Algorithm Analysis and Design: Final Report

# Bottleneck Path Algorithms

Benyu Wang

January 15, 2022

## 1 Introduction

### 1.1 Definitions and Problems

**Def** The **bottleneck path** (or widest path) for two nodes $s, t$ is a path from $s$ to $t$ in a weighted graph, maximizing the weight of the minimum-weight edge in the path.

**Def** The **bottleneck spanning tree** rooted at $s$ is a spanning tree (arborescence) in which every node can be reached from $s$, maximizing the weight of the minimum-weight edge in it.

**Problems** We consider three problems in this review: BP, SSBP, and BST.

The Bottleneck Path (BP) problem asks the bottleneck path weight for two nodes $s, t$.

The Single Source Bottleneck Path (SSBP) problem asks the bottleneck path weight to all nodes from a node $s$.

The Bottleneck Spanning Tree (BST) problem asks the bottleneck tree weight for a root $s$.

### 1.2 Introduction

Before we review the algorithms, we can have some assumptions:

We consider the number of nodes as $|V| = n$ and the number of edges as $|E| = m$. $\Omega(m)$ is necessary since we need to get every edge weight before calculating the exact values. We will see that Dijkstra's Algorithm gives complexity $O(m + nlogn)$, so we assume that the graph is sparse, where $m = o(nlogn)$.

In undirected graphs, the problem is relatively easy since the bottleneck spanning tree from any node is the maximum spanning tree, and the bottleneck paths are on the maximum spanning tree, so we can use any known algorithm for spanning tree to solve it (for example, the classical $O(n\beta(m, n)))$. Therefore, we can safely assume that the problem is in directed graphs.

We can also safely assume that the edge weights are distinct since it is easy to change them to distinct weights in linear time.

## 2 The reduction between BP and BST

We consider the proof by Chechik et al. The proof shows that the two problems can be easily reduction to each other with approximately same complexity.

**Lemma 1** If BST can be solved in $O(f(m,n))$ time, then BP can be solved in $O(f(m+n,n))$ time.

*Proof.* Consider linking $n$ edges with infinity weight from the destination $t$ to every node. Then we know the bottleneck path to $t$ will not change. Moreover, one $BST$ can be the bottleneck path plus some edges with infinity weight. Therefore, the $BST$ algorithm in this case can get the bottleneck path from $s$ to $t$.

**Lemma 2** If BP can be solved in $O(f(m,n))$ time, then using randomized algorithm, BST can be solved in $O(\sum_{i\geq 0} f(m/2^i, n/2^i))$ expected time.

*Proof.* Given a value, we can easily find every node whose BP value is higher or equal by DFS in linear time. Also by DFS, we can show that finding BST is as hard as finding the lowest BP value in the nodes.

Randomly sampling a node $u$ and let $S$ be the set of nodes with BP value is higher or equal than $u$. Then with at least $1/2$ probability $|S| \geq n/2$ and we can shrink $S$ to one node. This declines $n$ to a half with two tries expected.

Similarly, randomly sampling an edge $(u,v)$ and let $S$ be the set of nodes with BP value is higher or equal than $v$. Then with at least $1/2$ probability the number of edges going into $S \geq m/2$. This is because if we consider pairs (edge $(u,v)$, $v$) and sort them by $d(v)$, in this uniform sampling we can get one in the lower half with probability at least $1/2$.

So recursively doing these operations we can get $O(\sum_{i\geq 0} f(m/2^i, n/2^i))$ complexity since every round the complexity is $O(m', n')$.

## 3 Dijkstra's Algorithm

Dijkstra's Algorithm can be used in this case with small modifications. Consider a set $S$, initially empty, and every round we add a node $u$ with largest bottleneck value to $S$, and use $d(v) = max(d(v), min(d(u), w(u,v)))$ to update every $v$ where there exists an edge $u \to v$.

We use Fibonacci Heap to maintain these values $d(v)$. So there are $O(n)$ inserts and deletes, with $O(m)$ decrease-keys. We can see that the overall complexity is $O(m + nlogn)$.

The correctness proof is similar as the one we use in the shortest path. We can do induction on that the nodes in $S$ are with the maximum bottleneck path values among all nodes. After that we prove in every turn we really update the one with the maximum bottleneck path value in the rest nodes correctly.

## 4 Gabow and Tarjan's $O(mlog^*n)$ Algorithm for BST

If we need to sort the weight of these edges, we need $\Omega(mlogn)$ time at least, and if we actually sort the answers of the nodes such as in the algorithm above, we need $\Omega(nlogn)$ time, which can't be better than Dijkstra's algorithm. But partition is more efficient.

**Lemma** We can partition the $m$ edges into $k$ sets with size $O(m/k)$ with increasing edge values in $O(mlogk)$ time.

*Proof.* This is done using median finding. We recursively find the median in linear time and equally partition the set of edges. In every round, we do this for every set, until there are $k$ sets. Then the largest set will be at most twice as large as the smallest one, which shows that the size of all sets are in $O(n/k)$.

**Theorem** BST can be computed in $O(mlog^*n)$ time.

We partition the $m$ edges into $k$ sets with size $O(m/k)$. We use which set the edge is in to replace its weight, then we do BST using Dijkstra but since the value is from 1 to $k$, we don't need the Fibonacci Heap and it is in linear time.

After that, if we locate the answer in the $i$-th set, we use $\infty$ to replace all set with larger values, and delete all set with lower values. Recursively doing this will give $O(m'logk)$ time in every round, where $m'$ is the count of rest edges, and the rest edges are $O(m'/k)$.

If there are $r$ rounds, the linear part gives $O(mr)$ time complexity. If we want $m'logk$ to be the same in every round, we may let $k' = 2^k$ for every next round until $k' = m'$. Starting from $k = 2$ in the first round, this gives $m'logk = O(m)$ in every round. and in this situation we know $r = O(log^*m) = O(log^*n)$. So BST can be computed in $O(mlog^*n)$ time in total.

# 5 A Randomized Algorithm for BST with $O(m\beta(m,n))$ time

If we need $\Omega(mlogk)$ in the first round which is required by partitioning, then $k$ need to be small enough to make this round not too time consuming. The algorithm below gives a optimization using random sampling and locating without partitioning the edges.

To sample $k$ edges and to sort them are easy. This gives a expectation that the length of the answer interval consists $O(m/k)$ edges as expected, so it seems to be as good as the equal partition. With Markov bound, we can show that in constant rounds expected we can achieve this and give the new problem with $O(m/k)$ edges.

And the things remain to do is to locate the answer. In the $i$-th round, we can check if every unsearched node's value is lowe than the $i$-th value we choose, and than use the searched nodes to update its neighbors as in Dijkstra and maintain their values (if small enough, also marked as searched and consider its neighbors). This needs a $k$-round checking and a linear visiting in total. The complexity of it is $O(nk + m)$.

Now let $k = m/n$, and apply the algorithm above in next rounds (so in next round $k = 2^{(m/n)}$ and so on). We will see that if the algorithm ends in $r$ rounds, then the complexity will be $O(mr)$, and $log^{(r)}n = m/n$, which gives a overall complexity of $O(m\beta(m,n))$ expected time.

# 6 An $O(m\sqrt{logn})$ Time Randomized Algorithm for SSBP

The SSBP problem is harder since can't efficiently reduce the amount of nodes or edges in SSBP compared with BST. Suppose we can do a partition in $O(nlogk)$ time which is better than the rounds above, and we may think that $T(n) = kT(n/k) + O(nlogk)$, which is $O(nlogn)$ regardless of the choice of $k$. Therefore, it can't make advantage above Dijkstra's Algorithm.

The paper considered to give every node an initial capacity instead of linking $s$ to all nodes with some value. The version is slightly different but equivalent to $SSBP$. Then it also recursively do the $k$-partition, modify the edge weights, and give initial capacities.

## 6.1 Partition and its depth

As a lemma, we can prove that if there's zero or one edge with value not infinity (but with initial capacities), then the problem can be solved in linear time.

**Lemma** If there's zero or one edge with value not infinity, the problem can be solved in linear time.

*Proof.* If the edges are all with infinity capacity, we can see that every node in one strongly connected component will share the same answer. And by shrinking these components the graph will become a DAG. We can do the Dijkstra update in topological order in linear time then.

If there is only one edge with non-infinity capacity, delete it, compute the values and then add it back. If the edge is $(u, v)$, then $d(u)$ will be correct after the first part, and after adding it back, we may use $d(v)$ to update in the DAG. This is still in linear time.

Now we consider to partition the edges into $k$ sets by randomly choosing $k-1$ values. We also think it as good as a equal partition. In fact, with probability calculations and union bounds, the paper gives the conclusion:

**Lemma** With high probability, we will get every set with size $O(nlogk/k)$ in one round, and finally the recursion depth will be $O(logn/logk)$ until in every component there's zero or one edge with value not infinity and we can compute efficiently.

### 6.2 The complexity in one round

For edges, we use $min(d(u), I(w(u, v)))$ to update $d(v)$ in every round. However, there's no need to compute $I(w(u, v))$ exactly if $w(u, v)$ is large enough and larger than the lower bound of the interval of $d(u)$. Otherwise, we can see that after the partition, the edge will be deleted and not used any more. Therefore, for every edge we only need to evaluate its interval once in the whole algorithm.

However, we still cannot afford the $O(nlogk)$ complexity to consider the intervals of all initial capacities. Below we show the procedure in the paper.

**Lemma** For a tree with $n$ nodes and $s < n$, we can partition the tree into edge-disjoint subtrees with size in $[s, 3s)$.

*Proof.* A function can partition the tree into $[s, 2s)$ with a $< s$ rest part. We consider to recursively partition the subtrees first, then from left to right, merge as a subtree if the current size $\geq s$, and finally return the rest part with size $< s$. We can induction on the size of the returning rest part to prove that every subtree we partition will have size $< 2s$. Then the lemma comes after arbitrarily merging the rest part with an arbitrary connected one.

By lemma we find a spanning tree $T$ (see the edges as undirected), choose $s = logk$ and form $b = O(n/logk)$ groups by this partition. We then form two groups of nodes in the search: the updated nodes, which are the nodes getting value from some other node in this round, and the initializing nodes, which are the nodes getting value from the initial capacity.

We only evaluate the initial capacities for every node with the maximal initial capacity in its group. This gives $O(blogk) = O(m)$ time. Every time we choose one with smallest value from all nodes and update like in Dijkstra. If we choose an updated node, nothing else is needed to be done. However, if it is an initializing node, we need to find the next one in its group.

It can be shown that if there are $x$ different values in one group, then on the spanning tree at least $x - 1$ edges will not be considered in the next rounds. And checking whether the next one's value is in the same group as the one we considered is in $O(1)$. Therefore, we can show that actually every $O(logk)$ evaluation is still corresponding to a edge deletion in the next rounds, which composes a $O(mlogk)$ complexity.

### 6.3 The overall complexity

Finally, we find that we should do a linear part in $O(logn/logk)$ rounds in 6.1, with an extra cost $O(mlogk)$ in these rounds in 6.2. So the overall complexity is $O(m(logk + logn/logk))$. So we choose $logk = \sqrt{logn}$ to balance and give the complexity $O(m\sqrt{logn})$. It is faster than Dijkstra's algorithm when $m = o(n\sqrt{logn})$.

## References


[1] Gabow, H.N., & Tarjan, R.E. (1988). Algorithms for Two Bottleneck Optimization Problems. J. Algorithms, 9, 411-417.

[2] Chechik, S., Kaplan, H., Thorup, M., Zamir, O., & Zwick, U. (2016). Bottleneck paths and trees and deterministic Graphical games. In H. Vollmer, & N. Ollinger (Eds.), 33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016 [27] (Leibniz International Proceedings in Informatics, LIPIcs; Vol. 47). Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing. https://doi.org/10.4230/LIPIcs.STACS.2016.27

[3] Duan, R., Lyu, K., & Xie, Y. (2018). Single-Source Bottleneck Path Algorithm Faster than Sorting for Sparse Graphs. ArXiv, abs/1808.10658.