani-Elahabad, N. Mahdavi-Amiri & N. Kagan. On multi-state two separate minimal paths reliability problem with time and budget constraints. *International Journal of Operational Research*, **37**(4) (2020), 479–490.

[18] M. Forghani-elahabad & W.C. Yeh. An improved algorithm for reliability evaluation of flow networks. *Reliability Engineering & System Safety*, **221** (2022), 108371.

[19] Z. Hao, W.C. Yeh, Z. Liu & M. Forghani-elahabad. General multi-state rework network and reliability algorithm. *Reliability Engineering & System Safety*, **203** (2020), 107048.

[20] Z. Hao, W.C. Yeh, M. Zuo & J. Wang. Multi-distribution multi-commodity multistate flow network model and its reliability evaluation algorithm. *Reliability Engineering & System Safety*, **193** (2020), 106668.

[21] J.L. Hennessy & D.A. Patterson. "Computer architecture: a quantitative approach". Elsevier (2011).

[22] D.H. Huang, C.F. Huang & Y.K. Lin. Reliability Evaluation for a Stochastic Flow Network Based on Upper and Lower Boundary Vectors. *Mathematics*, **7**(11) (2019), 1115.

[23] Y. Lamalem, K. Housni & S. Mbarki. An Efficient Method to Find All d-MPs in Multistate Two-Terminal Networks. *IEEE Access*, **8** (2020), 205618–205624.

[24] S. Larsen & S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In "Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation", PLDI '00. Association for Computing Machinery, New York, NY, USA (2000), p. 145–156. doi:10.1145/349299.349320. URL `https://doi.org/10.1145/349299.349320`.

[25] J.S. Lin, C.C. Jane & J. Yuan. On reliability evaluation of a capacitated-flow network in terms of minimal pathsets. *Networks*, **25**(3) (1995), 131–138.

[26] Y.K. Lin & S.G. Chen. A maximal flow method to search for d-MPs in stochastic-flow networks. *Journal of computational science*, **22** (2017), 119–125.

[27] S.M. Mansourzadeh, S.H. Nasseri, M. Forghani-elahabad & A. Ebrahimnejad. A Comparative Study of Different Approaches for Finding the Upper Boundary Points in Stochastic-Flow Networks. *International Journal of Enterprise Information Systems (IJEIS)*, **10**(3) (2014), 13–23.

[28] Y.F. Niu, Z.Y. Gao & W.H. Lam. Evaluating the reliability of a stochastic distribution network in terms of minimal cuts. *Transportation Research Part E: Logistics and Transportation Review*, **100** (2017), 75–97.

[29] Y.F. Niu, Z.Y. Gao & W.H. Lam. A new efficient algorithm for finding all d-minimal cuts in multi-state networks. *Reliability Engineering & System Safety*, **166** (2017), 151–163.

[30] Y.F. Niu, X.Y. Wan, X.Z. Xu & D. Ding. Finding all multi-state minimal paths of a multi-state flow network via feasible circulations. *Reliability Engineering & System Safety*, **204** (2020), 107188.

[31] Y.F. Niu, X.Z. Xu, C. He, D. Ding & Z.Z. Liu. Capacity Reliability Calculation and Sensitivity Analysis for a Stochastic Transport Network. *IEEE Access*, **8** (2020), 133161–133169.

[32] T. Rauber & G. Rünger. "Parallel programming: for multicore and cluster systems". Springer-Verlag (2010).

[33] J. Shin. Introducing Control Flow into Vectorized Code. In "16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)" (2007), p. 280–291. doi:10.1109/PACT.2007.4336219.

[34] X.Z. Xu, Y.F. Niu & Y.F. Song. Computing the reliability of a stochastic distribution network subject to budget constraint. *Reliability Engineering & System Safety*, **216** (2021), 107947.

[35] W.C. Yeh. Search for MC in modified networks. *Computers & Operations Research*, **28** (2001), 177–184.

[36] W.C. Yeh. A simple algorithm for evaluating the k-out-of-n network reliability. *Reliability Engineering & System Safety*, **83** (2004), 93–101.

[37] W.C. Yeh. Fast Algorithm for Searching *d*-MPs for all Possible *d*. *IEEE Transactions on Reliability*, **67**(1) (2018), 308–315.

[38] W.C. Yeh, Z. Hao, M. Forghani-elahabad, G.G. Wang & Y.L. Lin. Novel binary-addition tree algorithm for reliability evaluation of acyclic multistate information networks. *Reliability Engineering & System Safety*, **210** (2021), 107427.

[39] W.C. Yeh & M.J. Zuo. A new subtraction-based algorithm for the d-MPs for all d problem. *IEEE Transactions on Reliability*, **68**(3) (2019), 999–1008.