Freie Universität Berlin
Fachbereich Informatik

Bottleneck Problems in Graphs

Linus Buddrus (Matrikelnummer: 5375100)

Contents

1 Introduction

Bottleneck problems are optimization problems on directed or undirected graphs with edge weights. They are related to problems that deal with finding structures that minimize the sum of edge weights, such as the well studied Shortest Path (SP) and Minimum Spanning Tree (MST) problems. Bottleneck problems deal with finding structures that maximize the minimum edge weight. The minimum edge weight is also called the limiting capacity. One can imagine a network where the edges represent connections between points and the edge weights represent the bandwidth of those connections. Our goal would be two find a path between a start and end point such that the network flow through that path is maximum, meaning the minimum bandwidth or bottleneck of that path is maximum.

Bottleneck problems have important applications in fields that deal with network routing such as logistics and computer networking, and others like digital compositing. The Bottleneck Path problem appears as a subproblem in the algorithm by Edmonds and Karp [7] for solving the Maximum Flow problem and in the k-splittable flow algorithm [1].

We concentrate on three bottleneck problems, namely the Bottleneck Path (BP) problem, the Bottleneck Spanning Tree (BST) problem and the Single-Source Bottleneck Path (SSBP) problem. We explain several algorithms that solve these problems and analyze their runtime complexity. These algorithms are presented in order of increasing complexity. In the first few sections we look at algorithms that are relatively simple but have a comparatively worse runtime complexity. In the later sections we look at algorithms that are more complex and improve on the runtime complexity of the algorithms discussed in the preceding sections. We focus especially on the algorithms by Chechik, Kaplan, Thorup, Zamir, and Zwick 2016 [3] and Duan, Lyu, and Xie 2018 [5], that have been discovered within the last couple of years and give the best running time for their problem currently known.

The algorithms are presented in such a way that the relationship between them should become apparent. In particular one should notice how each subsequent algorithm takes ideas from the previous algorithms and builds on them or modifies them in order to achieve a better runtime bound. These connections are also pointed out in the text. The descriptions for these algorithms in the original sources are given here in shortened form to only contain the main ideas. Some additional explanations are added and some arguments are made more explicit to help better understanding. For each algorithm a pseudocode is presented. In some cases the pseudocode is adopted from the original sources with little modification. This is true for Algorithm 6 in Section 7.1 and Algorithms 8, 9 and 10 in Section 8. Otherwise the pseudocode is provided based on the description of the algorithms given in the sources. For Algorithm 2 in Section 4 and Algorithm 3 in Section 5 no original sources exist. Algorithms 2 and 3 are however mentioned in the literature and brief descriptions are given for example in [3]. Algorithm 2 in Section 4 is explicitly stated in [10], which gives a good discussion on the topic. For Algorithm 4 and Algorithm 5 in Section 6 descriptions are given in [3] and [9]. The description given here for Algorithm 4 follows [9] while the description for Algorithm 5 follows [3]. This is done simply for ease of understanding.

The algorithms discussed work under the comparison-model, meaning the only oper-

ations allowed on edge weights are pairwise comparisons. Faster algorithm exist for the word-RAM model. These faster algorithm and the implications of the machine model used on algorithms for bottleneck problems are not a topic of this thesis.

2 Preliminaries

We consider a directed or undirected graph $G = (V, E)$ with weighted edges, where V denotes the set of nodes and E the set of edges. Let $n = |V|$ and $m = |E|$.

2.1 General Definitions

Definition 1 (Path)**.** *Let $s, t \in V$ be two distinct nodes. A path p from s to t is a sequence of edges $p = \langle e_1, e_2, \ldots, e_\ell \rangle$, where ℓ is the length of p, such that e_i and e_{i+1} are adjacent for $i = 1, 2, \ldots, \ell - 1$ and e_ℓ is incident to s and e_ℓ is incident to t. In a directed graph we require that the edges of a path are oriented in the same direction.*

Definition 2 (Spanning Tree)**.** *Given an undirected graph $G = (V, E)$, a spanning tree $T = (V', E')$ of G is a tree with $V' = V$ and $E' \subseteq E$. Given a directed graph $G = (V, E)$ and a root node $r \in V$ we call $T = (V', E')$ a spanning tree of G if $V' = V$, $E' \subseteq E$ and there is exactly one directed path from r to every other node $v \in V' \setminus \{r\}$.*

The definition for spanning tree in the directed case in Definition 2 we would normally associate with a spanning r-arborescence. However the term spanning tree is used in the literature both in the undirected and in the directed case.

Definition 3 (connected)**.** *We call an undirected graph G connected if for every pair $u, v \in V$ there is a path from u to v.*

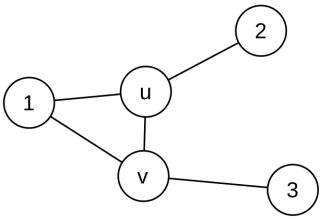
Definition 4 (weakly-connected)**.** *We call a directed graph G weakly-connected if changing every directed edge in G to be undirected results in a connected graph.*

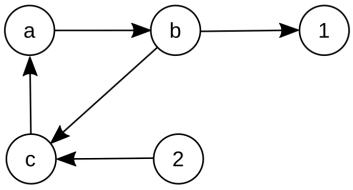
Definition 5 (strongly-connected)**.** *We call a directed graph G strongly-connected if for every pair $u, v \in V$ there is a directed path from u to v and a directed path from v to u.*

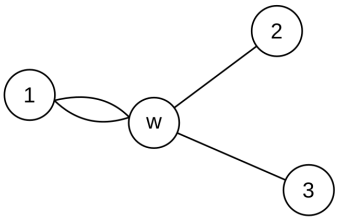
A *connected* (*weakly-connected*, *strongly-connected*) component is defined to be a maximal *connected* (*weakly-connected*, *strongly-connected*) subgraph G' of G, meaning that there is no node that could be added to G' such that G' is still *connected* (*weakly-connected*, *strongly-connected*).

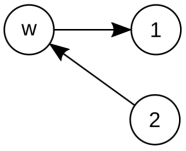
Edge contraction is an operation that joins two connected nodes $u, v \in V$ by removing the connecting edge (u, v) and nodes u and v from G and introducing a new node w that represents the joined node. The result of this operation is a new graph $G' = (V', E')$ where $V' = V \setminus \{u, v\} \cup \{w\}$ and E' consists of all edges in E that are not adjacent to (u, v) and edges (r, w) or (w, r) if there is an edge in E that is incident to node $r \in V' \setminus \{w\}$ and nodes u or v.
Contracting a connected (weakly-connected, strongly-connected) component involves contracting the edges that are connecting the nodes of the component in any sequence until only one node w remains. If there are multiple edges that are incident to nodes $r \in V' \setminus \{w\}$ and w in G' as a result, for our purposes we keep only one of those edges by either choosing the edge with the largest or smallest edge weight.

Figure 1: Examples of edge contraction









(a) Contracting a single edge

(b) Contracting a strongly-connected component

2.2 Bottleneck Problems

We call problems that deal with finding structures that minimize the sum of edge weights *MinSum* problems and problems that deal with finding structures that maximize the minimum edge weight *MaxMin* problems. Note that Bottleneck problems as described in the introduction are identical to MaxMin problems. Analogously to MaxMin we define *MinMax* problems to mean problems that deal with finding structures that minimize the maximum edge weight.

The MaxMin problem is equivalent to the MinMax problem. Given an algorithm that solves the MaxMin problem we can solve MinMax by negating the edge weights of the graph and then running the algorithm for MaxMin. One could alternatively define Bottleneck problems to mean MinMax problems. The algorithms we are looking at use both of these definitions. Be aware that a Bottleneck problem can refer to the MaxMin or the MinMax problem depending on which algorithm is being discussed.

As a first example we give a formal definition of the Bottleneck Path problem and the Shortest Path problem to see how they differ from one another. It should become clear how Bottleneck problems differ in general from MinSum problems such as SP and MST.

In the Bottleneck Path problem we are given a directed or undirected graph $G = (V, E)$, a weight function on the edges of that graph $w : E \to \mathbb{R}$ and a source and target node $s, t \in V$. Let $P = \{p_1, p_2, \ldots, p_k\}$ be the set of all paths between the nodes s and t in G. Let $E(p_i) = \{e_1, e_2, \ldots, e_{\ell_i}\}$ be the set of edges associated with path p_i, where ℓ_i is the length of p_i for all $i = 1, 2, \ldots, k$. Our goal is to find a path $p_i \in P$, such that the minimum edge weight of p_i is greater or equal to the minimum edge weight of every other path $p \in P \setminus \{p_i\}$. More formally we are trying to maximize the objective function $MinWeight : P \to \mathbb{R}$, where $MinWeight$ is defined as $MinWeight(p) = \min_{e \in E(p)} w(e)$. This means finding the input $p_i \in P$ such that $MinWeight(p_i) \geq MinWeight(p)$ for every $p \in P \setminus \{p_i\}$.

One thing to note is that the answer to this problem can be given in the form of a bottleneck path p_b or in the form of the bottleneck weight b, where b is the result of applying $MinWeight$ to the bottleneck path p_b. Given the bottleneck weight b we can find a bottleneck path in time $O(m)$ by deleting all edges e with $w(e) < b$ from G and using Depth First Search (DFS) or Breadth First Search (BFS) on the resulting graph G' to find a path between nodes s and t. Given a bottleneck path p_b the bottleneck weight b can be found by simply taking the minimum of all the edge weights in p_b. The algorithms discussed return the bottleneck weight b as their output. An additional $O(m)$ time cost does not effect the running time of any any of our algorithms. We write $b(s,t)$ to denote the bottleneck weight for all paths going from some source node s to a target node t. For simplicity we assume that the edges of a given graph have distinct weights. This restriction can be easily lifted. If the edge weights are not distinct, we can arbitrarily number all the edges of the graph with $\ell : E \to \{1, 2, \ldots, m\}$ and use ℓ to break ties in w, where w is the weight function $w : E \to \mathbb{R}$ on the edges of the graph.

The Shortest Path problem is a very well-known graph optimization problem with many theoretical and practical applications. It can be defined similarly to the Bottleneck Path Problem. The difference lies in the objective function being optimized. Again we are given a directed or undirected graph $G = (V, E)$, a weight function on the edges of that graph $w : E \to \mathbb{R}$ and a source and target node $s, t \in V$. Let $P = \{p_1, p_2, \ldots, p_k\}$ be the set of all paths between the nodes s and t in G. Let $E(p_i) = \{e_1, e_2, \ldots, e_{\ell_i}\}$ be the set of edges associated with path p_i, where ℓ_i is the length of p_i for all $i = 1, 2, \ldots, k$. Here our goal is to find a path $p_i \in P$, such that the sum of edge weights of p_i is less or equal to the sum of edge weights of every other path $p \in P \setminus \{p_i\}$. We are trying to minimize the objective function $SumWeight : P \to \mathbb{R}$, where $SumWeight$ is defined as $SumWeight(p) = \sum_{e \in E(p)} w(e)$. This means finding the input $p_i \in P$ such that $SumWeight(p_i) \leq SumWeight(p)$ for every $p \in P \setminus \{p_i\}$.

The Bottleneck Spanning Tree problem and Single Source Bottleneck Path problem are closely related to the Bottleneck Path problem. As with the Bottleneck Path problem the Bottleneck Spanning Tree problem and Single Source Bottleneck Path problem are defined on directed and undirected graphs. In the Bottleneck Spanning Tree problem we are asked to find a spanning tree of a given graph G such that the minimum edge weight of that spanning tree is maximum among all possible spanning trees of G. In the directed version of this problem we are additionally given a root node s and expected to find a directed spanning tree rooted at s as defined in Definition 2. The formal definition given for the Bottleneck Path problem can be easily adapted to the Bottleneck Spanning Tree problem. Instead of considering the set of all paths between two nodes s and t we consider the set of all spanning trees and define $MinWeight(T) = \min_{e \in E(T)} w(e)$ where $E(T)$ is the set of edges associated with spanning tree T. As with the Bottleneck Path problem it suffices to find the bottleneck weight b. Given b we can delete all edges with weight smaller than b and choose a spanning tree T in the resulting graph G'. Any spanning tree T in G' is a bottleneck spanning tree of G. This can be done in linear time using BFS or DFS. We write $b(G)$ in the undirected case and $b(G, s)$ in the directed case to denote the bottleneck weight for all spanning trees of G, where s is a given root node. Whereas in the Bottleneck Path problem we try to find the bottleneck weight $b(s,t)$ for a single target node t given a source node s in the Single Source Bottleneck Path problem we try to find the bottleneck weight $b(s,v)$ for all nodes v in G given

a source node s.

An overview of the problem definitions and algorithms for each problem is given here.

Problem 1: BP in undirected graphs

Input: Undirected graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$ and source and target nodes $s, t \in V$.
Output: Bottleneck weight $b(s, t)$.

Problem 2: BST in undirected graphs

Input: Undirected graph $G = (V, E)$ and weight function $w : E \to \mathbb{R}$.
Output: Bottleneck weight $b(G)$.

Problem 3: SSBP in undirected graphs

Input: Undirected graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$ and source node $s \in V$.
Output: Bottleneck weight $b(s, v)$ for all $v \in V$.

Problem 4: BP in directed graphs

Input: Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$ and source and target nodes $s, t \in V$.
Output: Bottleneck weight $b(s, t)$.

Problem 5: BST in directed graphs

Input: Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$ and root node $s \in V$.
Output: Bottleneck weight $b(G, s)$.

Problem 6: SSBP in directed graphs

Input: Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$ and source node $s \in V$.
Output: Bottleneck weight $b(s, v)$ for all $v \in V$.

	Undirected	**Directed**
BP	$O(m)$ by Algorithm 2 $O(m + n \log n)$ by Algorithm 1	$O(m\beta(m, n))$ by Algorithm 7 $O(m \log^* n)$ by Algorithm 5 $O(m \log n)$ by Algorithm 3 $O(m + n \log n)$ by Algorithm 1
BST	$O(m)$ by Algorithm 2 $O(m + n \log n)$ by Algorithm 1	$O(m\beta(m, n))$ by Algorithm 7 $O(m \log^* n)$ by Algorithm 5 $O(m \log n)$ by Algorithm 3 $O(m + n \log n)$ by Algorithm 1
SSBP	$O(m)$ randomized by solving MST $O(m + n \log n)$ by Algorithm 1	$O(\sqrt{nm \log n \log \log n} + m\sqrt{\log n})$ by Algorithm 10 $O(m + n \log n)$ by Algorithm 1

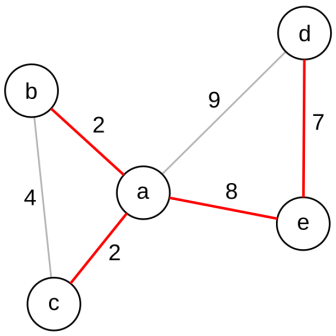
2.3 Observations

Considering the similarities between Bottleneck problems and MinSum problems it may not be surprising that there exist some relationships between solutions for both sets of problems.

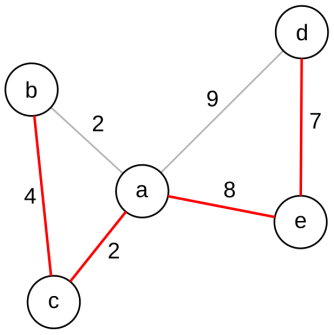
Given an undirected graph G finding the minimum spanning tree T of G gives us a solution to the BP, SSBP and BST problems. In undirected graphs a minimum spanning tree is also a bottleneck spanning tree. This is true because of the cut property of minimum spanning trees. A cut C is defined as a partition of the nodes of a graph $G = (V, E)$ into two subsets S and $V \setminus S$. The cut-set of a cut C is the set $\{(u, v) \in E \mid u \in S, v \in V \setminus S\}$ of edges that cross that cut. The cut property says that given any cut C of G the edge with minimum weight in the cut-set of C must be included in every minimum spanning tree of G. Let T be a minimum spanning tree of graph G. Suppose that T is not a bottleneck spanning tree. T must contain an edge $e = (u, v)$ whose weight is larger than the bottleneck weight $b(G)$ of a bottleneck spanning tree of G. If we remove $e = (u, v)$ from T, we get a graph that consists of two connected components such that u and v belong to different components. Let V_1 be the component that contains u and V_2 be the component that contains v. Now consider the cut C consisting of V_1 and V_2. Because T is a minimum spanning tree it must contain the edge with minimum weight in the cut-set of C. Therefore e must be the edge with minimum weight in the cut-set. However since the bottleneck spanning tree must include an edge f that crosses C as well and $w(f) \leq b(G) < w(e)$ it follows that e cannot be the edge with minimum weight in the cut-set.

Although every minimum spanning tree is a bottleneck spanning tree the reverse is not true. There can be two or more bottleneck spanning trees whose sum of edge weights differ from one another. In Figure 2a we see that the spanning tree outlined in red is a MST and BST. The spanning tree of the same graph outlined in Figure 2b is a BST but not a MST. For a spanning tree to be a bottleneck spanning tree we only require that the weights of its edges are less than or equal to $b(G)$. For our example in Figure 2 we have $b(G) = 8$.

Figure 2: Example showing that a BST must not be a MST



(a) MST and BST



(b) BST

Given an undirected graph $G = (V, E)$ for any nodes $s, t \in V$ a bottleneck path between s and t can be found in a minimum spanning tree T of G. This again

follows from the cut property of minimum spanning trees. Suppose that there is some edge $e = (u, v)$ along the path from s to t in T for which $w(e) > b(s, t)$. Let V_1 and V_2 be the connected components after removing e from T. We expect e to be the edge with minimum weight that crosses the cut C consisting of V_1 and V_2. However since a bottleneck path between s and t has to contain an edge f that crosses C as well and $w(f) \leq b(s, t) < w(e)$ it follows that e cannot be the edge with minimum weight in the cut-set of C.

We see that in undirected graphs by solving the MST problem we obtain a solution to the SSBP, BP and BST problems. The current best time bound for solving MST include expected $O(m)$ given by a randomized algorithm by Karget et al. [11]. There exists a deterministic $O(m\alpha(m, n))$ time algorithm for solving MST due to Chazelle [2] but no deterministic linear time algorithm for MST is known.

Note that the cut property does not hold for minimum spanning trees in directed graphs. In the directed case a solution to the MST problem does not give us a solution to the SSBP, BP or BST problems.

Given a directed or undirected graph $G = (V, E)$ and a source node $s \in V$ the bottleneck paths found by an algorithm for SSBP form a bottleneck spanning tree. Looking at the directed case we must have $b(s, v) \leq b(G, s)$ for all $v \in V$. This is easily shown. Suppose that there is some node $t \in V$ such that $b(s, t) > b(G, s)$. Let T be a bottleneck spanning tree of G rooted at s. Consider the path between s and t in T. Let f be the edge with maximum weight along that path. Since $w(f) \leq b(G, s) < b(s, t)$ it follows that $b(s, t)$ cannot be the bottleneck weight. The paths between a source node s and every other node $v \in V$ contained in a bottleneck spanning tree T however must not be bottleneck paths. We only require that the edge f with maximum weight along a path between s and v in T has $w(f) \leq b(G, s)$. It is possible however that $b(s, v) < w(f) \leq b(G, s)$, which would mean that the path between s and v in T is not a bottleneck path. The argument is analogous in the undirected case.

The BST problem is shown to be equivalent to the BP problem under polynomial-time randomized reductions by Chechik et al. [3]. If there is a $O(f(m, n))$ time algorithm for the BST problem, there is a $O(f(m, n))$ time algorithm for the BP problem and vice versa. The algorithms for solving BP or BST discussed here can be easily modified to solve either problem.

Lastly we want to introduce a basic threshold method for solving BST and BP in directed or undirected graphs that is described by Edmonds and Fulkerson in [6]. It is the basis for many of the algorithms that we are looking at. The BST and BP problems have two interesting properties that algorithms solving these problems can take advantage of. One property is that we know beforehand about the set of possible solutions to our problem. We know that the solution is the weight of one of the edges $e \in E$ of our graph. The other property is that we can easily discard solutions to our problem. If we wanted to know for example, given a graph $G = (V, E)$ and source and target nodes $s, t \in V$, if the weight $w(e)$ of some edge $e \in E$ is the bottleneck weight $b(s, t)$, we could delete all edges with weight larger than $w(e)$ and check if there is a path between s and t in the resulting graph G'. If there is no such path, we know that the bottleneck weight has to be larger than $w(e)$ and could discard $w(e)$ as a solution. If there is a path between s and t, we know that the bottleneck weight has to be smaller than or equal to $w(e)$. We say that $w(e)$ is a feasible solution.

The edge weight $w(e)$ could be the solution but we do not know for certain since there could be an edge $f \in E$ with $w(f) < w(e)$ such that $w(f)$ is a feasible solution as well. The bottleneck weight we are looking for is the smallest edge weight that satisfies our feasibility test. For the BST problem the steps above are the same except we would check if every node in G' could be reached starting from a given root node s or in the undirected case check that G' is connected.
The threshold method consists of going through every edge $e_i \in E$, $i = 1, 2, \ldots, m$ in ascending order of weight and checking for each edge e_i if $w(e_i)$ is a feasible solution to our problem. As soon as a feasible solution is found we stop. Let e_u be the final edge looked at. The bottleneck weight has to be $w(e_u)$. We know for certain that there is not an edge $f \in E$ with $w(f) < w(e_u)$ such that $w(f)$ is the bottleneck weight since we already checked every possible edge with weight smaller than $w(e_u)$. To introduce some terminology we say that the edge weights looked at in each iteration are thresholds where a threshold is denoted by λ. In this method we follow a simple pattern for picking thresholds. In each iteration we choose $\lambda = w(e)$ where e is the edge with the smallest weight in E not already looked at. Note that this method requires sorting the edges of our graph by weight, which is an $O(m \log n)$ time operation. Checking a solution involves deleting edges from our graph and then using BFS or DFS to search the resulting graph which has a runtime cost of $O(m)$. Since this is done for each edge in our graph our total runtime is $O(m^2)$. We see Algorithm 2 and Algorithm 3 using a more sophisticated version of this method where thresholds are picked following a binary search pattern. Here we avoid sorting the edges of our graph and reduce the number of iterations to $O(\log n)$.

3 Dijkstra's algorithm for SSBP

Dijkstra's algorithm for Single-Source Shortest Path (SSSP) [4] can be modified to solve the SSBP problem in directed or undirected graphs. The runtime complexity of the modified algorithm is $O(m+n\log n)$, the same as for SSSP. We know that the bottleneck paths found by an algorithm for SSBP form a bottleneck spanning tree. A solution to SSBP clearly gives us a solution to BP. Therefore Dijkstra's algorithm for SSBP also solves the BST and BP problems in both the directed and undirected case.

The algorithm is shown here. It is almost identical to Dijkstra's algorithm for SSSP. Each node $v \in V$ is in one of three states: *unsearched*, *active*, or *scanned*. We maintain the invariant that for each node $v \in V$ the label $d(v)$ is equal to the bottleneck weight $b(s,v)$ for all paths between s an v that only visit scanned nodes or v.

Algorithm 1 Modified Dijkstra's Algorithm for SSBP

Input: Directed or undirected graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$ and source node $s \in V$.
Output: Bottleneck weight $b(s, v)$ for all $v \in V$.
1: Mark all nodes as unsearched.
2: Label $d(s) \leftarrow -\infty$ and $d(v) \leftarrow \infty$ for all $v \neq s$.
3: Mark s as active.
4: **repeat**
5: Extract an active node u with minimum label $d(u)$ and mark u as scanned.
6: **for all** $(u, v) \in E$ **do**
7: Update $d(v) \leftarrow \min\{d(v), \max\{d(u), w(u, v)\}\}$.
8: Mark v as active if v is unsearched.
9: **until** all nodes are scanned

The labels $d(v)$ for $v \in V$ after the algorithm has terminated are the bottleneck weights $b(s,v)$ we are looking for.

The main difference between the modified algorithm for SSBP and the original algorithm for SSSP lies in how we update the label $d(v)$ for a node v in line 7. In the original Dijkstra algorithm for SSSP we compare the existing label $d(v)$ with the sum $d(u)+w(u,v)$ and update $d(v)$ with the minimum $d(v) = \min\{d(v), d(u)+w(u,v)\}$.

We achieve $O(m + n\log n)$ by using a Fibonacci Heap to keep track of the order of labels for active nodes. Each edge is looked at at most once in lines 6-8. In case $d(v)$ is updated with a new value for an active node v in line 7 we use a *decrease-key* operation which runs in amortized constant time. The total time spent on *decrease-key* operations is $O(m)$. In line 5 we use a *delete-min* operation which runs in time $O(\log n)$ and in line 8 we use an *insert* operation which runs in constant time. Each operation is performed at most once on each node. The total time spent on *delete-min* and *insert* operations is $O(n\log n)$.

If the edges of the graph are given to us in order of their weight, we can use Bucket Queue as an implementation for the Priority Queue instead. We use the rank of edges $e \in E$ $1, 2, \ldots, m$ as the indices of our buckets. A bucket $B_i, 1 \leq i \leq m$, stores the nodes v for which $d(v) = w(e)$ such that the rank of e is i. The *delete-min*

operation for extracting a node v with minimal label $d(v)$ in line 5 takes constant time because the buckets from which the nodes are extracted are looked at in monotonic order allowing us to use a pointer to keep track of the bucket looked at in the last *delete-min* operation. Thus we get a running time of $O(m)$.

We could modify the algorithm to specifically solve BP by terminating in line 5 as soon as our target node is scanned. To solve BST we could take the maximum of all labels $\max_{v \in V} d(v)$ after termination.

4 Linear Time Algorithm for BST and BP in Undirected Graphs

For undirected graphs BST and BP can be solved in $O(m)$ time.

In this algorithm we use edge contraction on connected components. If there are multiple edges incident to the joined node w and incident to some other node r of the graph as result of this operation, we keep the edge with the smallest weight.

The algorithm is shown here. We are using a more efficient version of the threshold method described in Section 2.3. Instead of picking thresholds incrementally we pick thresholds following a binary search pattern. In each iteration we choose the threshold λ to be the median edge weight of all edges in E and check if there is a path between s and t using only light edges $e \in E$ with $w(e) \leq \lambda$. If there is such a path, we know that the bottleneck weight has to be the weight of some light edge. If there is no such path, we know that the bottleneck weight has to be the weight of some heavy edge $e \in E$ with $w(e) > \lambda$. In either case we are able to cut the problem size in half by either removing all light edges or all heavy edges from our graph. This process is repeated until there is only one edge remaining in our graph.

Algorithm 2 BP Algorithm for Undirected Graphs

Input: Undirected graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$ and source and target nodes $s, t \in V$.
Output: Bottleneck weight $b(s, t)$.

```
1: while |E| > 1 do
2:     Let λ be the median edge weight among all edges in E.
3:     Delete all edges e with w(e) > λ. Let G' be the resulting graph.
4:     if there is a path from s to t in G' then
5:         G ← G'
6:     else
7:         Let V_1, ..., V_q be the connected components of G'.
8:         Contract V_1, ..., V_q in G.
```

The weight of the last remaining edge is the bottleneck weight $b(s, t)$.

Finding the median of a set of numbers can be done in linear time [12]. Checking if there is a path between s and t in line 4 can be done using BFS or DFS in linear time. All other operations of our algorithm are bounded by $O(|E|)$. In each iteration we eliminate half of the edges present in the graph. We either delete all heavy edges ($> \lambda$) in line 3 or contract all light edges ($\leq \lambda$) in line 8. There are $O(\log n)$ iterations. The runtime complexity is $\sum_{i=1}^{\log m} O(m/2^{i-1}) = O(m) + O(m/2) + O(m/4) + \cdots = O(m)$

To solve BST we can modify the above algorithm slightly. In line 4 instead of checking if there is a path from nodes s to t in G' we check whether or not the resulting graph G' is connected. The total running time remains the same.

The algorithm can be adapted to handle directed graphs as well. However we would not get the same runtime guarantees. In a directed graph to guarantee correctness we must only shrink strongly-connected components in lines 7-8. It must not be the case however that all light edges are part of a strongly-connected component. Contract-

ing the strongly-connected components in lines 7-8 therefore would not necessarily eliminate half of the edges in the graph.

5 Binary Search Algorithm for BST and BP in Directed Graphs

We show an algorithm for solving BST and BP in directed graphs that runs in $O(m \log n)$ time.

This algorithm is very similar to Algorithm 2 described in Section 4. We use a version of the threshold method (see Section 2.3) where thresholds are chosen according to a binary search pattern. Instead of picking thresholds from the set of edges E of our graph however we maintain a search set F of edges from which thresholds are chosen. The search set F is known to contain the bottleneck edge in each iteration. The reason this is necessary is because in contrast to Algorithm 2 we are not able to eliminate those edges from our graph that we have discarded as potential bottleneck edges. In Algorithm 2 we could reduce the size of the set of edges E in each iteration such that E always contains the bottleneck edge by making use of edge contraction.

In each iteration we let λ be the median edge weight of all edges in F and check if there is a path between s and t using only light edges $e \in E$ with $w(e) \leq \lambda$. If there is such a path, we know that the bottleneck weight has to be the weight of one of the edges $e \in F$ with $w(e) \leq \lambda$. If there is no such path, we know that the bottleneck weight has to be the weight of one of the edges $e \in F$ with $w(e) > \lambda$. We are able to cut the size of F in half by removing all edges e that have either $w(e) \leq \lambda$ or $w(e) > \lambda$. This process is repeated until there is only one edge remaining in F. Initially F is set to be equal to E.

Algorithm 3 BP Algorithm for Directed Graphs

Input: Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$ and source and target nodes $s, t \in V$.
Output: Bottleneck weight $b(s, t)$.

```
1: F ← E
2: while |F| > 1 do
3:     Let λ be the median edge weight among all edges in F.
4:     Delete all edges e with w(e) > λ. Let G' be the resulting graph.
5:     if there is a path from s to t in G' then
6:         F ← {e ∈ F | w(e) ≤ λ}.
7:     else
8:         F ← {e ∈ F | w(e) > λ}.
```

The weight of the remaining edge in F is the bottleneck weight $b(s, t)$. All operations in lines 3-8 are bounded by $O(m)$. There are $O(\log n)$ iterations. The algorithm has a total running time of $O(m \log n)$.

Again the above algorithm can be modified to solve BST by replacing line 5 to instead check whether or not every node in G' can be reached starting from a given root node s.

6 Algorithm by Gabow and Tarjan for BST and BP in Directed Graphs

Algorithm 3 is the basis for an improved algorithm by Gabow and Tarjan [9] for solving BST and BP in directed graphs that runs in time $O(m \log^* n)$. The function $\log^* n$ is the iterated logarithm function which is defined as $\log^* n = \min\{k \geq 1 \mid \log^{(k)} n \leq 1\}$, where $\log^{(1)} n = \log n$ and $\log^{(k)} n = \log \log^{(k-1)} n$, for $k > 1$. It is interesting to note that $\log^* n$ is an extremely slow growing function. For $\log^* n$ to be bigger than 5 we would need to have $n > 2^{65536}$.

The same approach as to the one of Algorithm 3 is used here. We are trying to successively narrow down the set of edges within which the bottleneck edge can be found. In Algorithm 3 we narrow down the set of potential bottleneck edges F by half in each iteration. We choose a threshold λ which we use to partition our search set F into two subsets $F = E_1 \cup E_2$ where $E_1 = \{e \in F \mid w(e) \leq \lambda\}$ and $E_2 = \{e \in F \mid w(e) > \lambda\}$ and reassign F to be either E_1 or E_2. In the algorithm by Gabow and Tarjan we are trying to speed up the process of narrowing down the set of potential bottleneck edges by partitioning our search set F into $k \geq 2$ subsets $F = E_1 \cup E_2 \cup \cdots \cup E_k$ instead of two. We do this by picking $k - 1$ thresholds $\lambda_1, \ldots, \lambda_{k-1}$ that are found by repeatedly calculating median values in F.

In the algorithm by Gabow and Tarjan we consider the BST problem.

6.1 Incremental Search

To find out which of the k subsets contains the bottleneck edge an incremental search algorithm is used, which runs in $O(m)$ time. The pseudocode is shown here. Each node $v \in V$ is one of three states: *labeled*, *unlabeled* or *scanned*.

Algorithm 4 Incremental Search Algorithm

Input: Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$, source node $s \in V$ and $k+1$ sets of edges $E_0, E_1, \ldots, E_k$ where if $e \in E_i$ and $f \in E_{i+1}$ then $w(e) < w(f)$ for $i = 0, 1, \ldots, k-1$. ($F = E_1 \cup E_2 \cup \cdots \cup E_k$ contains the bottleneck edge and $E_0 \subseteq E$ contains edges whose weights are known to be below the bottleneck weight)

Output: ℓ such that E_ℓ contains the bottleneck edge, $1 \leq \ell \leq k$.

1: $A(v) \leftarrow \{w \mid (v, w) \in E_0\}$ for every $v \in V$.
2: $\ell \leftarrow 0$.
3: Mark s as labeled and v as unlabeled for every $v \in V \setminus \{s\}$.
4: **while** there are labeled nodes **do**
5: Let v be a labeled node.
6: Mark v as scanned.
7: **for all** $w \in A(v)$ **do**
8: If w is unlabeled mark w as labeled.
9: **if** every node is scanned **then**
10: Terminate.
11: $\ell \leftarrow \ell + 1$.
12: **for all** $(v, w) \in E_\ell$ **do**
13: **if** v is unlabeled **then**
14: $A(v) \leftarrow A(v) \cup \{w\}$.
15: **else**
16: If w is unlabeled mark w as labeled.
17: **go to** 4

We start with $\ell = 0$. In each iteration we check if all nodes in G can be reached using only edges in $\bigcup_{i=1}^{\ell} E_i$. If not, we continue with $\ell \leftarrow \ell + 1$, where edges in $E_{\ell+1}$ are now considered in the search.

If the edges of the graph have small integer weights or are given in order of their weight, we can use the incremental search algorithm to solve BST in linear time. Instead of iterating over sets of edges E_ℓ for $\ell = 0, 1, 2 \ldots, k$ we iterate over the the edges $e_\ell \in E$ for $\ell = 0, 1, 2, \ldots, m$ sorted by weight, $w(e_i) \leq w(e_{i+1})$ for $i = 0, 1, 2, \ldots, m-1$. We can use Bucket Sort to sort the edges in time $O(m)$ if we are given edges with small integer weights. In each new iteration i the edge e_i gets included in the search. After termination we get e_ℓ as our bottleneck edge.

6.2 Main Algorithm

The k subsets are found by repeated median finding. We let λ be the median edge weight among all edges in F and split F into two subsets $F_\ell = \{e \in F \mid w(e) \leq \lambda\}$ and $F_r = \{e \in F \mid w(e) > \lambda\}$. We then repeat that step recursively on F_ℓ and F_r. We stop when there are k subsets. We have $\log k$ levels of recursion. Since the median finding algorithm runs in linear time splitting a set of edges into k subsets takes $O(|F| \log k)$ time in total. Note that the sizes of the k subsets are roughly equal $|E_i| = \lfloor |F|/k \rfloor$ or $\lceil |F|/k \rceil$ for $i = 1, 2, \ldots, k$.

The main algorithm is shown here.

Algorithm 5 BST Algorithm by Gabow and Tarjan

Input: Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$ and root node $s \in V$.

Output: Bottleneck weight $b(G, s)$.

1: $E_0 \leftarrow \emptyset$.
2: $F \leftarrow E$.
3: **while** $|F| > 1$ **do**
4: Partition F into k subsets $E_1, E_2, \ldots, E_k$ using repeated median finding.
5: Use INCREMENTAL SEARCH to find i such that E_i contains the bottleneck edge.
6: $E_0 \leftarrow E_0 \cup E_1 \cup \cdots \cup E_{i-1}$.
7: $F \leftarrow E_i$.

The weight of the remaining edge in F is the bottleneck weight $b(G, s)$.

We get our running time of $O(m \log^* n)$ by setting k appropriately in each iteration. In the i-th iteration, where $i > 1$, we set $k_i = 2^{k_{i-1}}$. Initially we set $k_1 = 2$.

Let $F^{(i)}$ be F in the i-th iteration. Then $|F^{(i)}| \leq |F^{(i-1)}|/k_{i-1} = O(m/k_{i-1})$. Thus the running time of partitioning $F^{(i)}$ into k subsets in line 4 is bounded by $O(m/k_{i-1} \cdot \log k_i) = O(m/k_{i-1} \cdot k_{i-1}) = O(m)$. The running time per iteration is $O(m)$. In each iteration we increment the number of times that we exponentiate 2 for setting k. After at most $O(\log^* n)$ iterations we have $|F|/k \leq 1$. Thus we get a total running time of $O(m \log^* n)$.

To solve BP we can modify INCREMENTAL SEARCH to terminate when the target node t is marked as labeled.

7 Algorithm by Chechik et al. for BST and BP in Directed Graphs

The algorithms discussed in this section and Section 8 are able to improve on the running time of the algorithms looked at in previous sections through use of randomization. We show an algorithm by Chechik et al. [3] for solving BST and BP in directed graphs. A similar approach of narrowing down the interval in which the bottleneck weight can be found is used here. The idea for improvement is to randomly sample edges and use their weights as a way of partitioning the set of potential bottleneck edges instead of using the repeated median finding approach of Gabow and Tarjan in Section 6. Chechik et al. in [3] and Duan et al. in [5] (discussed in Section 8) show that randomly sampling edge weights is likely to partition a set of edges evenly in their case meaning that no subset of the partition is significantly larger than another one. Based on that analysis they are able to derive an improved runtime complexity for their algorithms.

The algorithm by Chechik et al. further refines Algorithm 5 by Gabow and Tarjan achieving an expected running time of $O(m\beta(m,n))$ which is a little bit better than $O(m \log^* n)$. $\beta(m,n)$ is defined as $\beta(m,n) = \min\{k \geq 1 \mid \log^{(k)} n \leq m/n\}$. We have $\beta(m,n) \leq \log^* n - \log^*(m/n) + 1$. Since $\log^* n$ is already an extremely slow growing function this improved bound is of no practical importance. It is interesting to observe however that we come even closer to an $O(m)$ running time. In particular if $m \geq n \log^{(k)} n$ for some constant k we have $\beta(m,n) \leq k$ and get a total expected running time of $O(m)$.

In Algorithm 5 we initially partitioned the set of edges $F = E$ of the graph into k subsets $E_1, E_2, \ldots, E_k$ such that all edges in E_i have weights smaller than all edges in E_{i+1} for $i = 1, 2, \ldots, k-1$ and the subsets are roughly of equal size. Because of that we could make guarantees about the number of iterations needed until the size of our search set F is reduced to one at which point our algorithm terminates. A slightly different approach is used here. We take a random sample of size k from the set of edges E in the graph. Let $\lambda_1, \lambda_2, \ldots, \lambda_k$ be k thresholds, set to the weights of the k sampled edges in sorted order. Let $\lambda_0 = -\infty$ and $\lambda_{k+1} = \infty$. The thresholds naturally partition E into $E = E_0 \cup E_1 \cup \cdots \cup E_k$, where $E_i = \{e \in E \mid \lambda_i \leq w(e) < \lambda_{i+1}\}$. In Algorithm 5 the time needed for partitioning E and finding the subset E_i such that E_i contains the bottleneck edge is $O(m \log k)$. Chechik et al. show an algorithm for finding i such that the bottleneck weight $b(G,s)$ is located within $[\lambda_i, \lambda_{i+1})$ without explicitly computing the partition $E = \bigcup_{i=0}^{k} E_i$ that runs in $O(m + nk)$ or even $O(m + n \log k)$ time. Computing $F = E_i = \{e \in E \mid \lambda_i \leq w(e) < \lambda_{i+1}\}$ and $E_0 = \{e \in E \mid w(e) < \lambda_i\}$ takes $O(m)$ additional time.

7.1 Locate

The algorithm for finding i such that $\lambda_i \leq b(G,s) < \lambda_{i+1}$ is shown next. The algorithm goes through $k+1$ phases $i = 0, 1, 2, \ldots, k$. In each phase all nodes v for which there is a path from source node s to v in G all whose weights are smaller than λ_{i+1} are identified. If by the i-th phase all vertices $v \in V$ have been identified, we output i. We maintain a label $d(v)$ that keeps track of the smallest edge weight found so far such that there is a path from s to v using only edges e with $w(e) \leq d(v)$.

Algorithm 6 Locate Algorithm

Input: Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$, source node $s \in V$ and k thresholds $\lambda_1, \lambda_2, \ldots, \lambda_k$ in sorted order.
Output: i such that $\lambda_i \leq b(G, s) < \lambda_{i+1}$.

```
1:  λ_0 ← −∞, λ_{k+1} ← ∞.
2:  d(s) ← −∞ and d(v) ← ∞ for all v ≠ s.
3:  A ← ∅, B ← V.
4:  for i in 0, 1, 2, ..., k do
5:      for all v ∈ B do
6:          if d(v) < λ_{i+1} then
7:              Move v from B to A.
8:      while |A| > 0 do
9:          Extract a node v from A.
10:         for all (u, v) ∈ E do
11:             w̄ ← max{d(u), w(u, v)}.
12:             if w̄ < d(v) then
13:                 d(v) ← w̄.
14:                 if d(v) < λ_{i+1} and v ∈ B then
15:                     Move v from B to A.
16:     if |B| = 0 then
17:         return i
```

Each node is looked at at most k times in lines 5-7 and each edge is looked at at most once in lines 9-15. The overall time complexity is $O(m + nk)$.

7.2 Main Algorithm

The algorithm by Chechik et al. extends Algorithm 5 by Gabow and Tarjan. The LOCATE algorithm is used to initially narrow down the search set F. After that Algorithm 5 is run until the the size of F is further reduced to $O(m/\log n)$ at which point sorting F and finding the bottleneck weight $b(G, s)$ can be done in $O(m)$ time using the INCREMENTAL SEARCH algorithm from Section 6 (Algorithm 4).

Algorithm 7 BST Algorithm by Chechik et al.

Input: Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$ and root node $s \in V$.
Output: Bottleneck weight $b(G, s)$.

1: Sample k edges $e_1, e_2, \ldots, e_k$ from E uniformly randomly.
2: Sort the k edges by weight so that $w(e_i) < w(e_{i+1})$ for $i = 1, 2, \ldots, k-1$.
3: Set $\lambda_0 \leftarrow -\infty$, $\lambda_{k+1} \leftarrow \infty$ and $\lambda_i \leftarrow w(e_i)$ for $i = 1, 2, \ldots, k$.
4: Use LOCATE to find i such that $\lambda_i \leq b(G, s) < \lambda_{i+1}$.
5: Compute $F = E_i = \{e \in E \mid \lambda_i \leq w(e) < \lambda_{i+1}\}$ and $E_0 = \{e \in E \mid w(e) < \lambda_i\}$.
6: Run Algorithm 5 with F and E_0 until $|F| = O(m/\log n)$.
7: Sort F and use Algorithm 4 to find the bottleneck weight $b(G, s)$.

Initially we choose $k = \log^{(r)} n, r \geq 1$. Chechik et al. show that the expected size of any one subset E_i obtained by partitioning the set of edges E using k uniformly randomly sampled edge weights is $\mathbb{E}[|E_i|] \leq 2m/k$. Using Markov's Inequality we can say that the probability of $|E_i| \geq 4m/k$ is going to be at most $1/2$. Therefore

the expected size of F in line 5 is $|F| = O(m/k) = O(m/\log^{(r)} n)$. In line 6 we set $k = \log^{(r-1)} n$ and run one iteration of Algorithm 5. The running time is $O(m + |F| \log k) = O(m+|F| \log^{(r)} n) = O(m)$. The size of F is reduced to $O(m/\log^{(r-1)} n)$. Repeating this step with $k = \log^{(r-2)} n$ will reduce the size of F to $O(m/\log^{(r-2)} n)$. Thus running r iterations in total will give us $|F| = O(m/\log n)$. Line 4 takes $O(m + n \log^{(r)} n)$ time. In line 6 repeating Algorithm 5 r times takes $O(rm)$ time and line 7 can be done in $O(m)$ time. We have a total expected running time of $O(rm+n \log^{(r)} n)$. Setting $r = \beta(m, n)$ we get $O(m\beta(m, n)+nm/n) = O(m\beta(m, n))$.

The BP problem can be solved by slight modification of the LOCATE algorithm. We stop as soon as $d(t) < \lambda_{i+1}$ is found where t is our target node. Additionally the modified version of Algorithm 5 for solving BP is used.

8 Algorithm by Duan et al. for SSBP in Directed Graphs

Lastly we take a closer look at the Single Source Bottleneck Path problem and the algorithm by Duan et al. [5] which gives an improved running time of randomized $O(\sqrt{nm \log n \log\log n} + m\sqrt{\log n})$ compared to Dijkstra's algorithm for solving SSBP in directed graphs that is discussed in Section 3.

The algorithm by Duan et al. [5] continues the approach of the algorithms discussed in the previous sections of progressively narrowing down the interval within which the bottleneck weight can be found. In contrast to the previous algorithms we are concerned with finding more than one bottleneck weight at the same time. Duan et al. propose a Divide-and-Conquer method in which each subproblem represents a subgraph consisting of nodes v whose bottleneck weight $b(s, v)$ can be found in a certain interval. This is explained in more detail later. As with the algorithm by Chechik et al. in Section 7 we rely on uniformly randomly sampling edge weights to provide the intervals that are used to narrow down the search space. The weights of k sampled edges are used to form k thresholds. The assumption is that randomly picking edge weights partitions a set of edges evenly with high likelihood. This allows us to make guarantees about the runtime of the algorithm.

8.1 Arbitrary-Source Bottleneck Path ASBPIC

The algorithm by Duan et al. first reduces the Single Source Bottleneck Path problem to an instance of an equivalent problem called Arbitrary-Source Bottleneck Path with Initial Capacity (ASBPIC). In ASBPIC an initial capacity function, defined on the nodes of the graph, $h : V \to \mathbb{R} \cup \{\pm\infty\}$ is introduced as part of the problem input and the source node s that is part of SSBP is dropped. In SSBP the bottleneck path for a target node t given a source node s is defined to be the path $p = \langle e_1, e_2, \ldots, e_\ell \rangle$ going from s to t such that the minimum edge weight $\min\{w(e) \mid e \in E(p)\}$ is maximum. In ASBPIC for a path p starting form an arbitrary source node v we call the minimum of all edge weights in p and the initial capacity $h(v)$ of source node v $\min\{\{w(e) \mid e \in E(p)\} \cup h(v)\}$ the capacity of path p. The bottleneck path for a target node t is redefined to be the path starting from an arbitrary source node v and ending in t such that the capacity of that path is maximum. We use $d(t)$ to denote the capacity of a bottleneck path for a node t. The ASBPIC problem is given a graph $G = (V, E)$, weight function $w : E \to \mathbb{R} \cup \{\pm\infty\}$ and initial capacity function $h : V \to \mathbb{R} \cup \{\pm\infty\}$ output $d(v)$ for all $v \in V$.

The reduction from SSBP to ASBPIC is simple. Given an SSBP instance (G, w, s) we assign $h(s) = \max\{w(e) \mid e \in E\}$ and $h(v) = -\infty$ for all $v \in V \setminus \{s\}$. After solving ASBPIC we get $d(v) = b(s, v)$ for all $v \in V$.

We can use a variant of Dijkstra's algorithm to solve ASBPIC. This differs only slightly from Dijkstra's algorithm for SSBP in Section 3. Initially all nodes $v \in V$ are marked as active and their labels get initialized to $d'(v) \leftarrow h(v)$. We use $d'(v)$ to denote the label for a node v in Dijkstra's algorithm to distinguish it from the bottleneck capacity $d(v)$.

Algorithm 8 Modified Dijkstra's Algorithm for ASBPIC

Input: Graph $G = (V, E)$, weight function $w : E \to \mathbb{R} \cup \{\pm\infty\}$ and initial capacity function $h : V \to \mathbb{R} \cup \{\pm\infty\}$.
Output: Capacity $d'(v)$ for all $v \in V$.
1: Mark all nodes as active.
2: Label $d'(v) \leftarrow h(v)$ for all $v \in V$.
3: **repeat**
4: Extract an active node u with maximum label $d'(u)$ and mark u as scanned.
5: **for all** $(u, v) \in E$ **do**
6: Update $d'(v) \leftarrow \max\{d'(v), \min\{d'(u), w(u, v)\}\}$.
7: Mark v as active if v is unsearched.
8: **until** all nodes are scanned

After termination we have $d(v) = d'(v)$ for all $v \in V$.

8.2 Overview of the Main Algorithm

We walk through the basic steps of the algorithm by Duan et al. It is a recursive Divide-and-Conquer algorithm. Let us start at the point at which the algorithm is first called. The input we receive is the ASBPIC instance we get as a result of the reduction from SSBP to ASBPIC (see Section 8.1).

We are given a directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R} \cup \{\pm\infty\}$ and an initial capacity function $h : V \to \mathbb{R} \cup \{\pm\infty\}$. We define $E^{(r)} = \{e \in E \mid w(e) < +\infty\}$ and a parameter $k \geq 1$. Initially we have $E^{(r)} = E$. We sample $\ell = \min\{k, |E^{(r)}|\}$ edges $e_1, e_2, \ldots, e_\ell$, $w(e_i) < w(e_i + 1)$ for $i = 1, 2, \ldots, \ell - 1$, from $E^{(r)}$ and use their weights to form ℓ thresholds $\lambda_i = w(e_i)$. Additionally we let $\lambda_0 = -\infty$ and $\lambda_{\ell+1} = \infty$. The thresholds form $\ell+1$ intervals $[\lambda_0, \lambda_1), [\lambda_1, \lambda_2), \ldots, [\lambda_\ell, \lambda_{\ell+1})$. It is important for us to know in which interval $[\lambda_i, \lambda_{i+1})$ some number $x \in \mathbb{R}$ is located. Let us define a function $I : \mathbb{R} \to \{0, 1, \ldots, \ell\}$ such that if $I(x) = i$ we have $\lambda_i \leq x < \lambda_{i+1}$. We say that $I(x)$ is the index of $x \in \mathbb{R}$. We now compute for every $v \in V$ in which interval the bottleneck capacity $d(v)$ is located. Based on the result we partition V into $\ell+1$ levels $V = V_0 \cup V_1 \cup \cdots \cup V_\ell$ where $V_i = \{v \in V \mid I(d(v)) = i\}$. We construct a new ASBPIC instance $(G_i = (V_i, E_i), w_i, h_i)$ that consists of nodes in V_i and some of the edges that connect those nodes for $i = 0, 1, \ldots, \ell$ and solve each instance recursively. The critical part of the algorithm is finding out for a node v the interval $[\lambda_i, \lambda_{i+1})$ that $d(v)$ belongs to. This can be done by using Dijkstra's algorithm for ASBPIC. For every node v and every edge e in the graph we map $h(v)$ to $I(h(v))$ and $w(e)$ to $I(w(e))$ and use Dijkstra's algorithm to compute $d'(v)$. We then have $d'(v) = I(d(v))$ for every $v \in V$ where $I(d(v))$ is the interval we are looking for and can continue with partitioning V into $\ell + 1$ levels.

When constructing a new ASBPIC instance $(G_i = (V_i, E_i), w_i, h_i)$ we reduce the size of E to only contain edges that are needed for computing $d(v)$ for nodes $v \in V_i$. Additionally we reassign initial capacities and edge weights. For some of the edges $e \in E_i$ we let $w_i(e) = +\infty$. We call edges e that have $w(e) = +\infty$ *unrestricted*. Conversely we call edges e that have $w(e) < +\infty$ *restricted*. We use $E^{(r)}$ to denote the set of all restricted edges in our graph. When we have $|E^{(r)}| \leq 1$ in some call of our algorithm we can directly solve $d(v)$ for all nodes v present in the graph and return. This represents the base case of our recursive algorithm. Additionally if the

graph G we receive as input to our algorithm is not weakly-connected we make a recursive call for each weakly-connected component. By doing this we can be sure that our main algorithm operates on a weakly-connected graph. We use this fact when we discuss the runtime of our algorithm in the later subsections.

8.3 Constructing a New ASBPIC Instance

Each new ASBPIC instance (G_i, w_i, h_i) has to be constructed in such a way that the capacity $d_i(v)$ for node $v \in V_i$ in the subgraph equals $d(v)$ in the original graph. We explain how a new ASBPIC instance is created. This can be done in linear time by scanning all nodes and edges in the graph. The following terminology is used:

- An edge (u, v) is called *cross-level* when nodes u and v belong to different levels $I(d(u)) \neq I(d(v))$
- An edge (u, v) is called *below-level* when nodes u and v belong to the same level V_i and the weight of (u, v) falls below the i-th interval $I(w(u, v)) < i = I(d(u)) = I(d(v))$
- An edge (u, v) is called *above-level* when nodes u and v belong to the same level V_i and the weight of (u, v) falls above the i-th interval $I(w(u, v)) > i = I(d(u)) = I(d(v))$

We give a small illustration in Figure 3.

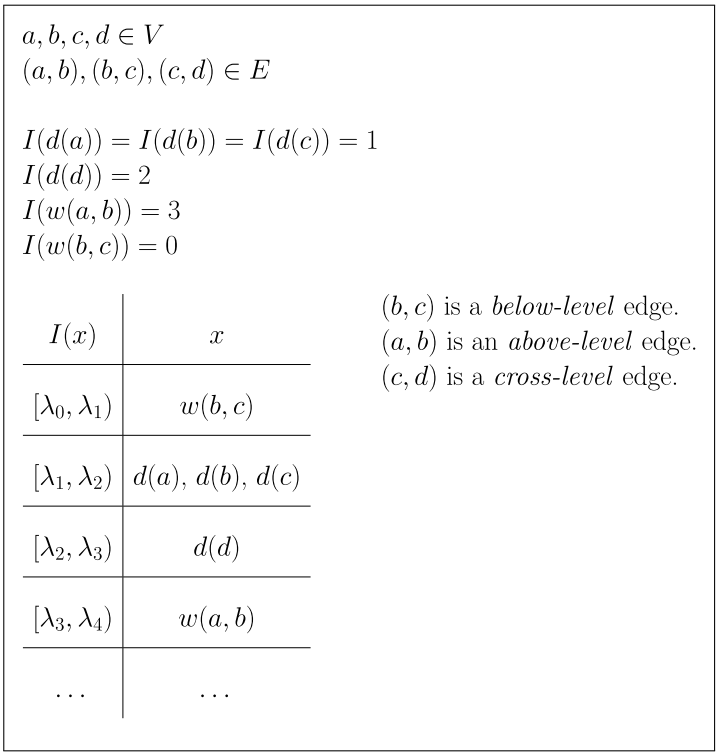


Figure 3: Illustration of below-level, above-level and cross-level edges

During our construction of a new ASBPIC instance (G_i, w_i, h_i) for a particular i we are only concerned with edges (u, v) for which at least one of u or v belongs to level i. Therefore by cross-level edges we mean edges (u, v) for which one of u or

v belongs to level i and the other belongs to a different level. By below-level and above-level edges we mean edges (u, v) for which both u and v belong to level i but $I(w(u, v)) < i$ or $I(w(u, v)) > i$.

All nodes belonging to a level smaller than i can be dropped since a bottleneck path ending in a node $v \in V_i$ cannot traverse them. Since an edge (u, v) incident to nodes $u \in V_{i+1} \cup V_{i+2} \cup \cdots \cup V_\ell$ and $v \in V_i$ could have a weight $w(u, v) = d(v)$ that determines the capacity of a bottleneck path ending in v we let $h_i(v) = \max\{\{h(v)\} \cup \{w(u, v) \mid u \in V_{i+1} \cup V_{i+2} \cup \cdots \cup V_\ell\}\}$ for every node $v \in V_i$ in our new ASBPIC instance. We effectively save potential bottleneck capacities using the initial capacity $h(\cdot)$. By doing this we can now drop all nodes belonging to a level larger than i and all cross-level edges. All below-level edges can be dropped since a bottleneck path ending in $v \in V_i$ cannot traverse them. Although above-level edges could be traversed by bottleneck paths ending in $v \in V_i$ their weights do not determine the capacity $d(v)$ of a bottleneck path. We keep above-level edges e but let $w_i(e) = +\infty$.

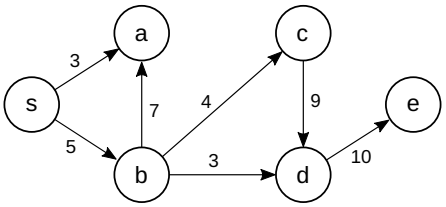
8.4 Algorithm for the Graph with at Most One Restricted Edge

Let us look at the algorithm for calculating $d(v)$ for all $v \in V$ if $|E^{(r)}| \leq 1$. If there are no restricted edges $|E^{(r)}| = 0$, the bottleneck capacity $d(v)$ for a node $v \in V$ is determined only by the maximum initial capacity $h(s)$ of possible source nodes s from which v can be reached. If nodes u and v belong to the same strongly-connected component, we have $d(u) = d(v)$. We can therefore simplify our calculation in the following way. We use Tarjan's algorithm [13] to determine all strongly-connected components and contract them to a single node each in linear time. For each new node w we set $h(w)$ to be the maximum $h(u)$ for all nodes u in the component. The resulting graph $G' = (V', E')$ is acyclic. We use Dijkstra's algorithm to compute $d'(w)$ for all $w \in V'$. Because G' is a DAG this can be done in linear time by considering nodes in topological order. After running Dijsktra's algorithm we set $d(v) = d'(w)$ for all $v \in V$ where w is the node that results from contracting the strongly-connected component in G that v belongs to. If there is one restricted edge $|E^{(r)}| = 1$, we let $e_0 = (u_0, v_0)$ be the only restricted edge. We remove e_0 and use the algorithm for $|E^{(r)}| = 0$ above to get $d(v)$ for all $v \in V$. Now all nodes v that cannot be reached from v_0 have the right $d(v)$. For nodes v that can be reached from v_0 we set $d(v) = \max\{d(v), \min\{d(u_0), w(e_0)\}\}$.

8.5 Example of the Main Algorithm

We want to demonstrate how our main algorithm works on an example graph. This should give us a clearer picture of the concepts discussed in the preceding subsections.

$V = \{s, a, b, c, d, e\}$
$E = \{(s, a), (s, b), (b, a), (b, c), (b, d), (c, d), (d, e)\}$
$h(s) = 10$
$h(a) = h(b) = h(c) = h(d) = h(e) = -\infty$
$w(s, a) = 3$
$w(s, b) = 5$
$w(b, a) = 7$
$w(b, c) = 4$
$w(b, d) = 3$
$w(c, d) = 9$
$w(d, e) = 10$



Bottleneck Capacities:
$d(s) = 10$
$d(a) = b(s, a) = 5$
$d(b) = b(s, b) = 5$
$d(c) = b(s, c) = 4$
$d(d) = b(s, d) = 4$
$d(e) = b(s, e) = 4$

① Pick thresholds

Let $k = 3$
Set of k sampled edges: $\{(s, b), (c, d), (b, d)\}$

$\lambda_0 = -\infty$
$\lambda_1 = w(b, d) = 3$
$\lambda_2 = w(s, b) = 5$
$\lambda_3 = w(c, d) = 9$
$\lambda_4 = \infty$

$$I(x) = \begin{cases} 0 & \text{if } x \in [-\infty, 3) \\ 1 & \text{if } x \in [3, 5) \\ 2 & \text{if } x \in [5, 9) \\ 3 & \text{if } x \in [9, \infty) \end{cases}$$

② Map edge weights $w(\cdot)$ and initial capacities $h(\cdot)$

For all edges $(u, v) \in E$ map $w(u, v)$ to $I(w(u, v))$:
$w'(s, a) = I(w(s, a)) = 1$
$w'(s, b) = I(w(s, b)) = 2$
$w'(b, a) = I(w(b, a)) = 2$
$w'(b, c) = I(w(b, c)) = 1$
$w'(b, d) = I(w(b, d)) = 1$
$w'(c, d) = I(w(c, d)) = 3$
$w'(d, e) = I(w(d, e)) = 3$

For all nodes $v \in V$ map $h(v)$ to $I(h(v))$:
$h'(s) = I(h(s)) = 3$
$h'(a) = I(h(a)) = 0$
$h'(b) = I(h(b)) = 0$
$h'(c) = I(h(c)) = 0$
$h'(d) = I(h(d)) = 0$
$h'(e) = I(h(e)) = 0$

③ Run Dijkstra's algorithm with G, w', h'

After termination we get:
$d'(s) = I(d(s)) = 3$
$d'(a) = I(d(a)) = 2$
$d'(b) = I(d(b)) = 2$
$d'(c) = I(d(c)) = 1$
$d'(d) = I(d(d)) = 1$
$d'(e) = I(d(e)) = 1$

④ Split V into $k + 1$ levels

For all nodes $v \in V$ put v into $V_{d'(v)}$:
$V_0 = \emptyset$
$V_1 = \{c, d, e\}$
$V_2 = \{a, b\}$
$V_3 = \{s\}$

⑤ Construct a new ASBPIC instance $(G_1 = (V_1, E_1), w_1, h_1)$

$I(x)$	x
$[-\infty, 3)$	
$[3, 5)$	$d(c), d(d), d(e)$
$[5, 9)$	$d(b)$
$[9, \infty)$	$w(c,d), w(d,e)$

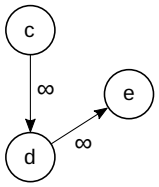
cross-level edges: $\{(b,c),(b,d)\}$
below-level edges: $\emptyset$
above-level edges: $\{(c,d),(d,e)\}$

Save the weights of edges coming from higher levels:
$h_1(c) = \max\{-\infty, w(b,c) = 4\} = 4$
$h_1(d) = \max\{-\infty, w(b,d) = 3\} = 3$
$h_1(e) = -\infty$

Drop all nodes belonging to different levels:
$V_1 = \{c, d, e\}$

Drop all cross-level and below-level edges:
$E_1 = \{(c,d),(d,e)\}$

Reassign the weights of all above-level edges:
$w_1(c,d) = +\infty$
$w_1(d,e) = +\infty$



⑥ Solve $d(v)$ for all v in $G_1 = (V_1, E_1)$

We have $|E^{(r)} = \emptyset| \leq 1$.

Contract all strongly-connected components:
$(V = \{c\}, E = \emptyset) \to w_1$
$(V = \{d\}, E = \emptyset) \to w_2$
$(V = \{e\}, E = \emptyset) \to w_3$
$V' = \{w_1, w_2, w_3\}$
$E' = \{(c,d),(d,e)\}$
$h'(w_1) = 4$
$h'(w_2) = 3$
$h'(w_3) = -\infty$
$w'(c,d) = +\infty$
$w'(d,e) = +\infty$

After running Dijsktra's algorithm with G', w', h':
$d'(w_1) = d(c) = b(s,c) = 4$
$d'(w_2) = d(d) = b(s,d) = 4$
$d'(w_3) = d(e) = b(s,e) = 4$

8.6 Runtime of the Main Algorithm

The previous subsections give a complete account of all operations of our algorithm. We now want to consider the runtime cost of these operations. We fill in some low level details that are needed for understanding how we achieve our overall runtime complexity. For analysis of the runtime it is important to observe that no cross-level or below-level edges appear in any subsequent recursive call and above-level edges become unrestricted. We let $r = |E| - \sum_{i=0}^{\ell} |E_i|$ be the number of edges that do not appear in any subsequent G_i and let $r' = |E^{(r)}| - \sum_{i=0}^{\ell} |E_i^{(r)}|$ be the number of restricted edges that become unrestricted or do not appear in any subsequent G_i. In analyzing the runtime of our algorithm we try to describe the runtime of different parts of our algorithm in terms of r or r'. This is because an operation that takes $O(r \cdot f(m,n))$ or $O(r' \cdot f(m,n))$ time per recursive call has a runtime that is bounded by $O(m \cdot f(m,n))$ across all recursive calls. Constructing a new ABPIC instance and computing $d(v)$ if $|E^{(r)}| \leq 1$ takes $O(m)$ time per level of recursion. Sampling and sorting ℓ edges from $E^{(r)}$ runs in $O((r+r')\log k)$ and Dijkstra's algorithm for splitting V into $\ell + 1$ levels runs in $O(m + n \log\log k + r \log k)$ time per level of recursion.

In Subsection 8.7 we discuss how $O((r + r') \log k)$ is achieved for sorting the set of sampled edges and in Subsection 8.8 we discuss the $O(m + n \log \log k + r \log k)$ time split algorithm.

8.7 Sorting the Set of Sampled Edges

Sorting the set of ℓ sampled edges can be done in $O(\ell \log \ell) = O((r + r') \log \ell) = O((r + r') \log k)$ time. For a sampled edge e_i with rank i we can say that either V_i is empty or V_i is not empty. Let ℓ_1 be the number of edges e_i for which V_i is empty and ℓ_2 be the number of edges e_i for which V_i is not empty. We have $\ell = \ell_1 + \ell_2$. If V_i is empty, $e_i = (u, v)$ must be a cross-level, below-level or above-level edge. If $I(d(u)) = I(d(v))$, we have $I(e_i) = i \neq I(d(u)) = I(d(v))$ meaning e_i is below-level or above-level and if $I(d(u)) \neq I(d(v))$, e_i is cross-level. This means that $\ell_1 \leq r'$. If there are ℓ_2 edges e_i for which V_i is not empty, there must be at least $\ell_2 - 1$ cross-level edges that connect the nodes of these ℓ_2 different levels. This is because our graph is guaranteed to be weakly-connected. Remember that we recurse on each weakly-connected component of G in case G is not weakly connected. We have $\ell_2 \leq r + 1$. Thus $\ell = \ell_1 + \ell_2 \leq r + r' + 1$ and the time needed for sorting the set of sampled edges is bounded by $O((r + r') \log k)$.

8.8 Splitting Nodes into Levels

We explain the algorithm that is used to split all nodes $v \in V$ into levels $V_0, V_1, \ldots, V_l$ where $V_i = \{v \in V \mid I(d(v)) = i\}$. The problem with the straight forward approach suggested in Section 8.2 of mapping initial capacities $h(v)$ to $I(h(v))$ and edge weights $w(e)$ to $I(w(e))$ and using a simple linear time implementation of Dijkstra's algorithm is the runtime cost associated with evaluating the index $I(\cdot)$ of initial capacities and edge weights. Duan et al. show that the number of recursive levels is $O(\log n / \log k)$ with high probability. Using binary search to find $I(x) = i$ for $x \in \mathbb{R}$, such that $\lambda_i \leq x < \lambda_{i+1}$ takes $O(\log k)$ time. If we perform an index evaluation for every edge and node in the graph, the time needed is $O(m \log k)$ per level of recursion and $O(m \log k) \cdot O(\log n / \log k) = O(m \log n)$ overall, which is worse than the $O(m + n \log n)$ time needed for Dijkstra's SSBP algorithm. In the split algorithm by Duan et al. the number of index evaluations is bounded by $O(r + n / \log k)$ and the time spent on index evaluations per level of recursion is thus $O(r + n / \log k) \cdot O(\log k) = O(r \log k + n)$. The idea is to instead of performing all index evaluations upfront delay an index evaluation until it becomes necessary. The split algorithm is an implementation of Dijkstra's algorithm in which we reduce the number of index evaluations. We discuss how certain operations of Algorithm 8 can be efficiently implemented and an overall time bound of $O(m + n \log \log k + r \log k)$ is achieved.

8.8.1 Index Evaluations for Edge Weights

First we look at how we can reduce the number of index evaluations for edge weights. During the update step of Dijkstra's algorithm (Algorithm 8 on line 6) we let $d'(v) = \max\{d'(v), \min\{d'(u), I(w(u, v))\}\}$. Here we can avoid calculating $I(w(u, v))$ if $\min\{d'(u), I(w(u, v))\} = d'(u)$. We check if $\lambda_{d'(u)} \leq w(u, v)$ and only calculate $I(w(u, v))$ if $\lambda_{d'(u)} > w(u, v)$. If $\lambda_{d'(u)} > w(u, v)$, (u, v) must be a cross-level or below-level edge. If node v ends up with a final label value of $d'(v) = d'(u)$ (u and v belong to the same level), we have $I(w(u, v)) < I(d(u)) = I(d(v))$ and (u, v)

is below-level. Otherwise (u, v) is cross-level. Thus the number of index evaluations for edge weights is at most r.

8.8.2 Index Evaluations for Initial Capacities

We look at how we can reduce the number of index evaluations for initial capacities. This is achieved by special construction of the priority queue that is used in Dijkstra's algorithm for maintaining the order of active nodes according to their label $d'(\cdot)$. We outline how the priority queue is used in our implementation of the split algorithm and derive from that an upper bound on the number of index evaluations for initial capacities.

Let us recall some key points of the split algorithm in Subsection 8.2. The label values $d'(v)$ for $v \in V$ are the indices $0, 1, \ldots, \ell$ of our edge weights and initial capacities. Initially all nodes $v \in V$ are active and the labels are set to $d'(v) = I(h(v))$. We repeatedly extract active nodes v with maximum label $d'(v)$. We can observe that the nodes extracted in this way have monotonically decreasing label values $d'(\cdot)$. We could therefore use an array of buckets that hold the nodes $v \in V$ according to their label $d'(v)$ and extract each node by iteratively looking at the buckets in reversed order. In the i-th iteration we look at the bucket with index i, where $i = \ell, \ell-1, \ldots, 0$ and extract all active nodes v with maximum label $d'(v) = i$.

For an active node v we can differentiate between two cases. Either $d'(v)$ has been set in a previous iteration of the algorithm using the update step $d'(v) = \max\{d'(v), \min\{d'(u), I(w(u, v))\}\}$ or $d'(v)$ has not been set in which case $d'(v) = I(h(v))$. We call v an *updated* node in the former case and an *initializing* node in the latter case. If v is an updated node, the value of $d'(v)$ is known and no index evaluation has to be performed. If v is an initializing node, we need to evaluate $I(h(v))$ to get a value for $d'(v)$. The general idea for reducing the number of index evaluations is to evaluate $I(h(v))$ for a node v during the i-th iteration only if v is an initializing node, meaning $d'(v)$ is not set, and $I(h(v)) = i$.

We initialize two arrays of buckets B_i and C_i that are each used to maintain the order of nodes according to their label $d'(\cdot)$, $i = 0, 1, \ldots, \ell$. B contains initializing nodes and C contains updated nodes. Since we do not evaluate $d'(v) = I(h(v))$ for all $v \in V$ upfront a bucket B_i cannot directly store nodes v with $I(h(v)) = i$. A bucket B_i stores groups U of nodes for which $\max\{I(h(v)) \mid v \in U\} = i$. We have $\bigcup_i U_i = V$. We discuss shortly how these groups are constructed. For a group U we find the node $v \in U$ with maximum $h(v)$ and evaluate $I(h(v))$. We can then put that group into the correct bucket $B_{I(h(v))}$. At the start of the i-th iteration we move all nodes v with $d'(v) = I(h(v)) = i$ from B_i into C_i. We repeatedly extract nodes u from C_i until C_i is empty and for each u update the label $d'(v)$ of adjacent nodes v. If the label $d'(v)$ of an adjacent node v is updated with a new value, we put v into $C_{d'(v)}$ and remove v from its previous location in one of the groups U in B or from C. Before starting with the next iteration we need to reassign the groups $U \in B_i$ to their correct bucket. For every $U \in B_i$ we evaluate $I(h(v))$ for the node $v \in U$ with maximum $h(v)$ and put U into $B_{I(h(v))}$.

To say more about the number of index evaluations for initial capacities we explain how the groups U are constructed. In our graph G we change every directed edge to be undirected. We find a spanning tree T in the resulting graph G' and partition that tree into $b = O(n/s)$ edge-disjoint subtrees $T_1, T_2, \ldots, T_b$ each of size $O(s)$. For

each tree T_i we now form a group U_i of nodes that contains the nodes belonging to T_i. To compute the node with maximum $h(\cdot)$ in a group we initialize for each U_i a priority queue that maintains the order of nodes $v \in U_i$ by their initial capacity $h(v)$.

We can now add some technical details to the description above. To move all nodes v with $d'(v) = I(h(v)) = i$ from B_i into C_i in the i-th iteration we consider all groups $U \in B_i$. For each group U we can first set $d'(v) = i$ for the node v with maximum $h(v)$ and move v into C_i. We now use the priority queue to get a new node $u \in U$ with maximum $h(u)$. Instead of evaluating $I(h(u))$ explicitly we can check if $h(u) \geq \lambda_{d'(v)}$, which can be done in constant time. If $h(u) \geq \lambda_{d'(v)}$, we have $I(h(u)) = d'(v)$ and can set $d'(u) = d'(v) = i$. We then move u into C_i and repeat this process with the next maximum node. If $h(u) < \lambda_{d'(v)}$, we have $I(h(u)) < I(h(v)) = i$ and can stop with that group. At the end of the i-th iteration we look at all groups $U \in B_i$ and for each group U evaluate $I(h(u))$ for the node $u \in U$ with maximum $h(u)$. For the maximum node u in a group $U \in B_i$ we can make the observation that u has a final label value $d'(u) < i = d'(v)$ where v is the last node that was removed from U at the start of the i-th iteration. This is because at this point in the algorithm all nodes with final label value $d'(v) \geq i$ have already been scanned. Therefore we can say that an index evaluation has to be performed every time we have $d'(u) < d'(v)$ for a new maximum node u in one of the groups. If we perform c index evaluations for a group U_i, this means that there are at least c different final label values $d'(\cdot)$ in U_i. From that we can conclude that there must be at least $c - 1$ cross-level edges in the subtree T_i corresponding to U_i that connect the nodes of these c different levels. Thus the total number of index evaluations performed for initial capacities is at most $r + b$, where b is the number of groups. We set $s = \min\{\log \ell, n\}$ and get $b = O(n/s) = O(n/\log k)$. The total number of index evaluations for both edge weights and index evaluations can be bounded by $2r + b = O(r + n/\log k)$.

Wee see that by connecting the nodes in a group with the nodes of a spanning tree we can identify for each index evaluation a cross-level edge in the tree. However this argument would also work if we form a single group from the spanning tree T of G' without partitioning into subtrees. The reason for partitioning into multiple subtrees is that the operations on the corresponding priority queue for each group of nodes become cheaper while the cost associated with partitioning does not negatively effect our runtime complexity.

8.8.3 Split Algorithm

The pseudocode for the split algorithm is shown next.

Algorithm 9 Split Algorithm

Input: Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R} \cup \{\pm\infty\}$, initial capacity function $h : V \to \mathbb{R} \cup \{\pm\infty\}$ and $\ell + 2$ thresholds $\lambda_0 = -\infty, \lambda_1, \ldots, \lambda_\ell, \lambda_{\ell+1} = \infty$ in sorted order.

Output: Partition of V into $\ell+1$ levels $V = V_0 \cup V_1 \cup \cdots \cup V_\ell$ where $V_i = \{v \in V \mid \lambda_i \leq d(v) < \lambda_{i+1}\}$.

1: Change every directed edge to be undirected in G to get G'. Find a spanning tree T of G'. Find $b = O(n/s)$ subtrees $T_1, \ldots, T_b$ of T. Form b groups $U_1, \ldots, U_b$ where each U_i contains the nodes that are in T_i.
2: Initialize a priority queue implemented by binary heap for each group U_i by the order of $h(\cdot)$.
3: Initialize $\ell+1$ buckets $B_0, \ldots, B_\ell$. For each group U_i put U_i into bucket $B_{I(h(v_i))}$ where v_i is the maximum node of U_i.
4: Initialize $\ell + 1$ buckets $C_0, \ldots, C_\ell$. Let $C_i \leftarrow \emptyset$ for each $i = 0, 1, \ldots, \ell$.
5: **for** i in $\ell, \ell - 1, \ldots, 0$ **do**
6: **for all** $U \in B_i$ **do**
7: **while** $|U| > 0$ and the maximum node $u \in U$ has $I(h(u)) = i$ **do**
8: Extract u from U and put u into C_i.
9: $d'(u) \leftarrow i$.
10: **while** $|C_i| > 0$ **do**
11: Extract a node u from C_i.
12: **for all** $(u, v) \in E$ **do**
13: $\bar{w} \leftarrow \min\{d'(u), I(w(u, v))\}$.
14: **if** $\bar{w} > d'(v)$ or $\lambda_{\bar{w}} > h(v)$ if $d'(v)$ does not exist **then**
15: Delete v from $C_{d'(v)}$.
16: Delete v from its group U and put v into $C_{\bar{w}}$.
17: $d'(v) \leftarrow \bar{w}$.
18: **while** $|B_i| > 0$ **do**
19: Extract a group U with $|U| > 0$ from B_i.
20: Evaluate $I(h(u))$ for the maximum node $u \in U$ and put U into $B_{I(h(u))}$.
21: **for all** $v \in V$ **do**
22: Put v into $V_{d'(v)}$.

All index evaluations for initial capacities are performed in lines 3 and 20. All index evaluations for edge weights are performed in line 13. As discussed in Subsubsection 8.8.1 and 8.8.2 there are $O(r + n/\log k)$ index evaluations in total each costing $O(\log k)$ time which gives us a runtime of $O(r+n/\log k)\cdot O(\log k) = O(r \log k+n)$ for all index evaluations performed. The delete operations on the binary heap for a group U in lines 8, 16 and 20 run in $O(\log |U|) = O(\log s) = O(\log \log k)$ time. There are at most $O(n)$ such operations. We get a runtime of $O(n) \cdot O(\log \log k) = O(n \log \log k)$ in total. Finding the subtrees $T_1, \ldots, T_b$ of our spanning tree T can be done in $O(n)$ time using a slight variant of the topological partition algorithm in Frederickson's paper [8]. All other operations run in linear time. The overall time complexity for the split algorithm is thus $O(m + n \log \log k + r \log k)$.

8.9 Main Algorithm

We show the pseudocode for the complete SSBP algorithm.

Algorithm 10 SSBP Algorithm by Duan et al.

Input: Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R} \cup \{\pm\infty\}$ and source node $s \in V$.

Output: Bottleneck weight $b(s, v)$ for all $v \in V$.

1: Reduce the SSBP instance (G, w, s) to an ASBPIC instance (G, w, h) by assigning $h(s) = \max\{e \in E \mid w(e)\}$ and $h(v) = -\infty$ for all $v \neq s$.
2: Call MAIN(G, w, h).
3: **function** MAIN(G, w, h)
4: **if** G is not weakly-connected **then**
5: Call MAIN(G_s, w, h) for each weakly-connected component G_s of G.
6: Let $E^{(r)} = \{e \in E \mid w(e) < +\infty\}$ be the set of restricted edges.
7: **if** $|E^{(r)}| \leq 1$ **then**
8: Compute $d(v)$ for all $v \in V$ and return.
9: Sample $\ell = \min\{k, |E^{(r)}|\}$ edges from $E^{(r)}$ uniformly randomly, sort them by weight and form ℓ thresholds $\lambda_i = w(e_i)$ where e_i is a sampled edge with rank i for $i = 1, 2, \ldots, \ell$.
10: $\lambda_0 \leftarrow -\infty$ and $\lambda_{\ell+1} \leftarrow \infty$.
11: Use the split algorithm to partition V into $\ell + 1$ levels $V_0, V_1, \ldots, V_\ell$ where $V_i = \{v \in V \mid \lambda_i \leq d(v) < \lambda_{i+1}\}$.
12: **for** i in $0, 1, \ldots, \ell$ **do**
13: Construct a new ASBPIC instance (G_i, w_i, h_i) that consists of nodes in V_i and some of the edges that connect those nodes.
14: Call MAIN(G_i, w_i, h_i).

The labels $d(v)$ for all $v \in V$ are the bottleneck weights $b(s, v)$ for our SSBP instance.

Computing $d(v)$ for all $v \in V$ in line 8 runs in $O(m)$ time. Sampling and sorting k edges in line 9 runs in $O((r + r') \log k)$ time. The split algorithm in line 11 has a runtime of $O(m + n \log\log k + r \log k)$. Constructing all new ASBPIC instances in line 13 runs in $O(m)$ time. Adding up the cost of our operations we get a runtime of $O(m + n \log\log k + (r + r') \log k)$ per level of recursion.

The parameter k is set at the beginning and remains constant during the running of our algorithm. We use k only to control the runtime of our algorithm. Duan et al. show that the maximum number of recursive levels is $O(\log n / \log k)$ with probability $1 - n^{\omega(1)}$ when setting $k = 2^{\Omega(\sqrt{\log n})}$. Therefore we achieve a total running time of $O(m \log n / \log k + n \log n / \log k \log\log k + m \log k)$. When $m \leq n \log\log n$ we set $k = 2^{\Theta(\sqrt{n \log n \log\log n / m})}$ which gives us a runtime of $O(\sqrt{nm \log n \log\log n})$. When $m > n \log\log n$ we set $k = 2^{\Theta(\sqrt{\log n})}$ which gives us a runtime of $O(m\sqrt{\log n})$. As our final result we can state that SSBP can be solved in $O(\sqrt{nm \log n \log\log n} + m\sqrt{\log n})$ with high probability.

9 Conclusion

We saw that in the undirected case simple solutions exist for our SSBP, BST and BP problems. BST and BP can be solved in $O(m)$ time by a binary search method (Section 4) that makes use of edge contraction to shrink the graph by half in each iteration. SSBP can be solved indirectly by solving the MST problem in randomized $O(m)$ time [11] or deterministically in $O(m\alpha(m,n))$ time [2]. In the directed case we saw solutions for BST and BP that come very close to a $O(m)$ time bound. The algorithm by Gabow and Tarjan [9] achieves a time bound of $O(m \log^* n)$ where $\log^* n$ is an extremely slow growing function. For practical applications $\log^* n$ is unlikely to ever exceed 5 since this would require n to be bigger than 2^{655336}, which is much larger than any quantity found in the physical world. Chechik et al. [3] still give a tiny improvement of randomized $O(m\beta(m,n))$ on the result of Gabow and Tarjan. For SSBP in directed graphs Duan et al. [5] present a randomized $O(\sqrt{nm \log n \log\log n} + m\sqrt{\log n})$ time algorithm. Duan et al. are able to improve on the traditional algorithm by Dijkstra for solving SSBP using Fibonacci Heap (Section 3), which has a time bound of $O(m + n \log n)$. Dijkstra's algorithm can be used to sort n numbers, so there is a sorting barrier which prevents a more efficient implementation from being found. It was an open question whether or not there existed a sorting barrier for SSBP. The algorithm by Duan et al. overcomes this sorting barrier when $m = \sigma(n\sqrt{\log n})$. There remains an open question whether the algorithm by Duan et al. can further be improved to $O(m\beta(m,n))$. For BP and BST in directed graphs there remains the question whether a linear time algorithm can be found. Furthermore, are there deterministic algorithms for solving BP, BST and SSBP in directed graphs with equally good time bounds?

References

[1] Georg Baier, Ekkehard Köhler, and Martin Skutella. "On the k-Splittable Flow Problem". In: *Algorithms — ESA 2002*. Ed. by Rolf Möhring and Rajeev Raman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 101–113. ISBN: 978-3-540-45749-7.

[2] Bernard Chazelle. "A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity". In: *J. ACM* 47.6 (Nov. 2000), pp. 1028–1047. ISSN: 0004-5411. DOI: 10.1145/355541.355562. URL: https://doi.org/10.1145/355541.355562.

[3] Shiri Chechik et al. "Bottleneck Paths and Trees and Deterministic Graphical Games". In: *33rd Symposium on Theoretical Aspects of Computer Science (STACS 2016)*. Ed. by Nicolas Ollinger and Heribert Vollmer. Vol. 47. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 27:1–27:13. ISBN: 978-3-95977-001-9. DOI: 10.4230/LIPIcs.STACS.2016.27. URL: http://drops.dagstuhl.de/opus/volltexte/2016/5728.

[4] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390.

[5] Ran Duan, Kaifeng Lyu, and Yuanhang Xie. "Single-Source Bottleneck Path Algorithm Faster than Sorting for Sparse Graphs". In: *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Ed. by Ioannis Chatzigiannakis et al. Vol. 107. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 43:1–43:14. ISBN: 978-3-95977-076-7. DOI: 10.4230/LIPIcs.ICALP.2018.43. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9047.

[6] Jack Edmonds and D.R. Fulkerson. "Bottleneck extrema". In: *Journal of Combinatorial Theory* 8.3 (1970), pp. 299–306. ISSN: 0021-9800. DOI: https://doi.org/10.1016/S0021-9800(70)80083-7. URL: https://www.sciencedirect.com/science/article/pii/S0021980070800837.

[7] Jack Edmonds and Richard M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *J. ACM* 19.2 (Apr. 1972), pp. 248–264. ISSN: 0004-5411. DOI: 10.1145/321694.321699. URL: https://doi.org/10.1145/321694.321699.

[8] Greg N. Frederickson. "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications". In: *SIAM J. Comput.* 14 (1985), pp. 781–798.

[9] Harold N Gabow and Robert E Tarjan. "Algorithms for two bottleneck optimization problems". In: *Journal of Algorithms* 9.3 (1988), pp. 411–417. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(88)90031-4. URL: https://www.sciencedirect.com/science/article/pii/0196677488900314.

[10] Volker Kaibel and Matthias Peinhardt. *On the Bottleneck Shortest Path Problem*. eng. Tech. rep. 06-22. Takustr. 7, 14195 Berlin: ZIB, 2006.

[11] David R. Karger, Philip N. Klein, and Robert E. Tarjan. "A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees". In: *J. ACM* 42.2 (Mar. 1995), pp. 321–328. ISSN: 0004-5411. DOI: 10.1145/201019.201022. URL: https://doi.org/10.1145/201019.201022.

[12] A. Schönhage, M. Paterson, and N. Pippenger. "Finding the Median". In: *J. Comput. Syst. Sci.* 13.2 (Oct. 1976), pp. 184–199. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(76)80029-3. URL: https://doi.org/10.1016/S0022-0000(76)80029-3.

[13] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010. eprint: https://doi.org/10.1137/0201010. URL: https://doi.org/10.1137/0201010.