 DOI: https://doi.org/10.26493/scores26.17

Two-Step Relaxation for All-Pairs Shortest Paths in Sparse Graphs

Dani Zugan
dani.zugan@fri.uni-lj.si
University of Ljubljana,
Faculty of Computer and Information Science
Večna pot 113
SI-1000 Ljubljana, Slovenia

ABSTRACT

The all-pairs shortest path problem is a fundamental problem in graph algorithms, with applications in telecommunications, logistics, and network analysis. In this paper, we study this problem in directed graphs and introduce a new relaxation based approach. The method decomposes the graph into strongly connected components, computes shortest paths within each component, and then determines shortest paths between different components. Unlike earlier component based methods that propagate distances directly through outgoing edges, the proposed approach uses a two-step initial-terminal relaxation scheme. Here, an initial vertex is the starting vertex of an outgoing edge of a component, and a terminal vertex is its ending vertex outside that component. In the first step, distances are propagated through these vertices; in the second step, the resulting values are extended through subsequent components. We prove that this decomposition correctly computes inter-component shortest path distances once shortest paths within the components are known. We also derive the running-time bound of the algorithm. In the worst case, it matches the cubic complexity of the classical Floyd–Warshall algorithm, while for graphs with small sets of initial and terminal vertices it can substantially reduce the number of required relaxations.

KEYWORDS

all-pairs shortest path, strongly connected components, Floyd-Warshall algorithm.

1 INTRODUCTION

Many real-world problems in bioinformatics, logistics, telecommunications, and related domains are modeled as graphs, where the main task is to find shortest paths between pairs of nodes [1]. The shortest path problem appears in two variants: finding shortest paths from a single source to all other nodes (single-source shortest path, SSSP), and finding shortest paths for all node pairs (all-pairs shortest path, APSP).

Two conceptually different strategies exist for APSP. The first reduces APSP to repeated SSSP computations—running an SSSP algorithm from every node and aggregating the results. The second constructs shortest paths for all pairs simultaneously. For graphs with non-negative edge weights, the repeated SSSP strategy uses Dijkstra-type algorithms. A classical representative of the simultaneous strategy is the Floyd–Warshall algorithm [7, 18], based on dynamic programming.

Despite their different viewpoints, both strategies rely on the same operation: *relaxation*. Since the overall running time depends on the number of relaxation attempts, repeated SSSP approaches tend to be preferable on sparse graphs, whereas Floyd–Warshall-like methods are well-suited for dense graphs.

The component-based framework used in this paper originates from [19]. In that work, the graph is first decomposed into strongly connected components (SCCs), and APSP is solved independently within each SCC, which is possible because all vertex pairs inside an SCC are mutually reachable.

The remaining task is to compute shortest paths between vertices that belong to different components. In [19], this is done by propagating distances across components through outgoing edges, using a reachability observation: if an arbitrarily chosen vertex in a source component cannot reach an arbitrarily chosen vertex in a destination component, then relaxations between these two components can be omitted. Our work adopts the same SCC decomposition and the same reachability viewpoint, but replaces the edge-based inter-component propagation of [19] with a computation based on classical Floyd–Warshall relaxations.

The goal of this paper is to introduce an SCC-guided approach to the APSP problem, where relaxations are performed only over relevant vertex triples.

2 RELATED WORK

The APSP problem has been studied extensively in the graph algorithms literature. The classical dynamic-programming solution is the Floyd–Warshall algorithm [7, 18], which runs in $O(n^3)$ time. A different approach is to solve APSP through repeated single-source shortest path computations. For graphs with non-negative edge weights, this is typically done with Dijkstra's algorithm [6], while Johnson's algorithm [10] extends this approach to graphs without negative cycles.

Several improvements of Floyd–Warshall aim to reduce the number of unsuccessful relaxations. In [3], the authors introduce the Tree and Hourglass data structures and obtain an expected running time of $O(n^2 \log^2 n)$ for complete directed graphs with edge weights drawn uniformly from $[0, 1]$. Similarly, [12] proposes the Smart-Force technique and a hybrid variant that processes only selected relaxations likely to improve distances.

Other work exploits sparsity or structural restrictions. The Rectangular Algorithm [2] skips inactive regions of the distance matrix, while [17] improves Floyd–Warshall on disconnected and very sparse graphs by maintaining reachability information. For sparse directed graphs, Pettie [13] improves the worst-case bound over Johnson's algorithm. A particularly relevant direction for our setting is APSP on graphs of low treewidth. For a vertex ordering d, Planken, de Weerdt, and van der Krogt [14] propose DPC-based APSP algorithms running in $O(n^2 w_d)$ time, where w_d denotes the

graph width induced by d. This results in $O(n^2)$ time for graphs of constant treewidth and $O(nm)$ time for chordal graphs.

A closely related line of work is based on strongly connected components (SCCs). SCC decomposition can be computed in linear time by Tarjan's algorithm [15], and its structural depth-first-search foundations are revisited in [16]. Building on this idea, [19] first solves shortest paths inside each component and then propagates distances between components by relaxing via outgoing edges from components.

For sparse graphs, repeated single-source methods remain a strong baseline. Their performance depends heavily on the priority queue used inside Dijkstra's algorithm. Fibonacci heaps give the classical theoretical bound [9], while pairing heaps are often competitive in practice [8].

Finally, for complete directed graphs with independent edge weights drawn uniformly from $[0, 1]$, sharper expected bounds are known. The Hidden Paths algorithm [11] achieves expected running time $O(n^2 \log n)$, and this bound was later matched by Demetrescu and Italiano, who also provided an experimental analysis of APSP algorithms [4, 5].

3 PRELIMINARIES

A *directed graph* is denoted by $G = (V, E)$, where V is a finite, nonempty set of vertices and $E \subseteq V \times V$ is a set of directed edges. We write $V = \{v_1, \ldots, v_n\}$.

An *edge* $e = (u, v) \in E$ is directed *from* u *to* v. The vertex u is called the *initial vertex* of e, denoted by $\mathrm{ini}(e) = u$, while v is called the *terminal vertex* of e, denoted by $\mathrm{ter}(e) = v$.

A *path* in G is a sequence of pairwise distinct vertices $P = (v_1, \ldots, v_k)$ such that $(v_i, v_{i+1}) \in E$ for all $1 \le i < k$. We say that v is *reachable* from u if there exists a path starting at u and ending at v.

If $P = (v_1, \ldots, v_k)$ is a path and $1 \le a \le b \le k$, then $(v_a, \ldots, v_b)$ is called a *subpath* of P. We denote this subpath by $v_a \rightsquigarrow v_b$.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$.

A *strongly connected component (SCC)* of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that every two vertices in C are mutually reachable.

A directed graph can be partitioned into t strongly connected components $C_1, C_2, \ldots, C_t$, ordered topologically so that there are no edges from C_j to C_i for $j > i$. Further, we denote the union $C_i \cup C_{i+1} \cup \cdots \cup C_t$ by $C^{(i\ldots t)}$.

Given an SCC $C_i \subseteq V$, the set of *outgoing edges* from C_i is defined as $\mathrm{Out}(C_i) = \{(u, v) \in E \mid u \in C_i,\ v \notin C_i\}$ and the set of *incoming edges* to C_i is defined as $In(C_i) = \{(u, v) \in E \mid u \notin C_i,\ v \in C_i\}$.

For an SCC C_i, we denote by $I_i = \{ini(e) \mid e \in \mathrm{Out}(C_i)\}$ the set of *initial vertices* of outgoing edges, and by $T_i = \{ter(e) \mid e \in \mathrm{Out}(C_i)\}$ the set of *terminal vertices* of outgoing edges.

Note that distinct edges in $\mathrm{Out}(C_i)$ may share the same initial vertex or the same terminal vertex. Thus, multiple outgoing edges may originate at a single vertex in I_i, and multiple outgoing edges may enter the same vertex in T_i.

For vertex $v \in C^{(i+1\ldots t)}$ we define $\mathrm{In}_i(v) = \{e \in \mathrm{Out}(C_i) \mid \mathrm{ter}(e) = v\}$.

A *weighted graph* is a directed graph $G = (V, E)$ with a *weight function* $w\colon E \to \mathbb{R}$ that assigns each directed edge $e \in E$ a weight $w(e)$. Any such weight function extends to a path $P = (v_1, v_2, \ldots, v_k)$ by $w(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$.

A *shortest path* from s to d is a path of minimum weight among all paths from s to d. The *distance* between two vertices s and d, denoted by $D_{s,d}$, is the weight of a shortest path from s to d in G.

For a component C_i, we write $D^{(i)}_{u,v}$ for the distance from u to v restricted to paths whose vertices lie entirely in C_i. Similarly, $D^{(i\ldots t)}_{u,v}$ denotes the distance from u to v restricted to paths whose vertices lie in $C^{(i\ldots t)}$.

For simplicity, we will refer to directed graphs simply as *graphs* throughout the rest of this paper.

4 OUR SOLUTION

We propose an alternative algorithm for computing all-pairs shortest paths in graphs, building directly on the SCC-based solution of [19]. The algorithm of [19] presented in Algorithm 1 computes inter-component shortest paths by iterating over all edges in $\mathrm{Out}(C_i)$ directly. This is efficient when the number of outgoing edges is small, but the approach becomes costly when $|\mathrm{Out}(C_i)|$ is large: in the worst case, every vertex in C_i has an outgoing edge to every vertex in the next component, causing the number of relaxation attempts to grow as $O(|C_i|^2)$ per component pair, which can push the overall complexity above $O(n^3)$.

```
1  for i = 1 to t do
2  |  (D^(i)_{s,d})_{s,d∈C_i} ← APSP(C_i)
3  end
4  for i = t − 1 to 1 do
5  |  for each node d ∈ C^(i+1...t) do
6  |  |  for each node s ∈ C_i do
7  |  |  |  D^(i...t)_{s,d} ← ∞;
8  |  |  |  for each edge e ∈ Out(C_i) do
9  |  |  |  |  D^(i...t)_{s,d} ←
   |  |  |  |    min(D^(i...t)_{s,d}, D^(i)_{s,ini(e)} + w(e) + D^(i+1...t)_{ter(e),d})
10 |  |  |  end
11 |  |  end
12 |  end
13 end
```

Algorithm 1: APSP algorithm in [19]

Our solution modifies line 9 of Algorithm 1. Instead of relaxing through each edge $e \in \mathrm{Out}(C_i)$ in a single step as $D^{(i)}_{s,\mathrm{ini}(e)} + w(e) + D^{(i+1\ldots t)}_{\mathrm{ter}(e),d}$, we decompose this into two vertex relaxations: first through $\mathrm{ini}(e) \in I_i$, which contains all initial vertices of $\mathrm{Out}(C_i)$, then through $\mathrm{ter}(e) \in T_i$, which contains all terminal vertices of $\mathrm{Out}(C_i)$. This reduces the inter-component work when $|I_i|$ and $|T_i|$ are small relative to $|\mathrm{Out}(C_i)|$.

4.1 Reducing the Floyd–Warshall Relaxation

The roles of I_i and T_i are structurally different. Initial vertices route paths from vertices in C_i to the vertices T_i (bridge step), terminal vertices then propagate those paths into $C^{(i+1\ldots t)}$ (reduced-relaxation step).

In the notation below, the left subscripts I and T indicate relaxation through the initial vertex set I_i and the terminal vertex set T_i, respectively. The superscript $(i)+$ denotes a distance obtained by following a path within C_i and then one outgoing edge from C_i.

Given distances $D^{(i)}_{s,d}$ and $D^{(i+1\ldots t)}_{s,d}$ between vertex pairs within C_i and within $C^{(i+1\ldots t)}$, respectively, we propose the following two-step decomposition:

Bridge step. For each $s \in C_i$ and $h \in T_i$, let

$$ {}_I D^{(i)+}_{s,h} = \min_{e \in \mathrm{In}_i(h)} \left(D^{(i)}_{s,\mathrm{ini}(e)} + w(e) \right). $$

Reduced-relaxation step. For each $s \in C_i$ and $d \in C^{(i+1\ldots t)}$, let

$$ {}_T D^{(i\ldots t)}_{s,d} = \min_{h \in T_i} \left({}_I D^{(i)+}_{s,h} + D^{(i+1\ldots t)}_{h,d} \right). $$

Theorem 4.1. *With notation above, for each $i \in \{1, \ldots, t\}$, each $s \in C_i$ and $d \in C^{(i+1\ldots t)}$ we have $D^{(i\ldots t)}_{s,d} = {}_T D^{(i\ldots t)}_{s,d}$.*

Proof. We prove the theorem by contradiction. Assume that for some $s \in C_i$ and some $d \in C^{(i+1\ldots t)}$

$$ D^{(i\ldots t)}_{s,d} < {}_T D^{(i\ldots t)}_{s,d}. $$

Let P be a path from s to d with $w(P) = D^{(i\ldots t)}_{s,d}$. Then:

$$ w(P) = D^{(i\ldots t)}_{s,d} < {}_T D^{(i\ldots t)}_{s,d} = \min_{h \in T_i} \left({}_I D^{(i)+}_{s,h} + D^{(i+1\ldots t)}_{h,d} \right). $$

Since $s \in C_i$ and $d \notin C_i$, the path P must leave component C_i. Therefore, there exists at least one edge $(u, v) \in \mathrm{Out}(C_i)$ along this path. The path P thus decomposes into two subpaths connected by (u, v):

$$ P = s \rightsquigarrow u, v \rightsquigarrow d. $$

Since the subpaths $s \rightsquigarrow u$ and $v \rightsquigarrow d$ are no shorter than the corresponding shortest paths, we have:

$$ w(P) \geq D^{(i)}_{s,u} + w(u, v) + D^{(i+1\ldots t)}_{v,d}. $$

By the definition of ${}_I D^{(i)+}_{s,v}$:

$$ D^{(i)}_{s,u} + w(u, v) \geq \min_{e \in \mathrm{In}_i(h)} \left(D^{(i)}_{s,\mathrm{ini}(e)} + w(e) \right) = {}_I D^{(i)+}_{s,v}, $$

and thus $w(P) \geq {}_I D^{(i)+}_{s,v} + D^{(i+1\ldots t)}_{v,d}$. Moreover, by the definition of ${}_T D^{(i\ldots t)}_{s,d}$:

$$ D^{(i\ldots t)}_{s,d} = w(P) \geq {}_I D^{(i)+}_{s,v} + D^{(i+1\ldots t)}_{v,d} \geq \min_{h \in T_i} \left({}_I D^{(i)+}_{s,h} + D^{(i+1\ldots t)}_{h,d} \right) = {}_T D^{(i\ldots t)}_{s,d}, $$

a contradiction. □

4.2 Algorithm

Algorithm 2 implements the above two-step relaxation for all components, processing them in reverse topological order so that distances within $C^{(i+1\ldots t)}$ are available when component C_i is processed.

In the reduced-relaxation step, one could in principle iterate over all vertices in $C^{(i+1\ldots t)}$. However, for a fixed head vertex $h \in T_i$, only vertices reachable from h can lead to a finite value of

$$ {}_I D^{(i)+}_{s,h} + D^{(i+1\ldots t)}_{h,d}, $$

```
1  Tarjan(G);
   /* computes SCCs C_1, ..., C_t and their topological
      order                                              */
2  for i = 1 to t do
3  |  Solve APSP inside C_i;
4  end
5  for i = t − 1 to 1 do
6  |  D^(i...t)_{s,d} ← ∞   ∀s ∈ C_i, d ∈ C^(i+1...t);
   |  /* Bridge step                                     */
7  |  for each edge e ∈ Out(C_i) do
8  |  |  for each node s ∈ C_i do
9  |  |  |  D^(i...t)_{s,ter(e)} ←
   |  |  |    min(D^(i...t)_{s,ter(e)}, D^(i)_{s,ini(e)} + w(ini(e), ter(e)));
10 |  |  end
11 |  end
   |  /* Reduced-relaxation step                        */
12 |  for each node h ∈ T_i do
13 |  |  for each node s ∈ C_i do
14 |  |  |  for each node d reachable from h in C^(i+1...t) do
15 |  |  |  |  D^(i...t)_{s,d} ← min(D^(i...t)_{s,d}, D^(i...t)_{s,h} + D^(i+1...t)_{h,d});
16 |  |  |  end
17 |  |  end
18 |  end
19 end
```

Algorithm 2: Our Two-Step relaxation APSP algorithm

and therefore only those vertices need to be considered.

This observation can be combined with the Tree algorithm of [3]. For each head vertex h, the predecessor information obtained during the APSP computation on $C^{(i+1\ldots t)}$ defines a shortest-path tree rooted at h containing the vertices reachable from h. Instead of relaxing all reachable vertices independently, we traverse this tree in depth-first order.

The Tree algorithm provides an additional reduction beyond reachability filtering. If relaxation through h improves the current distance to a vertex d, the traversal continues with the next vertex in DFS order. If the relaxation does not improve $D_{s,d}$, the entire subtree rooted at d can be skipped, since relaxation through h cannot improve the distance to any descendant of d in the shortest-path tree.

Thus, the tree-based reduced-relaxation step avoids both unreachable vertices and relaxation attempts that are guaranteed not to improve the current distances. In the worst case, all reachable vertices are still processed, but when unsuccessful relaxations occur near the roots of large subtrees, a substantial number of relaxation attempts can be avoided.

4.3 Time Complexity

Computing SCCs with Tarjan's algorithm takes $O(|V| + |E|)$ time. Solving APSP within each component C_i takes $O(|C_i|^3)$ time. For the bridge step, the algorithm considers every source $s \in C_i$ and every outgoing edge $(k, h) \in \mathrm{Out}(C_i)$, resulting in $O(|C_i| \cdot |\mathrm{Out}(C_i)|)$

relaxations. The reduced-relaxation step performs at most $O(|T_i| \cdot |C_i| \cdot |C^{(i+1\ldots t)}|)$ relaxations. Incorporating the Tree algorithm does not change this asymptotic bound, since in the worst case all vertices in $C^{(i+1\ldots t)}$ may still need to be processed.

Summing the computations within and between all components, the total running time is

$$O\left(\sum_{i=1}^{t} |C_i|^3 + \sum_{i=1}^{t-1} |C_i| \cdot \left(|\mathrm{Out}(C_i)| + |T_i| \cdot |C^{(i+1\ldots t)}|\right)\right). \quad (1)$$

To see that this is $O(n^3)$ in the worst case, we first note that $\sum_{i=1}^{t} |C_i| = n$. Consequently this gives $\sum_{i=1}^{t} |C_i|^3 \leq O(n^3)$, with this bound tight (up to constants) when $t = 1$, i.e., when G consists of a single SCC and the method degenerates to plain Floyd–Warshall.

For the second sum, each term is bounded using $|T_i| \leq |C_i| \leq n$, $|C^{(i+1\ldots t)}| \leq n$, and $|\mathrm{Out}(C_i)| \leq |C_i| \cdot |C^{(i+1\ldots t)}| \leq n^2$, giving

$$|C_i|\left(|\mathrm{Out}(C_i)| + |T_i| \cdot |C^{(i+1\ldots t)}|\right) = O(|C_i| \cdot n^2).$$

Summing over i and again using $\sum_i |C_i| \leq n$ gives:

$$\sum_{i=1}^{t-1} |C_i|\left(|\mathrm{Out}(C_i)| + |T_i| \cdot |C^{(i+1\ldots t)}|\right) = O(n^3)$$

.

Combining both parts, the total running time is $O(n^3)$ in the worst case, matching the classical Floyd–Warshall algorithm. For strongly disconnected graphs where the components, the sets T_i, and the sets of outgoing edges are small, the number of inter-component relaxations is substantially reduced.

The two orderings of the steps are not fixed: the scheme above first bridges through I_i within C_i and only then propagates through T_i into $C^{(i+1\ldots t)}$, so the reduced-relaxation step is governed by $|T_i|$ and $|C_i|$. By symmetry, the same computation can be carried out with the steps reversed. The propagation into $C^{(i+1\ldots t)}$ is performed first: for each $k \in I_i$ and $d \in C^{(i+1\ldots t)}$, combining every outgoing edge originating at k with the distance $D^{(i+1\ldots t)}_{\mathrm{ter}(e),d}$. The bridging through $D^{(i)}_{s,k}$ is then applied afterwards, for each $s \in C_i$. In this variant shortest paths from each component cost

$$O(|C^{(i+1\ldots t)}| \cdot (|\mathrm{Out}(C_i)| + |C_i| \cdot |I_i|))$$

instead of

$$O(|C_i| \cdot (|\mathrm{Out}(C_i)| + |T_i| \cdot |C^{(i+1\ldots t)}|).$$

Since both orderings compute the same inter-component distances, the smaller of the two sets can always be chosen.

5 CONCLUSIONS

We presented a new SCC-guided approach to the APSP problem. The method builds on the decomposition of the graph into strongly connected components and replaces direct edge-based inside the component propagation with a two-step initial-terminal relaxation scheme. The main result of the paper is a formal proof that this decomposition is correct: once APSP has been solved inside a component and in the subgraph induced by later components, the distances between them can be computed through the proposed two-step relaxation process.

The resulting algorithm preserves the worst-case $O(n^3)$ running time of Floyd–Warshall, while offering a more refined structural view of inter-component computation. In particular, the number of relaxations depends on the sizes of the boundary sets I_i and T_i rather than directly on all outgoing edges. This makes the method effective for strongly disconnected or structurally sparse graphs, with many outgoing edges.

As future work, we plan to complement this theoretical analysis with an empirical evaluation of the proposed algorithm. This includes implementing the two-step relaxation scheme and comparing its running time against the classical Floyd–Warshall algorithm, and other algorithms mentioned in Section 2, across graphs of varying density, SCC structure, and boundary-set sizes. Such experiments would show how closely practical performance tracks the derived worst-case bound, and to what extent the reduction in relaxations produces measurable speedups.

ACKNOWLEDGMENTS

This work was partially supported by ARIS grant P2-0359.

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. 1993. *Network flows: Theory, Algorithms and Applications.* Prentice-Hall, Inc.
[2] A. Aini and A. Salehipour. 2012. Speeding up the Floyd–Warshall algorithm for the cycled shortest path problem. *Applied Mathematics Letters* 25 (2012), 1–5.
[3] Andrej Brodnik, Marko Grgurovič, and Rok Požar. 2021. Modifications of the Floyd–Warshall algorithm with nearly quadratic expected-time. *Ars Mathematica Contemporanea* 22 (2021), 1–22.
[4] C. Demetrescu and G. F. Italiano. 2004. A new approach to dynamic all pairs shortest paths. *J. ACM* 51 (2004), 968–992.
[5] C. Demetrescu and G. F. Italiano. 2006. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms* 2 (2006), 578–601.
[6] E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271.
[7] R. W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5 (1962), 345.
[8] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. 1986. The pairing heap: A new form of self-adjusting heap. *Algorithmica* 1 (1986), 111–129.
[9] M. L. Fredman and R.E. Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34 (1987), 596–615. https://doi.org/10.1145/28869.28874
[10] D. B. Johnson. 1977. Efficient algorithms for shortest paths in sparse networks. *J. ACM* 24 (1977), 1–13.
[11] D. Karger, D. Koller, and S. J. Phillips. 1993. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM J. Comput.* 22 (1993), 1199–1217.
[12] Giuseppe Lancia and Marcello Dalpasso. 2025. Speeding Up Floyd–Warshall's Algorithm to Compute All-Pairs Shortest Paths and the Transitive Closure of a Graph. *Algorithms* 18, 9 (2025), 560.
[13] Seth Pettie. 2004. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science* 312 (2004), 47–74. https://doi.org/10.1016/S0304-3975(03)00402-X
[14] Léon Planken, Mathijs de Weerdt, and Roman van der Krogt. 2012. Computing all-pairs shortest paths by leveraging low treewidth. *J. Artif. Int. Res.* 43 (Jan. 2012), 353–388.
[15] R. E. Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1 (1972), 146–160.
[16] R. E. Tarjan and U. Zwick. 2024. Finding strong components using depth-first search. *European Journal of Combinatorics* 119 (2024).
[17] I. H. Toroslu. 2023. The Floyd–Warshall all-pairs shortest paths algorithm for disconnected and very sparse graphs. *Software: Practice and Experience* 53 (2023), 1287–1303.
[18] S. Warshall. 1962. A theorem on boolean matrices. *J. ACM* 9 (1962), 11–12.
[19] Dani Zugan, Rok Požar, and Andrej Brodnik. 2025. Floyd–Warshall Algorithm for Sparse Graphs. *Algorithms* 18, 12 (2025), 766. https://doi.org/10.3390/a18120766