XLIX International Symposium on Operational Research
SYM-OP-IS 2022
Vrnjačka Banja, 19 - 22.09.2022.
UNIVERSITY OF BELGRADE
Faculty of Economics and Business

NEW IDEAS TO SPEED-UP FLOYD-WARSHALL SHORTEST PATHS ALGORITHM

GIUSEPPE LANCIA[1], FRANCA RINALDI[2]

[1] University of Udine – D.M.I.F, giuseppe.lancia@uniud.it
[2] University of Udine – D.M.I.F., franca.rinaldi@uniud.it

Abstract: *Floyd and Warshall's algorithm for the all-pairs shortest path problem is a $\Theta(n^3)$ procedure which revisits n times all the cells of an $n \times n$ distance matrix. At each pass, all the cells are checked but only some of them get updated. In this paper, we report some preliminary results on a new version of the algorithm, designed to avoid checking cells which will not be updated, in order to reduce the overall time. Our procedure uses heaps to quickly identify which cells can be good candidates for an update. The new version improves over Floyd-Warshall's original for those input graphs in which the number of cells updated over all passes is substantially smaller than the number of checks. However, our procedure is worse than the original if the ratio between cell checks and updates is not large enough. To obtain an improvement independently of the particular instance type, we propose a hybrid combination of the two approaches, which starts with the original Floyd and Warshall version and then switches to the new one after some iterations. Preliminary experiments show the effectiveness of this strategy.*

Keywords: *Shortest path algorithm, Floyd Warshall, All-pairs shortest paths, Smart force*

1. INTRODUCTION

We are given in input a directed graph $G = (V,A)$ with arc costs $c(i, j)$. Wlog, $V = \{1,2,...,n\}$. The costs do not need to be nonnegative, but for the results to be correct there must be no negative-length cycle in G.

The goal is to compute the shortest path length between each pair of vertices i, j. We will denote the length of a shortest $i \to j$ path by $L[i, j]$.

This is a classical problem in combinatorial optimization, which is solved via a classical algorithm, named Floyd-Warshall (FW) after the authors who inspired it [3, 7]. The algorithm updates a matrix $L[i, j]$ initialized as $L[i, j] := c(i, j)$ for $(i, j) \in A$, while $L[i, j] := +\infty$ for $(i, j) \notin A$ and $L[i,i] := 0$ for all i. It then iterates three nested for-cycles:

```
1. for $k := 1,\dots,n$
2.     for $i := 1,\dots,n$
3.         for $j := 1,\dots,n$
4.             if $(L[i,j] > L[i,k]+L[k,j])$ then
5.                 $L[i,j] := L[i,k]+L[k,j]$;
```

At the end $L[i, j]$ is the shortest length of an $i \to j$ path, for each $i, j \in V$. The complexity of the algorithm is $\Theta(n^3)$. This is a worst-case complexity, but also the best-case and average-case are the same.

Major modifications of the basic Floyd-Warshall scheme, to reduce either its worst- or average-case computational complexity, have been proposed, for instance, by Brodnik et al. in [1] for complete graphs and by Sao et al. in [6] for sparse graphs. In this extended abstract, we introduce a relatively simple idea with a straightforward implementation, to speed-up the basic FW procedure. As discussed in Section 5, the idea yields quite promising results.

2. MAIN IDEA

Let us call line 4 of Floyd-Warshall algorithm an execution of `test`$(i, j|k)$ and line 5 an execution of `update`$(i, j|k)$ (the update of cell $[i, j]$ due to k).

The Floyd-Warshall algorithm can be seen as an enumeration of all triples (i, j, k) to execute the tests `test`$(i, j|k)$, some of which will cause an update and some will not. We could say that a test (and the triple) *succeeds* if it causes an update, while it *fails* otherwise. It is clear that when a triple fails the time spent to consider it was wasted, but we don't know in advance which triples will succeed, or we would enumerate only them. If $\mathcal{S}$ is the set of succeeding triples and $\mathcal{F}$ is the set of failing triples, it is expected (and observed in our experiments) that $|\mathcal{S}|$ is much smaller than $\binom{n}{3}$. In particular, after some iterations of the algorithm (as k approaches n), the number of new succeeding triples drops significantly, as the matrix $L[\cdot,\cdot]$ is almost entirely filled with the final correct values.

FWalgorithm is very lean and it appears very difficult to speed it up while maintaining its general scheme. Indeed, there are $\binom{n}{3}$ triples overall, and the time it takes Floyd-Warshall to create and test one triple is $O(1)$. This constant, call it a, is very small since there is a minimal cost for bookkeeping. Let us call b the constant time to execute $\texttt{update}(i,j|k)$. Then, the total running time is $a(|\mathcal{S}|+|\mathcal{F}|)+b|\mathcal{S}|$.

Ideally, in the best case, to beat it we would only enumerate the triples in $\mathcal{S}$. Suppose finding them costs us, on average, $f(n)$ each. The total cost would be $f(n)|\mathcal{S}|$ and so the new algorithm would be competitive as long as $f(n)|\mathcal{S}| < a(|\mathcal{S}|+|\mathcal{F}|)+b|\mathcal{S}|$. However, finding precisely only the successful triples seems quite hard to do. In this paper we propose a heuristic strategy, that exploits a necessary condition that all successful triples must satisfy, in order to lower the probability that a triple elected for the test will in fact fail. In particular, while not all triples will be enumerated, still our scheme will not miss any succeeding one.

3. THE SMART-FORCE ALGORITHM

The general question of Floyd-Warshall is:

QFW: *Having fixed k, which are the pairs (i,j) that complete a succeeding triple (i,j,k)?*

Notice that there are $\Theta(n)$ choices for k, while the answer takes $\Theta(n^2)$ time.
We will rephrase QFW into two questions:

Q1: *Having fixed (k,i), which are the indices j that complete a succeeding triple (i,j,k)?*
Q2: *Having fixed (k,j), which are the indices i that complete a succeeding triple (i,j,k)?*

There are $\Theta(n^2)$ choices for (k,i) (or (k,j)), but we aim at finding the answer in less than $\Theta(n)$ time, thus realizing a speed-up with respect to Floyd-Warshall.

An entry (i,j) gets updated if $L[i,k]+L[k,j] < L[i,j]$. Therefore, a necessary condition for a triple (i,j,k) to be succeeding is that

$$\left(L[i,k] < \frac{L[i,j]}{2}\right) \vee \left(L[k,j] < \frac{L[i,j]}{2}\right)$$

which we can rewrite as $\Big(L[i,j] > 2L[i,k]\Big) \vee \Big(L[i,j] > 2L[k,j]\Big)$.

When (k,i) are fixed we will check the first part of the condition, over all j, and when (k,j) are fixed, we will check the second part over all i. This way, all succeeding triples would be found. The above checks are not done by complete enumeration (brute-force) but rather by direct access to only the indices that can fulfill the condition (a techinque we called "smart-force" (SF) in our previous papers [4, 5]).

Let us consider the case of (k,i) being fixed, so that $2L[i,k] =: \alpha$ is a constant. The condition for some j to be part of a succeeding triple with i and k is that $L[i,j] > \alpha$. This implies, for instance, that if we knew that $\max_j L[i,j] \leq \alpha$, then we could avoid enumerating the n possible values for j. We therefore want to sample only values $j_1, j_2, \ldots, j_t$ such that $L[i,j_h] > \alpha$ for $h = 1,\ldots,t$. We achieve this by keeping a (row-)heap $\mathtt{H}_i^R$ for each i. The heap contains all elements j of row i, sorted by their $L[i,j]$ value. This way popping the maximum index takes time $O(1)$, while readjusting the heap after each pop takes $O(\log n)$. The total number of triples thus sampled is t rather than n at a cost of $O(t\log n)$ which is no worse than $\Theta(n)$ as long as t is $O(n/\log n)$.

In a perfectly analogous way, we work for the case in which (k,j) are fixed, by keeping n (column-)heaps $\mathtt{H}_j^C$, one for each j, and looking for indices i such that $L[i,j] > \alpha := 2L[k,j]$.

Building the heaps at the very beginning has cost $O(n)$ per heap, for a total cost $O(n^2)$. Each row-heap (and, respectively, column-heap) is implemented in the customary way as an array $\mathtt{H}_i^R[\cdot]$ of n records $\langle \mathtt{ndx}, \mathtt{value}\rangle$, where ndx represents a column- (respectively, row-) index, while value is the value of an entry of $L[i,\cdot]$ (respectively, $L[\cdot,j]$). E.g., if $\mathtt{H}_3^R[1] = \langle 5, 100\rangle$ it means that the highest value in row 3 is at column 5, and it is $L[3,5] = 100$. The second- and third- highest values of row 3 are in the next two entries of $\mathtt{H}_3^R[\cdot]$ (not necessarily in this order). In general, each index h identifies a node of the heap whose left subtree is rooted at $2h$ while the right subtree is rooted ar $2h+1$.

In the course of the algorithm, we need to keep the heaps consistent with L, i.e., each time an entry $L[i,j]$ is updated, we need to also update $\mathtt{H}_i^R$ and $\mathtt{H}_j^C$. In order to make these updates effectively, we use a back-pointer data

Algorithm 1 SF-FW

```
        /* create row- and column- heaps */
 1. for $i \leftarrow 1$ to $n$ do
 2.     for $j \leftarrow 1$ to $n$ do
 2.         $\mathtt{H}^R_i[j] \leftarrow \langle j, L[i,j]\rangle$; $\mathtt{Pos\_H}^R_i[j] \leftarrow j$
 3.     for $v \leftarrow \lfloor n/2 \rfloor$ downto 1 do heapify($\mathtt{H}^R_i, v$)
 4. for $j \leftarrow 1$ to $n$ do
 5.     for $i \leftarrow 1$ to $n$ do
 5.         $\mathtt{H}^C_j[i] \leftarrow \langle i, L[i,j]\rangle$; $\mathtt{Pos\_H}^C_j[i] \leftarrow i$
 6.     for $v := \lfloor n/2 \rfloor$ downto 1 do heapify($\mathtt{H}^C_j, v$);
 7. for $k \leftarrow 1$ to $n$ do
 8.     for $i \leftarrow 1$ to $n$ do /* run updates due to column k */
 9.         $\alpha \leftarrow 2L[i,k]$
10.         $t \leftarrow 0$
11.         while ( $\mathtt{H}^R_i[1]$.value $> \alpha$ )
12.             popped[$t$] $\leftarrow$ popHeapTop( $\mathtt{H}^R_i$ )
13.             $j$ := popped[$t$].ndx
14.             popped[$t$].value $\leftarrow L[i,j] \leftarrow$ eval($L, i, j, k$)
15.             $t$++
            /* restores heap. Popped entries get new value */
16.         for $e := 1$ to $t$ do insertHeap( $\mathtt{H}^R_i$, popped[$e$] )
17.     for $j \leftarrow 1$ to $n$ do /* run updates due to row k */
18.         $\alpha \leftarrow 2L[k,j]$
19.         $t \leftarrow 0$
20.         while ( $\mathtt{H}^C_j[1]$.value $> \alpha$ )
21.             popped[$t$] $\leftarrow$ popHeapTop( $\mathtt{H}^C_j$ )
22.             $i$ := popped[$t$].ndx
23.             popped[$t$].value $\leftarrow L[i,j] \leftarrow$ eval($L, i, j, k$)
24.             $t$++
            /* restores heap. Popped entries get new value */
25.         for $e := 1$ to $t$ do insertHeap( $\mathtt{H}^C_j$, popped[$e$] )
```

structure into the heap, i.e., we keep "parallel" arrays $\mathtt{Pos\_H}^R_i[\cdot]$ (respectively, $\mathtt{Pos\_H}^C_j[\cdot]$) that tell the position of each column index j (respectively, row index i) in the heap $\mathtt{H}^R_i$ (respectively, $\mathtt{H}^C_j$). In our example, $\mathtt{Pos\_H}^R_3[5] = 1$ since $j = 5$ is the root of heap $\mathtt{H}^R_3$. This way, whenever $L[i,j]$ is changed, we go to the node corresponding to j in $\mathtt{H}^R_i$, and change its value (which necessarily decreases). In doing so, we let the node slide down in the heap, by exchanging it with the largest of its two sons, if necessary, until it finds its correct position. We then record the new position of j in $\mathtt{Pos\_H}^R_i[j]$. The overall cost of this update is $O(\log n)$.

4. PSEUDOCODE

4.1. Procedure SF-FW.

The main procedure SF-FW is outlined in Procedure 1. We assume the input to be an $n \times n$ cost matrix $L[\cdot,\cdot]$, and the final result, i.e., all pairs shortest paths lenghts, will be overwritten on $L[\cdot,\cdot]$.

The procedure starts by creating the row-heaps $\mathtt{H}^R_i$ for each row i, and column-heaps $\mathtt{H}^C_j$ for each column j (lines 1–6). At this time, also the parallel arrays $\mathtt{Pos\_H}^R_i$ and $\mathtt{Pos\_H}^C_j$ are initialized. This is done within the procedure HEAPIFY (Procedure 2), which, while moving the elements in the heaps, keeps track of their positions.

The main loop starts at line 7 and ends at line 25. It is composed of two sub-loops. The first one (lines 8–16) takes care of all the updates to $L[\cdot,\cdot]$ due to the elements on column k, i.e., the pairs (i,k) for varying i. A perfectly analogous loop (lines 17–25) concerns the updates due to row k of $L[\cdot,\cdot]$, i.e., pairs (k,j) for varying j. We will comment only on the first loop since similar considerations hold also for the second.

The loop 8–16 starts by determining the threshold $\alpha = 2L[i,k]$ which must be beat by an index j to be considered in a possibly succeeding triple. In lines 11–15, all elements $X := \langle j(=\mathtt{ndx}), \mathtt{value}\rangle$ in the heap such that $L[i,j] > \alpha$ are popped from the heap and put in a local buffer popped[]. The triple (i,j,k) is then evaluated and the new value of the entry $L[i,j]$ is stored as the new `value` of X. At the end t is the total number of elements that were popped (i.e., of triples evaluated). At this point, the loop in line 16 puts back in the heap $\mathtt{H}_i^R$ the elements that were popped, with their new values. The back-pointers $\mathtt{Pos\_H}_i^R$ are updated within the procedure INSERTHEAP (Procedure 3).

4.10.1 Procedures for heaps management.

A heap H is a binary trees stored in consecutive cells of an array (see [2]). In general, the left son of node $H[i]$ is node $H[2i]$, while the right son is node $H[2i+1]$. The heap property requires that the value of each node is not smaller than the value of any of its sons. Thus, the largest value of any subtree is at the root node of the subtree.

HEAPIFY, depicted in Procedure 2, is a standard procedure that, given an index v and an array of n records such that the subtrees rooted at $2v$ and $2v+1$ are heaps, turns the subtree rooted at v into a heap. This is obtained by letting $H[v]$ "slide" down in the heap until it finds its final position. This version of HEAPIFY is such that the back-pointers to the heap are updated while the elements are moved around. Note that by calling HEAPIFY for $v = \lfloor n/2 \rfloor, \ldots, 1$, an entire array can be turned into a heap.

Procedure 2 HEAPIFY (H, v)

```
1. $ls \leftarrow 2v$  /* left son */
2. $rs \leftarrow 2v+1$  /* right son */
3. if $(H[ls].\mathtt{value} > H[v].\mathtt{value})$ then $large \leftarrow ls$ else $large \leftarrow v$
4. if $(H[rs].\mathtt{value} > H[large].\mathtt{value})$ then $large \leftarrow rs$
5. if $(large \neq v)$ then
6.     $\mathtt{Pos\_}H[H[v].\mathtt{ndx}] \leftarrow large$
7.     $\mathtt{Pos\_}H[H[large].\mathtt{ndx}] \leftarrow v$
8.     $H[v] \leftrightarrow H[large]$  /* swaps H[v] with the largest of its sons */
9.     heapify$(H, large)$
```

The procedure INSERTHEAP (procedure 3) is used to insert a new element `el` into a heap H of less than n elements. The element is initially put at the rightmost bottom leaf (position $H[H.\mathtt{SIZE}]$) and then is lifted up to its final position, i.e., when its value is not larger than its parent's. This is done in the loop 3–6. The position of each element which is moved down in the heap while `el` moves up is updated within the loop. Eventually, the element finds its final position in lines 7–8.

Procedure 3 INSERTHEAP (H,el)

```
1.     H.SIZE++
2.     curr ← H.SIZE
3.     while (curr > 1) AND (H[$\lfloor \frac{\mathtt{curr}}{2} \rfloor$].value < el.value)
4.         Pos_H[ H[$\lfloor \frac{\mathtt{curr}}{2} \rfloor$].ndx] ← curr
5.         H[curr] ← H[$\lfloor \frac{\mathtt{curr}}{2} \rfloor$]
6.         curr ← $\lfloor \frac{\mathtt{curr}}{2} \rfloor$
7.     Pos_H[el.ndx] ← curr
8.     H[curr] ← el
```

The procedure POPHEAPTOP (not reported, for space limitations) extracts from the heap, and returns, the element of maximum value (which is the root node, i.e., $H[1]$). It then rebuilds the heap by replacing the root with the leaf $H[H.\mathtt{SIZE}]$ which, by calling $\mathtt{heapify}(H,1)$, is moved down along the tree until the heap property is again fulfilled. The positions in the heap of the elements moved are updated accordingly, while the position of the element extracted from the heap is set to -1. The cost of this procedure is $O(\log(H.\mathtt{SIZE}))$.

5. DISCUSSION AND COMPUTATIONAL EXPERIMENTS

The cost of producing a triple to test is $O(\log n)$ for the SF_FW procedure, while for FW it is $O(1)$. Furthermore: (i) the hidden multiplicative constant for the complexity of SF_FW is certainly higher than for FW; (ii) for SF_FW there is the possibility that some triples (i,j,k) are evaluated twice (this happens when both $L[i,k] < L[i,j]/2$ and also $L[k,j] < L[i,j]/2$). For the above reasons, the only hope for SF_FW to outperform

FW is that the input graph is such that only *a very small number of triples* (compared to n^3) are updated in the course of FW procedure, and not many more than these are evaluated by SF_FW.

Graphs with few updates. For an extreme example, consider a complete graph in which the edge costs are random numbers in a range $\{m,\dots,M\}$ with $m \geq M/2$. For such a graph, each edge is in fact a shortest path and hence no triple would lead to an update. As far as SF_FW is concerned, for any target $\alpha = 2L[k,j]$ (or $\alpha = 2L[i,k]$) no entry of $L[,]$ would ever be popped from the heaps, with the exception of the phases in which $k = j$ or $k = i$ (i.e., the phases in which the target is given by one of the n diagonal elements). In these phases, $\alpha = 0$ and $n-1$ heap elements get popped). Therefore, overall, only $2n(n-1)$ triples get checked for SF_FW while n^3 triples get checked for FW. Notice that the time to recognize that each edge is indeed a shortest path cannot be smaller than $\Theta(n^2)$ and so SF_FWis optimal for these inputs.

If we perturb this type of input by altering a few entries, still SF_FW outperforms Floyd-Warshall. For an example, consider graphs of the above type in which we pick at random n arcs and give them a new random value in $[1,m]$, so that some triples will probably lead to an update (i.e., they are successful). Then we obtain the following average results (for $m = 200$, $M = 399$, over 5 instances for each n, rounded to integer)

n	successful triples	FW attempted triples	FW time	SF_FW attempted triples	SF_FW time
1000	322,487	1,000,000,000	5.6s	3,708,892	0.8s
2000	1,313,753	8,000,000,000	42.4s	15,009,216	3.0s
3000	3,011,423	27,000,000,000	135.4s	34,221,686	7.2s

To characterize the best inputs for SF_FW, it would be important to identify classes of graphs on which FW would run into a large ratio of attempted vs successful triples. One such type of graph could be a graph with some “fast” nodes (also called *hubs*) intermixed with general nodes. In such a graph each edge incident on a hub would be ”short”, while all the other edges can be of any length (like, in real life, large cities connected by highways are quicker to reach than small remote villages) and a shortest path most likely uses only hubs as intermediate nodes. Assuming the nodes are sorted according to their average distance from the rest, the hubs are processed first, and so for the remaining iterations probably very few entries of $L[,]$ will be updated.

We simulate a graph of this type by the following parameters: n_h is the number of hubs; m is the minimum cost of any arc; M_h is the maximum cost for an arc incident with a hub; M is the maximum cost for any other arc. We have performed tests on this type of graphs, with $n = 2000$ nodes and either 10, 20 or 40 hubs. We have set $m = 100$ and $M = 10000$ and have varied the length of the arcs incident on the hubs in $\{120, 150, 200\}$. The results (times in seconds) show a speed improvement from 3 times to 5 times of SF_FW versus the original Floyd-Warshall procedure:

	40 hubs			**20 hubs**			**10 hubs**		
M_h	FW	SF_FW	Speed-up	FW	SF_FW	Speed-up	FW	SF_FW	Speed-up
200	41.9s	16.1s	2.6×	42.0s	14.1s	3.0×	41.9s	13.4s	3.1×
150	41.8s	14.0s	3.0×	42.0s	12.0s	3.5×	41.5s	10.1s	4.1×
120	41.3s	12.6s	3.3×	41.6s	10.0s	4.2×	41.2s	8.1s	5.1×

More general graphs and the hybrid approach. The overhead due to the heap management compared to the minimal bookkeeping of FW is such that when the arc costs vary in a large interval (and therefore also the lengths of the paths can be very different among them), the original FW easily outperforms SF_FW. However, as soon as this variability in the arc costs starts to decrease, SF_FW becomes competitive and eventually can be better than Floyd-Warshall. Consider the following experiment run on complete graphs of size $n = 1000$ whose arc costs are drawn randomly in $\{1,\dots,\text{MAX}\}$ for various decreasing values of MAX:

MAX	successful triples	FW attempted triples	FW time	SF_FW attempted triples	SF_FW time
100	7,843,881	1,000,000,000	5.6s	61,760,461	10.4s
50	5,889,226	1,000,000,000	5.6s	55,583,872	9.3s
10	2,248,129	1,000,000,000	5.6s	20,532,006	3.4s
5	993,010	1,000,000,000	5.5s	8,289,043	1.6s

The table shows how SF_FW benefits from the arc costs varying in a small range (”small” should be intended in a relative way, as shown by our next experiment. That, is, if we keep MAX fixed, SF_FW benefits from the increase of n). For $n = 1000$, we see that SF_FW beats Floyd-Warshall for MAX$\leq$ 10, but this range is indeed too small to be really interesting.

In order to obtain a procedure which is effective also when the arc costs vary in a reasonably large range, we made the following observation. During the execution of FW algorithm, the entries get updated easily in the early phases, but then, once the values get closer to the true shortest path lengths, it becomes more and more difficult to beat them and so the number of updates drops dramatically. Therefore, it would make sense to use the original FW for some iterations, in order to improve the current matrix, and then continue on this matrix with the SF_FW version for the remaining iterations.

We then performed an experiment, for various values of n, on complete graphs with independent random arc costs drawn in the range $\{1,\ldots,100\}$. For each instance, we have tested what would happen if we start with S iterations of FW followed by $n-S$ iterations of SF_FW, with $S=\frac{h}{10}n$ and $h=0,1,\ldots,10$. Notice that when $h=0$ the hybrid approach degenerates into SF_FW, while for $h=10$ it is FW.

switch point	$n=1000$	$n=2000$	$n=3000$	$n=4000$	$n=5000$
n (FW)	5.6s	42.0s	130.1s	304.3s	607.4s
0.9 n	5.1s	39.6s	131.6s	290.3s	562.0s
0.8 n	5.4s	37.8s	128.9s	290.7s	534.0s
0.7 n	5.1s	35.9s	123.1s	276.3s	512.8s
0.6 n	4.8s	32.9s	118.2s	257.7s	489.1s
0.5 n	**4.5s**	31.0s	111.3s	245.5s	465.1s
0.4 n	4.6s	30.8s	104.9s	232.8s	438.3s
0.3 n	4.7s	**30.5s**	99.2s	220.1s	418.4s
0.2 n	5.5s	32.1s	**95.9s**	**204.2s**	392.7s
0.1 n	6.8s	36.9s	105.5s	208.4s	**395.5s**
0 (SF_FW)	11.0s	61.2s	167.5s	322.0s	585.6s

From the table, we can see that there is always some combination of FW+SF_FWwhich beats FW (indeed, with the exception of just one entry for $n=1000$, *all* combinations we tested are better than FW alone). We have highlighted the best combination for each n, and it looks that, when n increases, the number of iterations of FW needed to get the best speed-up of the search decreases. A heuristic rule that yields good results for $1000 \le n \le 5000$ appears to be "*run 500 iterations of* FW *and then switch to* SF_FW". Another interesting observation from the table is that, for increasing n, pure SF_FW becomes more and more competitive with pure FW, and eventually it beats it for $n=5000$.

6. CONCLUSIONS

We have suggested a new "smart-force" strategy to sample triples in Floyd-Warshall algorithm, in order to increase their probability of success. Preliminary results show the strategy to be very effective on some particular graphs, and a research question would be to characterize other classes of graphs on which the approach should perform well. Furthermore, we proposed a hybrid approach which is effective on more general graphs. An important question would be to characterize the best switching point from the old to the new strategy. This could be done by extensive testing and tuning, with possibly the use of machine-learning techniques. Other lines of future research concern finding permutations of the nodes that might help the running time, and incorporating simple changes to the strategy that can avoid the possibility of enumerating the same triple twice. Finally, we want to investigate if this type of idea can be applied to other algorithms, for this or other optimization problems.

REFERENCES

[1] Brodnik A., Grgurovič M. and Požar R. (2022). Modifications of the Floyd-Warshall algorithm with nearly quadratic expected-time. Ars Mathematica Contemporanea 22, 1–22.

[2] Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed.; The MIT Press: Cambridge, MA, USA, 2009.

[3] Floyd, R. W. (1962). Algorithm 97: Shortest Path. Communications of the ACM, 5(6), 345.

[4] Lancia, G., Dalpasso, M., (2020). Finding the Best 3-OPT Move in Subcubic Time. Algorithms, 13(11), 306–321.

[5] Lancia, G., Vidoni, P., (2020). Finding the largest triangle in a graph in expected quadratic time. European Journal of Operational Research, 286, 458–467.

[6] Sao P., Kannan R., Gera P., Vuduc R. (2020) A Supernodal All-Pairs Shortest Path Algorithm in Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[7] Warshall, S. (1962). A theorem on Boolean matrices. Journal of the ACM, 9(1), 11–12.