HIGH PERFORMANCE SHORTEST PATHS

by

Abhishek Chandratre

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Computing and Informatics

Charlotte

2017

ABSTRACT

ABHISHEK CHANDRATRE. High Performance Shortest Paths. (Under the direction of DR. ERIK SAULE)

In Graph theory, Single Source Shortest Paths (*SSSP*) is one of the problems which can be easily solved sequentially using Dijkstra's Algorithm, but it gets notoriously difficult in a parallel setting. This can be attributed to its transitive dependencies.

In this thesis, we present our ideas on efficiently calculating shortest paths for road networks on shared memory architecture. We first discuss the performance of *SSSP* algorithms on random graphs and actual road networks. Then we present our improvements on shared memory implementation of Delta Stepping algorithm. By far, this is the only work-efficient algorithm that can solve *SSSP* in parallel. We discuss the performance of our Improved Delta Stepping algorithm on graphs of U.S. states road networks. In the end, we report comparisons between existing algorithm and its proposed variant.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my thesis advisor Dr. Erik Saule for his extensive support. He was always there to look at my work, guide me and even with his hectic schedule he never denied any meeting request. Also, I would like to thank my thesis committee members Dr. Srinivas Akella and Dr. Dazhao Cheng for their constant support and feedback.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

DA	Dijkstra's Algorithm
Ligra	Lightweight Graph Processing Framework for Shared Memory
OSM	OpenStreetMap
SSSP	Single Source Shortest Path

CHAPTER 1: INTRODUCTION

There are many systems in the natural world and in the society that are amenable to mathematical and computational modeling. However, not everything is easily codified as a system of particles with coordinates and momenta. Some systems and problems such as social networks, ecologies, and genetic regulatory schemes are intrinsically divorced from space and time descriptions and instead are more naturally expressed as graphs that reflect their topological properties[1].

A Graph $G = (V, E)$ is defined as a group of nodes V, and nodes $u, v \in V$ which share relation are modeled as edges E. Graphs can be undirected for such as friends in a social network graph or directed such as water flowing through pipes. In this thesis, we will be focusing on directed graphs, specifically road networks. Using algorithms we can solve multiple real-world problems by correctly modeling these problems into graphs. In shortest path, we can solve problems like finding all pair shortest path [2] or single source shortest path where we have a fixed source and we find the shortest distance from it to all the other nodes.

Finding shortest paths in a graph is used in large number of domains such as, GPS help create road networks and finding shortest can help us answer questions like "How can I reach to point B from point A quickly", shortest paths algorithms are also used in financial asset trading [3], currency conversions [3]. In social networks, we can deduce degree of separation. Routing protocols in networks use shortest paths. When we talk about betweenness centrality [4] [5] , we can answer "Who is the most important or central node of the graph", betweenness centrality is defined as the "number of times a node acts as a bridge along the shortest path between two other nodes", knowing this fact we can answer questions like "Selection which group of people from social media would help spread the news quicker", "which cellular tower is most loaded". Betweenness centrality of node v is given by,

$$g(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is a total number of shortest paths from node s to t and $\sigma_{st}(v)$ is number of those paths passing through v. In such cases, it is tempting to first find all shortest path pair using Floyd-Warshall algorithm in $\theta(v^3)$. The problem arises when we have a large number of nodes and it becomes impossible to keep a 2D matrix of the shortest path in Memory. Thus in this thesis, since we will be dealing with millions of nodes in a road network, we chose to solve Single Source Shortest Path ($SSSP$) problem.

Since the advent of the self-driving car, we find a lot of application of shortest paths for operations like route planning and motion planning.

In the parallel environment, we have two main architecture namely shared memory and distributed memory architecture. In shared memory, we have multiple CPU cores accessing single physical memory. Alternatively, in distributed memory, the physical memory is distributed over the network. In this thesis, we particularly look at shared memory architecture for solving $SSSP$ in parallel.

1.1 Motivation

$SSSP$ problem has been solved a couple of decades ago, we believe there has not been an extensive study on solving this problem in parallel. Finding shortest path is one the basic algorithms which in turn is used as base by many other algorithms to find their optimal solutions. Thus improving the efficiency of $SSSP$ will have a huge impact on other algorithms. Thus, in this thesis, we focus on Delta Stepping [6] - an algorithm to find shortest paths in parallel and present improvements to its implementation.

Also, we observed that authors used randomly generated graphs to report performance of their algorithm. We strongly believe that using random graphs to model performance does not portray its actual performance when compared with real-world data.

1.2 Research Contributions

In this thesis, we answer the question of whether performance observed on randomly generate graph actually models the performance of those algorithms on real road networks. Next, we present improvements to Delta Stepping, optimizations are achieved by modifying data structures and adding clever techniques. We also show that performance of our implementation compare with performance obtained by using optimized heaps.

Moreover, we compared our implementation with *Ligra* framework [7] and our improved Delta Stepping outperformed *Ligra* in several test, more details in 4.3.

1.3 Thesis Outline

The thesis is organized as follows: In the next Chapter 2, we present formal problem definition 2.1 and Related work 2.2, where we talk about sequential and parallel solutions to $SSSP$. Chapter 3 will discuss the importance of graph topology and specifically the road networks. In Chapter 4 we present Delta Stepping algorithm and the optimizations which we performed with their results. In the end, in Chapter 5 concludes the thesis with remarks and raises a few question for future research.

CHAPTER 2: Problem Definition and Related Work

2.1 Problem Definition

Let $G = (V, E)$ be a directed graph with V vertices and E edges. We will be using term vertices and nodes interchangeably. Let (u, v) be two nodes such that $(u, v) \in V$ and $(u, v) \in E$ if and only if there is a direct non negative edge from u to v. In $SSSP$ we select a source node s from V and compute $\delta(v)$ for all nodes in G, $\delta(v)$ is the shortest path from node s to node v. $\delta(v) = \infty$ is v is not reachable from s and $\delta(s) = 0$.

$SSSP$ is inherently a sequential problem, i.e., if $\delta(v)$ is not zero and node w lies in the shortest path of v then we first need to find the value of $\delta(w)$ to compute $\delta(v)$. Delta Stepping [6] is first successful algorithm which can compute shortest paths in parallel. According to our findings, this algorithm for shared memory system was then studied twice in [8] and [9]. Both of them introduced optimizations to *DeltaStepping* and reported their results.

In Chapter3 we discuss effect of having real road network dataset compare to a graph which are randomly generated in evaluating the $SSSP$ results.

2.2 Related Work

In 2005-2006, 9th $DIMACS$ (Center for Discrete Mathematics and Theoretical Computer Science) challenge was on shortest paths. Thus, around that year many studies were conducted on finding shortest paths. Some of the studies involved BFS on massive graphs [10], parallel shortest paths implementation [11], high performance multi-level graph [12], K shortest paths [13].

This chapter has been divided into two segments concerning $SSSP$ by sequential and parallel algorithms.

2.2.1 Sequential Shortest Path

In sequential algorithm section, we present two types of algorithm i.e Label setting algorithm and Label correcting algorithm.

2.2.1.1 Label Setting Algorithms

These algorithms mark one vertex v as permanent after each iteration. Dijkstra's is one such algorithm.

Edsger W. Dijkstra - in 1959, then a twenty-nine years old computer scientist proposed algorithms for the solution of two fundamental graph theoretic problems: the minimum weight spanning tree problem and the shortest path problem [14]. Even today Dijkstra's Algorithm (DA) for the shortest path problem is one of the most celebrated algorithms in computer science[14]. DA is summarized as:

1. Assign $\delta(s) = 0$ for source vertex s and $\delta(x) = \inf$ for $x \in V - (s)$.
2. Find vertex v with smallest $\delta(v)$. If there are no vertices or if $\delta(v) = \inf$ then goto step 4.
3. Make v permanent and for nodes y adjacent to v, update $\delta(y) = \delta(v) + E(v, y)$. Goto step 2.
4. Print δ for all values and exit.

Finding the next non permanent vertex v with minimum $\delta(v)$ is computationally intensive part of this algorithm. Original DA did not use min priority queue to find next shortest vertex and ran in time $O(V^2 + E)$. Using Fibonacci Heap [15], DA can run in $O(Vlog(V) + E)$ [16], this is due to the fact that $decrease - key$ operation in Fibonacci heap is constant. Whereas if we use binary heap we get complexity of $O((E + V)log(V))$, as $decrease - key$ operation takes $O(log(V))$ running time.

Dijkstra's algorithm have the flexibility to choose heap and the way we can use it i.e. while relaxing we can either decrease-key or we can simply add a new pair of $(v, \delta(v))$ where $v \in V$ to the heap. In following table we discuss some types of heap with thier insert/delete-Min operation [17].

Table 2.1: Priority Queue that can be used in Dijkstra's algorithm

Priority Queue	Insert/Delete-Min
Bottom-up Binary Heap	$O(log_2(V))$
Aligned 4-ary Heap	$O(log_4(V))$
Aligned 16-ary Heap	$O(log_{16}(V))$
Fibonacci Heap	$O(1)$

While we were comparing various implementations we found out we got most performance from 4-ary heap, more details at result section 4.3. Following table summaries difference in complexities for a graph $G(V, E)$ with D decrease key operations, using heap with decrease-key or heap without decreasing-key operations [17].

Table 2.2: Dijkstra's with and without Decrement Key

	Insert	Decrease-Key	Delete/Delete-Min	Total
Dijkstra's Dec	V	D	V	$2V + D$
Dijkstra's No-Dec	$V + D$	$None$	$V + D$	$2V + 2D$

We can observe, in case of DA without decreasing key we performs more operation on heap but it outperforms DA with decrease-key operation on sparse graphs, details discussed in 3.1.1.

2.2.1.2 Label Correcting Algorithms

These algorithms can relax non settled vertices, continuous relaxing results in convergence to shortest paths. The label are not permanent until all iterations have been completed. Bellman-Ford [18], is one such label correcting algorithm with running time of $O(VE)$. Next, we present pseudo code for Bellman-Ford.

Algorithm 1 Bellman-Ford

```
**Input**: Graph $G(V, E), s$
**Output**: $\delta(v)$ for $v \in V$, shortest path for $v$ from $s$
function BELLMAN-FORD
    for all $v \in V$ do
        $\delta(v) \leftarrow \inf$
    $delta(s) \leftarrow 0$
    for all $i$from 1 to $|V|$ - 1 do
        for all ( do$(u, v) \in E$)
            $relax(u, v)$
function RELAX(u, v)
    if $\delta(v) > \delta(u) + dist(u, v)$ then
        $\delta(v) = \delta(u) + dist(u, v)$                ▷ Update min to v
```

We can observe that we have a loop while run over the length of vertices and for each iteration it again scans for all the edges, thus we the running time complexity as $O(VE)$, which is worse that what Dijkstra's algorithm offer. The noteworthy point of this algorithm is that it can detect negative cycles, which cannot be achieved in Dijkstra's algorithm.

2.2.2 Parallel Shortest Path

In this section, we talk about some algorithms that can process graphs in parallel and result in shortest paths. There are various algorithm that run in parallel can solve all pair shortest path such as, we can run multiple instance of Dijkstra's algorithm on multiple nodes simultaneously.

For Parallel Bellmen Ford, there are multiple variation possible such are distribute nodes across the networks, using GPU's. paper [19], talk about it in more detail.

Also, we have parallel implementation of Floyd-Warshall algorithm has been studied in [20], [21].

In this thesis, we will talk about Delta Stepping in detail under chapter 4. Also, we present our own implementation and discuss the motivations behind choosing certain data structures.

2.2.3 Graph Analysis Libraries

In this thesis, we compared our Improved Delta Stepping performance with Delta Stepping provided by Ligra [7]. According to our information Galios [22], is another lightweight parallel graph processing framework. Networkit [23] and Combinatorial BLAS [24] are two highly used high-performance software libraries for graph processing.

CHAPTER 3: Road Networks

In this thesis, we present a way to extract road networks from OpenStreetMap [25]. OpenStreetMap (OSM) is Wikipedia for geographic locations, where users upload geographic information. After every 7 hours, OSM releases new Planet file. Another website called geofabrik [26] parses the planet file and maintain geographic information in a hierarchical manner. In this thesis, we picked up states geographic files from geofabrik and wrote a script against it to parse them and create graphs in a simple format of the directed edge from vertex u to v with distance d.

The main advantage of going through this process is that, we can model various types of graphs, where the edge weight can be the distance between two nodes $u, v \in V$ or it could be time taken by vehicle, bike, walk to travel from node v to u. We kept it simple, using minimum speed and maximum speed for roads, we modeled the network from a perspective of car travel.

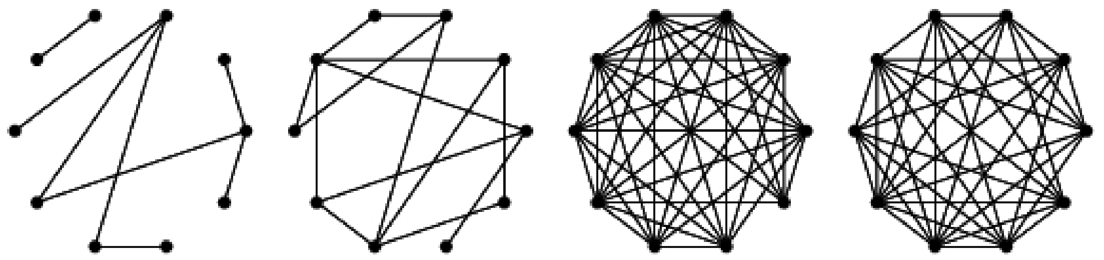
3.1 Graph Topology

Most of the times, when authors publish algorithm they test their algorithms on graphs which are generated randomly. We believe for shortest paths, the performance of algorithms on road networks vs randomly generated graph is quite different.

Random graphs are mostly generated using following ways.

3.1.1 Random Graph

Random Graph is the first prototype of random graph generator given by Erdos-Renyi in 1960[27]. These graphs can be defined as a Probability distribution over a graph. Given a random graph $G(V, p)$, V are nodes of graph and p gives the probability of having an edge between any two nodes. Following diagram portrays the how different graphs are generated by adjusting the value of p. We can observe if we increase the value of p we get a denser graph and at $p \leftarrow 1$ we get a complete graph.



(a) Random Graph

Figure 3.1: Effect of increasing p value on a graph with 10 nodes.

3.1.2 Variations of Random Graphs

For long time random graphs have been studied and used to modeling complexities of algorithms like shortest paths, scheduling problems, approximation algorithms. Researcher have come up with various have proposed variations of random graphs to simulate graph for a particular problem.

3.1.2.1 Weighted Random Graph

Weighted random graph $G(V, p, d)$ can be defined as random graph of $G(V, p)$ which has edges which weight is randomly distributed in radius of $]0; d]$.

3.1.2.2 Random under constraints

Certain types of random graph generator create graph under conditions. One of such graph is precedence graphs [28], these graph have a property that these graphs can be topologically sorted. There are no back cycle, thus these graphs can be used to create graphs for modeling and simulating scheduling problems.

3.1.3 Kronecker graph

Kronecker graph are used to generate realistic networks. These graphs are based on how graphs evolve over the time, such as number of nodes in networks increase, diameter of graphs reduce. These graphs are created recursively using $Kronecker\ matrix$ and $Kronecker\ Product$ [29].

3.1.4 RMAT graphs

RMAT graphs are another types of recursive matrix graphs which aims to created real like graphs over multiple domains [30]. By setting Kronecker matrix to $2x2$ we generate RMAT graphs, thus we can say Kronecker graphs are generalized RMAT graphs [31]. Also, we need to explicitly pass E edges in case of RMAT graphs, but in Kronecker encodes the number of edges [31].

3.1.5 Arguments on Random Graphs

Random graphs have been studied throughly [32], since they are simple and we can make use of the probability p associated with it. These graphs are widely used by researchers to express their work in terms of random graph. Thus we make a statement that "Using random graph for modeling performance does not guarantee the same performance on real network graphs". To this statement we put forth following case:

When we consider DA, we can have multiple implementation priority queue. Heaps are best way to abstract priority queue. In DA, heaps can be modeled to relax nodes either by decreasing key or by reinserting the same node with reduced distance. Following summaries the complexity of heap operation in both of the cases.

Table 3.1: Dijkstra's with and without Decreasing Key

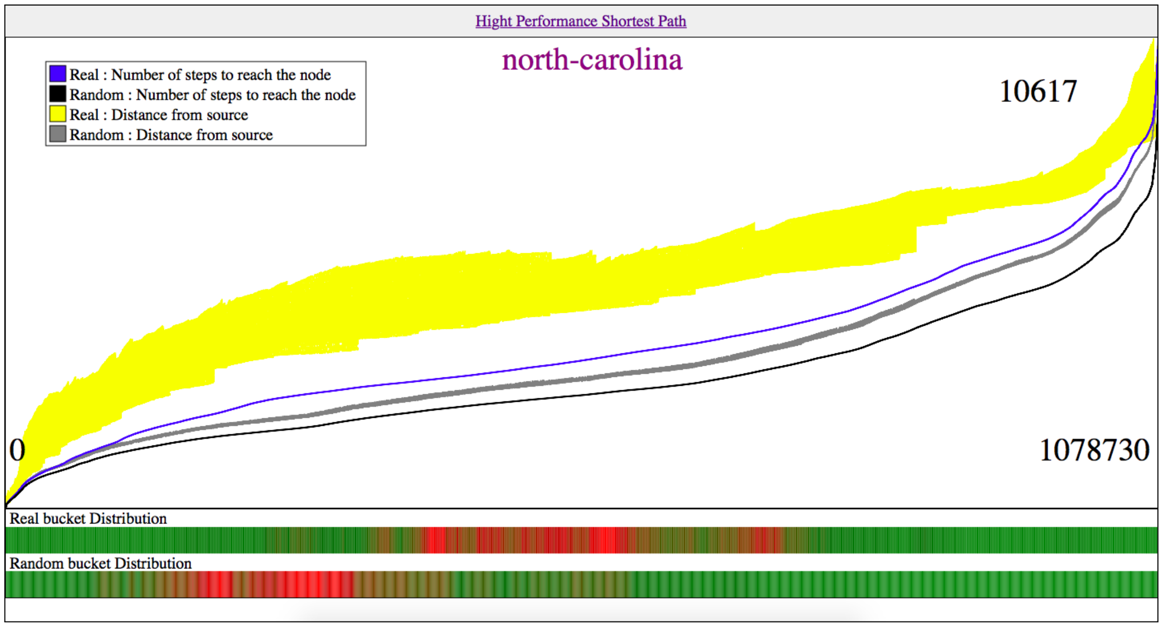
	Insert	Decrease-Key	Delete/Delete-Min	Total
Dijkstra's Dec	n	D	n	2n + D
Dijkstra's No-Dec	n + D	None	n + D	2n + 2D

From table 3.1 we can observe that the heap with no decrease-key is doing more work, but in when we consider a sparse graph studies have shown that heap without decrease-key is more efficient and heap with decrease-key is more efficient on dense graphs [17]. Thus we can say that until and unless we know the topology of graph performance of algorithm cannot be compared.

3.1.6 Real Graphs and Random Edges

Random Graphs have Poisson distribution of edges, and road networks are not based on specific distributions. Next, we consider graphs which are road networks but with randomly weighted edges and we make an argument that, even if we modeling performance on network graphs based on random weights, we still cannot predict its performance on network graphs. Following case bolsters our argument:

Some algorithms are sensitive to edge weight, for example In Delta Stepping we have a configurable parameter Δ which is used to divide adjacent nodes into two categories, in result 4 we show the effect of value of Δ on a particular graph. We observe that different values of Δ are suited for unique graphs. Thus if an algorithm has high performance on road networked modeled graph, does not necessarily mean it would have high performance on a social network graph.

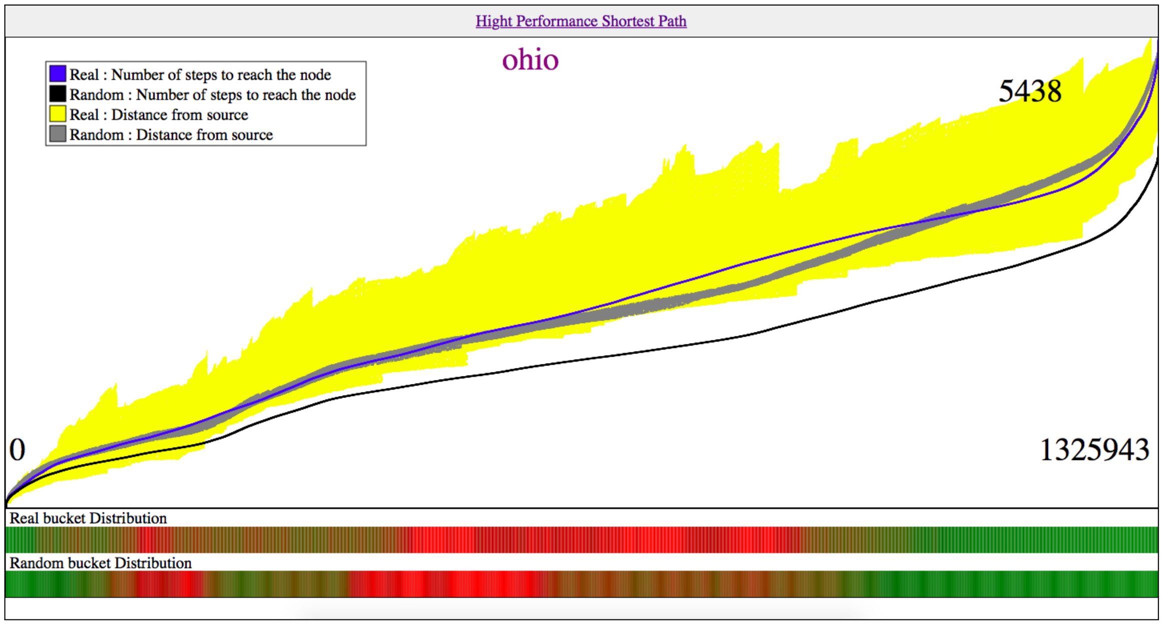


(a) Real Edges vs Random Edges on based on geography of North Carolina

Figure 3.2: Distribution of shortest path on nodes, sorted by BFS for North Carolina.

Visualizing graphs based on random edge weight is difficult. In figure 3.2, we have captured shortest paths for NC state and have sorted nodes based on BFS. Yellow colored spectrum is shortest paths based on network graph and gray colored spectrum is based on network graph with random edges.

We can observe that gray color spectrum is tighter than yellow colored spectrum. Thus when an edge sensitive algorithm tries to perform optimizations on random graphs, we are unsure that the same algorithm would perform similar to what it did on random graph, due to its sensitivity to edge lengths.



(a) Single thread implementation comparisions

Figure 3.3: Distribution of shortest path on nodes, sorted by BFS for Ohio.

For Ohio 3.3, again we have x-axis as nodes sorted by BFS from source node and on y-axis we have four parameters namely,

1. Number of steps required to reach from soruce
2. Shortest path distance

each for real graph and a real graph with random edges. We see the same pattern as North-Carolina graph.

1. The Shortest paths in real graph are spread in a spectrum where as for real graph with random edges shortest paths of neighboring are close.
2. The steps required to reach a particular node in real graph with random edges is much smaller compared to its counterpart in real graphs.

Moreover, at bottom of figure 3.2 and 3.3 we see bucket distribution of nodes, more detail on buckets is mentioned in next section. Buckets holds vertices, the time required to process each bucket depends on number of vertices it keeps. As we can observer the bucket distribution for real graph and real graph with random edges is different, then execution of same algorithm on these graph would result in different execution time.

In results section 4.3, we show the effect of changing Δ on a graph. Also, we show with same Δ and threads two algorithms vary their performance based on the graph.

3.1.7 Short Summary

In this section using different random generated graph and graph with random edge weight, we have shown that if an algorithm A performs well on graph G, it does not imply that A would perform similarly on random graph $\hat{G}$.

CHAPTER 4: Delta Stepping

In 1998, Delta Stepping was introduced by U. Meyers and P. Sander [6]. This was first work efficient parallel shortest path algorithm[6] and is discussed in 4.1.

4.1 Algorithm

In this section we discuss the working of basic Delta Stepping algorithm. We have directed graph $G(V, E)$, source node s and Δ parameter as inputs, and we expect to get shortest path from s i.e. $\delta(v) : v \in V$.

Algorithm 2 Delta Stepping

```
**Input**: Graph $G(V,E), s, \Delta$
**Output**: $\delta(v)$ for $v \in V$, shortest path for $v$ from $s$
1: function DELTA_STEPPING
2:     for all $v \in V$ do
3:         $heavy[v][w] \leftarrow \{dist(v,w) : (v,w) \in E$ and $dist(v,w) <= \Delta\}$
4:         $light[v][w] \leftarrow \{dist(v,w) : (v,w) \in E$ and $dist(v,w) > \Delta\}$
5:         $\delta(v) \leftarrow \inf$
6:     $relax(s,0)$
7:     $i \leftarrow 0$
8:     while $B$ is not empty do
9:         $S = \theta$
10:        while $B[i]! = \theta$ do
11:            $Req = \{(w; \delta(v) + dist(v;w)) : v \in B[i]$ and $(v;w) \in light(v)\}$
12:            $S = S \cup B[i]; B[i] = \theta$
13:            for $(v,x) \in Req$ do                              ▷ Relax light edges
14:                $relax(v,x)$
15:        $Req = \{(w; \delta(v) + dist(v;w)) : v \in S$ and $(v;w) \in heavy(v)\}$
16:        for $(v,x) \in Req$ do                                  ▷ Relax heavy edges
17:            $relax(v,x)$
18:        $i = i + 1$                                             ▷ Next Bucket
19: function RELAX(v, w)
20:     if $w < \delta(v)$ then
21:         $B[\delta(v)/\Delta] := B[\delta(v)/\Delta] \setminus v$            ▷ Remove $v$ from old bucket
22:         $\delta(v) = w$                                        ▷ Update shortest path to $v$
23:         $B[\delta(v)/\Delta] := B[\delta(v)/\Delta] \cup v$              ▷ Place $v$ in new bucket
```

The algorithm takes three parameter namely, graph $G(V, E)$, source nodes s, and Δ. In this algorithm size of each bucket is exactly Δ. We process for non-empty buckets and if all the bucket have been completed, then we terminate the problem. Algorithm can be divided in three main phases as mentioned below.

The relaxation that we perform are done in parallel.

4.1.1 Classification of Edges

This is the phase where parameter Δ sets up the edges into categories for further processing. From line 2 to 5, for $v \in V$ we categorize edges into light or heavy depending on whether their edge weight is less than or equal or more than Δ respectively.

For edges $(u, v) \in Light$, signify that upon relaxation of u, v would fall in same bucket or the next bucket. Similarly, for $(u, v) in Heavy$, upon relaxation of u, we are guaranteed that v would never fall in current bucket.

After completion of Phase 1. We loop over all and to find first non-empty bucket. Then we start with following phases.

4.1.2 Light Edges Processing

Once we find first non-empty bucket we perform following operations. Empty S data structure.

1. Empty Req data structure
2. Collect all vertices and their light edge neighbors and push them in Req data structure.
3. Push current vertices in S set.
4. Empty current Bucket.
5. Relax all elements from Req data structure.

After we complete one iteration of above steps, since nodes at the end of light edge are within distance Δ, these nodes either fall in current bucket or the next one. Then we check are there more vertices in current bucket, if true then repeat above 4 steps, else move to next phase.

4.1.3 Heavy Edges Processing

In this phase, we relax the heavy edges of the vertices which were processed in previous phase. As we have pushed all the vertices in S data structure, we perform following actions.

1. Empty Req
2. Loop over S
3. Push heavy edges and updated distance to Req
4. Relax all elements from Req data structure.

We just make perform iteration of current phase, as the nodes in this phase are beyond length of Δ none of the edges fall in current bucket. Thus we move forward to search next empty bucket.

As we notice, we never set a label on vertex. Thus we classify Delta Stepping as Label Correcting algorithm. Once all the buckets are empty, we are sure that all nodes are converged and we have the final result in δ.

4.2 Delta Stepping Improved

In this section we present our implementation of the idea Delta Stepping.

Removing Req Data structure completely Taking S as vector but making sure that there are no reinsertions. Having limited number of buckets Copying just once per iteration.

Algorithm 3 Delta Stepping Improved

```
  Input: Graph $G(V, E), s, \Delta$
  Output: $\delta(v)$ for $v \in V$, shortest path for $v$ from $s$
  #define CORRECT_BUCKET(x,y) ((x)%y)
  struct {
          Vertex barray[count(V)]];
          atomic<int> index=0;
  } bucket;

  struct {
          Vector<Vertex> t_vec[width];
  } thread_buckets;

 1: function DELTA_STEPPING
 2:     for all $v \in V$ do
 3:         $heavy[v][w] \leftarrow \{dist(v,w) : (v,w) \in E$ and $dist(v,w) <= \Delta\}$
 4:         $light[v][w] \leftarrow \{dist(v,w) : (v,w) \in E$ and $dist(v,w) > \Delta\}$
 5:         $\delta(v) \leftarrow \inf$
 6:     $max\_edge \leftarrow max(E)$
 7:     $width \leftarrow (max\_edge/\Delta) + 1$
 8:     $buckets \leftarrow bucket[width]$
 9:     $t\_buckets \leftarrow thread\_bucket[number\_of\_threads]$
10:     $S \leftarrow vector < Vertex > [number\_of\_threads]$
11:     $relax(s, 0)$
12:     $i \leftarrow 0$
```

Algorithm 4 Delta Stepping Improved

```
13:     while $buckets$ are not empty do
14:         $tid \leftarrow thread\_id$
15:         $cb \leftarrow next\ non\ emptyCORRECT\_BUCKET(i, widht)$
16:         $S[tid] = \theta$
17:         while $buckets[cb].index! = 0$ do
18:             $relax(w, \delta(v) + dist(v; w)) : v \in buckets[i]$ and $(v; w) \in light(v)\}$
19:             $S[tid] = S[tid] \cup buckets[cb].array$
20:             $buckets[cb].index = 0$
21:             $merge\_thread\_buckets(cb, true)$
22:         $Req = \{(w; \delta(v) + dist(v; w)) : v \in S[tid]$ and $(v; w) \in heavy(v)\}$
23:         for $(v, x) \in Req$ do                              ▷ Relax heavy edges
24:             $relax(v, x)$
25:         $merge\_thread\_buckets(cb, false)$
26:         $i = i + 1$                                          ▷ Next Bucket
27: function MERGE_THREAD_BUCKETS(bucket_index, is_cb)
28:     $start \leftarrow bucket\_index$
29:     $end \leftarrow 1\ if is\_cb = True\ else\ width$
30:     for $i \in 0, end$ do
31:         $buckets[i] \leftarrow buckets[i] \cup t\_buckets[tid][i]$
32: function RELAX(v, w)
33:     if $w < \delta(v)$ then
34:         $B[\delta(v)/\Delta] := B[\delta(v)/\Delta] \setminus v$          ▷ Remove $v$ from old bucket
35:         $\delta(v) = w$                                    ▷ Update shortest path to $v$
36:         $B[\delta(v)/\Delta] := B[\delta(v)/\Delta] \cup v$          ▷ Place $v$ in new bucket
```

4.2.1 Data structures

In this section we discuss the data structure that we use and our motivation behind using them. We are making use of atomic which are provided by C++, instead of locks since they are more expensive than atomics. We wanted to achieve most parallelism from the program and at the same time we wanted to minimize conflicts on atomics while processing in parallel.

- $\delta(v)$: To store minimum distance of vertex $v \in V$ we use array of atomics <typeof(V)>. This to ensure that minimum distance have been updated atomically and δ always keeps updated minimum value.
- Thread buckets: Each thread bucket is a list of vector<typeof(V)> owned by a thread, the size of list is equal to width. We use thread buckets while relaxation, so that while relaxing we do not get conflicts between threads.
- buckets: Buckets contain array of structure of size width. Where each bucket structure contain an array is of size(V) to hold vertices, and has a an atomic pointer *index* which which when accessed tells number of nodes currently present in the bucket. When we need to merge nodes from thread bucket to actual buckets we first lock the space by incrementing the atomic *index* to *index* + (elements in equivalent thread bucket), and then copy elements from thread bucket to buckets in a single pass.
- S: S structure is list of vector<typeof(V)> of size number of threads. Since we use vector instead of set we expect multiple similar entries in S, see S_buckets for details on how we achieve set like properties on vector S. Each thread will fill its own S data structure, that we do not get conflicts when we fill S data structure with nodes.

- S_bucket: This is an array of size V, which keep mapping of vertex to its bucket. We make use of S_bucket to check whether nodes bucket have been changed, if true then push nodes to S[tid]; else skip. Although we have use S as vector, using S_bucket we minimize chances of reinsertions in vector S. This simple trick gave us a performance boost of up to 10% on graph with million nodes. We tried to do atomic insertion for S_bucket, but found that the cost of duplicate insertion was smaller than the cost of atomics, so we would rather have duplicates.

4.2.2 Working of Improved Delta Stepping

We again break the algorithm in three different phases namely: edge classification, light edges processing and heavy edges processing.

4.2.2.1 Classification of edges

This section is similar to 4.1.1. Moreover we can perform classification of edges in parallel using simple omp loop construct.

4.2.2.2 Light Edges Processing

In this phase we process light edges of the nodes which are present in current bucket. Initially we find first non empty bucket by looping over bucket[CORRECT_BUCKET(i, width)].index. Thus before exiting we scan just width number of times.

While current bucket has elements, we spread those elements amongst threads. Each thread then relaxes the light edges, since each threads has its own t_bucket there are no conflicts during relaxation. Moreover, while pushing vertex in S, we make use of S_bucket, so that we do not add redundand elements, if we had redundant vertices's in S, then we would have to process them for their heavy edges with no change in result whatsoever. After all threads have completed their processing, we empty current bucket and then merge t_bucket[current_bucket] to actual buckets and repeat above steps.

4.2.2.3 Heavy Edges Processing

Once current bucket is empty, we move to heavy edge processing. In this phase, each thread loops over its S data structure and relaxes heavy edges. Next, the thread will merge it's first non empty t_bucket to actual buckets.

4.2.3 Explanation of Optimization

In this section we talk about, motivation behind using some data structure and techniques.

- Width: In this thesis is defined as maximum edge in graph divided by Δ plus 1. It gives us an estimate of number of maximum buckets that can utilized without side effect of relaxation. This allow us to reuse previous buckets and at the same time it reduces time for searching vertices for next non empty bucket.

- Removal of Req data structure: We believe that Req data structure was used so that relaxation can be done in parallel. Instead, we created set of buckets with each thread so that relaxation does not introduce conflicts. Later by merging thread buckets to actual buckets we achieve same its desired state with optimizations in merging of buckets.

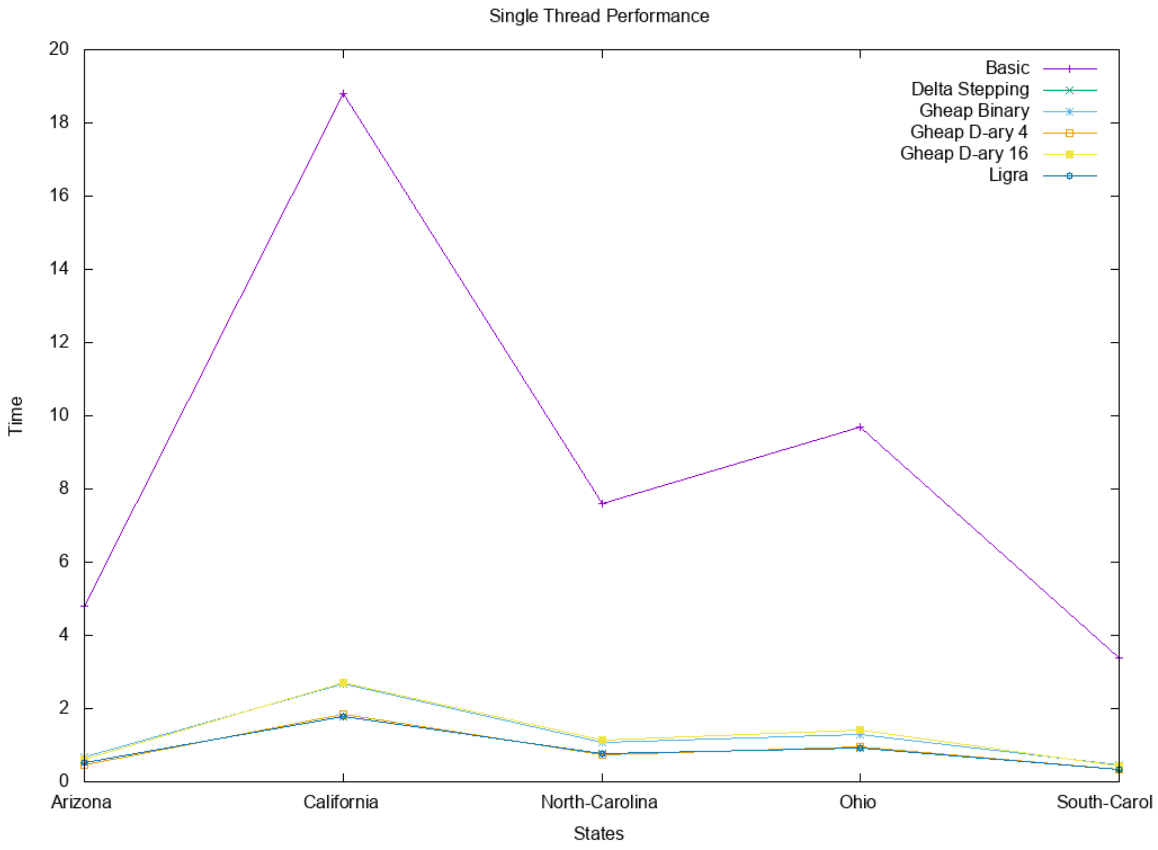
- Merging of buckets: For a thread to merge its thread bucket to actual bucket, it first has to increment buckets.index by number of element current thread bucket has. Once the thread is successful in pushing buckets.index, thread can copy its nodes without a risk of overwriting nodes of other elements.

  Moreover, we can control the number of buckets that we can merge. Upon testing we found out, instead of merging all the thread bucket to actual buckets at end of phase 3, we can merge first non empty bucket. This optimization helped us achieve better even with lower Δ.

4.3 Results

All the result were performed on UNCC HPC cluster. Cluster consisted of 16 (8*2) cores Intel(R) Xeon(R) CPU E5-2667 v3, which were clocked to 3.20GHz and it had 131 GB physical memory. On the software side of the environment, cluster ran on "Red Hat Enterprise Linux 7" with 3.10.0-693.5.2.el7.x86_64 Linux Kernel, with GCC 5.5.0 compiler and -O3 -std=c++11 fopenmp flags.

4.3.1 Single Thread Delta Stepping and Dijkstra's

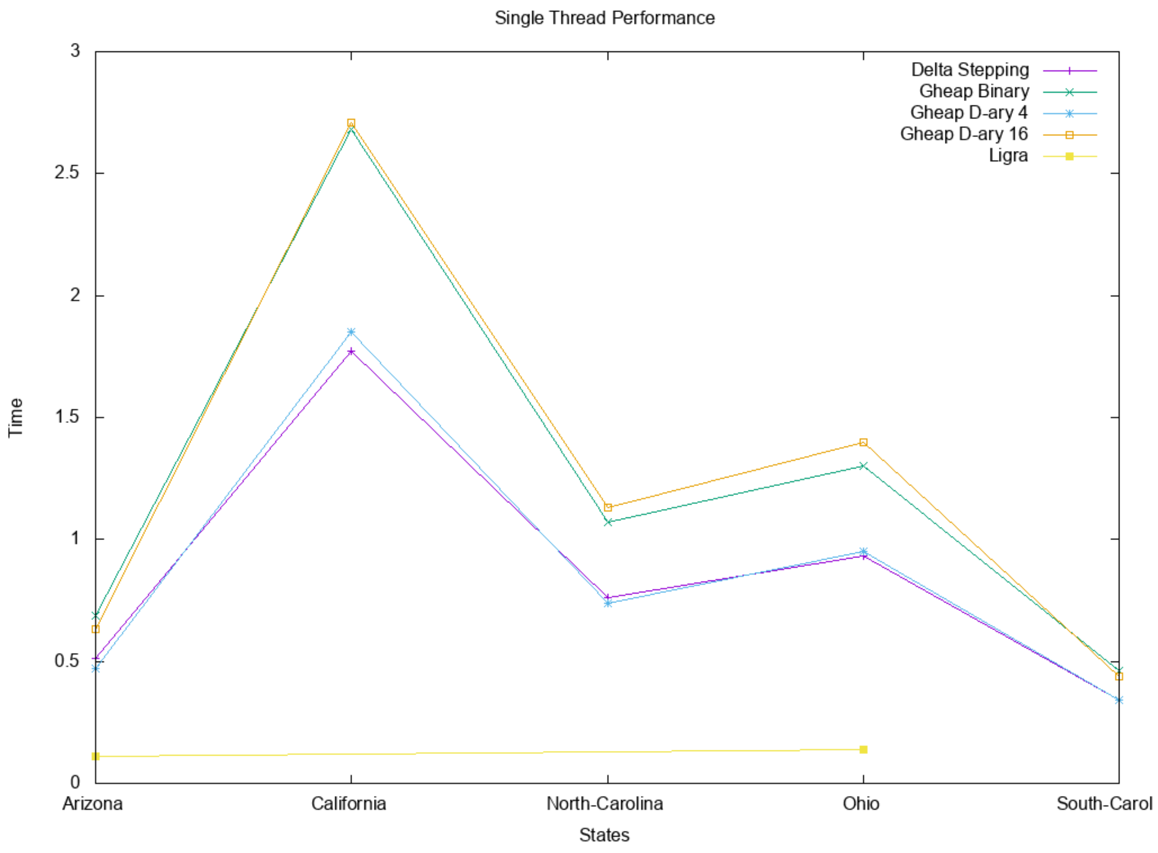


(a) Single thread implementation comparisions

Figure 4.1: Comparison of single thread Delta Stepping with various Dijkstra's implementations.

In figure 4.1, we compare single thread implementation of Delta Stepping with various Dijkstra's implementations. Basic Dijkstra's implementation is simple loop over priority queue and the priority queue is created using Boost Library's Fibonacci heap. As we can observe that basic Dijkstra's I taking lot of time, we then implemented DA using gheap [33] which is fast generalized heap tree algorithms in C++. Using gheap we get comparable results to Imporved Delta Stepping with single thread.

We also test with n-ary heap using gheap. In this study we tested with 4-ary and 16-ary heap. Although, both of them performed better than basic implementations of Dijkstra's, 4-ary heap was better than 16-ary heap. In the next figure we take the same results and exclude basic implementation from it.



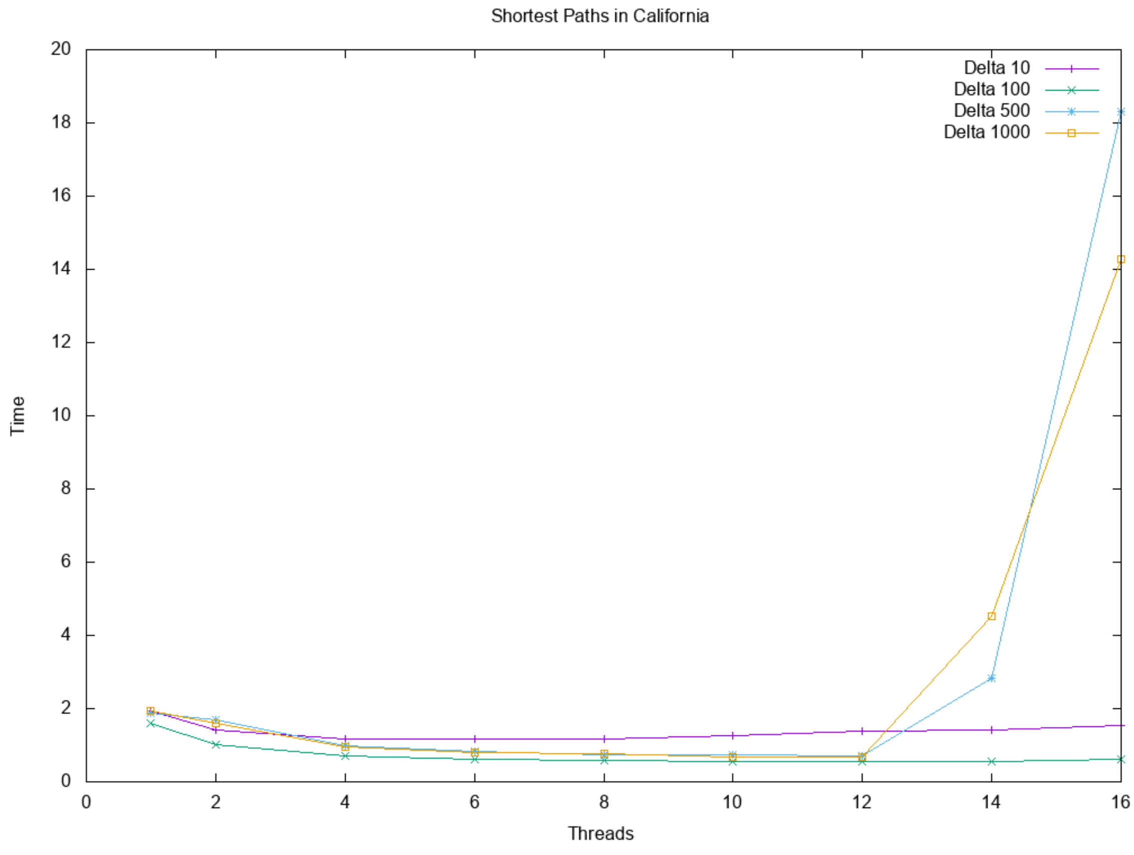
(a) Single thread implementation comparisions without Fibonacci Heap

Figure 4.2: Comparison of single thread Delta Stepping with various Dijkstra's implementations excluding basic implementation of DA.

In figure 4.8, we excluded basic implementation so that we can zoom in performance of Delta Stepping and n-ary heaps. Also, we have added Ligra's single thread implementation to the graph, more details on Ligra in 4.3.4[7]. In this result, Ligra's Delta Stepping implementation with single thread outperforms every other algorithm. Improved Delta Stepping with single thread compared neck to neck with 4-ary heap implementation of Dijkstra's.

4.3.2 Performance on increasing threads

In this experiment, we studied effects of adding thread on performance. Improved Delta Stepping was performed on graph of California, various Δ values and with increasing number of threads.

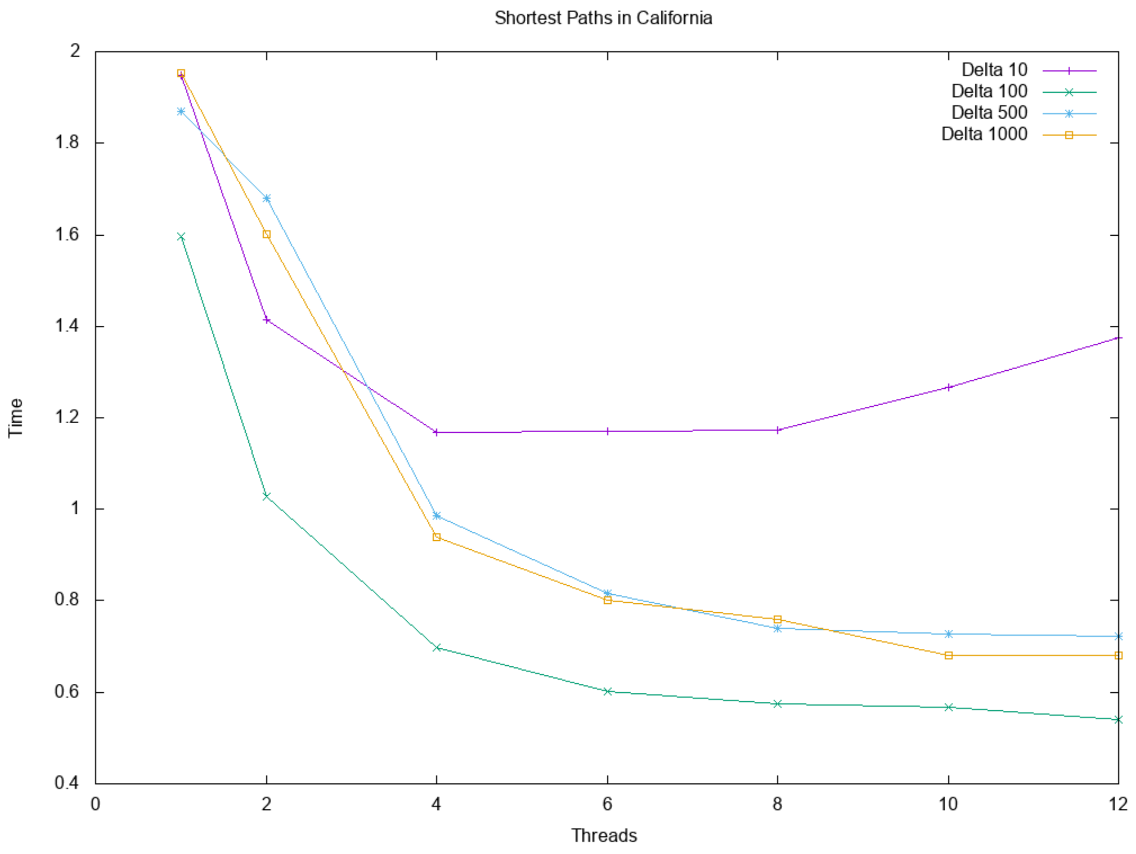


(a) Threads vs Performance

Figure 4.3: Performance gain on addition of threads.

As we can observe, for smaller Δ of 10 and 100 we did notice decrease in time, whereas for Δ of 500 and 1000 we see sudden increase in time for 14 and 16 threads.

Since the performance of graphs are also dependent on value of Δ, choosing right number of threads and right Δ is important. In next figure we exclude thread 14 and 16 so that we can view other threads in more detail.



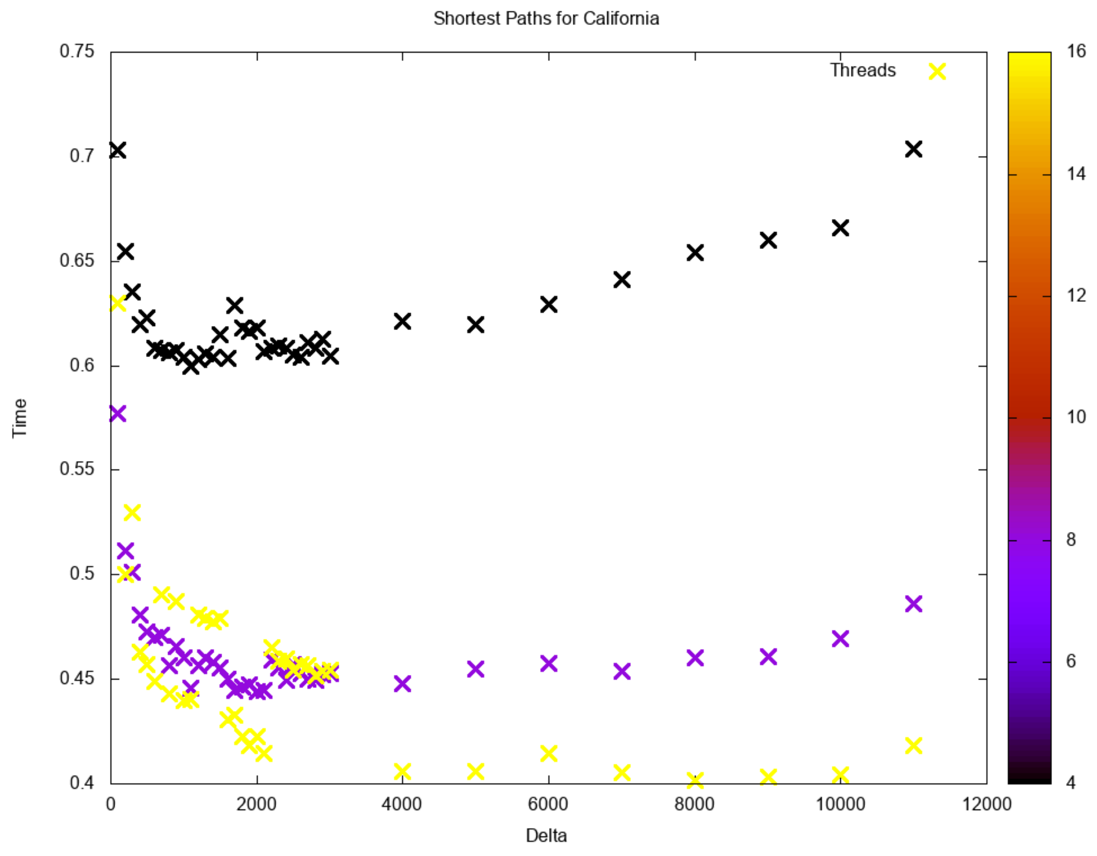
(a) Threads vs Performance without 14 and 16 threads

Figure 4.4: Performance gain on addition of threads.

Here we can observe that as we increase threads we see proportional decrease in time up to 8 threads beyond that we might not be able to see proportional decrease in time. Also we can see that the value of Δ affect the performance. Thus in next section we look at how value of Δ affects the performance.

4.3.3 Effect of increasing Δ

In figure 4.5, we again use the same California graph and this time we test for 4, 8 and 16 threads and vary value of Δ from 10 to 12000 and observe the performance.



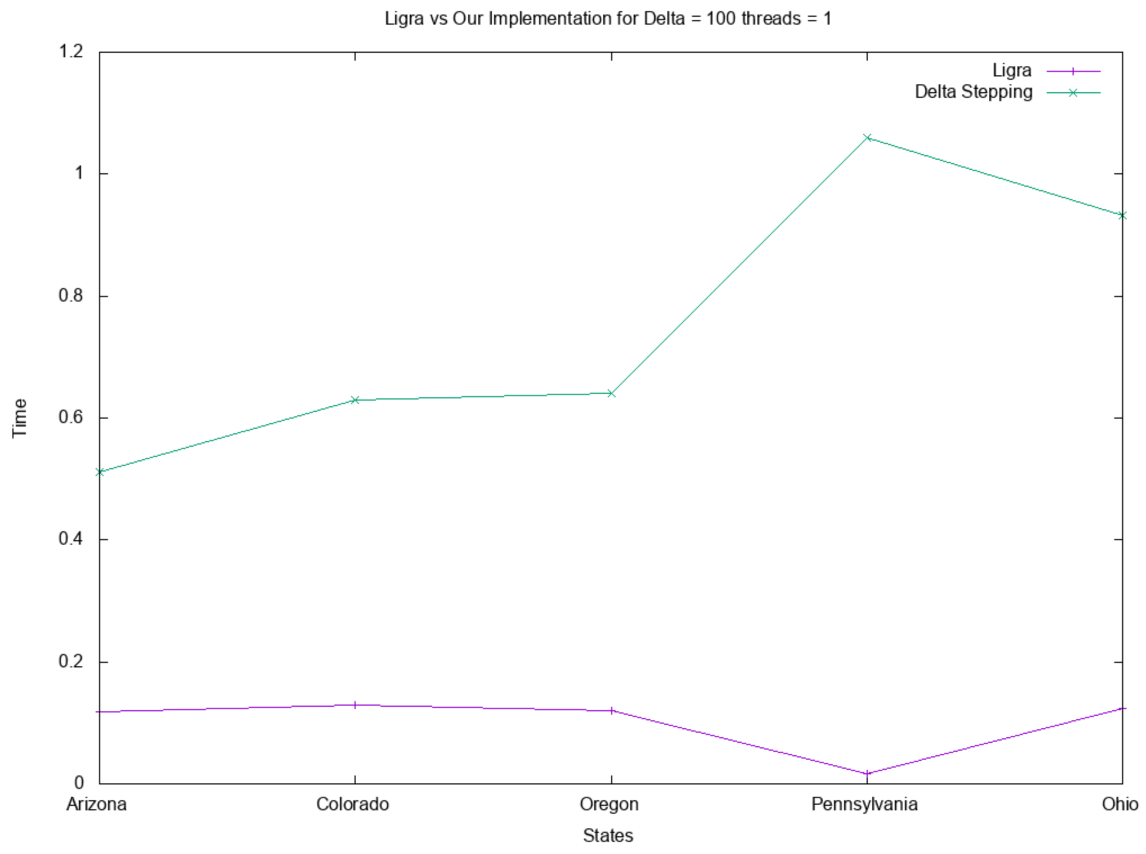
(a) Performance vs Delta

Figure 4.5: Performance under varying Δ.

We notice that as we increased Δ performance of 4 threads beyond Δ 1000 decreased. Whereas, for 8 threads performance was somewhat better compared to 4 threads. But 16 threads showed nice performance even after increasing Δ beyond 1000. One thing to notice here is around Δ value of 1500, we observe a spike in time axis for 4 threads. As of now there is no good way to find Δ on real graphs. For now, trail and error is the only good way to find plausible Δ.

4.3.4 Comparison with Ligra

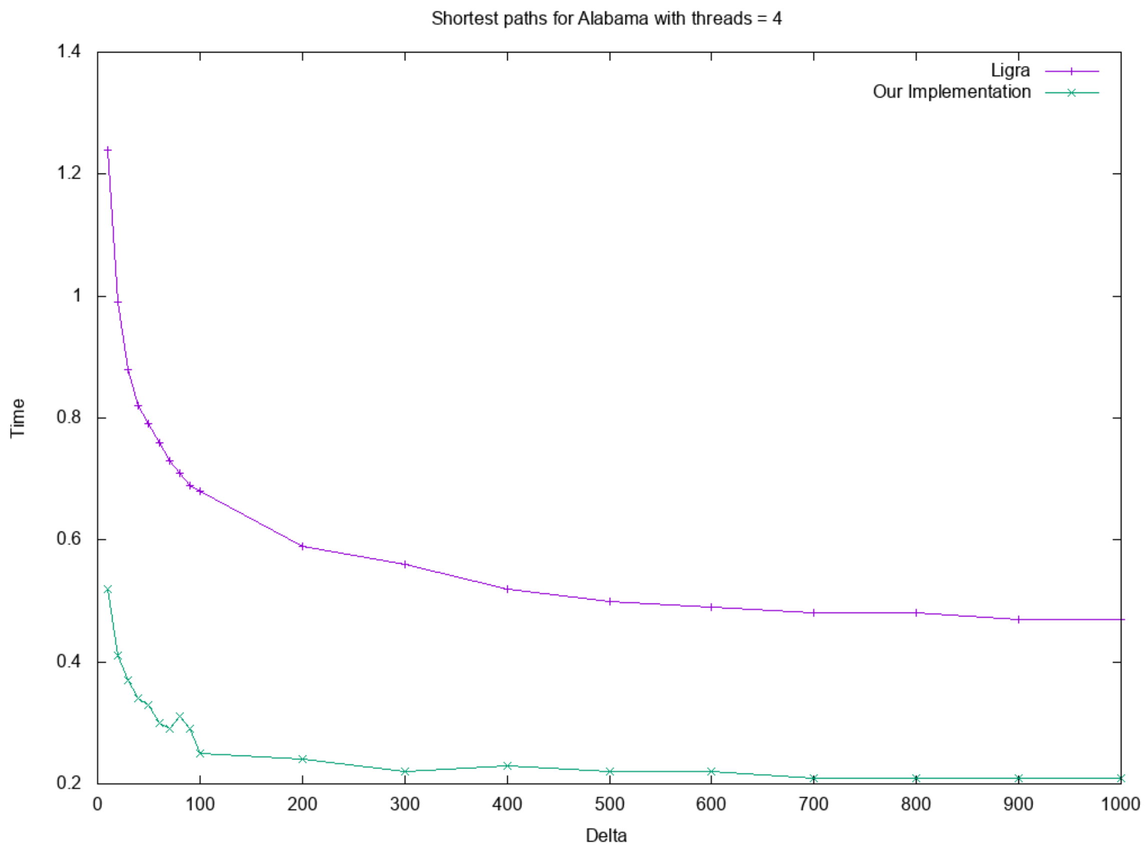
In this section we show performance comparison of our implementation against Ligra's Delta Stepping [7].



(a) Improved Delta Stepping vs Ligra's Delta Stepping with single thread

Figure 4.6: Comparison of single thread Improved Delta Stepping with Ligra's Delta Stepping.

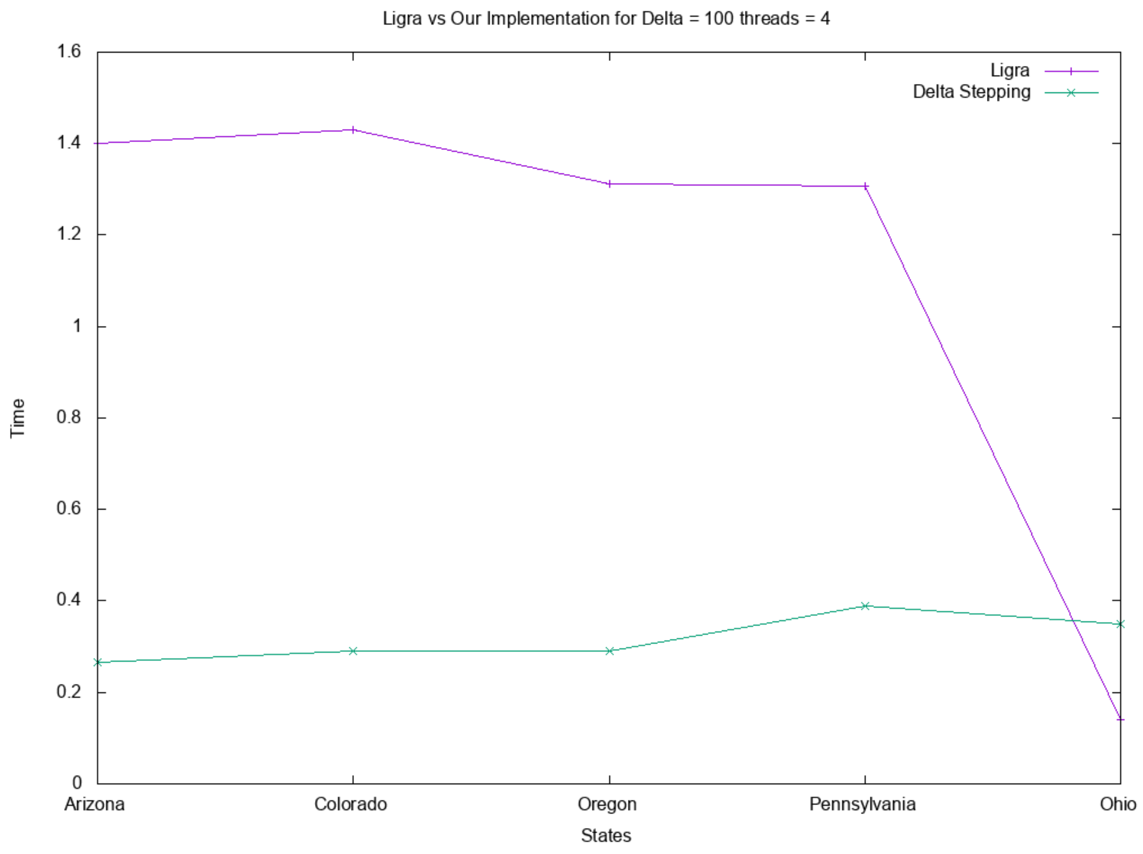
Figure 4.6, we tested Ligra and improved Delta Stepping over graphs of different states with 1 thread and delta as 100. We observe that, ligra with one thread outperforms improved Delta Stepping with one thread. Moreover, Ligra's single threads also outperform almost all Dijkstra's implementations that we implemented. The reason for this performance boost as of is unknown.



(a) Delta Stepping vs Ligra over various Δ

Figure 4.7: Plots of Improved Delta Stepping and Ligra's Delta Stepping over various Δ values.

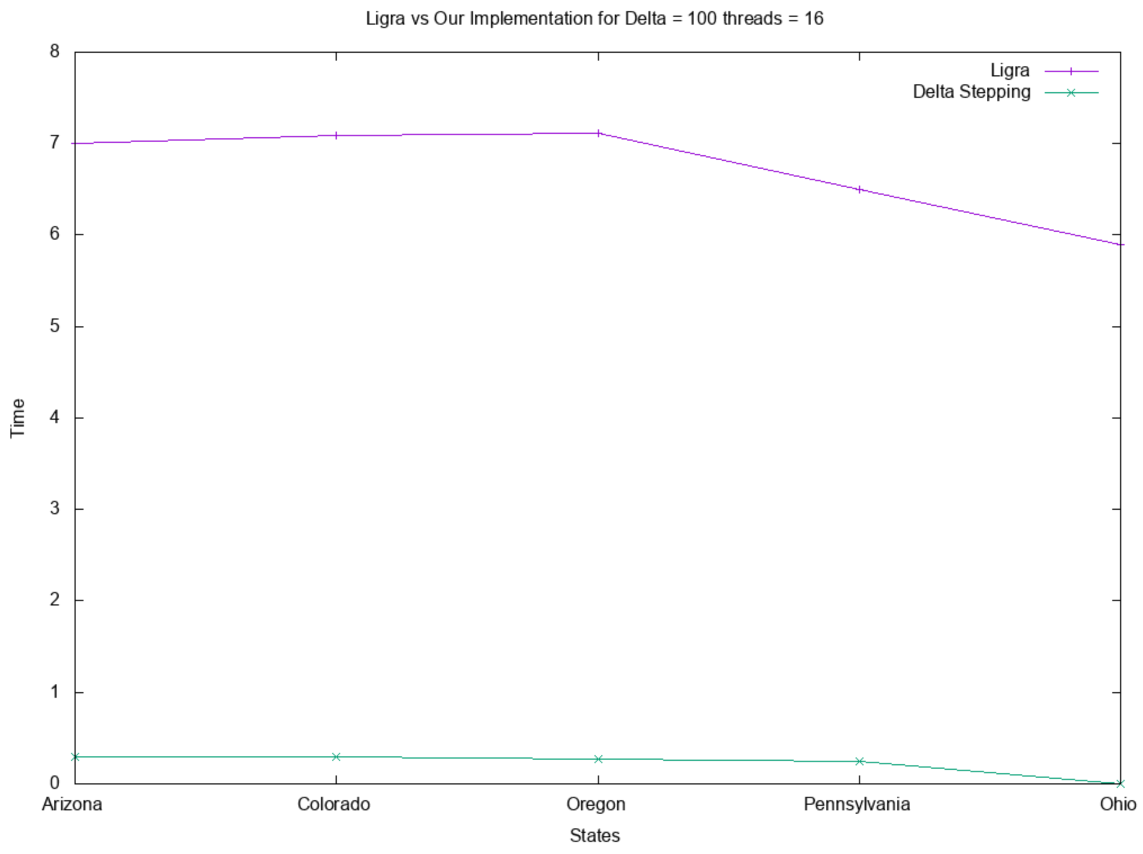
In figure 4.7, we test graph of Alabama with 4 threads and by increasing value of Δ. In this experiment we wanted to check, how do both of these algorithms react to different Δ values. One thing to notice is our implementation, outperformed Ligra's implementation being twice as fast.



(a) Delta Stepping vs Ligra for 4 thread and 100 Δ

Figure 4.8: Plots of Improved Delta Stepping and Ligra's Delta Stepping over various states with 4 threads and 100 Δ value.

Next, we compare for same thread and delta value, we measure performance of these two algorithms over different graphs. As we can notice, Improved Delta Stepping outperformed Ligra for most of the states, while Ligra's implementation in this experiment well only for Ohio.



(a) Delta Stepping vs Ligra for 16 thread and 100 Δ

Figure 4.9: Plots of Improved Delta Stepping and Ligra's Delta Stepping over various states with 16 threads and 100 Δ value.

Also, we tested same graphs again, but this time we increase number of threads to 16. We found out that our Improved Delta Stepping performs better in parallel environment, refer to figure 4.9. Whereas, Ligra's 1 thread outperformed improved Delta stepping.

CHAPTER 5: CONCLUSIONS

In this thesis, we explained why we shouldn't model performance based on random graphs for shortest paths algorithms, instead performance should be modeled based on the domain, it our case it was road networks. Moreover, we discussed Delta Stepping algorithm in detail, then we presented improved version of Delta Stepping.

Our implementation was compared with various implementations of Dijkstra's algorithm. We saw that single thread Delta Stepping had a similar running time as Dijkstra's algorithm using 4-ary heap and without heap decrement. Moreover, we compared our implementation with Ligra (Lightweight Graph Processing Framework), and we observed most of the time our implementation outperformed better than Ligra, the only exception being performance comparison with one thread. Where, Ligra outperforms our implementation, with a huge margin.

In future research, we still need to study on how to create a perfect Δ value for a particular graph such that we get maximum speedup. Also, we need to investigate why Ligra's Delta Stepping with 1 thread outperformed all other implementations.

REFERENCES

[1] K. M. T. A. B. T. K. M. Thaddeus Abiy, Beakal Tiliksew, “Graphs,” 2017.

[2] R. W. Floyd, “Algorithm 97: Shortest path,” *Commun. ACM*, vol. 5, pp. 345–, June 1962.

[3] J. Oldham, “A simpler generalized shortest paths algorithm,”

[4] L. Freeman, “A set of measures of centrality based on betweenness,” vol. 40, pp. 35–41, 03 1977.

[5] U. Brandes, “A faster algorithm for betweenness centrality,” *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

[6] U. Meyer and P. Sanders, “Delta-stepping: A parallel single source shortest path algorithm,” in *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA ’98, (London, UK, UK), pp. 393–404, Springer-Verlag, 1998.

[7] J. Shun and G. 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, (New York, NY, USA), pp. 135–146, ACM, 2013.

[8] T. Panitanarak and K. Madduri, “Performance analysis of single-source shortest path algorithms on distributed-memory systems,” in *Proc. 6th SIAM Workshop on Combinatorial Scientific Computing (CSC)*, pp. 60–63, July 2014.

[9] “Parallel delta-stepping algorithm for shared memory architectures,” *CoRR*, vol. abs/1604.02113, 2016. Withdrawn.

[10] “Breadth first search on massive graphs,” in *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, vol. 74, 2009.

[11] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak, “An experimental study of a parallel shortest path algorithm for solving large-scale graph instances,” in *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, SIAM, Jan. 2007.

[12] “High-performance multi-level graphs,” in *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, vol. 74, 2009.

[13] “K shortest path algorithms,” in *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, vol. 74, 2009.

[14] M. Sniedovich, “Dijkstra’s algorithm revisited: the dynamic programming connexion,” *Control and Cybernetics*, vol. 35, no. 3, pp. 599–620, 2006.

[15] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, pp. 596–615, July 1987.

[16] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," *J. ACM*, vol. 46, pp. 362–394, May 1999.

[17] M. Chen, R. A. Chowdhury, V. Ramachandran, D. L. Roche, and L. Tong, "Priority queues and dijkstra's algorithm," 2007.

[18] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[19] H. Gaurav and P. Manish, "Parallel implementations for solving shortest path problem using bellman-ford," vol. 95, June 2014.

[20] M. Jian, L. Ke-ping, and L. y. Zhang, "A parallel floyd-warshall algorithm based on tbb," in *2010 2nd IEEE International Conference on Information Management and Engineering*, pp. 429–433, April 2010.

[21] S. of the Parallel Processing Systems course, "Parallelizing the floyd-warshall algorithm on modern multicore platforms: Lessons learned,"

[22] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 456–471, ACM, 2013.

[23] C. Staudt, A. Sazonovs, and H. Meyerhenke, "Networkit: An interactive tool suite for high-performance network analysis," *CoRR*, vol. abs/1403.3005, 2014.

[24] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, pp. 496–509, Nov. 2011.

[25] Wikipedia, "Openstreetmap — wikipedia, the free encyclopedia," 2017. [Online; accessed 8-November-2017].

[26] Geofabrik, "Geofabrik," 2017. [Online; accessed 8-November-2017].

[27] P. Erdos and A. Renyi, "On the evolution of random graphs," in *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*, pp. 17–61, 1960.

[28] A. R. Monica Van Horn and D. Lopez, "A random graph generator,"

[29] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, Mar. 2010.

[30] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *SIAM International Conference on Data Mining*, 2004.

[31] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, *Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication*, pp. 133–145. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[32] A. Frieze and M. Karoński, *Introduction to Random Graphs.* Introduction to Random Graphs, Cambridge University Press, 2015.

[33] Valyala, "Generalized heap implementation," 2011.