ISSN 1855-3966 (printed edn.), ISSN 1855-3974 (electronic edn.)
ARS MATHEMATICA CONTEMPORANEA 22 (2022) #P1.01
https://doi.org/10.26493/1855-3974.2467.497
(Also available at http://amc-journal.eu)

Modifications of the Floyd-Warshall algorithm with nearly quadratic expected-time*

Andrej Brodnik †
University of Primorska, UP FAMNIT, Glagoljaška 8, 6000 Koper, Slovenia
University of Primorska, UP IAM, Muzejski trg 2, 6000 Koper, Slovenia
University of Ljubljana, UL FRI, Večna pot 113, 1000 Ljubljana, Slovenia

Marko Grgurovič
University of Primorska, UP FAMNIT, Glagoljaška 8, 6000 Koper, Slovenia
University of Primorska, UP IAM, Muzejski trg 2, 6000 Koper, Slovenia

Rok Požar ‡
University of Primorska, UP FAMNIT, Glagoljaška 8, 6000 Koper, Slovenia
University of Primorska, UP IAM, Muzejski trg 2, 6000 Koper, Slovenia
IMFM, Jadranska 19, 1000 Ljubljana, Slovenia

Received 25 October 2020, accepted 10 April 2021, published online 25 November 2021

Abstract

The paper describes two relatively simple modifications of the well-known Floyd-Warshall algorithm for computing all-pairs shortest paths. A fundamental difference of both modifications in comparison to the Floyd-Warshall algorithm is that the relaxation is done in a smart way. We show that the expected-case time complexity of both algorithms is $O(n^2 \log^2 n)$ for the class of complete directed graphs on n vertices with arc weights selected independently at random from the uniform distribution on $[0, 1]$. Theoretically best known algorithms for this class of graphs are all based on Dijkstra's algorithm and obtain a better expected-case bound. However, by conducting an empirical evaluation we prove that our algorithms are at least competitive in practice with best know algorithms and, moreover, outperform most of them. The reason for the practical efficiency of the presented algorithms is the absence of use of priority queue.

*A preliminary version of this work has been published in Shortest Path Solvers: From Software to Wetware, volume 32 of Emergence, Complexity and Computation (2018). The authors would like to thank the reviewer for excellent comments that substantially improved the quality of the paper.

†This work is sponsored in part by the Slovenian Research Agency (research program P2-0359 and research projects J1-2481, J2-2504, and N2-0171).

‡Corresponding author. This work is supported in part by the Slovenian Research Agency (research program P1-0285 and research projects N1-0062, J1-9110, J1-9187, J1-1694, N1-0159, J1-2451).

Keywords: All-pairs shortest paths, probabilistic analysis.

Math. Subj. Class. (2020): 05C85, 68W40

1 Introduction

Finding shortest paths in graphs is a classic problem in algorithmic graph theory. Given a (directed) graph in which arcs are assigned weights, a shortest path between pair of vertices is such a path that infimizes the sum of the weights of its constituent arcs. The problem pops up very frequently also in practice in areas like bioinformatics, logistics, VLSI design (for a more comprehensive list of applications see e.g. [2]). Two of the most common problem's variants are the single-source shortest path problem and the all-pairs shortest path problem (APSP). In the first variant of the problem, we are searching for paths from a fixed vertex to every other vertex, while the APSP asks for a shortest path between every pair of vertices. In this paper we focus exclusively on the APSP variant of the problem.

In general, the APSP can be solved by using the technique of **relaxation**. The relaxation consists of testing whether we can improve the weight of the shortest path from u to v found so far by going via w, and updating it if necessary. In fact, the number of attempts to perform relaxation corresponds to the time complexity under the RAM model. A trivial text-book relaxation-based solution to the APSP is a dynamic programming Floyd-Warshall algorithm [11] running in $O(n^3)$ time on graphs with n vertices.

Moreover, also Dijkstra's algorithm [10] solving single-source shortest path problem is relaxation-based. However, since the order in which the relaxations are performed is greedy, it uses an additional priority queue data structure. Obviously we can solve the APSP running Dijkstra's algorithm from each vertex of the graph obtaining $O(mn \log n)$ solution where m is the number of arcs in the graph, provided we use the binary heap implementation of the priority queue. This is an improvement over the Floyd-Warshal solution for sparse graphs. Asymptotically we get an even better solution by using Fibonacci heaps over binary heaps yielding $O(n^2 \log n + mn)$ time complexity. We refer to such approaches as a **Dijkstra-like**, which inherently use some kind of a priority queue implementation.

However, the described solutions to the APSP using the Dijkstra's algorithm have at least two limitations. The first one is that all arc weights have to be non-negative. This can be alleviated by using Johnson's approach [15], which reweighs all arc weights making them non-negative. On such a graph we can now run Dijkstra's algorithm. The second limitation is related to the efficiency of the solution implementation. Namely, due to computer architecture efficient implementations exploit the issue of data locality; i.e. in consecutive memory accesses they try to access memory locations that are "close together". A similar observation is made in [5] for the solutions to the single-source shortest path problem, where authors show that a Fibonacci heap, as asymptotically better implementation of a priority queue, in practice underperform simple binary heap.

For dense graphs, a slightly better worst-case running time of $O(n^3 \log\log n / \log^2 n)$ over the $O(n^3)$-time Floyd-Warshall algorithm can be achieved by using an efficient matrix multiplication technique [13]. For sparse graphs on n vertices and with m non-negative weighted arcs fastest known solution [20] runs in time $O(mn + n^2 \log\log n)$.

E-mail addresses: andrej.brodnik@upr.si (Andrej Brodnik), marko.grgurovic@famnit.upr.si (Marko Grgurovič), rok.pozar@upr.si (Rok Požar)

Considering expected-case running-time of APSP algorithms we can find in the literature a number of good solutions assuming that input instances are generated according to a probability model on the set of complete directed graphs with arc weights. In the uniform model, arc weights are drawn at random, independently of each other, according to a common probability distribution. A more general model is the endpoint-independent model [3, 24], where, for each vertex v, a sequence of $n-1$ non-negative arc weights is generated by a deterministic or stochastic process and then randomly permuted and assigned to the outgoing arcs of v. In the vertex potential model [5, 6], arc weights can be both positive and negative. This is a probability model with arbitrary real arc weights but without negative cycles.

In the uniform model with arc weights drawn from the uniform distribution on $[0, 1]$, the $O(n^2 \log n)$ expected time complexity algorithms for solving the APSP were presented by Karger et al. [16] and Demetrescu and Italiano [8, 9], where the latter was inspired by the former. Another algorithm with the same expected time complexity was presented by Brodnik and Grgurovič [4]. Peres et al. [19] improved the Demetrescu and Italiano algorithm to theoretically optimal $O(n^2)$ by replacing the priority queue implementation with a more involved data structure yielding theoretically desired time complexity. In the endpoint-independent model, Spira [24] proved an expected-case time bound of $O(n^2 \log^2 n)$, which was improved by several authors. Takaoka and Moffat [25] improved the bound to $O(n^2 \log n \log\log n)$. Bloniarz [3] described an algorithm with expected-case running time $O(n^2 \log n \log^* n)$. Finally, Moffat and Takaoka [18] and Mehlhorn and Priebe [17] improved the running time to $O(n^2 \log n)$. In the vertex potential model, Cooper et al. [6] gave an algorithm with an expected-case running time $O(n^2 \log n)$. All the above algorithms use Dijkstra-like approach.

In this paper, we present two modifications of the Floyd-Warshall algorithm, which we name the Tree algorithm and the Hourglass algorithm. A fundamental difference of both modifications in relation to the Floyd-Warshall algorithm is a smarter way to perform the relaxations. This is done by introducing a tree structure that allows us to skip relaxations that do not contribute to the result. The worst-case time complexity of both algorithms remains $O(n^3)$, however, in the analysis we show that their expected running time is substantially better. To simplify the analysis, we consider the uniform model which gives us the following main result.

Theorem 1.1. *For complete directed graphs on n vertices with arc weights selected independently at random from the uniform distribution on $[0, 1]$, the Tree algorithm and the Hourglass algorithm both have an expected-case running time of $O(n^2 \log^2 n)$.*

The proof of our main result relies on the following well-known properties of the complete directed graph on n vertices with uniformly distributed arc weights on $[0, 1]$. First, *a maximum weight* of a shortest path in such a graph is $O(\log n/n)$ with high probability; second, *a longest shortest path* has $O(\log n)$ arcs with high probability; and third, *the maximum outdegree of the subgraph* consisting of all arcs that are shortest paths is $O(\log n)$ with high probability. These properties, together with the observation that if the relaxation on some vertex of the introduced tree structure fails, we can skip relaxations on the entire subtree defined by this vertex (see Lemma 3.1), then give the desired result. Since theoretically best expected-case APSP algorithms are based on Dijkstra's algorithm, it is interesting that a competitive approach can also be obtained by a modification of the Floyd-Warshall algorithm.

To prove the practical competitiveness of our algorithms, we supplement the theoretical analysis with an empirical evaluation. It should be pointed out, that all algorithms mentioned above with $o(n^2 \log^2 n)$ expected-case running time obtain a better theoretical bound. Moreover, Brodnik and Grgurovič in [4] show, for the same family of graphs as studied in this paper, practical supremacy of their algorithm over the algorithms due to Karger et al. [16] and Demetrescu and Italiano [8, 9] and consequently over the algorithm of Peres et al. [19], since its improvement of Demetrescu and Italiano solution does not improve the practical efficiency of the original algorithm. Therefore in the practical evaluation of Tree and Hourglass algorithms we compare them to the algorithm of Brodnik and Grgurovič [4] only. The reason for the practical efficiency of the presented algorithms is the absence of use of priority queue. Indeed, the Tree and Hourglass algorithms are simple to implement and use only simple structures such as vectors and arrays, which also exhibit a high data locality.

The structure of the paper is the following. Section 2 contains the necessary notation and basic definitions to make the paper self-contained. In Section 3 we describe the Tree and Hourglass algorithms. Properties of certain shortest paths in complete graphs with independently and uniformly distributed arc weights are analyzed in Section 4. The proof of the main result is presented in Section 5, while Section 6 contains empirical evaluation of the algorithms. In Section 7 we give some concluding remarks and open problems.

2 Preliminaries

All logarithms are base e unless explicitly stated otherwise. The model of computation used in algorithm design and analysis is the comparison-addition model, where the only allowed operations on arc weights are comparisons and additions.

A **digraph** (or **directed graph**) G is a pair (V, A), where V is a non-empty finite set of **vertices** and $A \subseteq V \times V$ a set of **arcs**. We assume $V = \{v_1, v_2, \ldots, v_n\}$ for some n. The two vertices joined by an arc are called its **endvertices**. For an arc $(u, v) \in A$, we say that u is its **tail**. The **outdegree** of $v \in V$, is the number of arcs in A that have v as their tail. The maximum outdegree in G is denoted by $\Delta(G)$.

A digraph $G' = (V', A')$ is a subdigraph of the digraph $G = (V, A)$ if $V' \subseteq V$ and $A' \subseteq A$. The **(vertex-)induced subdigraph** with the vertex set $S \subseteq V$, denoted by $G[S]$, is the subgraph (S, C) of G, where C contains all arcs $a \in A$ that have both endvertices in S, that is, $C = A \cap (S \times S)$. The **(arc-)induced subdigraph** with the arc set $B \subseteq A$, denoted by $G[B]$, is the subgraph (T, B) of G, where T is the set of all those vertices in V that are endvertices of at least one arc in B.

A path P in a digraph G from $v_{P,0}$ to $v_{P,m}$ is a finite sequence $P = v_{P,0}, v_{P,1}, \ldots, v_{P,m}$ of pairwise distinct vertices such that $(v_{P,i}, v_{P,i+1})$ is an arc of G, for $i = 0, 1, \ldots, m-1$. The **length** of a path P, denoted by $|P|$, is the number of vertices occurring on P. Any vertex of P other than $v_{P,0}$ or $v_{P,m}$ is an **intermediate** vertex. A k**-path** is a path in which all intermediate vertices belong to the subset $\{v_1, v_2, \ldots, v_k\}$ of vertices for some k. Obviously, a 0**-path** has no intermediate vertices.

A **weighted digraph** is a digraph $G = (V, A)$ with a **weight function** $w\colon A \to \mathbb{R}$ that assigns each arc $a \in A$ a **weight** $w(a)$. A weight function w can be extended to a path P by $w(P) = \sum_{i=0}^{m-1} w(v_{P,i}, v_{P,i+1})$. A **shortest path** from u to v, denoted by $u \rightsquigarrow v$, is a path in G whose weight is infimum among all paths from u to v. The **distance** between two vertices u and v, denoted by $D_G(u, v)$, is the weight of a shortest path $u \rightsquigarrow v$ in G.

The **diameter** of G is $\max_{u,v\in V} D_G(u,v)$, that is, the maximum distance between any two vertices in G. Given a subset $S \subseteq V$, the distance between S and a vertex v in G, denoted by $D_G(S,v)$, is $D_G(S,v) = \min_{u\in S} D_G(u,v)$. A shortest k-path from u to v is denoted by $u \overset{k}{\rightsquigarrow} v$. Further, we denote the set of arcs that are shortest k-paths in G by $A^{(k)}$ and the subdigraph $G[A^{(k)}]$ by $G^{(k)}$.

Finally, we will need some tools from combinatorics. In the balls-into-bins process M balls are thrown uniformly and independently into N bins. The maximum number of balls in any bin is called the **maximum load**. Let L_i denote the load of bin i, $i \in \{1,2,\ldots,N\}$. The next lemma, used in Subsection 4.3, provides an upper bound on the maximum load. It is a simplified version of a standard result, c.f. [23], tailored to our present needs. For completeness we provide the proof.

Lemma 2.1. *If M balls are thrown into N bins where each ball is thrown into a bin chosen uniformly at random, then $\mathbb{P}(\max_{1\le i\le N} L_i \ge e^2(M/N + \log N)) = O(1/N)$.*

Proof. First, we have $\mu = \mathbb{E}(L_i) = M/N$, $i = 1,2,\ldots,N$, and we can write each L_i as a sum $L_i = X_{i1} + X_{i2} + \cdots + X_{iM}$, where X_{ij} is a random variable taking value 1, if ball j is in bin i, and 0 otherwise. Next, since L_i is a sum of independent random variables taking values in $\{0,1\}$, we can apply, for any particular bin i and for every $c > 1$, the multiplicative Chernoff bound [12], which states that

$$\mathbb{P}(L_i \ge c\mu) \le \left(\frac{e^{c-1}}{c^c}\right)^{\mu} \le \left(\frac{e}{c}\right)^{c\mu}.$$

We consider two cases, depending on whether $\mu \ge \log N$ or not. Let $\mu \ge \log N$. Take $c = e^2$. Then,

$$\mathbb{P}(L_i \ge e^2\mu) \le \left(\frac{1}{e}\right)^{e^2\mu} \le \left(\frac{1}{e}\right)^{e^2 \log N} = \frac{1}{N^{e^2}} \le \frac{1}{N^2}.$$

Consider now $\mu < \log N$. Take $c = e^2 \frac{N}{M} \log N$. Since $x^{-x} \le \left(\frac{1}{e}\right)^x$ for all $x \ge e$, we have

$$\begin{aligned}\mathbb{P}\left(L_i \ge \mu e^2 \frac{N}{M} \log N\right) = \mathbb{P}(L_i \ge e^2 \log N) &\le \left(\frac{e}{c}\right)^{c\mu} = \left(\left(\frac{c}{e}\right)^{-\frac{c}{e}}\right)^{e\mu} \\ &\le \left(\left(\frac{1}{e}\right)^{\frac{c}{e}}\right)^{e\mu} = \left(\frac{1}{e}\right)^{e^2 \log N} \le \frac{1}{N^2}.\end{aligned}$$

Putting everything together, we get that

$$\begin{aligned}\mathbb{P}(L_i \ge e^2(\mu + \log N)) &\le \mathbb{P}(L_i \ge e^2\mu \mid \mu \ge \log N) + \mathbb{P}(L_i \ge e^2 \log N \mid \mu < \log N) \\ &\le \frac{1}{N^2} + \frac{1}{N^2} = \frac{2}{N^2}.\end{aligned}$$

This, by the union bound, implies that

$$\mathbb{P}\left(\max_{1\le i\le N} L_i \ge e^2(\mu + \log N)\right) \le \sum_{i=1}^{N} \mathbb{P}(L_i \ge e^2(\mu + \log N)) \le N\frac{2}{N^2} = O(1/N).$$

□

3 Speeding up the Floyd-Warshall algorithm

The Floyd-Warshall algorithm [11, 26] as presented in Algorithm 1 is a simple dynamic programming approach to solve APSP on a graph $G(V, A)$ represented by a weight matrix W, where $W_{ij} = w(v_i, v_j)$ if $(v_i, v_j) \in A$ and ∞ otherwise. Its running time is $O(n^3)$ due to three nested **for** loops.

Algorithm 1 FLOYD-WARSHALL(W)

```
1 for k := 1 to n do
2     for i := 1 to n do
3         for j := 1 to n do
4             if W_ik + W_kj < W_ij then                ▷ Relaxation
5                 W_ij := W_ik + W_kj
```

3.1 The Tree algorithm

Let us consider iteration k, and let OUT_k denote a shortest path tree rooted at vertex v_k in $G^{(k-1)}$. Intuitively, one might expect that the relaxation in lines 4-5 would not always succeed in lowering the value of W_{ij} which currently contains the weight $w(v_i \overset{k-1}{\rightsquigarrow} v_j)$. This is precisely the observation that we exploit to arrive at a more efficient algorithm: instead of simply looping through every vertex of V in line 3, we perform the depth-first traversal of OUT_k. This permits us to skip iterations which provably cannot lower the current value of W_{ij}. As the following lemma shows, if $w(v_i \overset{k}{\rightsquigarrow} v_j) = w(v_i \overset{k-1}{\rightsquigarrow} v_j)$, then $w(v_i \overset{k}{\rightsquigarrow} v_y) = w(v_i \overset{k-1}{\rightsquigarrow} v_y)$ for all vertices v_y in the subtree of v_j in OUT_k.

Lemma 3.1. *Let $v_j \in V \setminus \{v_k\}$ be some non-leaf vertex in OUT_k, $v_y \neq v_j$ an arbitrary vertex in the subtree of v_j in OUT_k, and $v_i \in V \setminus \{v_k\}$. Consider the path $v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j$. If $w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) \geq w(v_i \overset{k-1}{\rightsquigarrow} v_j)$, then $w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_y) \geq w(v_i \overset{k-1}{\rightsquigarrow} v_y)$.*

Proof. Since v_j is neither a leaf nor the root of OUT_k, we have $j < k$, and so $v_i \overset{k-1}{\rightsquigarrow} v_j \overset{k-1}{\rightsquigarrow} v_y$ is a $(k-1)$-path between v_i and v_y. Because $v_i \overset{k-1}{\rightsquigarrow} v_y$ is a shortest $(k-1)$-path between v_i and v_y, we have

$$\begin{aligned} w(v_i \overset{k-1}{\rightsquigarrow} v_y) &\leq w(v_i \overset{k-1}{\rightsquigarrow} v_j \overset{k-1}{\rightsquigarrow} v_y) = w(v_i \overset{k-1}{\rightsquigarrow} v_j) + w(v_j \overset{k-1}{\rightsquigarrow} v_y) \\ &\leq w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) + w(v_j \overset{k-1}{\rightsquigarrow} v_y) = w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j \overset{k-1}{\rightsquigarrow} v_y), \end{aligned}$$

where the last inequality follows by the assumption. Finally, since v_y is in the subtree rooted at v_j, we have $v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j \overset{k-1}{\rightsquigarrow} v_y = v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_y$, and so the last term is equal to $w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_y)$. This completes the proof. □

The pseudocode of the modified Floyd-Warshall algorithm augmented with the tree OUT_k, named the Tree algorithm, is given in Algorithm 2. To perform depth first search we first construct the tree OUT_k in line 3 using CONSTRUCTOUT given in Algorithm 3. For the construction of tree OUT_k an additional matrix π, where π_{ij} specifies the penultimate vertex on a k-shortest path from v_i to v_j (i.e. the vertex "before" v_j)[1] is used. More

[1] C.f. $\pi_{ij}^{(k)}$ in [7, Sec. 25.2].

Algorithm 2 TREE(W)

```
1  Initialize π, an n × n matrix, as π_ij := i.
2  for k := 1 to n do
3      OUT_k := CONSTRUCTOUT_k(π)
4      for i := 1 to n do
5          Stack := empty
6          Stack.push(v_k)
7          while Stack ≠ empty do
8              v_x := Stack.pop()
9              for all children v_j of v_x in OUT_k do
10                 if W_ik + W_kj < W_ij then                ▷ Relaxation
11                     W_ij := W_ik + W_kj
12                     π_ij := π_kj
13                     Stack.push(v_j)
```

Algorithm 3 CONSTRUCTOUT$_k(\pi)$

```
1  Initialize n empty trees: T_1, ..., T_n.
2  for i := 1 to n do
3      T_i.Root := v_i
4  for i := 1 to n do
5      if i ≠ k then
6          Make T_i a subtree of the root of T_{π_ki}.
   return T_k
```

precisely, the tree OUT_k is obtained from π by making v_i a child of $v_{\pi_{ki}}$ for all $i \neq k$. This takes $O(n)$ time. Finally, we replace the iterations in lines 3-5 in Algorithm 1 with depth-first tree traversal of OUT_k in lines 5-13 in Algorithm 2. Note that if, for a given i and a child v_j, the condition in line 10 evaluates to false we do not traverse the subtree of v_j in OUT_k.

Corollary 3.2. *The Tree algorithm correctly computes all-pairs shortest paths.*

Proof. The correctness of the algorithm follows directly from Lemma 3.1. □

Time complexity

Let T_k denote the running time of the algorithm TREE(W) in lines 3-13 at iteration k. As already said, line 3 requires $O(n)$ time. To estimate the time complexity of lines 4-13, we charge the vertex v_x in line 8 by the number of its children. This pays for lines 9-13. Furthermore, this means that on the one hand leaves are charged nothing, while on the other hand nobody is charged for the root v_k. To this end, let $SP_k^{(k)}$ be the set of all shortest k-paths that contain v_k and end at some vertex in the set $\{v_1, v_2, \ldots, v_k\}$. Now v_x in line 8 is charged at most $|SP_k^{(k)}|$ times over all iterations of i. Since the number of children of v_x is bounded from above by $\Delta(OUT_k)$, we can bound T_k from above by

$$T_k \leq |SP_k^{(k)}| \cdot \Delta(OUT_k) + O(n). \tag{3.1}$$

Practical improvement

Observe that in Algorithm 2 vertices of OUT_k are visited in a depth-first search (DFS) order, which is facilitated by using the stack. However, this requires pushing and popping of each vertex, as well as reading of all its children in OUT_k. We can avoid this by precomputing two read-only arrays *dfs* and *skip* to support the traversal of OUT_k. The array *dfs* consists of OUT_k vertices as visited in the DFS order. On the other hand, the array *skip* is used to skip OUT_k subtree when relaxation in line 10 of Algorithm 2 does not succeed.

In detail, for a vertex v_z, $skip_z$ contains the index in *dfs* of the first vertex after v_z in the DFS order that is not a descendant of v_z in OUT_k. Utilizing the arrays outlined above, we traverse OUT_k by scanning *dfs* in left-to-right order and using *skip* whenever a relaxation is not made. Consequently, we perform only two read operations per visited vertex. It should be pointed out that the asymptotic time remains the same, as this is solely a technical optimization.

3.2 The Hourglass algorithm

We can further improve the Tree algorithm by using another tree. The second tree, denoted by IN_k, is similar to OUT_k, except that it is a shortest path "tree" for paths $v_i \overset{k-1}{\rightsquigarrow} v_k$ for each $v_i \in V \setminus \{v_k\}$. Strictly speaking, this is not a tree, but if we reverse the directions of the arcs, it turns it into a tree with v_k as the root. Traversal of IN_k is used as a replacement of the **for** loop on variable i in line 4 of Algorithm 2 (in line 2 of Algorithm 1). As the following lemma shows, if $w(v_i \overset{k}{\rightsquigarrow} v_j) = w(v_i \overset{k-1}{\rightsquigarrow} v_j)$, then $w(v_y \overset{k}{\rightsquigarrow} v_j) = w(v_y \overset{k-1}{\rightsquigarrow} v_j)$ for all vertices v_y in the subtree of v_i in IN_k.

Lemma 3.3. *Let $v_i \in V \setminus \{v_k\}$ be some non-leaf vertex in IN_k and let $v_y \neq v_i$ be an arbitrary vertex in the subtree of v_i in IN_k, and $v_j \in V \setminus \{v_k\}$. Consider the path $v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j$. If $w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) \geq w(v_i \overset{k-1}{\rightsquigarrow} v_j)$, then $w(v_y \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) \geq w(v_y \overset{k-1}{\rightsquigarrow} v_j)$.*

Proof. Due to the choice of v_i and v_y we have: $v_y \overset{k-1}{\rightsquigarrow} v_k = v_y \overset{k-1}{\rightsquigarrow} v_i \overset{k-1}{\rightsquigarrow} v_k$. We want to show, that:

$$w(v_y \overset{k-1}{\rightsquigarrow} v_j) \leq w(v_y \overset{k-1}{\rightsquigarrow} v_i) + w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j).$$

Observe that $i < k$, since v_i is neither a leaf nor the root of IN_k. Thus we have:

$$w(v_y \overset{k-1}{\rightsquigarrow} v_j) \leq w(v_y \overset{k-1}{\rightsquigarrow} v_i) + w(v_i \overset{k-1}{\rightsquigarrow} v_j).$$

Putting these together we get the desired inequality:

$$w(v_y \overset{k-1}{\rightsquigarrow} v_j) \leq w(v_y \overset{k-1}{\rightsquigarrow} v_i) + w(v_i \overset{k-1}{\rightsquigarrow} v_j) \leq w(v_y \overset{k-1}{\rightsquigarrow} v_i) + w(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j).$$

□

The pseudocode of the modified Floyd-Warshall algorithm augmented with the trees OUT_k and IN_k, named the Hourglass algorithm[2], is given in Algorithms 4 and 5. To construct IN_k efficiently, we need to maintain an additional matrix ϕ_{ij} which stores the second

[2]The hourglass name comes from placing IN_k tree atop the OUT_k tree, which gives it an hourglass-like shape, with v_k being at the neck.

vertex on the path from v_i to v_j (cf. π and π_{ij}). Algorithm 6 constructs IN_k similarly to the construction of OUT_k, except that we use the matrix ϕ_{ik} instead. The only extra space requirement of the Hourglass algorithm that bears any significance is the matrix ϕ, which does not deteriorate the space complexity of $O(n^2)$. The depth-first traversal on IN_k is performed by a recursion on each child of v_k in line 7 of Algorithm 4. In the recursive step, given in Algorithm 5, we can prune OUT_k as follows: if v_i is the parent of v_y in IN_k and $v_i \overset{k-1}{\rightsquigarrow} v_j \leq v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j$, then the subtree of v_j can be removed from OUT_k, while inspecting the subtree of v_i in IN_k. Before the return from the recursion the tree OUT_k is reconstructed to the form it was passed as a parameter to the function.

Algorithm 4 HOURGLASS(W)

```
1  Initialize π, an n × n matrix, as π_ij := i.
2  Initialize φ, an n × n matrix, as φ_ij := j.
3  for k := 1 to n do
4      OUT_k := CONSTRUCTOUT_k(π)
5      IN_k := CONSTRUCTIN_k(φ)
6      for all children v_i of v_k in IN_k do
7          RECURSEIN(W, π, φ, IN_k, OUT_k, v_i)
```

Algorithm 5 RECURSEIN$(W, \pi, \phi, IN_k, OUT_k, v_i)$

```
1  Stack := empty
2  Stack.push(v_k)
3  while Stack ≠ empty do
4      v_x := Stack.pop()
5      for all children v_j of v_x in OUT_k do
6          if W_ik + W_kj < W_ij then                    ▷ Relaxation
7              W_ij := W_ik + W_kj
8              π_ij := π_kj
9              φ_ij := φ_ik
10             Stack.push(v_j)
11         else
12             Remove the subtree of v_j from OUT_k.
13 for all children v_i' of v_i in IN_k do
14     RECURSEIN(W, π, φ, IN_k, OUT_k, v_i')
15 Restore OUT_k by reverting changes done by all iterations of line 12.
```

In practice, the recursion can be avoided by using an additional stack, which further speeds up an implementation of the algorithm.

Corollary 3.4. *The Hourglass algorithm correctly computes all-pairs shortest paths.*

Proof. Observe, that lines 5-10 of Algorithm 5 are effectively the same as in Algorithm 2. Line 12 of Algorithm 5 does not affect the correctness of the algorithm due to Lemma 3.3, which states that, for any $v_{i'}$ that is a child of v_i in IN_k, these comparisons can be skipped, as they cannot lead to shorter paths. However, Lemma 3.3 does not apply to a sibling v_{i^*}

Algorithm 6 CONSTRUCTIN$_k(\phi)$

```
1  Initialize n empty trees: T_1, ..., T_n.
2  for i := 1 to n do
3      T_i.Root := v_i
4  for i := 1 to n do
5      if i ≠ k then
6          Make T_i a subtree of the root of T_{π_{ki}}.
   return T_k
```

of v_i, arising from line 6 of Algorithm 4. Therefore line 15 restores the tree OUT_k, which maintains the correctness of the algorithm. □

Finally, note that the worst-case time complexity of the Hourglass (and Tree) algorithm remains $O(n^3)$. The simplest example of this is when all shortest paths are the arcs themselves, at which point all leaves are children of the root and the tree structure never changes.

4 Properties of shortest k-paths in complete graphs

Let K_n denote a complete digraph on the vertex set $V = \{v_1, v_2, \ldots, v_n\}$.

4.1 Distances

We assume that arc weights of K_n are exponential random variables with mean 1 and that all $n(n-1)$ random arc weights are independent. Due to the memoryless property, it is easier to deal with exponentially distributed arc weights than directly with uniformly distributed arc weights. The aim of this subsection is to show that the diameter of $K_n^{(k)}$, the subdigraph of K_n consisting of all (weighted) arcs that are shortest k-paths in K_n, is $O(\log n/k)$ with very high probability. We note, however, that by the same argument as given in the beginning of Subsection 4.3, all results derived in this subsection for exponential arc weights also hold, asymptotically for $[0,1]$-uniformly distributed arc weights as soon as $k \geq \log^2 n$.

We start by considering for a fixed $u \in V$, the maximum distance in $K_n^{(k)}$ between u and other vertices in V. To this end, let $S = \{u, v_1, \ldots, v_k\} \subseteq V$, and let $\overline{S} = V \setminus S$. We clearly have

$$\max_{v\in V} D_{K_n^{(k)}}(u,v) \leq \max_{v\in S} D_{K_n[S]}(u,v) + \max_{v\in\overline{S}} D_{K_n[S\times\overline{S}]}(S,v), \tag{4.1}$$

that is, the maximum distance in $K_n^{(k)}$ between u and other vertices in V is bounded above by the sum of the maximum distance in $K_n[S]$ between u and other vertices in S, and by the maximum distance in $K_n[S \times \overline{S}]$ between S and vertices in $\overline{S}$. We note that $K_n[S]$ is a complete digraph on $|S|$ vertices and $K_n[S \times \overline{S}]$ is a complete bipartite digraph with bipartition $(S, \overline{S})$.

To provide an upper bound on $\max_{v\in S} D_{K_n[S]}(u,v)$, we use the following result, which follows from the equation (2.8) in the proof of Theorem 1.1 of Janson [14].

Theorem 4.1 ([14, Theorem 1.1]). *Let $u \in V$ be a fixed vertex of K_n. Then for every $a > 0$, we have*

$$\mathbb{P}\left(\max_{v \in V} D_{K_n}(u,v) \geq \frac{a \log n}{n} \right) = O(e^a n^{2-a} \log^2 n).$$

Lemma 4.2. *Let $8 \leq k \leq n$, and let $S \subseteq V$ with $|S| = k$. Then, for a fixed $u \in S$ and for any constant $c > 0$, we have*

$$\mathbb{P}\left(\max_{v \in S} D_{K_n[S]}(u,v) \geq \frac{c \log n}{k} \right) = O(n^{2-c/2} \log^2 n).$$

Proof. By Theorem 4.1, for any $a > 0$ we have

$$\mathbb{P}\left(\max_{v \in S} D_{K_n[S]}(u,v) \geq \frac{a \log k}{k} \right) = O(e^a k^{2-a} \log^2 k).$$

Setting $a = c \log n / \log k$ we get

$$e^a k^{2-a} \log^2 k = e^{c \log n / \log k} k^2 k^{-c \log n / \log k} \log^2 k \leq (e^{\log n})^{c/2} k^2 (k^{\log_k n})^{-c} \log^2 k.$$

In the last step we used the fact that $1/\log k \leq 1/2$ for $k \geq 8$ and that $\log n / \log k = \log_k n$. Furthermore,

$$(e^{\log n})^{c/2} k^2 (k^{\log_k n})^{-c} \log^2 k = n^{c/2} k^2 n^{-c} \log^2 k = O(n^{2-c/2} \log^2 n),$$

and the result follows. □

Next, we provide an upper bound on $\max_{v \in \overline{S}} D_{K_n[S \times \overline{S}]}(S,v)$.

Lemma 4.3. *Let $1 \leq k \leq n$, let $S \subseteq V$ with $|S| = k$, and let $\overline{S} = V \setminus S$. Then for any constant $c > 0$, we have*

$$\mathbb{P}\left(\max_{v \in \overline{S}} D_{K_n[S \times \overline{S}]}(S,v) \geq \frac{c \log n}{k} \right) = O(n^{1-c} \log n).$$

Proof. Let $Z = \max_{v \in \overline{S}} D_{K_n[S \times \overline{S}]}(S,v)$. Arguing similarly as in the proof of Theorem 1.1 of Janson [14], Z is distributed as

$$\sum_{j=k}^{n-1} X_j,$$

where X_j are independent exponentially distributed random variables with mean $\frac{1}{k(n-j)}$. First, for any constant $c > 0$, the Chernoff bound [12] states that

$$\mathbb{P}(Z \geq c \log n/k) \leq e^{-tc \log n} \mathbb{E}(e^{ktZ}).$$

Further, for $-\infty < t \leq 1$, we have

$$\mathbb{E}(e^{ktZ}) = \prod_{j=k}^{n-1} \mathbb{E}(e^{ktX_j}) = \prod_{j=k}^{n-1} \left(1 - \frac{t}{n-j}\right)^{-1}.$$

Using the inequality $-\log(1-x) \le x + x^2$ for all $0 \le x \le 1/2$, we can bound, for all $0 < t < 1$ and $k \le j \le n-2$, each term $(1 - t/(n-j))^{-1}$ as follows

$$\left(1 - \frac{t}{n-j}\right)^{-1} = \exp\left(-\log\left(1 - \frac{t}{n-j}\right)\right) \le \exp\left(\frac{t}{n-j} + \left(\frac{t}{n-j}\right)^2\right).$$

This gives us

$$\begin{aligned}\mathbb{P}(Z \ge c\log n/k) &\le (1-t)^{-1}\exp\left(-tc\log n + \sum_{j=k}^{n-2}\left(\frac{t}{n-j} + \left(\frac{t}{n-j}\right)^2\right)\right)\\ &= (1-t)^{-1}\exp(-tc\log n + t\log(n-k) + O(1)).\end{aligned}$$

Taking $t = 1 - 1/\log n$, we finally get

$$\mathbb{P}(Z \ge c\log n/k) \le (1/\log n)^{-1}\exp(-c\log n + \log n + O(1)) = O(n^{1-c}\log n).$$

□

We are now ready to show that the diameter of $K_n^{(k)}$ is $O(\log n/k)$ with very high probability.

Theorem 4.4. *Let $8 \le k \le n$. Then, for any constant $c > 0$, we have*

$$\mathbb{P}\left(\max_{u,v\in V} D_{K_n^{(k)}}(u,v) \ge \frac{c\log n}{k}\right) = O(n^{3-c/4}\log^2 n).$$

Proof. Let $S = \{u, v_1, \dots, v_k\} \subseteq V$, let $\overline{S} = V \setminus S$, and write $\alpha = c\log n/k$. Then, by inequality (4.1), we have

$$\begin{aligned}\mathbb{P}\left(\max_{v\in V} D_{K_n^{(k)}}(u,v) \ge \alpha\right) &\le \mathbb{P}\left(\max_{v\in S} D_{K_n[S]}(u,v) + \max_{v\in\overline{S}} D_{K_n[S\times\overline{S}]}(S,v) \ge \alpha\right)\\ &\le \mathbb{P}\left(\max_{v\in S} D_{K_n[S]}(u,v) \ge \frac{\alpha}{2}\right) + \mathbb{P}\left(\max_{v\in\overline{S}} D_{K_n[S\times\overline{S}]}(S,v) \ge \frac{\alpha}{2}\right).\end{aligned}$$

By Lemma 4.2, we have

$$\mathbb{P}\left(\max_{v\in S} D_{K_n[S]}(u,v) \ge \frac{\alpha}{2}\right) = O(n^{2-c/4}\log^2 n),$$

and, by Lemma 4.3,

$$\mathbb{P}\left(\max_{u\in\overline{S}} D_{K_n[S\times\overline{S}]}(S,v) \ge \frac{\alpha}{2}\right) = O(n^{1-c/2}\log n).$$

Putting everything together, we get

$$\mathbb{P}\left(\max_{v\in V} D_{K_n^{(k)}}(u,v) \ge \alpha\right) = O(n^{2-c/4}\log^2 n),$$

which, by the union bound, implies

$$\mathbb{P}\left(\max_{u,v\in V} D_{K_n^{(k)}}(u,v) \ge \alpha\right) \le n\mathbb{P}\left(\max_{u\in V} D_{K_n^{(k)}}(v,u) \ge \alpha\right) = O(n^{3-c/4}\log^2 n).$$

□

4.2 Lengths

Let all arc weights of K_n be either independent $[0,1]$-uniform random variables or independent exponential random variables with mean 1. In this subsection, we bound the length of the longest shortest k-path in K_n.

The proof of our next lemma follows directly from Theorem 1.1 of Addario-Berry et. al [1] on the longest shortest path in K_n.

Theorem 4.5 ([1, Theorem 1.1])**.** *The following two properties hold:*

(i) For every $t > 0$, we have

$$\mathbb{P}\left(\max_{u,v \in V} |u \rightsquigarrow v| \geq \alpha^* \log n + t \right) \leq e^{\alpha^* + t/\log n} e^{-t},$$

where $\alpha^* \approx 3.5911$ is the unique solution of $\alpha \log \alpha - \alpha = 1$.

(ii) $\mathbb{E}(\max_{u,v \in V} |u \rightsquigarrow v|) = O(\log n)$.

Lemma 4.6. *The following two properties hold:*

(i) For every $c > 5$ and $8 \leq k \leq n$, we have $\mathbb{P}(\max_{u,v \in V} |u \overset{k}{\rightsquigarrow} v| \geq c \log n) = O(n^{2-c/2})$.

(ii) $\mathbb{E}(\max_{u,v \in V} |u \overset{k}{\rightsquigarrow} v|) = O(\log k)$.

Proof. Let $S = \{v_1, v_2, \ldots, v_k\}$, and let $u \to w \overset{k}{\rightsquigarrow} z \to v$ be a shortest k-path in K_n. Since $w \overset{k}{\rightsquigarrow} z$ is a shortest path from w to z in $K_n[S]$, we have

$$\max_{u,v \in V} |u \overset{k}{\rightsquigarrow} v| \leq \max_{w,z \in S} |w \overset{k}{\rightsquigarrow} z| + 2. \tag{4.2}$$

By (i) of Theorem 4.5, for any $t > 0$,

$$\mathbb{P}\left(\max_{w,z \in S} |w \overset{k}{\rightsquigarrow} z| \geq \alpha^* \log k + t \right) \leq e^{\alpha^* + t/\log k} e^{-t},$$

where $\alpha^* \approx 3.5911$ is the unique solution of $\alpha \log \alpha - \alpha = 1$. Using $t = (c - \alpha^*) \log n - 2$ gives us

$$\begin{aligned} \mathbb{P}\left(\max_{w,z \in S} |w \overset{k}{\rightsquigarrow} z| + 2 \geq \alpha^* \log(k/n) + c \log n \right) &\leq e^{\alpha^* - 2/\log k + 2} e^{(c-\alpha^*)(\frac{\log n}{\log k} - \log n)} \\ &\leq e^{\alpha^* - 2/\log k + 2} (e^{\log n})^{1/2(\alpha^* - c)} \\ &= O(n^{2-c/2}). \end{aligned}$$

By inequality (4.2), we have

$$\begin{aligned} \mathbb{P}\left(\max_{u,v \in V} |u \overset{k}{\rightsquigarrow} v| \geq c \log n \right) &\leq \mathbb{P}\left(\max_{w,z \in S} |u \overset{k}{\rightsquigarrow} v| + 2 \geq c \log n \right) \\ &\leq \mathbb{P}\left(\max_{w,z \in S} |w \overset{k}{\rightsquigarrow} z| + 2 \geq \alpha^* \log(k/n) + c \log n \right), \end{aligned}$$

and (i) follows.

To prove (ii), we note that, by (ii) of Theorem 4.5, $\mathbb{E}(\max_{u,v \in S} |u \overset{k}{\rightsquigarrow} v|) = O(\log k)$, and, by inequality (4.2), the result follows.

□

4.3 Maximum outdegree

Let arc weights of K_n be independent $[0,1]$-uniform random variables. Our goal in this subsection is to show that the maximum outdegree of a shortest path tree OUT_k in $K_n^{(k)}$ is $O(\log k + (n-k)/k)$ with high probability for all $k \geq \log^2 n$.

Let now $S = \{v_1, v_2, \ldots, v_k\}$ and $\overline{S} = V \setminus S$. We can consider OUT_k as consisting of the subtree $OUT_k[S]$ to which each vertex from $\overline{S}$ is attached as a leaf. To see how these vertices are attached to $OUT_k[S]$, let us assume for the moment that arc weights are exponentially distributed with mean 1. Then, it is easy to see that a vertex $v \in \overline{S}$ is attached to that one in S with which it forms a shortest arc, say a^v, between S and v. Let $(K_n[S \times \overline{S}])^*$ be the subdigraph of $K_n[S \times \overline{S}]$ with the set V of vertices and the set $\{a^v \mid v \in \overline{S}\}$ of arcs. By observing that $OUT_k[S]$ is a subdigraph of the graph $(K_n[S])^{(k)}$ consisting of all arcs that are shortest paths in $K_n[S]$, we have

$$\Delta(OUT_k) \leq \Delta((K_n[S])^{(k)}) + \Delta((K_n[S \times \overline{S}])^*). \tag{4.3}$$

To extend the latter bound to uniform distribution, we use a standard coupling argument as in [1]. Let U be a random variable uniform on $[0,1]$. Then $-\log(1-U)$ is an exponential random variable with mean 1, and so we can couple the exponential arc weights $W'(u,v)$ to uniform arc weights $W(u,v)$ by setting $W'(u,v) = -\log(1-W(u,v))$. As $x \leq -\log(1-x) \leq x + 2x^2$ for all $0 \leq x \leq 1/2$, we have that, for all arcs (u,v) of K_n, $|W'(u,v) - W(u,v)| = O((W'(u,v))^2)$, uniformly for all $W'(u,v) \leq 1/2$. In particular, if $W'(u,v) \leq 12\log n/k$, say, and $k \geq \log^2 n$, then $|W'(u,v) - W(u,v)| = O(1/\log^2 n)$ for n large enough, and so for a path P with $O(\log n)$ vertices and with $W'(P) \leq 12\log n/k$, we have

$$|W'(P) - W(P)| = O(1/\log n)$$

for n large enough. By Theorem 4.4, with very high probability a shortest $(k-1)$-path in K_n with the exponential arc weights has weight less than $12\log n/k$, while by (i) of Lemma 4.6, with very high probability it has $O(\log n)$ vertices. It then follows easily that, for all n sufficiently large and $k \geq \log^2 n$, the bound as in (4.3) holds for uniform distribution, as well.

The following result on the maximum outdegree in the subgraph $(K_n[S])^{(k)}$ of the complete graph $K_n[S]$ on k vertices with $[0,1]$-uniform arc weights can be found in Peres et al. [19].

Lemma 4.7 ([19, Lemma 5.1])**.** *Let $1 \leq k \leq n$ and let $S \subseteq V$ with $|S| = k$. Then, for every $c > 6$, we have $\mathbb{P}(\Delta((K_n[S])^{(k)}) > c\log k) = O(k^{1-c/6})$.*

The maximum outdegree in $(K_n[S \times \overline{S}])^*$ is directly related to the maximum load in the balls-into-bins process, which is used in the proof of the following lemma.

Lemma 4.8. *Let $1 \leq k \leq n$, let $S \subseteq V$ with $|S| = k$, and let $\overline{S} = V \setminus S$. Then,*

$$\mathbb{P}(\Delta((K_n[S \times \overline{S}])^*) \geq e^2((n-k)/k + \log k)) = O(k^{-1}).$$

Proof. Consider vertices from S as bins and vertices from $\overline{S}$ as balls. For $v \in \overline{S}$, each arc in $S \times v$ is equally likely to be the shortest, so v is thrown into a bin chosen uniformly at random, and the result follows by Lemma 2.1 for $N = k$ and $M = n - k$. □

We are now ready to prove the main result of this subsection.

Theorem 4.9. *For every $k \geq \log^2 n$, we have*

$$\mathbb{P}\left(\Delta(OUT_k) \geq (e^2+12)\log k + e^2\frac{n-k}{k}\right) = O(k^{-1}).$$

Proof. Let $S = \{v_1, v_2, \ldots, v_k\}$ and $\overline{S} = V \setminus S$. Further, let us write $\alpha = 12\log k$ and $\beta = e^2((n-k)/k\log k)$. By the inequality (4.3), for every $k \geq \log^2 n$, we have

$$\begin{aligned}\mathbb{P}(\Delta(OUT_k) \geq \alpha+\beta) &\leq \mathbb{P}(\Delta((K_n[S])^{(k)}) + \Delta((K_n[S\times\overline{S}])^*) \geq \alpha+\beta)\\ &\leq \mathbb{P}(\Delta((K_n[S])^{(k)}) \geq \alpha) + \mathbb{P}(\Delta((K_n[S\times\overline{S}])^*) \geq \beta).\end{aligned}$$

By Lemma 4.7, we have $\mathbb{P}(\Delta((K_n[S])^{(k)}) \geq \alpha) \leq 1/k$. Similarly, by Lemma 4.8, we have $\mathbb{P}(\Delta((K_n[S\times\overline{S}])^*) \geq \beta) \leq 1/k$. Hence, $\mathbb{P}(\Delta(OUT_k) \geq \alpha+\beta) \leq 1/k + 1/k = O(1/k)$. □

5 Expected-case analysis

We perform an expected-case analysis of the Tree algorithm for the complete directed graphs on n vertices with arc weights selected independently at random from the uniform distribution on $[0,1]$. Recall that $SP_k^{(k)}$ is the set of all shortest k-paths that contain v_k and end at some vertex in the set $\{v_1, v_2, \ldots, v_k\}$. We first show that the expected number of paths in $SP_k^{(k)}$ is $O(n\log k)$.

Lemma 5.1. *For each $k = 1, 2, \ldots, n$, we have $\mathbb{E}(|SP_k^{(k)}|) = O(n\log k)$.*

Proof. For $v_i \in V$, let $SP_i^{(k)}$ denote the set of all shortest k-paths that contain v_i and end at some vertex in the set $\{v_1, v_2, \ldots, v_k\}$. Note that

$$\sum_{i=1}^{k} |SP_i^{(k)}| \leq \sum_{i=1}^{n}\sum_{j=1}^{k} |v_i \overset{k}{\rightsquigarrow} v_j|.$$

By symmetry, we have $\mathbb{E}(|SP_i^{(k)}|) = \mathbb{E}(|SP_j^{(k)}|)$ for arbitrary $i, j \in \{1, 2, \ldots, k\}$, and hence

$$k\mathbb{E}(|SP_k^{(k)}|) = \sum_{i=1}^{k}\mathbb{E}(|SP_i^{(k)}|) \leq \sum_{i=1}^{n}\sum_{j=1}^{k}\mathbb{E}(|v_i \overset{k}{\rightsquigarrow} v_j|) \leq kn\mathbb{E}(\max_{u,v\in V}|u \overset{k}{\rightsquigarrow} v|).$$

By (ii) of Lemma 4.6, we get that $\mathbb{E}(|SP_k^{(k)}|) = O(n\log k)$. □

We are now ready to analyse the expected time of the Tree algorithm.

Theorem 5.2. *The Tree algorithm has an expected-case running time of $O(n^2\log^2 n)$ for the complete directed graphs on n vertices with arc weights selected independently at random from the uniform distribution on $[0,1]$.*

Proof. To estimate the number of comparisons T_k at iteration k, we consider two cases. First, for $k < \log^2 n$ we bound T_k from above by n^2. Second, we estimate $\mathbb{E}(T_k)$ for $k \geq \log^2 n$. For every $c > 0$, we have

$$\begin{aligned}\mathbb{E}(T_k) = \mathbb{E}(T_k \mid \Delta(OUT_k) < c) \cdot \mathbb{P}(\Delta(OUT_k) < c) \\ + \mathbb{E}(T_k \mid \Delta(OUT_k) \geq c) \cdot \mathbb{P}(\Delta(OUT_k) \geq c).\end{aligned}$$

Using inequality (3.1) we get

$$\begin{aligned}\mathbb{E}(T_k \mid \Delta(OUT_k) < c) &\leq \mathbb{E}(|SP_k^{(k)}| \cdot \Delta(OUT_k) + O(n) \mid \Delta(OUT_k) < c) \\ &\leq c \cdot \mathbb{E}(|SP_k^{(k)}|) + O(n).\end{aligned}$$

As T_k is always at most n^2, we have $\mathbb{E}(T_k \mid \Delta(OUT_k) \geq c) \leq n^2$. Further, taking into account that $\mathbb{P}(\Delta(OUT_k) < c) \leq 1$, we get

$$\mathbb{E}(T_k) \leq c \cdot \mathbb{E}(|SP_k^{(k)}|) + O(n) + n^2 \cdot \mathbb{P}(\Delta(OUT_k) \geq c).$$

Take $c = (e^2 + 12)\log k + e^2 \frac{n-k}{k}$. Then, by Lemma 4.9, we have $\mathbb{P}(\Delta(OUT_k) \geq c) = O(k^{-1})$. Moreover, by Lemma 5.1, we have $\mathbb{E}(|SP_k^{(k)}|) = O(n \log k)$, which gives us

$$\begin{aligned}\mathbb{E}(T_k) &= O((e^2 + 12)n \log^2 k + e^2(n-k)n \log k/k) + O(n) + O(n^2/k) \\ &= O(n \log^2 n + n^2 \log n/k).\end{aligned}$$

Putting everything together, we bound the expected time of the algorithm from above as

$$\begin{aligned}\mathbb{E}\left(\sum_{k=1}^{n} T_k\right) &= \sum_{k=1}^{\log^2 n - 1} \mathbb{E}(T_k) + \sum_{k=\log^2 n}^{n} \mathbb{E}(T_k) \\ &\leq \sum_{k=1}^{\log^2 n - 1} n^2 + \sum_{k=\log^2 n}^{n} O(n \log^2 n + n^2 \log n/k) = O(n^2 \log^2 n),\end{aligned}$$

as claimed. □

We conclude the section with a proof of the main theorem.

Proof of Theorem 1.1. The Hourglass algorithm does not have a worse bound than the Tree variant, so the result follows by Theorem 5.2. □

6 Empirical evaluation

All algorithms were implemented in `C++` and compiled using `g++ -march=native -O3`. The tests were ran on an Intel(R) Core(TM) i7-2600@3.40GHz with 8GB RAM running Windows 7 64-bit.

To make the comparison between Floyd-Warshall and its modified versions fairer, we improved the Floyd-Warshall algorithm with a simple modification skipping combinations of i and k where $W_{ik} = \infty$, and consequently reducing the number of relaxations of the algorithm to $R_{FW} \leq n^3$.

The experiments were conducted on the following random digraphs: (i) uniform random digraphs with arc weights uniformly distributed on the interval $[0, 1]$, and (ii) unweighted random digraphs. In both cases, the digraphs were constructed by first setting the desired vertex count and density. Then, a random Hamiltonian cycle was constructed, ensuring the strong connectivity of the digraph. After the cycle was constructed, the remaining $n(n-2)$ arcs were put into a list and randomly permuted, and then added into the digraph until the desired density was reached. Finally, algorithms were executed on the instance, and their running times were recorded. Tests were conducted ten times and averaged, with each test running on a different randomly generated graph.

6.1 Empirical comparison of the number of relaxations

Our motivation when designing the Hourglass and Tree algorithms was to skip relaxations that are not contributing to the result. To verify the theoretical results on the expected number of relaxations in practice we conducted two experiments in which we counted the number of relaxations by different algorithms. For the first experiment we generated a subfamily of digraphs from (i), mentioned above, consisting of complete digraphs of varying size vertex set. On contrary, for the second experiment we generated another subfamily of digraphs from (i), now consisting of sparse digraphs with fixed vertex set and variable arc density. The results of experiments are presented in the plots relative to the number of relaxations performed by the Floyd-Warshall algorithm; i.e. all numbers of relaxations are divided by R_{FW}.

The results of the first experiment, in which $R_{FW} = n^3$ since digraphs are complete, are presented in Figure 1. To relate the theoretical upper bound of $O(n^2 \lg^2 n)$ of the Tree algorithm and the experimental results, we added also the plot of the function $60\frac{n^2 \lg^2 n}{n^3}$. We chose the constant 60 so that the plots of the Tree algorithm and the added function start at the same initial point, namely at 2^8 vertices. The results of the second experiment for

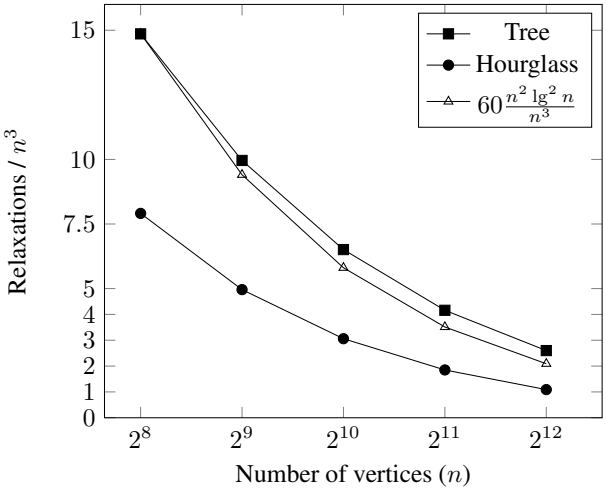

Figure 1: Complete digraphs of various sizes with the number of relaxations of algorithms divided by n^3.

$n = 1024$ vertices and sizes of the arc set varying between $n^2/10$ and $8n^2/10$ are shown in Figure 2.

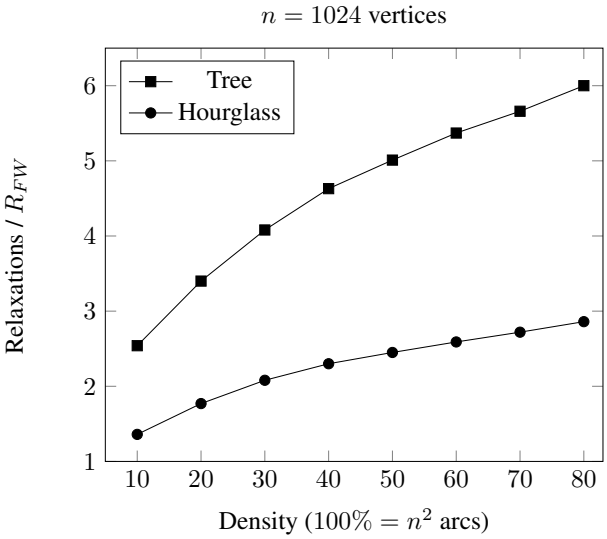

Figure 2: Digraphs with $n = 1024$ vertices and various arc densities with the number of relaxations of algorithms divided by R_{FW}.

In Figure 1 we see a significant reduction of relaxations which also implies the decrease of running time of the Tree and Hourglass algorithms. From the plot we can also see that the experimental results indicate that the theoretical upper bound of the Tree algorithm is asymptotic tight. The experiments on sparse digraphs (see Figure 2) also show a reduction in relaxations as digraphs become sparser.

6.2 Empirical comparison of running times

As discussed in the introduction, we compared the Tree[3] and Hourglass algorithms with the Floyd-Warshall [11, 26] and Dijkstra [10] algorithms, as well as the algorithm of Brodnik and Grgurovič [4], which we refer to as Propagation. These algorithms were chosen since they proved to perform best out of a larger group of algorithms compared in [4].

It should be pointed out, that breadth-first search is extremely efficient in solving APSP on unweighted digraphs. However, we did not include breadth-first search in comparisons, because we consider unweighted graph instances only as the worst-case instances of the general shortest path problem (each arc is part of at least one shortest path in such instances).

The algorithms were tested on the graph families (i) and (ii) described at the beginning of this section, with sizes of the vertex set varying between 512 and 4096, and sizes of the arc set varying between $n^{1.1}$ and n^2. As the priority queue in the Dijkstra and Propagation algorithms we used pairing heaps since they are known to perform especially well in solving APSP in practice [22], even though the amortized complexity of their decrease key

[3] In the tests, we used the implementation of the algorithm with improvements from Subsection 3.1.

operation takes $O(2^{2\sqrt{\lg \lg n}})$ in comparison to O(1) of Fibonacci heaps [21]. We used the implementation of pairing heaps from the Boost Library, version 1.55.

The results for uniform random digraphs presented in Figure 3 show that both, Propagation and Tree, outperform the other algorithms on all vertex and arc densities. As the size n of graphs increases, the running time of Hourglass approaches the running time of Tree, but the constant factors still prove to be too large for Hourglass to prevail because of a more clever exploration strategy. Moreover, it is interesting to see that Floyd-Warshall based Tree and Hourglass outperform Dijkstra on sparse graphs.

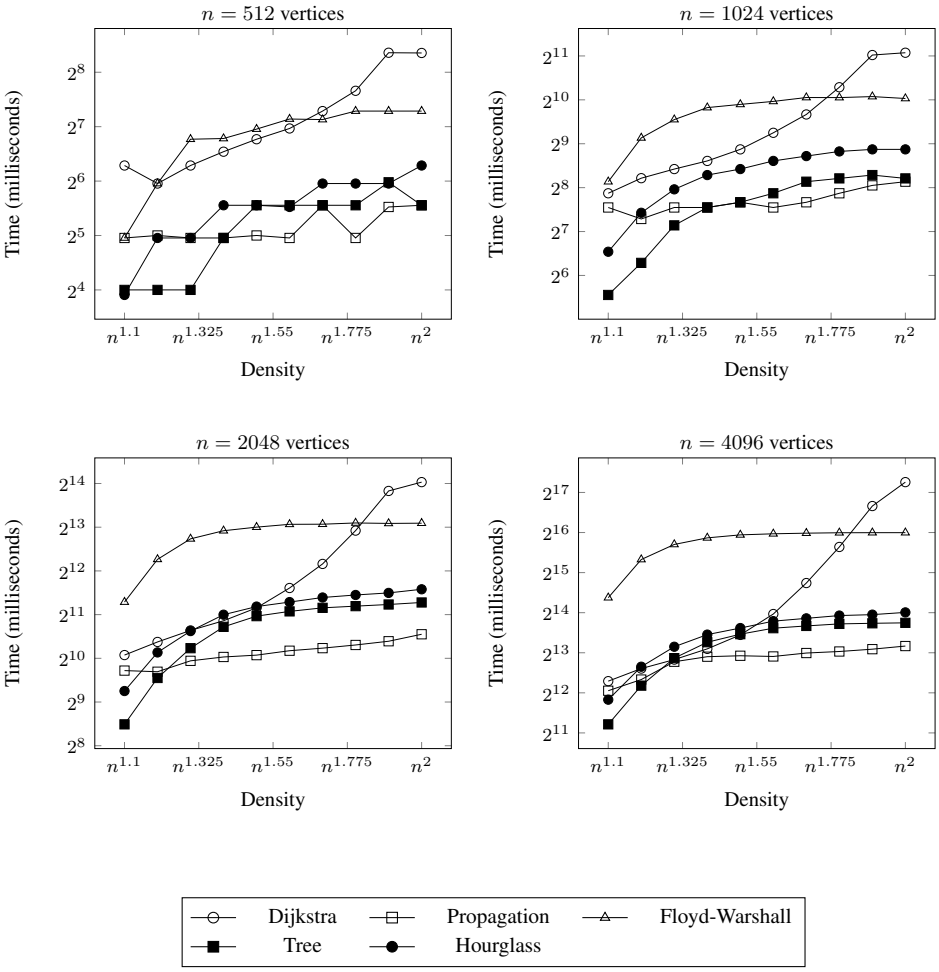

Figure 3: Experimental results on family (i) – uniform digraphs.

The results for unweighted random digraphs are shown in Figure 4. What is interesting is that Tree and Hourglass remain competitive with Dijkstra, and even outperforming it on smaller graphs in some instances. In contrast, the performance of Propagation falls short of Dijkstra because each arc is part of at least one shortest path in these graphs.

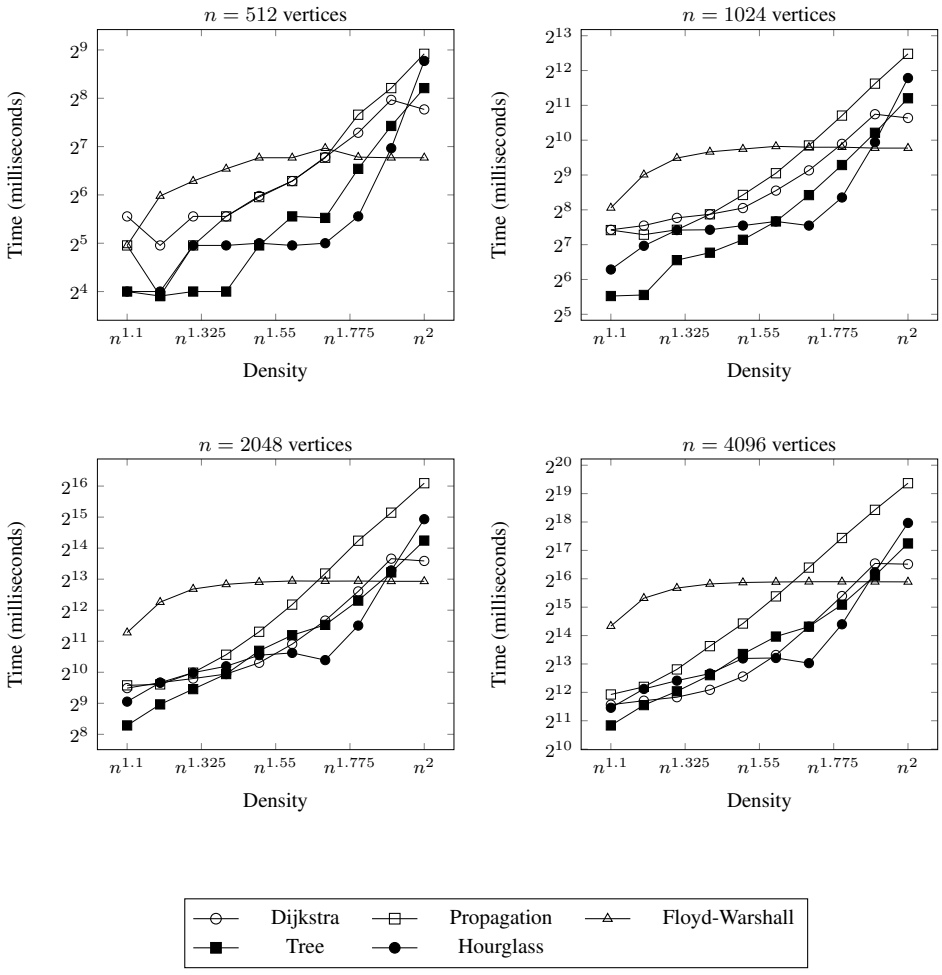

Figure 4: Experimental results on family (ii) – unweighted digraphs.

7 Conclusion

We theoretically analyzed the Tree algorithm which is a relatively simple modification of the Floyd-Warshall algorithm. The analysis gives its expected-case time complexity in the uniform model of $O(n^2 \log^2 n)$, which also explains the algorithm's good practical performance presented in Section 6. We also presented the Hourglass algorithm as a further improvement of the Tree algorithm, but it remains an open question whether its expected-case time complexity in the uniform model is $o(n^2 \log^2 n)$.

Next, since both the Tree and Hourglass algorithms allow negative arc weights, it would be interesting to analyze their expected-case running time complexity for a model that permits negative arcs such as the vertex potential model [5, 6].

Overall, the Tree algorithm is simple to implement and offers very good performance. The Hourglass algorithm has the potential to be even better but probably requires a more

complex implementation. It is also worthwhile to note that the space requirement of the Tree algorithm is not worse than the space requirement of any algorithm that reports all shortest paths. The Hourglass algorithm requires an additional matrix of size n^2.

ORCID iDs

Andrej Brodnik https://orcid.org/ 0000-0001-9773-0664
Rok Požar https://orcid.org/ 0000-0002-2037-9535

References

[1] L. Addario-Berry, N. Broutin and G. Lugosi, The longest minimum-weight path in a complete graph, *Comb. Probab. Comput.* **19** (2010), 1–19, doi:10.1017/s0963548309990204.

[2] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, Inc., USA, 1993.

[3] P. A. Bloniarz, A shortest-path algorithm with expected time $O(n^2 \log n \log^* n)$, *SIAM J. Comput.* **12** (1983), 588–600, doi:10.1137/0212039.

[4] A. Brodnik and M. Grgurovič, Solving all-pairs shortest path by single-source computations, *Discrete Appl. Math.* **231** (2017), 119–130, doi:10.1016/j.dam.2017.03.008.

[5] B. V. Cherkassky, A. V. Goldberg and T. Radzik, Shortest paths algorithms: theory and experimental evaluation, *Math. Programming* **73** (1996), 129–174, doi:10.1016/0025-5610(95)00021-6.

[6] C. Cooper, A. Frieze, K. Mehlhorn and V. Priebe, Average-case complexity of shortest-paths problems in the vertex-potential model, *Random Struct. Algorithms* **16** (2000), 33–46, doi:10.1002/(sici)1098-2418(200001)16:1⟨33::aid-rsa3⟩3.0.co;2-0.

[7] T. H. Cormen, C. Stein, R. L. Rivest and C. E. Leiserson, *Introduction to Algorithms*, McGraw-Hill Higher Education, 2nd edition, 2001.

[8] C. Demetrescu and G. F. Italiano, A new approach to dynamic all pairs shortest paths, *J. ACM* **51** (2004), 968–992, doi:10.1145/1039488.1039492.

[9] C. Demetrescu and G. F. Italiano, Experimental analysis of dynamic all pairs shortest path algorithms, *ACM Trans. Algorithms* **2** (2006), 578–601, doi:10.1145/1198513.1198519.

[10] E. W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik* **1** (1959), 269–271, doi:10.1007/bf01386390.

[11] R. W. Floyd, Algorithm 97: Shortest path, *Commun. ACM* **5** (1962), 345, doi:10.1145/367766.368168.

[12] T. Hagerup and C. Rüb, A guided tour of Chernoff bounds, *Inf. Process. Lett.* **33** (1990), 305–308, doi:10.1016/0020-0190(90)90214-I.

[13] Y. Han and T. Takaoka, An $O(n^3 \log\log n/\log^2 n)$ time algorithm for all pairs shortest paths, in: *Proceedings of the 13th Scandinavian Conference on Algorithm Theory*, Springer-Verlag, Berlin, Heidelberg, SWAT'12, 2012 pp. 131–141, doi:10.1007/978-3-642-31155-0_12.

[14] S. Janson, One, two and three times $\log n/n$ for paths in a complete graph with random weights, *Combin. Probab. Comput.* **8** (1999), 347–361, doi:10.1017/s0963548399003892.

[15] D. B. Johnson, Efficient algorithms for shortest paths in sparse networks, *J. ACM* **24** (1977), 1–13, doi:10.1145/321992.321993.

[16] D. Karger, D. Koller and S. J. Phillips, Finding the hidden path: time bounds for all-pairs shortest paths, *SIAM J. Comput.* **22** (1993), 1199–1217, doi:10.1137/0222071.

[17] K. Mehlhorn and V. Priebe, On the all-pairs shortest-path algorithm of Moffat and Takaoka, *Random Struct. Algorithms* **10** (1997), 205–220, doi:10.1002/(sici)1098-2418(199701/03)10:1/2⟨205::aid-rsa11⟩3.3.co;2-J.

[18] A. Moffat and T. Takaoka, An all pairs shortest path algorithm with expected time $O(n^2 \log n)$, *SIAM J. Comput.* **16** (1987), 1023–1031, doi:10.1137/0216065.

[19] Y. Peres, D. Sotnikov, B. Sudakov and U. Zwick, All-pairs shortest paths in $O(n^2)$ time with high probability, *J. ACM* **60** (2013), Article No. 26 (25 pages), doi:10.1145/2508028.2505988.

[20] S. Pettie, A new approach to all-pairs shortest paths on real-weighted graphs, *Theor. Comput. Sci.* **312** (2004), 47–74, doi:10.1016/s0304-3975(03)00402-x.

[21] S. Pettie, Towards a final analysis of pairing heaps, in: *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, IEEE Computer Society, Washington, DC, USA, 2005 pp. 174–183, doi:10.1109/sfcs.2005.75.

[22] S. Pettie, V. Ramachandran and S. Sridhar, Experimental evaluation of a new shortest path algorithm, in: *Proceedings of the 4th International Workshop on Algorithm Engineering and Experiments 2002 (ALENEX '02)*, Springer-Verlag, Berlin, Heidelberg, 2002 pp. 126–142, doi:10.1007/3-540-45643-0_10.

[23] M. Raab and A. Steger, "Balls into bins"—a simple and tight analysis, in: *Randomization and Approximation Techniques in Computer Science (Barcelona, 1998)*, Springer, Berlin, volume 1518 of *Lecture Notes in Computer Science*, pp. 159–170, 1998, doi:10.1007/3-540-49543-6_13.

[24] P. M. Spira, A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$, *SIAM J. Comput.* **2** (1973), 28–32, doi:10.1137/0202004.

[25] T. Takaoka and A. Moffat, An $O(n^2 \log n \log \log n)$ expected time algorithm for the all shortest distance problem, in: *Mathematical Foundations of Computer Science 1980*, Springer, Berlin, Heidelberg, volume 88 of *Lecture Notes in Computer Science*, 1980 pp. 643–655, doi:10.1007/bfb0022539, proceedings of the 9th Symposium Held in Rydzyna, Poland, September 1 – 5, 1980.

[26] S. Warshall, A theorem on boolean matrices, *J. ACM* **9** (1962), 11–12, doi:10.1145/321105.321107.