UNIVERSITY OF CALIFORNIA
RIVERSIDE

An Evaluation of Shortcutting Strategies for Parallel Bellman-Ford and Other Parallel Single-Source Shortest Path Algorithms

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Daniel Thomas Li

September 2023

Thesis Committee:

Dr. Yihan Sun, Chairperson
Dr. Zhijia Zhao
Dr. Elaheh Sadredini

The Thesis of Daniel Thomas Li is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am incredibly grateful towards my advisor Professor Sun. She has been incredibly patient, supportive, and understanding. Whenever we ran into a problem, she always had many ideas on how to test our assumptions and figure out the root causes. I would not have been able to complete this without her.

To my parents for their love and encouragement.

To my sister for providing an endless supply of snacks.

To the ISPC gang for making my time at UCR memorable.

To my friends for their support.

ABSTRACT OF THE THESIS

An Evaluation of Shortcutting Strategies for Parallel Bellman-Ford and Other Parallel Single-Source Shortest Path Algorithms

by

Daniel Thomas Li

Master of Science, Graduate Program in Computer Science
University of California, Riverside, September 2023
Dr. Yihan Sun, Chairperson

A fundamental question in graph theory is the Single-Source Shortest Path (SSSP) problem. This is well-studied in classical algorithm literature, but is only more recently studied in the parallel setting. A relatively simple way to solve SSSP in parallel is with a Parallel Bellman-Ford(BF). BF shows strong performance on dense graphs, when $m >> n$. But due to its frontier-based approached, BF is bounded by the diameter of the graph. This thesis proposes 2 different preprocessing strategies to alleviate this. The first strategy is to generate shortcuts such that each vertex attempts to have at most degree k. The second approach is graph contraction, which removes specific vertices and replaces them with a single shortcut. We show that both preprocessing strategies reduce the overall rounds required to complete all testing algorithms. Additionally, we evaluate both preprocessing strategies with our own implementation of BF and state of the art parallel SSSP algorithms. In general, δ-stepping and ρ-stepping show improved times after contraction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are incredibly useful abstractions that model natural and synthetic data across many domains. For example, graphs are commonly used to represent social networks, the structure of the internet, data dependencies for compiler analysis, and protein-protein interaction networks. [1, 27, 23, 31].

One of the most fundamental questions for a graph is the Single-Source Shortest Path (SSSP) problem: Given a source node s, compute the minimum-weighted (shortest) path to all other nodes within the graph. The weight of a path p is the summation of the weight of each edge that constructs p.

There are many well-studied sequential algorithms to solve this problem, including Dijkstra's Algorithm and Bellman-Ford. But computer hardware has been advancing at a rapid pace for the past several decades. DRAM is now accessible and multicore processors are the norm. A newer branch of algorithms have evolved to fully utilize these advanced hardware capabilities - parallel algorithms.

A parallel algorithm can be loosely defined as an algorithm that does multiple things in a single step [12]. A simple example is a reduction. Given an array of n numbers, we want to reduce all elements into a single element. Here we assume the reduction function is the sum. One way to do this sequentially is to iterate through each element one at a time, adding that to the current sum. To do this in parallel, we can simply add every 2 neighbors together (like $A[1] + A[2]$, $A[3] + A[4]$) until the number of elements are halved. Using multiple processors, we can do this in a single step. Then repeat this until the final sum is computed. It's easy to see that this does as much work as the sequential version, but finishes in exactly $\log n$ steps.

Surprisingly, there are inherently some algorithms that do not scale well depending on the structure of the input data. Parallel Bellman-Ford is one such example of this. When the span of the graph is large with relatively small frontiers, the performance of a parallel Bellman-Ford can be worse than the sequential version. There are several reasons for this. Many parallel SSSP algorithms are not as work-efficient as there sequential counterparts. Additionally, a parallel Bellman-Ford has frontier-based parallelism, which means that only vertices within the same frontier can be processed simultaneously to ensure correctness. Between every frontier is a forced barrier to synchronize any threads spawned in that round. The sequential version has no such restriction and can simply process its queue until empty.

The goal of this thesis is to explore several preprocessing algorithms and the effect they have upon subsequent traversals. These preprocessing algorithms transform the graph by adding or deleting vertices and edges. We then evaluate the impact of these transformations by running a suite of Parallel SSSP algorithms.

The remainder of the paper is outlined as follows. Previous research in related topics is discussed. Then we discuss a preliminary background on graphs and parallel algorithms. We then give an overview of the testing algorithms. Next, we do an in-depth discussion about each preprocessing algorithm. Afterwards, we discuss implementation and optimization details. We then discuss the experimental results. Lastly, we summarize key findings in the conclusion and present further ideas to explore in later research.

Chapter 2

Related Work

There is a rich literature related to SSSP algorithms, parallel algorithms, graph-based frameworks, parallel SSSP algorithms, and parallelism outside of shared-memory multicore systems. These are a selected few works to discuss.

2.1 Single-Source Shortest Path Algorithms

Arguably the most famous SSSP algorithm is Dijkstra's algorithm, which has bounds of $O(m \log n)$ if a binary heap is used as the priority queue[17]. Fredman and Tarjan improve Dijkstra's algorithm with a Fibonacci heap, achieving a time of $O(n \log n + m)$ [19]. If all weights are positive integers, then Thorup shows a bucket-based approach to achieve linear time bounds[35]. If a graph has negative weight edges, then Bellman-Ford is another classic solution to SSSP, achieving bounds of $O(mn)$[6]. Constraining edge weights to integers, recent work by Bernstein et. al. show a randomized algorithm that achieves bounds of $O(m \log^8(n) \log W)$ [7].

2.2 Parallel Algorithms for the Binary Fork Join Model

Blelloch et. al. do an in-depth overview of many algorithms with the binary-forking model, providing cost analysis, proofs, and detailed algorithms. They include list ranking, sorting, and balanced tree operations, to name a few[10]. Dhulipala et. al. also present many parallel algorithms, including SSSP, BFS, k-core, low-diameter decomposition, strongly connected components, maximal independent set, and triangle counting [16].

2.3 Graph Frameworks

There has been keen interest in developing general, high performance frameworks to operate on graphs. Shun et. al. developed Ligra, a framework for shared memory algorithms that only requires the user to define 2 functions, a *vertex map* and *edge map* function. They implemented a simple parallel BFS with this framework, but only with unweighted graphs[33]. Dhulipala et. al. shows a work-efficient parallel weighted BFS using a bucket-based graph framework Julienne, which outperforms Ligra for the same task [15]. Zhang et. al. show a general graph framework called GraphIT, which has high performance for Domain Specific Languages. The key detail is that algorithms are separated from the scheduling, which can lead to better performance depending on the input[36].

2.4 Parallel SSSP

Meyers and Sanders invented the δ-stepping algorithm, which shows strong practical parallel performance[28]. Madduri et. al. validated this claim by showing fast perfor-

mance of δ-stepping on low diameter, sparse graphs [26]. Blelloch et. al. build on top of delta stepping with radius stepping, relaxing all edges within a variable distance, enabling tight bounds [11]. Dong et. al. abstract these previous algorithms by developing a general framework for parallel stepping algorithms, which they applied to δ-stepping, Bellman-Ford, and a new algorithm called ρ-stepping [18]. Key to their implementation is a new Abstract Data Type, the Lazy Batched Priority Queue. Andoni et. al. developed an parallel $(1+\epsilon)$-approximate algorithm to compute SSSP for undirected graphs. They do so with m poly $\log n$ work and poly $\log n$ span. They develop the notion of a low hop emulator, which is a graph in which every path has at most $O(\log \log n)$ edges. [2]

2.5 Parallelism beyond Multicore Shared Memory

Parallelism is not just for multicore systems. There are many levels of parallelism, from distributed systems to HPC workloads to GPUs and FPGAs. Davidson et. al. show several SSSP implementations designed for GPUs, noting several challenges considering the architectural differences[14]. Harish et. al. also show a SSSP algorithm for GPUs, but give a detailed analysis of Parallel BFS and connected components [20].

Chapter 3

Preliminaries

3.1 Graph Notation

Graphs will be represented as $G = (V, E, w)$, where V represents the vertices, E represents the edges, and w is a function that maps an edge to a non-negative real number $w : E \to \mathbb{R}_+$. It is noted as $w(e)$, where $e \in E$. Unweighted graphs are shown as $G = (V, E)$. Unless otherwise specified, graphs depicted here are simple and undirected. n and m represent the number of vertices and edges in the graph respectively. Additionally, the neighbor set of a vertex v will be noted as $N(v)$. We define $N(v) = \{u|(v, u) \in E\}$. The degree of a vertex v is $deg(v)$.

3.2 Definitions

We use several common and uncommon definitions later in this paper. We define them here as they may be referenced in later proofs.

Definition 1 *Shortcut: A shortcut from node x to node y in G is a edge from $x \to y$ with distance > 1 that contains the minimum-weighted path between $x \to y$.*

Definition 2 *Chain vertex: A vertex with a degree of 2.*

Definition 3 *Endpoint: An endpoint is a vertex v where $deg(v) \neq 2$ and at least one of $N(v)$ is a chain vertex.*

Definition 4 *With High Probability (w.h.p.): A bound $O(f(n))$ happens with high probability if the bound is $O(kf(n))$ with probability at least $1 - 1/n^k$, for any $k > 1$.*

3.3 Computational Model

All analysis on parallel algorithms in this paper follow the binary-forking model with Compare and Swap (CAS) [10, 32]. In this model, a multiprocessor system shares a pool of threads with access to shared memory. Each thread has the typical capabilities and behavior under the standard RAM model (such as a limited number of registers and a program counter), but also has an additional *fork* instruction. When *fork* is invoked, the parent thread spawns two child threads and suspends until both child threads finish execution (i.e. they join), and then the parent resumes resumes. Under this model, a parallel for-loop with n elements has a work of $O(n)$ and span of $O(\log n)$.

We also augment the binary-fork model with the atomic instruction Compare and Swap (CAS). CAS is an atomic instruction that takes 3 parameters: *address, oldValue, newValue*. If the value stored at *address* matches *oldValue*, the value at *address* will atomically update to $newValue$ and the instruction succeeds. Otherwise, no updates are made

to *address* and the instruction fails. CAS has an expected time of $O(1)$. CAS is a realistic capability to add to the binary-fork model since many Instruction Set Architectures support this (e.g. x86, ARM64)[21, 3].

This model also assumes a randomized work-stealing scheduler is used, which has been shown to have strong theoretical bounds and good performance in practice [4, 13].

Work

Work is the total number of instructions executed. Let p denote the number of processors available and let T_1 be the total time to execute instructions on a single processor. The Work Law can summarized as: $T_p \geq T_1/p$.

Span

Span is longest dependency chain within the DAG. The span indicates the minimum time required to execute a program even with infinite processors.

3.4 Parallel Primitives

This set of parallel algorithms and instructions (referred to as primitives) is used as the building blocks for more interesting algorithms shown in the paper.

3.4.1 Scan

Given an input sequence A of size n and an output sequence B of size n, scan computes $B[i] \leftarrow A[i] + B[i-1]$. Scan can be computed in parallel with $O(n)$ work and $O(\log n)$ span using the upsweep-downsweep method proposed by Blelloch[8]. Although

scan works with any associative binary operation, it is assumed to only do addition within this paper. By default, scan is inclusive. An exclusive scan will be labelled explicitly as ScanExclusive.

3.4.2 Pack

Given an input sequence A and a boolean array $flag$ of the same size, PACK returns a new sequence B where each element in B maps to a *true* value in $flag$. There's an alternative version called PACKINDEX that only takes in a boolean array $flag$ and returns the indices such that each index corresponds to a true value in $flag$. The latter is useful when data is represented as a bit vector. Both Pack and PackIndex have $O(n)$ work and $O(\log n)$ span.

3.4.3 Flatten

Given a nested sequence of k arrays with the head pointer called A, writes a single contiguous sequence into an output B, maintaining ordering of all elements in k and between indices $1 \dots k$. In other words, it "flattens" a 2D array into a single contiguous array, maintaining its order. This can be done in parallel in $O(n)$ work and $O(\log n)$ span.

3.4.4 WriteMin

Given a *address* and $newVal$, *writeMin* writes $newVal$ to *address* only if $newVal$ is strictly less than the current value stored in *address*. Internally, this uses CAS in a while loop which guarantees the minimum value even among competing threads. It has $O(1)$ time in expectation and has been shown to have good performance in practice [34].

Chapter 4

Graph Traversal Algorithms

This section describes the algorithms used before and after the preprocessing done in Section 5 to test performance and correctness. There are 4 primary algorithms here: Dijkstra's Algorithm, Parallel Bellman-Ford, δ-stepping, and ρ-stepping.

4.1 Dijkstra

Since all graphs in this work are assumed to be positively weighted, Dijkstra's algorithm is used as a the sequential baseline for performance. If implemented with a binary-heap, it has a work of $O(m \log n)$. This algorithm was chosen because it is simple to implement and quite fast in practice. The algorithm is described below.

Algorithm 1 Dijkstra

Input: $G = (V, E, w), s$

Output: $dist$ contains SSSP distances rooted at s

```
1: dist ← {∞...}
2: dist[s] ← 0
3: q ← {(0, s)}
4: while q do
5:     [d, u] ← q.extractMin
6:     q.pop
7:     if dist[u] < d then
8:         continue
9:     for v ∈ N(u) do
10:        if dist[v] > dist[u] + w(v) then
11:            dist[v] ← dist[u] + w(v)
12:            q.push((dist[v], v))
```

4.2 Parallel Bellman-Ford

Here will cover all portions of our Bellman-Ford implementation.

4.2.1 Main Method

Algorithm 2 describes the full Bellman-Ford algorithm. Note that we implement it as the hybrid approach first described by [5].

Algorithm 2 Parallel Bellman-Ford

Input: $frontier, frontierSize, G = (V, E, w)$

Output: Write SSSP distances from s into $dist$

1: $sparseMode \leftarrow true$

2: $frontier \leftarrow \{s\}$

3: $inFrontier, outFrontier, visited \leftarrow$ empty array of size n

4: $size \leftarrow 1$

5: **while** $size > 0$ **do**

6: **if** $size \leq$ SparseThreshold **then**

7: **if** not $sparseMode$ **then** $frontier \leftarrow$denseToSparse(frontier)

8: $size \leftarrow$ nextFrontier$(G = (V, E, w), frontier, frontierSize, dist, visited)$

9: **else**

10: **if** $sparseMode$ **then** $frontier \leftarrow$ sparseToDense$(frontier)$

11: $size \leftarrow$ nextFrontierDense$(G = (V, E, w), frontier, dist, inFrontier, outFrontier)$

Algorithm 2 is described here. In lines 1 and 2, we set $sparseMode$ to true and $frontier$ initially contains only the source s. $sparseMode$ is boolean that indicates if the current iteration of Bellman-Ford has sparse frontiers or dense frontiers. The remaining lines 3 - 9 are in the same while loop. We run either Algorithm 3 if frontier size $<$ predetermined threshold, or we run Algorithm 4. We switch frontier modes $sparse \leftrightarrow dense$ as necessary. The work of this algorithm is $O(diam(G)m)$ and the span is $O(diam(G) \log n)$ [16].

4.2.2 Sparse Frontier

This algorithm describes the sparse frontier generation given a sparse frontier.

Algorithm 3 Sparse next frontier

Input: $G = (V, E, w), frontier, frontierSize, dist, visited$

Output: next sparse frontier as a single array

1: $ranges \leftarrow frontierSize$ nested arrays

2: **parallel for** $u \in frontier$ **do**

3: $flag \leftarrow$ boolean array of size $deg(u)$, initialized to false

4: **parallel for** $v \in N(u)$ **do**

5: **if** WRITEMIN($\&dist[v], dist[u]+W(v)$) and CAS($\&visited[v], false, true$) **then**

6: $flag[v] \leftarrow true$

7: $ranges[k] \leftarrow$ PACK($N(u), flag$)

8: clear $visited$

9: $frontier \leftarrow$ FLATTEN($ranges$)

10: **return** size of $frontier$

We describe Algorithm 3 here. In line 1, we allocate space for $ranges$, which contains n nested arrays. Each $range[i]$ indicates is an array to indicate which vertices i visited this round. Then in line 2, we have the main parallel for loop over each frontier node. We temporarily allocate space for a local flag array called $flag$ of length $deg(u)$. Each index in $flag$ corresponds indicates a successful visit for vertex u. Initially, $flag$ is cleared since nothing has been visited yet. In lines 4-6, we have another loop that iterates over $N(u)$. If u successfully visits v, we set the $flag[v]$ to true. A successful visit indicates

that v now has the minimum distance possible from the all frontier nodes and it is only added to the next frontier one time. In line 7, we finish the inner for loop and PACK the neighbors of u based of the flag array. Doing this causes each inner array in *ranges* to only contain the vertices that were successfully visited. In line 8 we clear the *visited* array. This is unnecessary for unweighted graphs, but required for weighted graphs since it's possible to revisit the same node multiple times in different frontiers. Note that to clear the *visited* array, only $\sum_{i=0}^{frontierSize} deg(frontier[i])$ vertices must be written to clear the array, not necessarily n of them. The last step is to FLATTEN(*ranges*) in line 9. This collects each range into a contiguous array, which contains the frontier for the next round. All elements are guaranteed to be unique.

4.2.3 Dense Frontier

Algorithm 4 shows the generation of the next frontier in dense-mode.

Algorithm 4 Dense next frontier

Input: $G = (V, E, w), dist, inFrontier, outFrontier$

Output: $inFrontier$ holds the next dense frontier

1: **parallel for** $v \in V$ **do**

2: **if** $inFrontier[v]$ **then**

3: $inFrontier[v] \leftarrow false$

4: **for** $u \in N(v)$ **do**

5: **if** WRITEMIN$(\&dist[u], dist[v] + w(u))$ **then**

6: $outFrontier[u] \leftarrow true$

7: SWAP$(inFrontier, outFrontier)$

8: **return** COUNT$(inFrontier, true)$

We describe Algorithm 4 here. Nearly all of it is done within a single parallel for loop in lines 1 - 6. First we iterate over all vertices in V in line 1. This is because $inFrontier$ is a dense $flag$ array, so all elements have to be visited to know which are the current frontier nodes. In lines 2 and 3, if we detect that the current vertex v is a frontier node, we clear that index in $inFrontier[v]$ and proceed. In lines 4-6, we look at all neighbors of v and attempt to visit each one. When we successfully visit a node u, we relax $dist[u]$ with $dist[v] + w(u)$ and update $outFrontier[u]$ to true. Finally, we SWAP$(inFrontier, outFrontier)$ so that $inFrontier$ holds the next dense frontier and $outFrontier$ is now a cleared array. Lastly we count the number of visited elements in $inFrontier$ and return this value.

4.2.4 A Note on Bellman-Ford vs BFS

If the *visited* array and related CAS calls are removed, actually the behavior of Algorithm 2 matches the behavior of a Parallel BFS for unweighted graphs. Every vertex is visited only a single time, and all vertices within a frontier are 1-hop further than their predecessors. We know this because WriteMin functions as a CAS since each node in the same frontier is the same distance away from its neighbors. However, looking at other literature, a positive integral-weighted BFS is commonly implemented with a bucket based approach (similar to δ-stepping) [33, 15]. So we mainly refer to Algorithm 2 as Bellman-Ford, although its bounds on unweighted graphs are actually tighter than a weighted graph.

4.3 Delta-Stepping

The high level idea in δ-stepping is to relax vertices close to the source (within a certain δ). All of these "close" vertices can be relaxed in parallel using Bellman-Ford. It achieves this by distinguishing edges between light edges ($w(e) \leq \delta$) and heavy edges ($w(e) > \delta$). The idea is that after relaxing a light edge, it may get reinserted into another bucket, thus potentially requiring another relaxation. Once all light edges are processed, relaxing a heavy edge is guaranteed to not cause a reinsertion into the same bucket. For more detail on δ-stepping, consult [28].

4.4 Rho-Stepping

ρ-stepping is similar to δ-stepping since it also follows the same stepping framework. For ρ-stepping, the key idea is to extract the ρ nearest vertices in the current frontier.

Once these vertices are found, they can be relaxed in parallel and successful visitors are pushed to the next frontier. For more detail, see [18].

Chapter 5

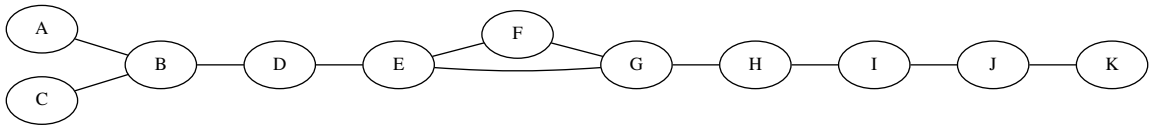
Preprocessing Algorithms

Since the span of a parallel Bellman-Ford is bounded as $O(diam(G) \log n)$, the main proposal of this work is to simply decrease $diam(G)$ to achieve better performance in practice. To do this, we explore two different preprocessing techniques. A high level description and diagram is provided for both, then detailed analysis is discussed.

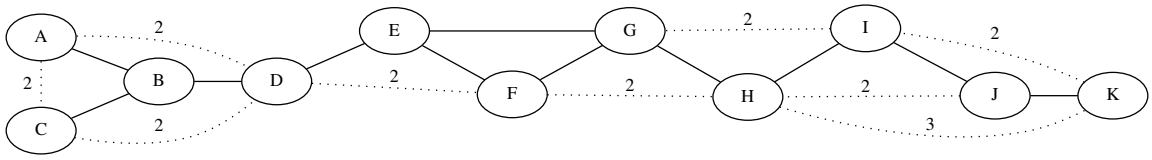
5.1 K-nearest Edges

The first approach is to increase the level of parallelism available by potentially adding shortcuts to each vertex. There is a tradeoff - we're explicitly adding more work, which can take longer to execute, but we also are shortening total rounds required due to increased parallelism.

Figure 5.1 depicts the transformation of an input graph when k=3 after running Algorithm 5.



(a) Original graph.



(b) After running KNE with k=3

Figure 5.1: Illustrated Example of KNE. Solid lines have weight 1, dotted lines are new shortcuts. (a): Original Graph. (b): Add shortcuts with $k = 3$.

Algorithm 5 K-nearest edges

Input: $G = (V, E, w), k$

Output: G' with shortcuts

1: $E' \leftarrow n$ nested arrays

2: **parallel for** $v \in V$ **do**

3: $kDist \leftarrow$ Run BFS from v, but stop when there is k total edges.

4: **for each** $(node, weight) \in kDist$ **do**

5: **if** $weight > 1$ **then**

6: Append $(v, node, weight)$ to $E'[v]$

7: $E' \leftarrow \textsc{Flatten}(E')$

8: Add duplicate edges to E' such that there is a (v, u, d) for each (u, v, d)

9: Combine E' with E and remove redundant edges, adjust V', E', w' as needed.

10: **return** $G' = (V', E', w')$

We describe Algorithm 5 here. We initialize n nested arrays to hold future shortcuts in line 1. In line 2, we iterate over each vertex in parallel. In line 3, we run BFS to find the up to k nearest neighbors. We return a map where the key is the node and the value is distance to that node from the source v. Note that we consider any immediate neighbors of v as part of the k count, so any vertices that already have $\geq k$ neighbors return an empty map. Additionally, we only store distances > 1. We store the result in $kDist$. In lines 4 - 6 we update $E'[v]$ with a new shortcut if the distance is > 0. In line 7, we flatten our nested shortcuts into a single contiguous array. Then in line 8 we duplicate each edge in E' such that there is a $(v, u, d)|(u, v, d) \in E'$. Lastly in line 9 we combine E' with E and remove any redundant edges.

5.1.1 Correctness

To show this algorithm is correct we need to show 2 things:

1. We must show that the shortcuts generated are valid shortcuts, i.e. that each edge $(u, v, d) \in E'$ contains the minimum weighted-path from u to v in E.

2. We must show that running SSSP with the same source s in G' leads to the equivalent distances in G.

Lemma 5 *After the completion of Algorithm 5, any $d|(u, v, d) \in E'$ is the minimum weighted path between $u \to v \in E$.*

Proof. Pick an arbitrary pair of vertices in G, say they are u and v. Let k be a sufficiently large integer. Let $d(u, v)$ be the minimum path distance between u and v in G. Consider the distance between $u \to v$ in G', denoted as $d'(u, v)$. We will show that

$d(u,v) = d'(u,v)$. Assume to the contrary that $d(u,v) \neq d'(u,v)$. Any edges in G' are either of weight 1 or weight > 1. Then let's go thorugh each scenario, $d(u,v) = 1, d'(u,v) > 1$. But this is not possible, since $d'(u,v) > 1$ implies a shortcut was constructed. But if such a shortcut did exist, it would have been removed in line 9. $d(u,v) > 1, d'(u,v) = 1$. But this is also not possible. if $d(u,v) = 1$ after the call to remove redundant edges, that can only mean that there was not shortcut generated at all and $d'(u,v)$ is from the original E. Since we know k is sufficiently large, then the BFS would have generated the correct distance. And the last case if $d(u,v) > 1, d'(u,v) > 1$. We know that for sure this will got to line 6, where the distances originate from the BFS. Since BFS is valid on unweighted graphs, the distance values generated must be correct. Since all cases are impossible, we know that the assumption is false. Therefore, $\forall d | (u,v,d) \in E'$ are equivalent to minimum path distances from $u \to v$. ■

Theorem 6 *Running a SSSP algorithm in G' from source s has equivalent distance values to running SSSP in G from source s.*

Proof. Pick any source $s \in V$ from G. Let the SSSP distances tree rooted at s be *dist* for G and *dist'* for G'. Pick any non-source node $x \in V$ where $x \neq s$. SSSP will calculate the shortest path distance from s to all other nodes in G', including x. We proved in Lemma 5 that each $(u,v,d) \in E'$ consists of the minimum-weighted path distances from $u \to v$. Thus, by the validity of a SSSP algorithm, the calculated shortest path distance to from $s \to x$ in G' must be equal to the shortest path distance from $s \to x$ in G. ■

5.1.2 Cost Analysis

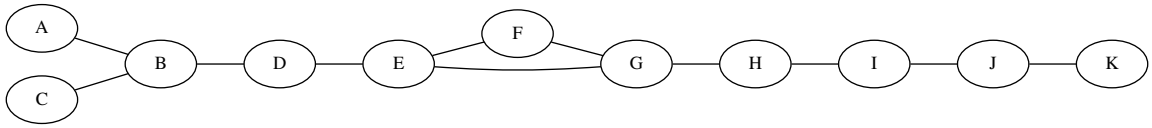
Theorem 7 *The KNE algorithm can be executed with $O(nk + m)$ work in expectation and $O(\log n)$ span w.h.p.*

Proof. Within the parallel for loop (lines 3 to 12), we operate over each vertex in parallel. Within a vertex we look for process up to k nodes, making that individual BFS with a work of O(k). The rest of the operations from lines 6 - 12 have k work since they operate with that many elements in a single pass. In total, that parallel loop has a work of $O(nk)$ and a span of $O(\log n)$. Both FLATTEN calls have a work of $O(nk)$ and span of $O(\log n)$. Adding duplicate edges is also linear work and $\log n$ span. Removing redundant edges can be done by filtering empty values and then inserting values into a parallel hash table, only keeping minimum weights when updating the same key entry of (u, v). It costs $O(m)$ work in expectation and $O(\log n)$ span w.h.p. to insert m elements into a parallel hash table. ■

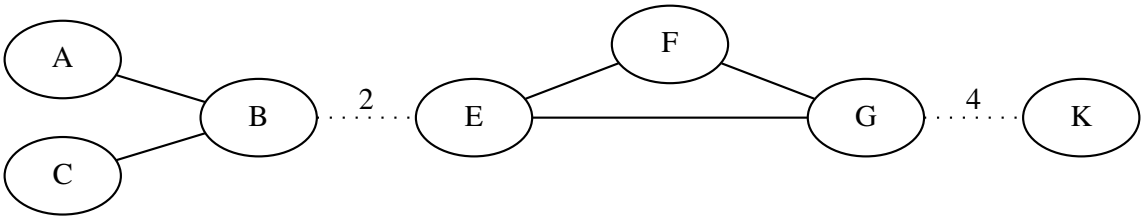
5.2 Contraction

In the previous section, we showed a purely additive approach - we only ever increased m. However, this approach may not always yield a better performance than the original graph. Essentially, there is a tradeoff between parallelism and work: larger frontiers enable greater parallelism, at the cost of more work (we show this in-depth in the Experimental Results). To account for this extra work, we want a method to achieve similar results in terms of parallelism, but does not necessarily create more edges. And thus, we have the next method, contraction.

The idea behind graph contraction is to replace $G = (V, E, w)$ with a $G' = (V', E', w')$ such that $\|V'\| + \|E'\| < \|V\| + \|E\|$. In our case, we have a simple contraction scheme. We remove chain vertices and add a shortcut between the endpoints of that chain. Refer to Figure 5.2 for a visual depiction.



(a) Original graph



(b) Contracted graph

Figure 5.2: Illustrated Example of Contraction. Solid lines have weight 1, dotted lines are shortcuts. (a): Original graph. (b): Contracted graph.

Algorithm 6 contracts the given graph by replacing chain vertices with a single weighted edge between them and storing necessary information for later postprocessing. In line 1, we allocate empty arrays of size n for *removed*, *recovery*, and *newEdge*. *removed* will hold vertices that are removed by contraction. *recovery* holds a pair of pairs, each contains an endpoint and distance to that endpoint. And *newEdge* contains the newly computed edge to add. Lines 2 - 10 contain the main parallel for loop over each vertex in V. We skip any vertex v that has already been processed. Otherwise, we continue if v is a chain vertex in line 4. In line 5 we look for endpoints of v. This can be done simply

Algorithm 6 Contraction

Input: $G = (V, E, w)$

Output: $G' = \{V', E', w'\}$

$removed, recovery, newEdge \leftarrow [n]$

parallel for $v \in V$ **do**

    if $v \in removed$ **then** continue

    if $deg(v) == 2$ **then**

        let $u1, u2$ be the nearest endpoints of v

        add each node with degree = 2 in path between $u1, u2$ to $removed$

        let $d1, d2$ contain the nearest distance from v to $u1, u2$ respectively

        $recovery[v] \leftarrow ((u1, d1), (u2, d2))$

        Traverse to the endpoints, setting end1,d1 and end2,d2 for all vertices in path

        $newEdge[v] \leftarrow (u1, u2, d1 + d2)$

Duplicate $newEdge$ such that each $u, v, d \in newEdge$ has a v, u, d

Remove empty and redundant edges from $newEdge$

Filter E such that are no longer any edges that belong to a vertex in $removed$

$E' \leftarrow$ Combine E and $newEdges$ then remove duplicates

Adjust V', w' as necessary

return $removed, recovery, G' = (V', E', w')$

with a BFS that only enqueues nodes that have not been visited and have a degree of 2. When processing a node that does not have a degree of 2, that is an endpoint for v. In this same traversal, we can keep track of all vertices visited in this path, later adding these to *removed* in line 6. We also set $recovery[v]$ to hold endpoint,distance for each endpoint pair in line 7. We do one final traversal to the same endpoints, this time setting the endpoint pairs for each node in the path (if not already set). In line 10, we set $newEdge[v]$ to contain this shortcut, which is from one endpoint to another, with the weight equal to the distance from $e1$ + the distance to $e2$. In line 11 we filter out any empty or redundant edges from the shortcuts we just made. In line 12, we filter out the original E such that there are no longer any $(u, v, d) \in E$ where $u \in removed$ or $v \in removed$. Then we combine the newly filtered E with the shortcuts generated $newEdges$, remove any redundant edges, and store in E' in 13. When removing redundant edges, we only keep the smallest possible weight for for any given (u, v, d). The last step is to adjust other parts of the graph as necessary, then return this new result as G' in line 15. An example of an adjustment would be updating offsets with new the vertex degrees if the graph is in CSR format.

5.2.1 Correctness

To prove the overall correctness of Algorithm 6, we need to show 3 things:

1. We only remove chain vertices.

2. There exists a shortcut in G' between the endpoints of all removed vertices.

3. *recovery* maintains the shortest path distance between endpoints for all vertices in *removed*.

Lemma 8 *After the completion of Algorithm 6, any shortcuts added to G' contain the minimum weighted path distance between its two vertices.*

Proof. Algorithm 6 only generates shortcuts in line 10, where we set $newEdge[v]$ to be $(u, v, d1 + d2)$. Since $d1$ and $d2$ are defined to be the nearest distance from v to the endpoints (line 7), we must be setting $newEdge[v]$ to the minimum weighted path distance since it is the sum of minimum weighted path from $v \to u1$ with the minimum weighted path from $v \to u2$. ■

Lemma 9 *For all $v \in removed$, there must be a valid shortcut in G' between the endpoints $e1 \to e2$, where $e1, e2$ are endpoints of v.*

Proof. For a vertex v to be in *removed*, it must have been added in line 6 of Algorithm 6. We know $e1, e2$ are the endpoints of v from line 5. Then in line 10 we create a shortcut from $e1 \to e2$. We proved earlier in Lemma 8 that any shortcut created is valid. Thus, we have proved all $v \in removed$ contain a valid shortcut between their respective endpoints. ■

Lemma 10 *For every vertex $v \in removed$, $recovery[v]$ contains the minimum weighted path distances to its endpoints.*

Proof. We add a node to removed only in line 6 of Algorithm 6. Note that we remove all vertices in that chain (line 6). In line 9, we set the endpoints and minimum weighted path distances to all vertices in that same chain. Therefore, after completion of the algorithm, all vertices in *removed* also contain the appropriate endpoint and minimum path data. ■

5.2.2 Cost Analysis

Theorem 11 *The work of the contraction algorithm is $O(nm)$ in expectation and the span is $O(\log m)$ w.h.p.*

Proof. First, we look at each vertex once, causing a parallel for loop with that much work. Within an iteration: we run BFS, but only for a limited chain length. We also try to not to repeat these by updating removed (although its actually possible to have multiple BFSs throughout a chain). In the worst case, we have a graph that looks something like a singular long chain of length n. In this case, the inner loop work is O(nm). Duplicate the edge in line 11 is O(m) work and $O(\log m)$ span. One way to remove redundant edges is to use a parallel hashtable, where inserting m elements has a work of $O(m)$ in expectation and span of $O(\log m)$ with high probability. In line 13, we filter m elements in E, which has linear work and $O(\log n)$ span. Thus the total work is $O(nm)$ in expectation and the total span is $O(\log m)$ w.h.p. ■

5.3 Recover Dists

After running contraction, certain vertices will be deleted and associated edges are removed. Thus, running any SSSP algorithm on this G' will not fully settle all distances, but only the ones reachable from the source that are not part of the *removed* set. However, recall that the contraction algorithm also stores the distances from a vertex to each of its endpoints in a format like $((e1, d1), (e2, e2))$, where $e1/e2$ are endpoints and $d1/d2$ are their respective distances. Using this information, we can settle all distances with Algorithm 7.

Given a contracted graph $G', s, removed, recovery$, recover dists settles all tenta-

tive distances in *dist*.

Algorithm 7 Recover Dists

Input: $G' = (V', E', w'), dists, recovery, removed, s$

Output: All values in *dists* are settled

1: **if** $s \in removed$ **then**

2: Run SSSP on G' with source s and initial frontier of two vertices, $\{e1, e2\}$. $e1, e2$ are equal to $recovery[s].end1, recovery[s].end2$ respectively. Set $dist[e1], dist[e2]$ to $recovery[s].dist1, recovery[s].dist2$ respectively.

3: **else**

4: Run SSSP on G' with source s. The initial frontier is s with $dist[s] \leftarrow 0$.

5: **parallel for** $v \in removed$ **do**

6: $d1 \leftarrow dist[recovery[v].end1] + recovery[v].dist1$

7: $d2 \leftarrow dist[recovery[v].end2] + recovery[v].dist2$

8: $dist[v] \leftarrow \text{MIN}(d1, d2)$

5.3.1 Correctness

To show correctness of Algorithm 7, we must show 3 things:

1. Running SSSP with a source in removed will settle all non-removed vertices.
2. Running SSSP with a source not in removed will settle all non-removed vertices.
3. After the completion of Recover Dists, the final distance array is settled.

Lemma 12 *Any vertex not in removed only has neighbors that are also not removed.*

Proof. Start with a node x not in removed. Assume to the contrary, that there exists a

$y \in N(x)$ that is removed. We know from Lemma 9 that there must be a valid shortcut between $x \to y$ with distance greater than 1. However, it's not possible for y to have a distance greater than 1 because a removed vertex is cut and has a degree of 0. Thus by contradiction we have proved all neighbors of vertex not in removed are also not in removed.

■

Lemma 13 *Running SSSP on any source vertex $s \notin$ removed will settle all other vertices not removed.*

Proof. Given a source node s not in removed, we want to prove SSSP will settle all other vertices not removed. This proof proceeds by induction. The base case is true since we know that the source is not removed and the SSSP distance from the source is always 0. Assume there is a node x that is also not removed and is settled. We want to show all $N(x)$ are not removed and can be settled from SSSP. From Lemma 12, we know all $y \in N(x)$ must not be removed. Then by definition, $deg(y) > 0$. We know $dist[x]$ is settled. Due to the correctness of SSSP algorithms, we know SSSP from the source x must also settle all $N(x)$ since they are reachable from x. Thus, the claim holds for $N(x)$ and we have proved that all non-removed vertices can be settled with SSSP with induction. ■

Lemma 14 *Given a source node s that is in removed, we can SSSP on s and settle s and the non-removed nodes by first setting the initial frontier equal to $recovery[s].end1, recovery[s].end2$ and setting distances to the endpoints with $recovery[s].dist1$ and $recovery[s].dist2$ respectively.*

Proof. We begin with a removed source vertex s. Assume we initialize the first frontier within the SSSP algorithm to have 2 endpoints for s, denoted as $e1$ and

$e2$, found in $recovery[s].end1$ and $recovery[s].end2$. Also assume we set $dist[end1]$ to $recovery[s].dist1$ and $dist[end2]$ to $recovery[s].dist2$. We will denote $recover[s].dist1$ as $d1$ and $recovery[s].dist2$ as $d2$ from now on. $end1$ and $end2$ are not in removed because because we only remove vertices with degree 2, and by definition an endpoint can not have a degree of 2. Furthermore, $dist[end1]$ and $dist[end2]$ must be settled. From Lemma 10 we know that $d1$ and $d2$ are the minimum path distances from s. Then, setting $dist[end1] \leftarrow d1$ and $dist[end2] \leftarrow d2$ are obviously the final distances, since the source starts with a distance of 0 and there are no smaller paths to $end1$ and $end2$. Since each vertex in this frontier of $end1, end2$ is not-removed and settled, we know that the remaining not-removed vertices can be settled with Lemma 13. ■

Lemma 15 *Any removed vertices are settled after running Algorithm 7.*

Proof. From Lemma 10, we know that any vertex $v \in removed$ contains a corresponding index in $recovery$ with endpoints $e1, e2$ and the minimum distances from v to $e1, e2$ as $d1, d2$ respectively. Additionally, we know that both $dist[e1]$ and $dist[e2]$ must be settled according to Lemma 13 and Lemma 14. Then setting $dist[v]$ to be the minimum between $dist[recovery[v].end1] + recovery[v].dist1$ and $dist[recovery[v].end2] + recovery[v].dist2$ must settle v. ■

5.3.2 Cost Analysis

Theorem 16 *Settling removed vertices has a work of $O(n)$ and a span of $O(\log n)$.*

Proof. There is a single parallel for loop iterating over at most n elements. Each statement inside this loop (lines 6-8) are all constant time operations. Therefore, the total

work and span are simply based off the time to process the for loop in parallel, which is $O(n)$ work and $O(\log n)$ span. ■

Chapter 6

Implementation and Optimizations

This chapter outlines how several key algorithms were implemented, tested, and optimized.

6.1 Bellman-Ford

This section describes key implementation details for the BF algorithm.

6.1.1 Local Updates First

In both sparse and dense relaxation, we actually do a local update to the frontier node with its minimum tentative neighbor distances. We found that this has a definite impact on the total number of rounds required, since it reduces contention for WriteMin later on.

6.1.2 No Visited Array

We originally maintained a visited array (as described in Algorithm 3), but after further experimentation, we found that maintaining this visited array is actually expensive in terms of runtime. This is because we have to clear portions of the visited array after each frontier is generated. Overall the time adds up since there are many rounds of execution in sparse graphs. We ended up implementing that algorithm without the visited array, but of course we end up having slightly larger frontiers as they may have redundant vertices in them. After looking into it, there is typically < 100 repeated vertices, even with large frontier sizes.

6.2 Container Choice

We actually spent some time to determine what container choices have the best performance for out datasets. We tested with raw C-style arrays, Parlay sequences of integer types different integer types, and also atomic arrays and sequences. Because of the convenience of many functions and primitives provided by ParlayLib, we mainly utilize Sequences.

6.3 Coarsening

Coarsening is definitely something we experimented with since it can have pretty drastic results. If the parallelism is too fine, there is not sufficient work to do per thread, and there ends up being excess overhead due to scheduling and synchronization. However, if the parallelism is too coarse, then there may not be sufficient parallelism to achieve optimal

results. In reality, we simple experimented by setting several parameters and seeing the performance result after tuning them one at a time. For coarsening, we have this in several forms. Within sparse relax, we let the inner loop be a sequential for loop if the number of neighbors is less than some parameter, which we set to 10000.

6.4 Delayed Sequencing

Whenever possible, we implemented delayed sequencing to reduce the memory overhead of large arrays. Delayed sequencing is passing a function that returns a value instead of passing an actual array that returns the value.

6.5 Contraction

We actually experimented with 2 versions of the contraction algorithm. One is more aggressive - any possible chain vertex was cut. The other is less aggressive - only chains with length ≥ 5 were cut. After some testing, we found that the less aggressive version has a better performance on sparse graphs. We think it's because with very aggressive shortcutting, it's somehow possible to prematurely relax a vertex. By only focusing on certain chain lengths, it helps limit the amount of redundant work. Unfortunately, there is still a lot of revisited vertices in Bellman-Ford, as can be seen in the next chapter.

Chapter 7

Experimental Results

This chapter covers experimental setup, input data, and the analysis of preprocessing metadata and runtime performance.

7.1 Experimental Setup

The machine used for all of the experiments is a 2.4 GHz 64-bit Intel Xeon machine with 32 cores and 64 hyperthreads. It has 125 GB of RAM and a L3 cache size of 24MB. Both sequential and parallel programs were compiled using G++ 8.5.0, with the flags -Ofast and -std=C++17. The program was implemented with ParlayLib [9] and uses their built-in scheduler and many of their primitives, such as $Pack$, $WriteMin$, and $Sort$. Note that any parallel framework that also follows the binary-fork join model should be able to attain comparable results. This includes OpenMP and Cilk+[24, 29].

Times reported are the median result from 10 runs. These times do not include memory allocation time, unless that is done within the sparse relax or dense relax function.

Time to convert frontiers (ie. sparse $\leftrightarrow$ dense) is also included, although this typically has a negligible impact on time. Lastly, all contracted graphs include the RecoverDist time since it is required to settle all distances. This is more or less a negligible time increase.

7.1.1 3rd-Party Algorithm Utilization

To show more robust results, we utilize several different Parallel SSSP algorithm implementations. Table 7.1 shows the algorithm name, what we denote the implementation as in the data, and where this implementation comes from.

Algorithm	Denoted As	Source
Bellman-Ford	BF	Ours
Bellman-Ford	BF*	Parallel-SSSP
Delta-Stepping	DELTA*	Parallel-SSSP
Rho-Stepping	RHO*	Parallel-SSSP

Table 7.1: Overview of Parallel SSSP Algorithms used in Testing

We use them as-is with no attempt to meaningfully alter the code. We only add a way to extract metrics such as frontier size and runtime. But that is not included in the reported time, so it is a faithful implementation of their work and results should definitely be comparable. The source code for all Parallel-SSSP algorithms can be found here.

7.1.2 Input Graphs

We use four input graphs for testing. Orkut (OK) and LiveJournal (LJ) are both social media networks and thus are quite dense. The other two graphs are RoadUSA (US) and RoadGermany (GE), both of which have a small average degree of 2. We picked these 4 graphs because they all have sufficiently large n and m while maintaining different

properties, enabling a greater range of testing. Note that we explicitly ignore any given edge weights since we apply our own transformations later. We obtained both US and GE datasets from the Network Repository [30] and obtained OK and LJ from SNAP [25]. We show the following table depicting some key metrics for these datasets.

Dataset	**n**	**m**	**Average Degree**	**Max Degree**
US	23947347	5770862	2.4098	9
GE	12277375	32266600	2.6281	13
LJ	4846609	85702474	17.6795	20333
OK	3072441	234370166	76.2768	33313

Table 7.2: Original Dataset Metadata

The leftmost column identifies the dataset. Columns n and m show the number of vertices and edges respectively. Average Degree shows the average degree of a vertex. And Max Degree shows the largest degree of any vertex in that dataset.

7.2 K-nearest Edges Results

In this section we analyze data obtained for KNE. We start with how varying values of k affect the datasets themselves and then look into performance metrics.

Table 7.3 shows metadata information of the input graphs after running KNE when k = 2 and k = 10. The remaining data values for k are skipped for brevity here. We can see that increasing k leads to an increase in m and the average degree. For sparse graphs like the US and GE datasets, there is a dramatic increase in edges (4.36x and 4.12x respectively). On the other hand, there is almost no increase to edges in the OK dataset. Since that dataset has a high average degree initially, only setting the k up to 10 will have

a limited effect. If we set k higher, we would see similar results.

graph	k	n	m	avg_degree	max_degree	time
US	2	23947347	67023442	2.7988	12	2.819
US	10	23947347	292800584	12.2268	52	15.7446
GE	2	12277375	36065736	2.9376	14	1.7574
GE	10	12277375	148840202	12.1231	59	8.089
LJ	2	4846609	87815562	18.1154	25242	4.0469
LJ	10	4846609	127024250	26.2037	54295	5.8278
OK	2	3072441	234505700	76.3209	33313	13.6461
OK	10	3072441	238376554	77.5807	51927	14.0851

Table 7.3: Metadata for Input Graphs when k = 2 and k = 10

Figure 7.1 shows the total number of edges after running Algorithm 5 with specific values of k. We can see a strong correlation between the the total number of edges in the graph and construction time. We can note that the US dataset has the most aggressive growth. GE also grows consistently, but just not at the rate that US does. We can also see LJ has a consistently small growth compared to the road networks. Notice that there is almost no growth in the OK dataset. This is because the values of k chosen were not large enough to cause major changes in m, since KNE will not attempt to generate shortcuts on a vertex that it determines to be dense. Recall from Table 7.3 that the average degree increased only by about 1 for the OK dataset, which is far lower than all other graphs.

Figure 7.2 depicts how increasing values of k affects runtime for road networks. We can see that there are not many consistent patterns, so it is hard to draw conclusions from the figure alone. For example, if we focus on the bottom row (US), we can see that increasing k has a consistent increase in time for the BF* and DELTA* algorithms. However, those

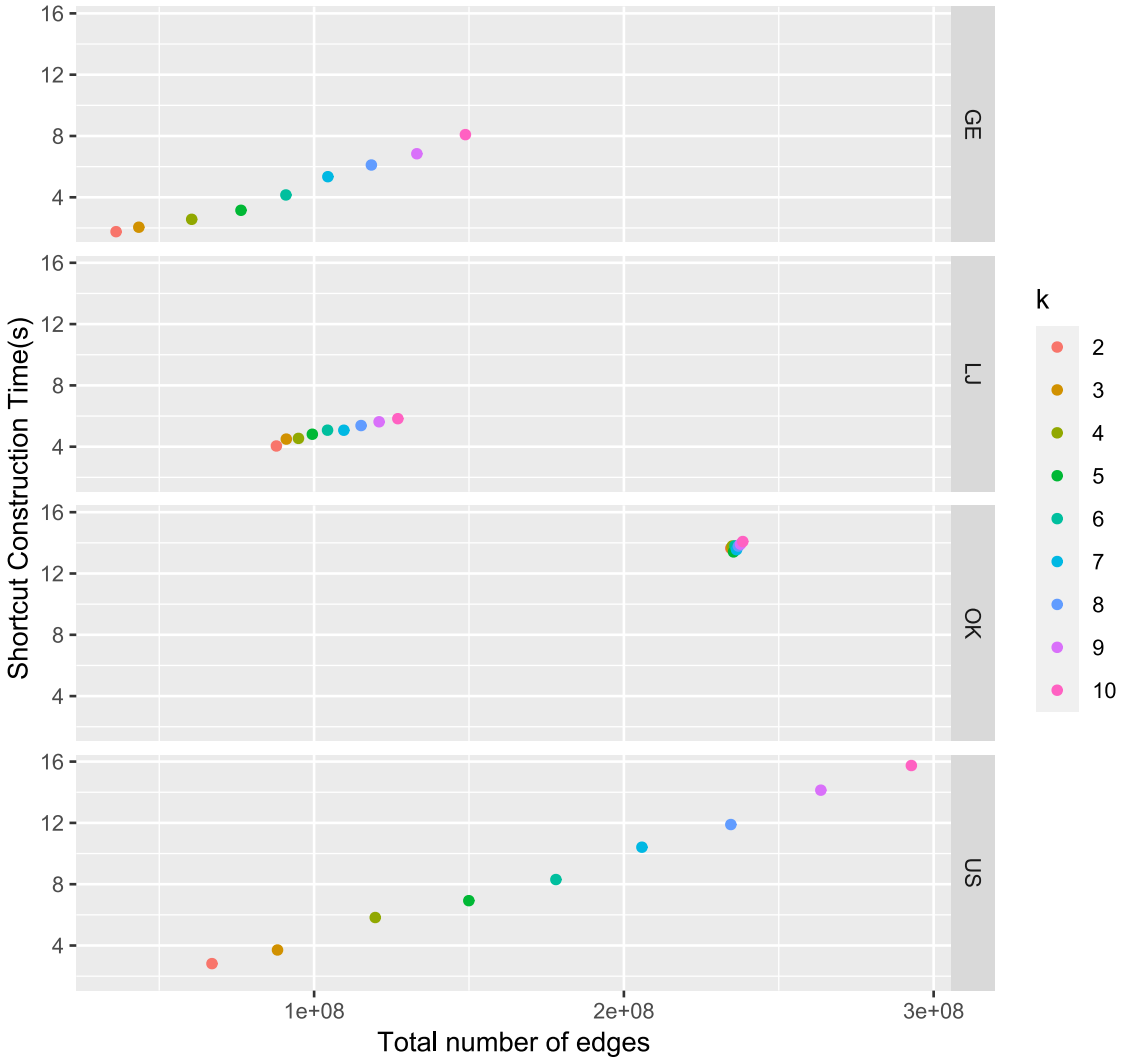


Figure 7.1: Shortcut Construction Time vs Edges

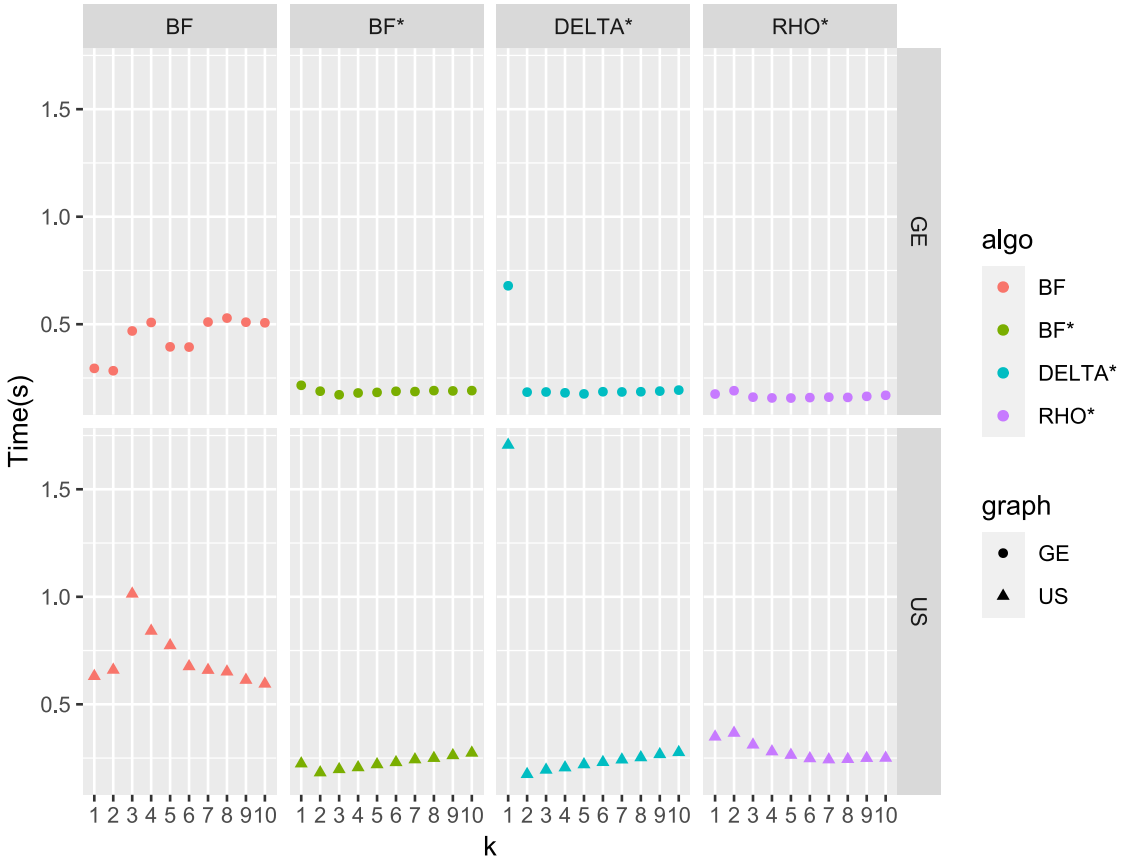


Figure 7.2: Execution Time vs k for Road Networks

same algorithms show minimal change for GE dataset, which is the most similar dataset to US in terms of density. BF also has a very different pattern between GE and US. In US, BF has something like a normal curve in terms of timing. Having a small k has a small time, and then having a large k also has a small time. but values in the middle increase the time. On the other hand, the same algorithm shows not the same pattern on GE. In GE, we can see there's not a clear shape of timing execution.

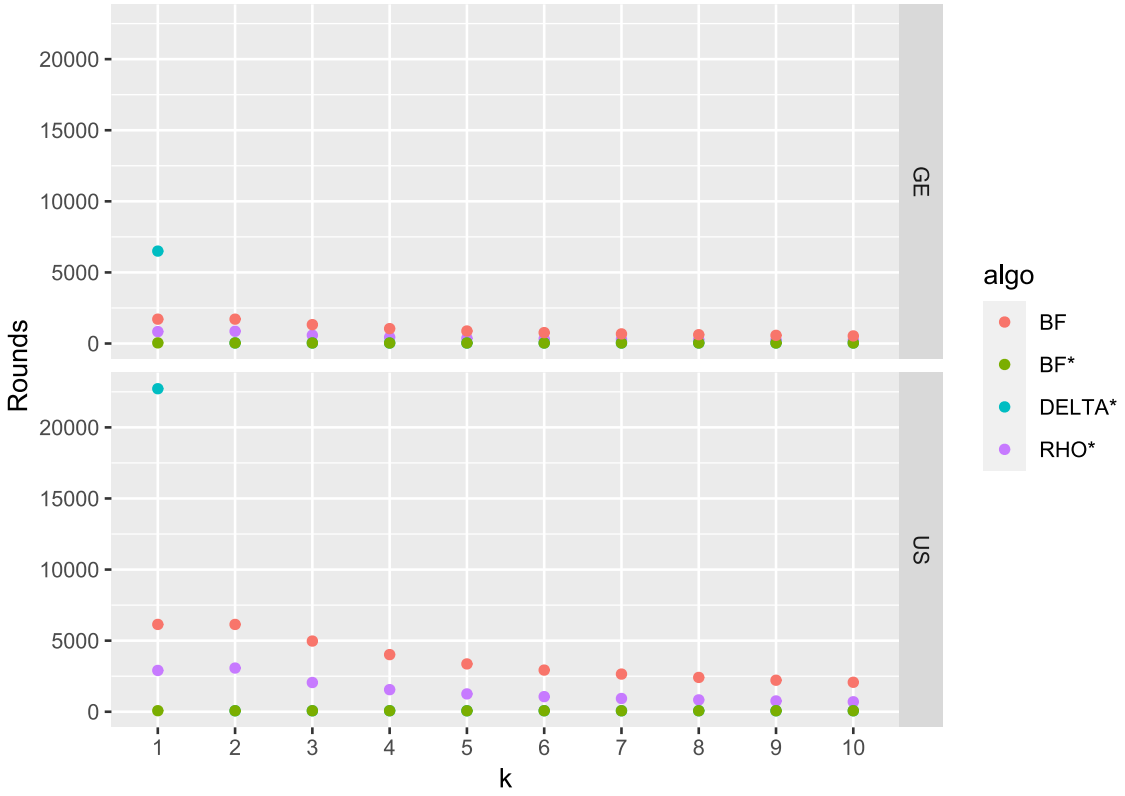


Figure 7.3: Total Rounds vs k for Road Networks

We can see from Figure 7.3 that in both graphs, we can see a clear reduction in total number of rounds. The pattern is stronger in US compared GE dataset, but it is still there. BF and ρ-stepping show the largest overall reduction in rounds for both graphs.

While BF* does not show much change in terms of rounds, all other algorithms show a clear progression as k increases.

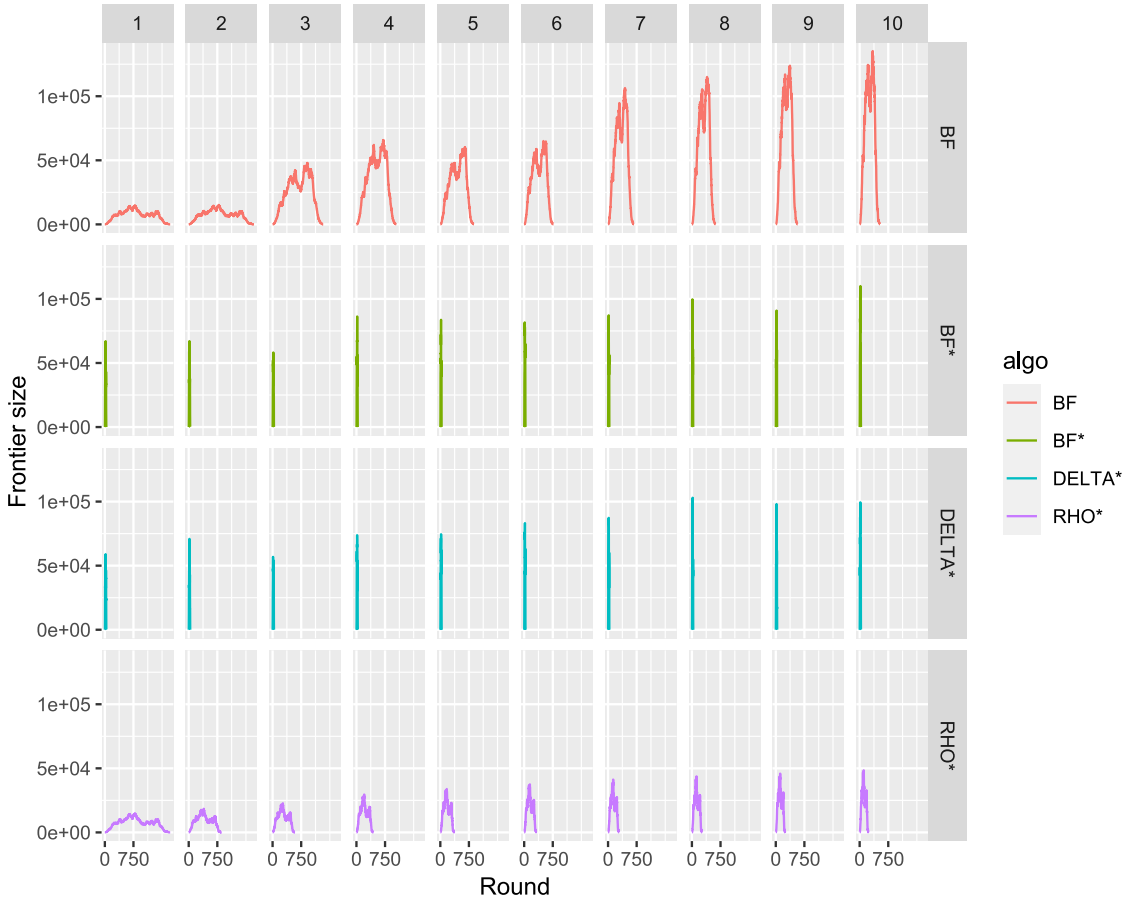


Figure 7.4: Frontier Size each Round for GE

Figure 7.4 shows the frontier size during each round of execution for each value for k for the GE dataset. Only GE is shown for brevity. The top value in each column shows the value of k picked in that run. In this case it varies from 1 to 10 (1 effectively means the original graph). Each row shows a different algorithm, labelled in the legend on the right. The most important thing is if we look across a single row, we can see how the overall frontiers shape transforms over k. BF is a great example of this. We can clearly see the

initial pattern is many small frontiers, which is a large number of rounds but with shallow depth. As we increase k, we can see the frontier curve consistently becomes sharper and the overall rounds decrease. ρ-stepping shows the same kind of pattern. It's not as noticeable for $BF*$ and δ-stepping. This is primarily because both $BF*$ and δ-stepping already show a frontier pattern that has minimal rounds with large frontiers. After increasing k, both of those algorithms maintain the same pattern.

7.3 Analysis of Contracted Graph Results

In this section we will look at results after running Algorithm 6. We begin with the metadata and then analyze the runtime statistics.

7.3.1 Contracted Graph metadata

Table 7.4 contains the metadata after contraction is executed.

Dataset	**n**	**m**	**Average Degree**	**Max Degree**
US	22710198	55208022	2.3054	9
GE	12065194	31800762	2.5902	13
LJ	4815184	85609014	17.6602	20333
OK	3071883	234368492	76.2763	33313

Table 7.4: Contracted Dataset Metadata

We can see from the above tables that after running contraction, all datasets have a reduction in the average degree. While there is little change in the number of edges for LJ and OK, about $2.5M$ edges are cut from US and about $450K$ edges are cut from GE.

7.3.2 Runtime Performance of Parallel Bellman-Ford

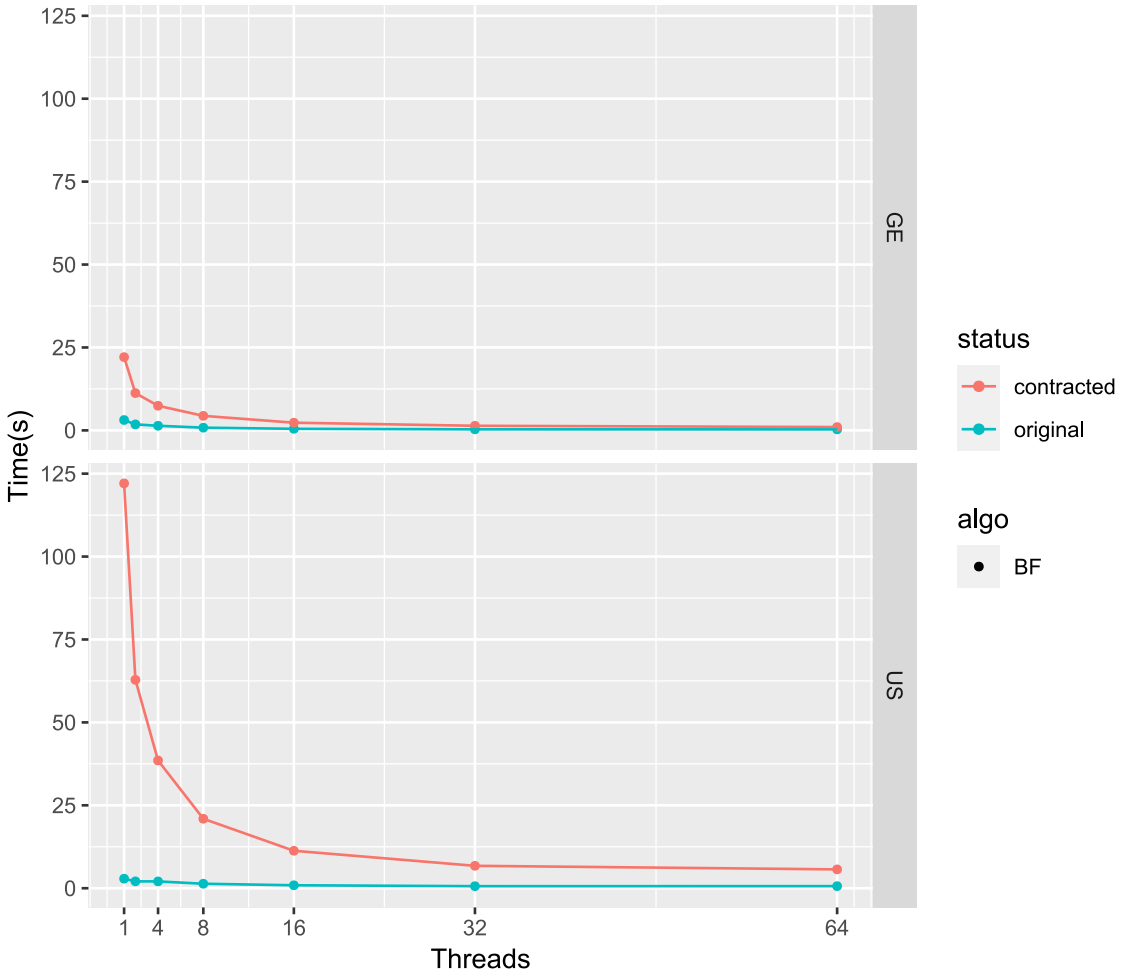


Figure 7.5: Execution Time vs Threads for BF on Road Networks

Figure 7.5 shows Time vs Thread Count for Bellman-Ford on the US and GE datasets. We directly compare the performance before and after contraction. We can see that in both datasets, the contracted graph never shows a better time than the original for BF. We can also see that the US dataset has very good parallelism after contraction - as we double the number of threads, we can see a time decrease by nearly half.

7.3.3 Runtime Performance of the State of the Art Algorithms

Non-trivial optimizations are required to see the benefits of the contraction algorithm on sparse graphs, such as road networks. As previously mentioned, Dong et. al. developed a stepping framework for parallel SSSP algorithms and opensourced their implementations of δ-stepping, ρ-stepping, and Bellman-Ford [18]. These are highly optimized implementations that rely on a novel abstract data type called a *Lazy Batched Priority Queue(LaB-PQ)*. The *LaB-PQ* is a priority queue that can batch multiple updates and queries in parallel. They also utilize the idea of *bucket fusion* introduced by [15], where they fuse multiple frontiers into a single frontier. We will see soon that it shows strong performance in practice.

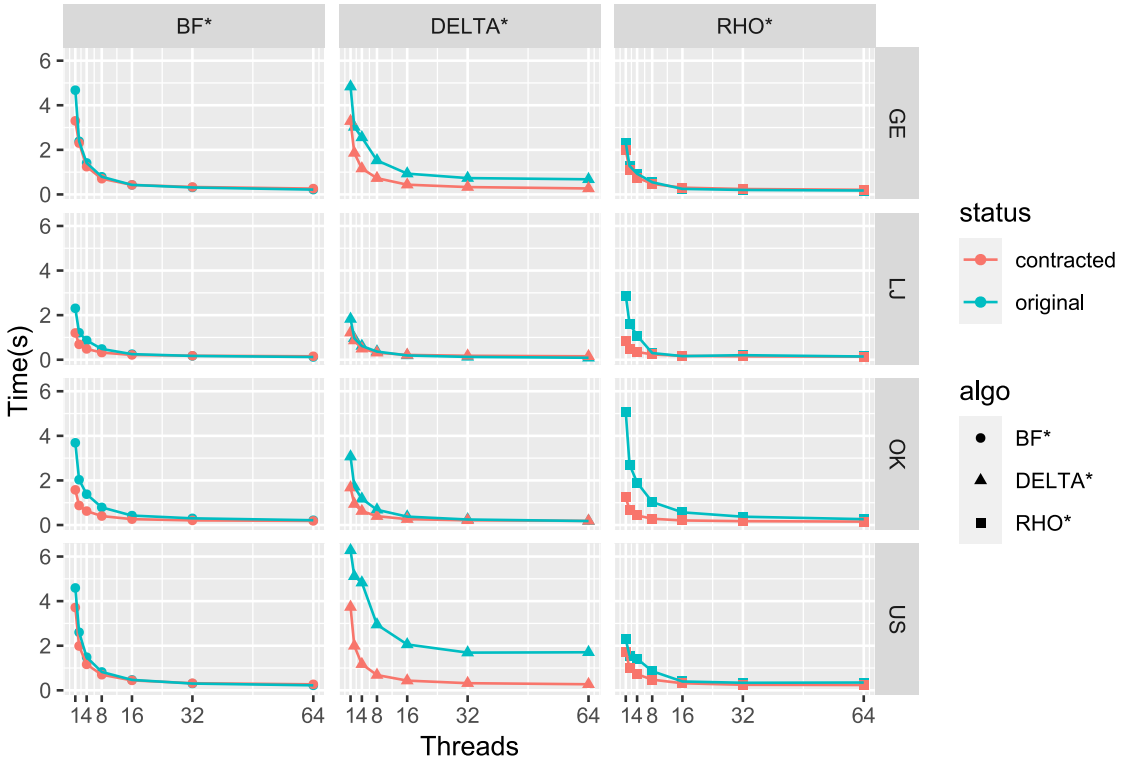


Figure 7.6: Execution Time vs Threads for Selected Algorithms

Figure 7.6 shows the the execution time in seconds compared to threads utilized for the BF*, DELTA*, and RHO* on all datasets. We directly compare the performance of the contracted graph vs the original as we overlay them in each sub-chart. If we look at the middle column for DELTA*, we can see overall the greatest amount of improvement after applying contraction. For both the GE and US datasets, DELTA* has better performance with every measured thread count. If we look at the column for RHO*, we can also see improvement for all datasets except GE. Surprisingly, RHO* shows the greatest overall improvement for LJ and OK, both of which are still quite dense after running contraction. In a similar fashion, BF* shows the most improvement for LJ and OK. With the exception of BF*, all algorithms are not worse after running contraction as evidenced by their runtimes.

Figure 7.7 shows the frontier size each round when run with 64 threads. Only the US dataset is shown here for simplicity, although GE behaves similarly. We can see DELTA* and RHO* both obtain a different structure after contraction. Instead of many small frontiers, they end up having a very short spike. For reference, RHO* goes from 2906 rounds to 489 rounds after contraction, and the max frontier size increases from 11408 to 19060. Similarly DELTA* goes from 22719 rounds to 73 rounds after contraction, and its max frontier size increases from 10470 to 68118. Interestingly, BF* is the only algorithm to show almost no change in frontier shape after contraction. Somehow, BF ends up doing lots of additional work after contraction. After contraction, BF goes from 6142 to 5558 rounds. But the max frontier size increases from 9410 to 244091. For BF, we can also see that the average frontier size balloons as well to 100383, compared to 3898. Despite a reduction in rounds, there is a far larger increase in work for BF, causing worse times overall.

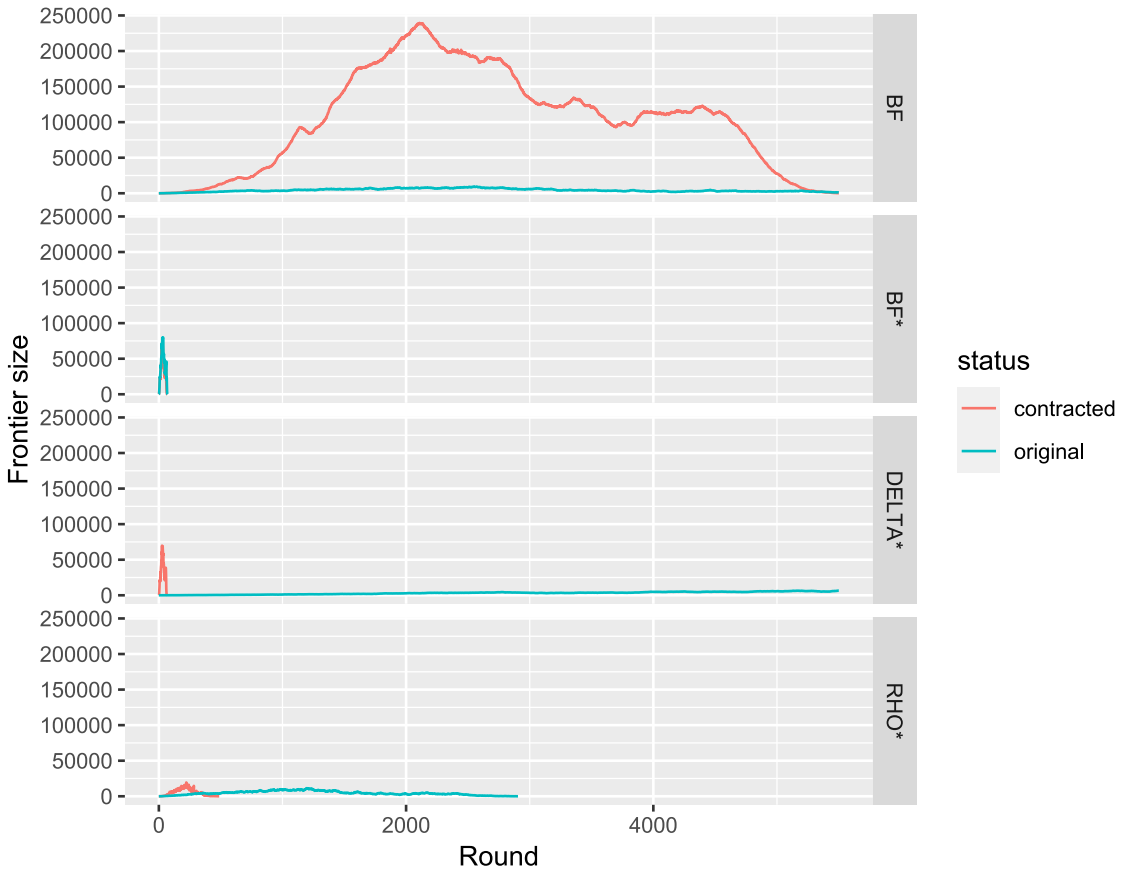


Figure 7.7: Frontier Size each Round in RoadUSA

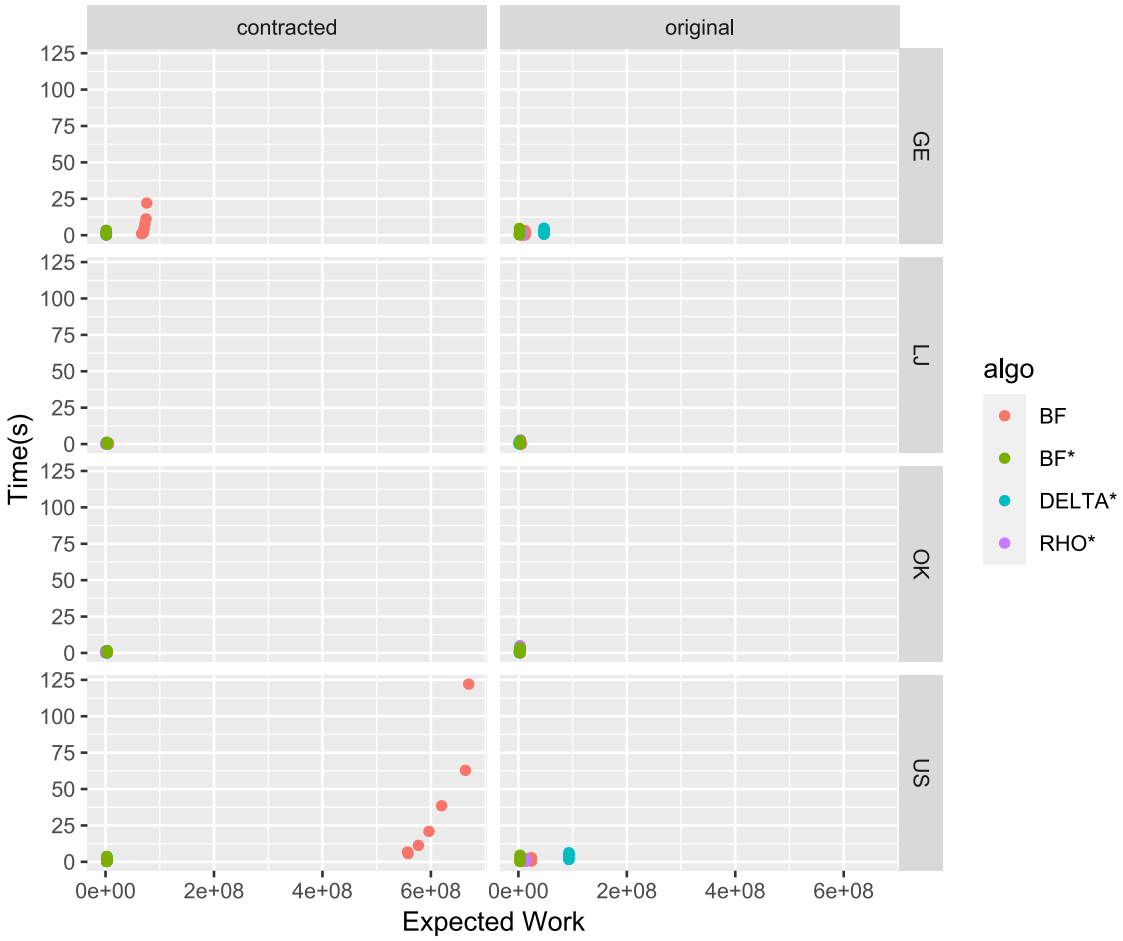


Figure 7.8: Runtime vs Expected Work

Figure 7.8 shows the execution time of all algorithms vs their expected works. We define expected work simply as the $\|averagefrontier\| * rounds$. Each row represents a different graph, and each column indicates if the graph is contracted or not. We can see from the middle rows that LJ and OK show no real difference after contraction. This is expected behavior since contraction does not cut many vertices and edges from these graphs. We can see 2 major patterns with regards to GE and US. First, the expected work for both DELTA* and RHO* decrease after contraction (although it is hard to see RHO* decrease). Conversely, the expected work for BF increases after contraction.

graph	status	algo	rounds	Average Frontier	Max Frontier	time	threads
GE	original	BF	1714	7162	14985	0.2947	64
GE	original	RHO*	837	7951	15944	0.175348	64
GE	original	DELTA*	6500	7264	14477	0.678846	64
GE	original	BF*	45	38743	76115	0.215973	64
GE	contracted	BF	1215	54673	147608	0.9981	64
GE	contracted	RHO*	179	8464	34416	0.208306	64
GE	contracted	DELTA*	33	40403	96912	0.267993	64
GE	contracted	BF*	35	34701	73212	0.262457	64
US	original	BF	6142	3898	9410	0.6306	64
US	original	RHO*	2906	4356	11408	0.348895	64
US	original	DELTA*	22719	4099	10470	1.70606	64
US	original	BF*	74	39246	84135	0.224052	64
US	contracted	BF	5558	100383	244091	5.6795	64
US	contracted	RHO*	489	4904	19060	0.237733	64
US	contracted	DELTA*	66	33641	68118	0.271372	64
US	contracted	BF*	66	34055	77662	0.26895	64

Table 7.5: Runtime Data for Road Networks with 64 Threads

Table 7.5 shows the raw performance metrics before and after contraction. Only data for the US and GE datasets with 64 threads is shown to conserve space. We can note several interesting observations. After contraction, all algorithms decrease in total rounds. DELTA* shows an increase in average frontier size after contraction by about 5.6x and 8.2x for GE and US respectively. RHO* shows different results, only increasing the average

frontier size by 1.06x and 1.12x for GE and US respectively. And BF* actually shows a reduction in average frontier size by 10.4% and 13.2% after contraction. Even though each of those 3 algorithms show a different pattern in average frontier size after contraction, it's not exactly indicative of runtime performance.

Chapter 8

Conclusions

SSSP is a fundamental question of interest within the network community. While sequential algorithms are well-studied with strong bounds, deterministic parallel algorithms often make a trade-off between work and span to achieve strong theoretical and practical performance. In particular, Parallel SSSP is a notoriously difficult problem, especially on certain input topologies like road networks. We noticed that while Parallel Bellman-Ford has strong performance on low-diameter dense graphs, it struggles to perform well with sparser networks. We theorized this is due to the span bound of $O(diam(g) \log n)$. To combat this, we proposed 2 different preprocessing algorithms. The first is k-nearest edges, which guarantees each vertex to have degree $\geq k$ after completion. The second is contraction, which removes specific chain vertices and replaces them with a shortcut from endpoint to endpoint. We implemented these preprocessing algorithms and benchmarked them by comparing the performance of various parallel SSSP algorithms, including δ-stepping, ρ-stepping, and 2 differing implementations of Bellman-Ford. We show for both preprocessing algorithms,

the shape of the frontiers drastically change for sparse graphs like road networks, especially for Bellman-Ford and ρ-stepping. For contraction in particular, we show improved time for δ-stepping and ρ-stepping. Future work could be trying different contraction strategies and comparing the runtime results. For example, we could maintain contraction chains of length at most k to see if this graph has interesting properties to explore. Additionally, another future work possibility is to be more selective with KNE and only attempt to add shortcuts to vertices that seem relatively sparse.

Bibliography

[1] Md Altaf-Ul-Amin, Yoko Shinbo, Kenji Mihara, Ken Kurokawa, and Shigehiko Kanaya. Development and implementation of an algorithm for detection of protein complexes in large interaction networks. *BMC Bioinformatics*, 7(1), 2006.

[2] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Parallel approximate undirected shortest paths via low hop emulators. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, page 322–335, New York, NY, USA, 2020. Association for Computing Machinery.

[3] ARM. Arm a64 instruction set architecture. `https://developer.arm.com/documentation/ddi0596/2020-12/Base-Instructions/CAS--CASA--CASAL--CASL--Compare-and-Swap-word-or-doubleword-in-memory-l`, 2020. Online; accessed 1 June 2023.

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, page 119–129, New York, NY, USA, 1998. Association for Computing Machinery.

[5] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.

[6] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[7] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time, 2022.

[8] Guy E. Blelloch. Prefix sums and their applications. 1990.

[9] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib - a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, page 507–509, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, page 89–102, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. Parallel shortest-paths using radius stepping, 2016.

[12] Guy E. Blelloch and Bruce M. Maggs. Parallel algorithms. *ACM Comput. Surv.*, 28(1):51–54, mar 1996.

[13] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[14] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359, 2014.

[15] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, page 293–304, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Trans. Parallel Comput.*, 8(1), apr 2021.

[17] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, dec 1959.

[18] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. Efficient stepping algorithms and implementations for parallel shortest paths. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 184–197, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, jul 1987.

[20] Pawan Harish, Vibhav Vineet, and P Narayanan. Large graph algorithms for massively multithreaded architectures. 03 2009.

[21] Intel. Intel® 64 and ia-32 architectures software developer's manual. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`, 2023. Online; accessed 1 June 2023.

[22] Kevin Kelley and Tao B. Schardl. Parallel single-source shortest paths. 2010.

[23] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tompkins, and Eli Upfal. The web as a graph. In *Proceedings of the Nineteenth ACM*

SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '00, page 1–10, New York, NY, USA, 2000. Association for Computing Machinery.

[24] Charles E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, page 522–527, New York, NY, USA, 2009. Association for Computing Machinery.

[25] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[26] Kamesh Madduri, David A. Bader, Jonathan W. Berry, and Joseph R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, page 23–35, USA, 2007. Society for Industrial and Applied Mathematics.

[27] Julian Mcauley and Jure Leskovec. Discovering social circles in ego networks. *ACM Trans. Knowl. Discov. Data*, 8(1), feb 2014.

[28] Ulrich Meyer and Prashanthan Sanders. Delta-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49:114–152, 10 2003.

[29] OpenMP. Openmp homepage. `https://www.openmp.org/`, 2023. Online; accessed 1 June 2023.

[30] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[31] G. Sander. Graph layout for applications in compiler construction. *Theoretical Computer Science*, 217(2):175–214, 1999. ORDAL'96.

[32] Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. Many sequential iterative algorithms can be parallel and (nearly) work-efficient. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '22, page 273–286, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery.

[34] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing contention through priority updates. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, page 152–163, New York, NY, USA, 2013. Association for Computing Machinery.

[35] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, may 1999.

[36] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with graphit. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 158–170, New York, NY, USA, 2020. Association for Computing Machinery.