Optimization Letters
https://doi.org/10.1007/s11590-024-02113-0

ORIGINAL PAPER

# On efficient algorithms for bottleneck path problems with many sources

Kirill V. Kaymakov[1] · Dmitry S. Malyshev[2]

Received: 12 October 2023 / Accepted: 19 March 2024

**Abstract**
For given edge-capacitated connected graph and two its vertices $s$ and $t$, the bottleneck (or max min) path problem is to find the maximum value of path-minimum edge capacities among all paths, connecting $s$ and $t$. It can be generalized by finding the bottleneck values between $s$ and all possible $t$. These problems arise as subproblems in the known maximum flow problem, having applications in many real-life tasks. For any graph with $n$ vertices and $m$ edges, they can be solved in $O(m)$ and $O(t(m, n))$ times, respectively, where $t(m, n) = \min(m + n\log(n), m\alpha(m, n))$ and $\alpha(\cdot, \cdot)$ is the inverse Ackermann function. In this paper, we generalize of the bottleneck path problems by considering their versions with $k$ sources. For the first of them, where $k$ pairs of sources and targets are (offline or online) given, we present an $O((m + k)\log(n))$-time randomized and an $O(m + (n + k)\log(n))$-time deterministic algorithms for the offline and online versions, respectively. For the second one, where the bottleneck values are found between $k$ sources and all targets, we present an $O(t(m, n) + kn)$-time offline/online algorithm.

**Keywords** Design and analysis of algorithms · Computational complexity · Bottleneck path problem

The work of the author Malyshev D.S. was conducted within the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE).

✉ Dmitry S. Malyshev
dsmalyshev@rambler.ru

Kirill V. Kaymakov
kaymakov.kirill@huawei-partners.com

[1] Coleman Tech LLC, 40 Mira Avenue, Moscow, Russia 129090

[2] Laboratory of Algorithms and Technologies for Networks Analysis, National Research University Higher School of Economics, 136 Rodionova Str., Nizhny Novgorod, Russia 603093

Published online: 18 April 2024

## 1 Introduction

In this paper, we consider generalizations of the known *Bottleneck Path Problem*, abbreviated as the BPP, and *Single-Source Bottleneck Paths Problem*, abbreviated as the SSBPP. In both these problems, a simple, connected graph $G = (V, E)$ with $V = \{v_1, v_2, \dots, v_n\}$ and $E = \{e_1, e_2, \dots, e_m\}$ is given and, for every edge $e_i$, its capacity $c_i$ is also given. Additionally given a source vertex $s \in V$ and a target vertex $t \in V$, the BPP is to find the value $b(s,t) = \max_{P \in \mathcal{P}_{st}} \min_{e \in P} c(e)$, where $\mathcal{P}_{st}$ is the set of all paths between $s$ and $t$. The SSBPP asks to find $b(s, t)$, for a given source vertex $s \in V$ and all $t \in V$.

The BPP appears as a subproblem in the algorithm by Edmonds and Karp [12] for solving the Maximum Flow Problem and in an algorithm for the $k$-Splittable Flow Problem [3]. The Maximum Flow Problem serves as a (relaxation of a) mathematical model for many real-life problems, arising in electrical power transmission, airline scheduling, communication networks. The SSBPP arises as a subroutine, for example, in railway timetabling [19]. The BPP can be solved in $O(m)$ time by the Camerini's threshold approach, see Section 2. The SSBPP can be solved in $O(m + n\log(n))$ time or in $O(m\alpha(m,n))$ time by the Prim's algorithm or the Chazelle's algorithm, modified by breadth-first search, respectively, see Section 2. Descriptions of the original Camerini's, Chazelle's, and Prim's algorithms can be found in [7, 8, 20]. Duan, Lyu, and Xie presented an algorithm with the complexity $O(\sqrt{mn\log(n)\log\log(n)} + m\sqrt{\log(n)})$ for solving the SSBPP [10]. There are several papers, in which algorithms with computational complexity guaranties are presented for the directed versions of the BPP and SSBPP, see, for example, [9, 10, 17].

We generalize the BPP and SSBPP by considering statements with many sources as follows. Given a simple, connected, edge-capacitated graph and vertices $s_1, \dots s_k$, the *Multi-Pair Bottleneck Path Problem* (MPBPP, for short) is to find the values of $b(s_1, t_1), \dots, b(s_k, t_k)$, where $t_1, \dots, t_k$ are additionally given, and the *Multi-Source Bottleneck Paths Problem* (MSBPP, for short) is to find $b(s_i, t)$, for any $i$ and all $t \in V$. The version of the MSBPP, when $k = n$, is called the *All-Pairs Bottleneck Paths Problem*, abbreviated as the APBPP.

The BPP, SSBPP, MPBPP, MSBPP, APBPP are max min problems. Any algorithm, solving one of them, can be used to solve the corresponding min max problem, which is obtained by taking all capacities with the opposite sign. The MPBPP, MSBPP, and APBPP can be used as parts in algorithms for solving variations and generalizations of the Maximum Flow Problem.

Obviously, the MPBPP can be solved in $O(km)$ time by $k$ calls of a linear-time algorithm for the BPP. Similarly, the MSBPP can be solved in $O(kt(m, n))$ time. These approaches could be considered as naive or baseline solutions for the MPBPP and MSBPP. Several algorithms for solving the APBPP with upper complexity bounds are presented in [11, 21, 23]. In this paper, we propose $O((m + k)\log(n))$-time randomized and $O(m + (n + k)\log(n))$-time deterministic algorithms for the offline and online versions of the MPBPP, respectively, i.e., when $s_i$ and $t_i$ are given in advance or subsequently enter. We also propose an

$O(t(m,n)+kn)$-time algorithm for the offline/online MSBPP. For large $k$, our algorithms are better than the baseline solutions.

## 2 Baseline solutions

A *spanning tree* of a connected graph is any its tree subgraph, containing all vertices of the graph. The *minimum spanning tree problem*, abbreviated as the MSTP, is to find in a given edge-weighted graph a spanning tree with the minimum sum of weights of its edges. The *Bottleneck Spanning Tree Problem* (the BSTP, for short) is to find in a given edge-weighted graph a spanning tree, in which the maximum weight of its edges achieves the minimum value. The maximization version of the MSTP (respectively, the $\max\min$ version of the BSTP) can be solved with any algorithm for the MSTP (respectively, for the BSTP) by sign changing for all edge weights.

The BSTP can be solved in linear time on the quantity of edges by the Camerini's algorithm [7], but no linear-time algorithm for the MSTP is known. There are many classic algorithms for solving the MSTP, like the Boruvka's algorithm [6], Prim's algorithm [20], Kruskal's algorithm [18], Chazelle's algorithm [8], and others, but all of them are super-linear. To solve the BPP, the Camerini's threshold method can be applied as follows:

**Algorithm 1** Linear-time BPP algorithm

```
Input: A simple, edge-capacitated, connected graph $G = (V, E, c)$, source $s$ and target $t$
vertices.
Output: $b(s,t)$.
$b(s,t) \leftarrow 0$;
while $(|E(G)| > 1)$ do
    Find the median $c^*$ among capacities of edges;
    Delete from $G$ all edges $e$ with $c(e) > c^*$ to obtain a graph $G'$;
    if ($s$ is reachable from $t$ in $G'$) then
        $G \leftarrow G'$; $b(s,t) \leftarrow c^*$;
    end
    else
        Determine connected components of $G'$ and contract each of them in $G$;
    end
end
return $b(s,t)$;
```

The computational complexity analysis of Algorithm 1 can be done in a similar way to one presented in [7], giving the complexity bound $O(m)$. Therefore, the BPP can be solved in $O(m)$ time.

Optimal solutions of the MSTP are appeared to be useful for solving the SSBPP. Indeed, the following statement is true (it is known, no doubtely, but the authors did not find the corresponding reference):

**Statement 1** If $T$ is a minimum spanning tree of $G=(V,E,c)$, then, for any $s,t \in V$, the $st$-path in $T$ is an optimal solution of the $\min\max$ version of the BPP with the source $s$ and the target $t$.

***Proof*** Assume the contrary, i.e., for the maximum-capacity edge $\hat{e}$ on the $st$-path in $T$, we have $c(\hat{e}) > \hat{b}(s,t) = \min_{P \in \mathcal{P}_{st}} \max_{e \in P} c(e)$. Denote by $V_1$ and $V_2$ the vertex sets of the connected components of $T \setminus \{\hat{e}\}$, where $s \in V_1, t \in V_2$. For any edge $e' = \{a,b\} \in E(G)$ with $a \in V_1, b \in V_2$, we have $c(e') \geq c(\hat{e})$. Otherwise, a spanning tree $(T \setminus \{\hat{e}\}) \cup \{e'\}$ has a smaller weight, than $T$. Therefore, for any $st$-path $P$ in $G$, we have $\max_{e'' \in P} c(e'') \geq c(\hat{e})$. Hence, $\hat{b}(s,t) \geq c(\hat{e})$. We have a contradiction. □

The MSTP can be solved by the Prim's algorithm, which pseudocode is presented below:

**Algorithm 2** Prim's algorithm for the MSTP

```
Input: A simple, edge-weighted, connected graph $G = (V, E, c)$.
Output: A minimum spanning tree $T$ of $G$.

Add an arbitrary vertex to $T$;
while ($|E(T)| < n - 1$) do
    Among all edges $\{a, b\}$, where $a \in V(T), b \in V(G) \setminus V(T)$, find an edge
      $e^* = \{a^*, b^*\}$ with the minimum weight;
    Add $e^*$ and $b^*$ to $T$;
end
return $T$;
```

Using Fibonacci heaps [13], the Prim's algorithm can be implemented in $O(m + n\log(n))$ time. B. Chazelle invented the so-called soft heaps and applied them for solving the MSTP in $O(m\alpha(m,n))$ time [8]. Having a minimum spanning tree $T$ and a source $s$, breadth-first search in $T$, started at $s$, gives an optimal solution of the SSBPP in $O(n)$ time by Statement 1. Hence, the SSBPP can be solved in $O(t(m, n))$ time.

## 3 Our solutions for the MPBPP

### 3.1 The offline MPBPP

In this Subsection, we assume that all $(s_1, t_1), \dots, (s_k, t_k)$ are given offline. In other words, all input data are known in advance, before the algorithm starts running.

A disjoint-set data structure, abbreviated as a DJS, is a data structure that stores a partition of a set into disjoint subsets. It supports operations for adding new subsets, replacing two subsets by their union, and finding a representative member of a subset. A DJS is usually implemented as a disjoint-subset forest, allowing to perform the three basic operations in near-constant time, see [14, 16, 22]. More precisely, insertion and join can be performed in $O(1)$ time in the worst case, but search can be performed in amortized time, bounded from above by a value of the inverse Ackermann function.

DJSs play an important role in an efficient implementation of the Kruskal's algorithm for solving the MSTP or its maximization version. Namely, at each its step, a disjoint-set data structure keeps a (monotonically growing) forest, which always is a part of a minimum/maximum spanning tree, obtained after the last step. First, the Kruskal's algorithm sorts edges by their weights (non-decreasingly for the MSTP or non-increasingly for its maximization variant), next, it scans the sorted set and determines whether a current edge can be added to an optimal solution or not. This verification is based on the union and search operations with DJSs. It can be adopted to the MPBPP.

Our DJS-based solution for the MPBPP is presented in Algorithm 3. It uses the following notations:

- *ind*[*v*] is the set of those *i*, such that $s_i$ or $t_i$ belongs to the subset of a DJS, containing *v*;
- *Find*(*v*) returns a canonical element of the subset, containing *v*;
- *Join*(*x*, *y*) replaces the two subsets, having canonical elements *x* and *y*, by their union, arranges *x* as the canonical element of the new subset, and swaps the arguments of *ind*[*x*] and *ind*[*y*], if $\sharp ind[y] > \sharp ind[x]$;
- *answer*[*i*] is $b(s_i, t_i)$, for any *i*;
- $A \otimes B$ means the symmetric difference of sets *A* and *B*.

**Algorithm 3** Algorithm for the offline MPBPP

```
Input: A simple, edge-capacitated, connected graph $G = (V, E, c)$, pairs
$(s_1, t_1), \dots, (s_k, t_k)$ of source and target vertices.
Output: $b(s_1, t_1), \dots, b(s_k, t_k)$.
Initialize an $n$-element array $ind$ with $n$ empty sets as its elements;
Initialize a disjoint-set data structure with $n$ singletons, each corresponding to a graph's
  vertex;
for $(i \leftarrow 1;\ i \leq k;\ i \leftarrow i + 1)$ do
    $answer[i] \leftarrow -\infty$;
    $ind[s_i] \leftarrow ind[s_i] \cup \{i\}$;
    $ind[t_i] \leftarrow ind[t_i] \cup \{i\}$;
end
Non-increasingly sort $E$ by edge weights;
forall $(e = \{a, b\} \in E)$ do
    $x \leftarrow Find(a)$;
    $y \leftarrow Find(b)$;
    if $(x \neq y)$ then
        $Join(x, y)$;
        forall ($z$ in $ind[x] \cap ind[y]$) do
            $answer[z] \leftarrow c(e)$;
        end
        $ind[x] \leftarrow ind[x] \otimes ind[y]$;
        delete $ind[y]$;
    end
end
return $answer[]$;
```

The correctness of Algorithm 3 is based on Statement 1. Indeed, according to this statement, the value of $b(s_i, t_i)$ is determined at the first moment, when a path arises between $s_i$ and $t_i$ in the partial optimal solution. In other words, *e* connects vertices $a$ and $b$ from distinct connected components, $a, s_i$ belong to one of them and $b, t_i$ belong to another. Hence, $i \in ind[x]$ and $i \in ind[y]$. At this moment, $answer[i] = c(e) = b(s_i, t_i)$ and the same is true for the whole $ind[x] \cap ind[y]$. The assignment $ind[x] \leftarrow ind[x] \otimes ind[y]$ guarantees that the subset $T_x$ (or the tree) with the canonical element $x$ from the partial optimal solution keeps only those $s_i$ or $t_i$ that $s_i \in V(T_x), t_i \notin V(T_x)$ or $s_i \notin V(T_x), t_i \in V(T_x)$.

Let us estimate the computational complexity of Algorithm 3, assuming that all $ind[i]$ are stored by hash-sets. It is known that insertion of an element into a hash-set and deletion of an element from a hash-set have constant randomized times [2]. The computational complexity of all algorithm's actions, non-connected to work with $inds$, can be estimated by $O(mlog(n))$. Computing $ind[x] \cap ind[y]$ and the assignment $ind[x] \leftarrow ind[x] \otimes ind[y]$ are performed in $O(\min(\sharp ind[x], \sharp ind[y]))$ expected time by iterating on a minimum-size of these two sets. Let us show that the expected running time with all $inds$ is $O(k\log(n) + n)$.

Work of the second cycle in Algorithm 3 can be represented by a binary tree $Tr$, all whose vertices correspond to subsets of a DJS, assuming that any such a subset $S$ is equipped with its size $n_S$ and the number $k_S$ of $i$ that $s_i \in S$ or $t_i \in S$. Clearly, it holds that $\sharp ind[x_S] \le k_S$, where $x_S$ is the canonical element of $S$. By $E(p, q)$ we denote the expected maximum quantity of atomic operations, needed for obtaining subsets $S$ with $n_S = p$ and $k_S = q$ overall input data. As the root of $Tr$ is obtained by joining some two subsets, we have

$$E(n,k) \le E(t,k-l) + E(n-t,l) + C \cdot l, \text{ for some } C > 0, 1 \le t \le n-1, 0 \le l \le \frac{k}{2}.$$

By the mathematical induction method on $n$ and $k$ we will show that $E(n,k) = O(k\log(n) + n)$. Its basis are the statements $E(n,1) = O(n)$ and $E(1,k) = O(k)$, which are clearly true, for any $n$ and $k$. The induction step assumes that $E(n',k') \le \hat{C}(k'\log_2(n') + n')$, for some $\hat{C} > C$ and all $n \ge n' \ge 2, k' \le k, (n',k') \ne (n,k)$. Hence, we have

$$\begin{aligned} E(n,k) - \hat{C}(k\log_2(n) + n) &\le \hat{C}((k-l)\log_2(t) + l\log_2(n-t) + l - k\log_2(n)) \\ &= \hat{C}((k-l)\log_2(\frac{t}{n}) + l\log_2(\frac{2(n-t)}{n})) \le 0, \end{aligned}$$

if $t \ge \frac{n}{2}$. If $t < \frac{n}{2}$, then, by $l \le \frac{k}{2}$, we have

$$\begin{aligned} E(n,k) - \hat{C}(k\log_2(n) + n) &\le \hat{C}(k-l)(\log_2(\frac{t}{n}) + \log_2(\frac{2(n-t)}{n})) \\ &= \hat{C}(k-l)\log_2(\frac{2t(n-t)}{n^2}) < 0. \end{aligned}$$

So, we proved that $E(n,k) = O(k\log(n) + n)$. By this fact and the previous reasonings, the total expected computational complexity of Algorithm 3 is $O((m+k)\log(n))$.

## 3.2 The online MPBPP

The input data for Algorithm 3 are assumed to be given offline. For the online MPBPP, i.e., when $(s_i, t_i)$ are entering in the online regime, it is also possible to design an algorithm with the same complexity bound $O((m + k)\log(n))$. It is based on modifications of some efficient algorithms for the so-called *lowest common ancestor problem*, abbreviated as the LCAP and introduced by A. Aho, J. Hopcroft, J. Ullman in [1].

For a given rooted tree (or a directed acyclic graph) $G$ and its vertices $v$ and $u$, the LCA of $v$ and $u$ is the deepest node, for which both $v$ and $u$ are descendants, assuming that each vertex is a descendent of itself. The LCAP is to find the LCA for an offline given $G$ and its subsequently given queries $(v_1, u_1), \dots, (v_k, u_k)$. We will assume that $G$ is a rooted tree.

Usually, algorithms for solving the LCAP firstly preprocess $G$ in linear time on its vertex number and then return the LCAs in constant time per query. D. Harel and R. Tarjan invented in [15] the first algorithm of this type, based on the heavy-light decomposition technique, but their solution is difficult to understand and implement. O. Berkman and U. Vishkin discovered in [5] a new way to solve the LCAP, also consumpting linear-time preprocessing time with constant query time. Their approach uses the depth-first search from the root of a given tree and indexing vertices in accordance to it, a reduction of the LCAP to the range minimum query problem (the RMQP, for short) within some subinterval in the sequence of indices. This RMQP is solved, using several techniques, one of them is precomputing the answers on large intervals that have sizes that are powers of two.

The mentioned approach of O. Berkman and U. Vishkin was simplified by M. Bender and M. Farach-Colton in [4]. Their algorithm is known as the jump-pointers algorithm. It uses a preprocessing step, working in $O(n\log(D))$ time, where $D$ is the tree diameter. Namely, for each vertex $x$, all liftings from $x$ are organized to vertices that are higher than $x$ on powers of two. We modify the preprocessing step by splitting paths from current vertices to the tree root into segments and computing minimum edge capacities in them.

In Algorithm 4, $T$ is an edge-capacitated tree, stored by an adjacency list and rooted at an arbitrarily chosen vertex $r$. It also uses the following notations:

- $x$ is a current vertex;
- $d[x]$ is the depth of $x$ with respect to $r$;
- $p[x][0 \dots \lceil\log_2(D)\rceil]$ and $v[x][0 \dots \lceil\log_2(D)\rceil]$ are arrays of jump-pointers and minimum edge capacities in the corresponding segments.

**Algorithm 4** Modified jump-pointers preprocession

```
Input: An edge-capacitated tree.
Output: Arrays $d[], p[][], v[][]$.
Call a breadth-first search, started at $r$, computing $d[]$ and $D$, orienting all edges of $T$ to
  its root;
forall (arcs $y_x x$ in the breadth-first order) do
    BuildLCAIndex($x$);
end
return $d[], p[][], v[][]$;
Function BuildLCAIndex($x$):
    $p[x][0] \leftarrow y_x$;
    $v[x][0] \leftarrow c(y_x x)$;
    for ($i \leftarrow 1;\ i < \lceil \log_2(D) \rceil;\ i \leftarrow i+1$) do
        $pp \leftarrow p[x][i-1]$;
        $p[x][i] \leftarrow p[pp][i-1]$;
        $v[x][i] \leftarrow \min(v[x][i-1], v[pp][i-1])$;
    end
    return $p[x][]$ and $v[x][]$;
```

Clearly that the computational complexity of Algorithm 4 is $O(n\log(D))$. In Algorithm 5 below, for each query, two vertices are aligned, so that they are located at the same depth. Next, for any $j$, after $j$ jumps the vertices $s$ and $t$ will have the depth $d[s] - 2^j = d[t] - 2^j$. Hence, if $d[p[s][j]] \leq d[l]$, where $l$ is the LCA of $s$ and $t$, then $p[s][j] = p[t][j]$.

**Algorithm 5** Computation of the bottleneck value

```
Input: The modified jump-pointers preprocession by Algorithm 4 in the form of arrays
$d[], p[][], v[][]$ and vertices $s, t$.
Output: $anwer = b(s,t)$.
Function FindMin($s, t$):
    $answer \leftarrow +\infty$;
    if ($d[s] \neq d[t]$) then
        Align($s, t, answer$);
    end
    if ($s = t$) then
        return $answer$;
    end
    Up($s, t, answer$);
    return $\min(answer, v[s][0], v[t][0])$;

Function Align($s, t, answer$):
    if ($d(s) > d(t)$) then
        swap $s$ and $t$;
    end
    for ($i = \lceil \log_2(D) \rceil - 1; i \geq 0; i \leftarrow i-1$) do
        if ($d[s] \leq d[p[t][i]]$ then
            $answer \leftarrow \min(answer, v[t][i])$;
            $t \leftarrow p[t][0]$;
        end
    end

Function Up($s, t, answer$):
    for ($i = \lceil \log_2(D) \rceil - 1; i \geq 0; j \leftarrow i-1$) do
        if ($p[s][i] \neq p[t][i]$) then
            $answer \leftarrow \min(answer, v[s][i], v[t][i])$;
            $s \leftarrow p[s][i]; t \leftarrow p[t][i]$;
        end
    end
```

It is easy to see that the computational complexity of Algorithm 5 is $O(\log(D))$. So, by Statement 1, to solve the MPBPP, it is enough to find the maximum spanning tree for $(V, E, c)$, preprocess it by Algorithm 4, and call Algorithm 5, for every pairs of queries, see Algorithm 6.

**Algorithm 6** Algorithm for the online MPBPP

```
Input: A simple, edge-capacitated, connected graph $G = (V, E, c)$, online entering pairs
$(s_1, t_1), \ldots, (s_k, t_k)$ of source and target vertices.
Output: $b(s_1, t_1), \ldots, b(s_k, t_k)$.

Find a maximum spanning tree of $G$ by the Prim's algorithm with a Fibonacci heap;
Apply a modified jump-pointers algorithm by Algorithm 4;
For any pair $(s_i, t_i)$, call FindMin $(s_i, t_i)$ by Algorithm 5;
return $b(s_1, t_1), \ldots, b(s_k, t_k)$;
```

The computational complexity of Algorithm 6 is $O(m + (n + k) \log(n))$.

## 4 Our solution for the MSBPP

Our algorithm for the offline/online MSBPP is simple, see Algorithm 7. Firstly, find the maximum spanning tree for a given graph. Next, call breadth-first search, started at every source, to determine the bottleneck values. Its correctness is based on Statement 1 and the computational complexity is $O(t(m, n) + kn)$.

**Algorithm 7** Algorithm for the offline/online MSBPP

```
Input: A simple, edge-capacitated, connected graph $G = (V, E, c)$, offline or online
entering sources $s_1, \ldots, s_k$.
Output: $b(s_1, t), \ldots, b(s_k, t)$, for all $t$.

Find a maximum spanning tree of $G$ by the Prim's algorithm with a Fibonacci heap or
  by the Chazelle's algorithm;
For any $s_i$, call breadth-first search to determine $b(s_i, t)$, for all $t$;
return $b(s_1, t), \ldots, b(s_k, t)$, for all $t$;
```

## 5 Conclusions and future work

In this paper, we considered bottleneck path problems with many sources. Previously, only single-source such problems were considered and algorithms for them were designed. We presented several efficient algorithms for multi-sources bottleneck path problems in this paper. When the sources quantity is large, our solutions work faster than baseline algorithms with sequential calling single-source algorithms. Developing new algorithms and improving the existing ones is a challenging research problem for future research.

**Author contributions** All authors contributed to the study conception and design. Material preparation was performed by Kirill Kaymakov and Dmitriy Malyshev. The first draft of the manuscript was written by Kirill Kaymakov and Dmitriy Malyshev and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

**Funding** The work of the author Malyshev D.S. was conducted within the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE).

**Data availability** Our paper has no associated data.

## References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: On finding lowest common ancestors in trees. In: Aho A.V. et al. (eds.) Proceedings of the 5th Annual ACM Symposium on Theory of Computing, ACM, pp. 253–265 (1973)
2. Aumüller, M., Dietzfelbinger, M., Woelfel, P.: Explicit and efficient hash families suffice for cuckoo hashing with a stash. Algorithmica **70**, 428–456 (2014)
3. Baier, G., Köhler, E., Skutella, M.: On the $k$-splittable flow problem. In: Möhring, R., Raman, R. (eds.). Proceedings of European Symposium on Algorithms, pp. 101–113, Springer (2002)
4. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. **321**(1), 5–12 (2004)
5. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. **22**(2), 221–242 (1993)
6. Boruvka, O.: About a certain minimal problem. Proc. Morav. Soc. Nat Sci. **3**(3), 37–58 (1926)
7. Camerini, P.M.: The min–max spanning tree problem and some extensions. Inf. Process. Lett. **7**(1), 10–14 (1978)
8. Chazelle, B.: A minimum spanning tree algorithm with inverse-ackermann type complexity. J. ACM **47**(6), 1028–1047 (2000)
9. Chechik, S. et al.: Bottleneck paths and trees and deterministic graphical games. In: Olliger, N., Vollmer, H. (ed.) Proceedings of the 33rd Symposium on Theoretical Aspects of Computer Science, Dagstuhl: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 27:1-27:13 (2016)
10. Duan, R., Lyu, K., Xie, Y.: Single-source bottleneck path algorithm faster than sorting for sparse graphs. In: Chatzigiannakis, I. et al. (eds.) Proceedings of the 45th International Colloquium on Automata, Languages, and Programming, Dagstuhl: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 43:1-43:14 (2018)
11. Duan, R., Pettie, S.: Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. In: Johnson, D., Fiege, U. (eds.) Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, pp. 384–391 (2009)
12. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM **19**(2), 264–284 (1972)
13. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3), 596–615 (1987)
14. Galler, B.A., Fischer, M.J.: An improved equivalence algorithm. Commun. ACM **7**(5), 301–303 (1964)
15. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13**(2), 338–355 (1984)
16. Hopcroft, J.E., Ullman, J.D.: Set merging algorithms. SIAM J. Comput. **2**(4), 294–303 (1973)
17. Kaibel, V., Peinhardt, M. On the bottleneck shortest path problem. Tech. rep. 06-22., Takustr. 7, 14195 Berlin: ZIB (2006)
18. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. Am. Math. Soc. **7**(1), 48–50 (1956)
19. Ljunggren, L., et al.: Railway timetabling: a maximum bottleneck path algorithm for finding an additional train path. Public Transp. **13**, 597–623 (2021)

20. Prim, R.C.: Shortest connection networks and some generalizations. Bell Syst. Tech. J. **36**(6), 1389–1401 (1957)
21. Shinn, T.-W., Takaoka, T.: Variations on the bottleneck paths problem. Theor. Comput. Sci. **575**, 10–16 (2015)
22. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2), 245–281 (1984)
23. Vassilevska, V., Williams, R., Yuster, R.: All-pairs bottleneck paths for general graphs in truly sub-cubic time. In: Johnson, D., Fiege, U. (eds.) Proceedings of the 39th Annual ACM Symposium on Theory of Computing, ACM, pp. 585–589 (2007)