A comparison of algorithms and heuristics for solving the O 1 I n D 1 shortest route problems

Steven D. Young
James Madison University

Follow this and additional works at: https://commons.lib.jmu.edu/honors201019
Part of the Other Computer Engineering Commons

Recommended Citation
Young, Steven D., "A comparison of algorithms and heuristics for solving the O 1 I n D 1 shortest route problems" (2016). Senior Honors Projects, 2010-current. 222.
https://commons.lib.jmu.edu/honors201019/222

This Thesis is brought to you for free and open access by the Honors College at JMU Scholarly Commons. It has been accepted for inclusion in Senior Honors Projects, 2010-current by an authorized administrator of JMU Scholarly Commons. For more information, please contact dc_admin@jmu.edu.
A Comparison of Algorithms and Heuristics for Solving O1P1D1 Shortest Route Problems

An Honors Program Project Presented to
the Faculty of the Undergraduate
College of Integrated Science and Technology
James Madison University

by Steven David Young

May 2016

Accepted by the faculty of the Department of Computer Science, James Madison University, in partial fulfillment of the requirements for the Honors Program.

FACULTY COMMITTEE:

Project Advisor:  David H. Bernstein, Ph.D.,
Professor, Computer Science

Reader:  Nathan Sprague, Ph.D.,
Assistant Professor, Computer Science

Reader:  Michael Lam, Ph.D.,
Assistant Professor, Computer Science

HONORS PROGRAM APPROVAL:

Bradley R. Newcomer, Ph.D.,
Director, Honors Program
## Contents

1. **Acknowledgments Page** .............................................. 7

2. **Abstract** ............................................................... 8

1. **Introduction** ............................................................ 1

   1.1 Context ........................................................... 1

   1.2 Motivation for this Research .................................... 7

   1.3 Organization of the Document ................................... 8

2. **Background** ............................................................. 9

   2.1 Definitions ......................................................... 9

   2.2 $O^1D^1$: Single-Source Shortest Path Problems ............ 12

   2.3 $O^1D^n$ and $O^nD^1$ ............................................. 18

   2.4 $O^1I^1D^1$: Shortest Route through an Intermediate Node .... 19

3. **The Problem** .......................................................... 21

4. **Algorithm Descriptions** .............................................. 24

   4.1 Brute Force Algorithm ........................................... 24

   4.2 Bi-Directional Heuristic ........................................ 32

   4.3 Multi-Label Heuristic ........................................... 42
5 Design and Implementation of the Empirical Analysis

5.1 Obtaining the Data ................................................................. 49

5.2 Design of Code ................................................................. 54
  5.2.1 Graph Structure ........................................................... 54
  5.2.2 Pre-Processing .............................................................. 55
  5.2.3 The Algorithms and Heuristics ....................................... 56

5.3 Utilities ................................................................. 56

5.4 Test Cases ................................................................. 56

6 Results of the Empirical Analysis ........................................... 66

6.1 Run Timing ................................................................. 66

6.2 Discussion ............................................................... 69

7 Conclusion and Directions for Future Research ...................... 71

8 Appendix ............................................................... 73

8.1 Code Listings ............................................................... 73
  8.1.1 Brute Force Algorithm .................................................. 73
  8.1.2 Bi-Directional Heuristic Algorithm ................................ 77
  8.1.3 Multi-Label Heuristic .................................................. 87
  8.1.4 DHeap ................................................................. 92
List of Figures

1. Fastest Routes Separately. .................................................. 6
2. Fastest Routes Stitched Together. ....................................... 6
3. Node visualization. ........................................................... 9
4. Node visualization. ........................................................... 9
5. Path including the same edge twice. ..................................... 11
6. A simple graph with a negative weight cycle. ....................... 14
7. A route with an intermediate destination thought of as a single route. . . . . . . 19
8. A route with an intermediate destination thought of as two distinct routes. . . . . . . 20
9. Route versus path distinction. ............................................. 20
10. Commuter problem. .......................................................... 22
11. Long Trip problem. .......................................................... 22
12. Simple intermediate set example. ....................................... 23
13. Brute Force algorithm example. ......................................... 31
14. Bi-Directional Heuristic algorithm example. ....................... 41
15. Multi-Label Heuristic algorithm example. ......................... 48
16. Depiction of some imaginary roadway. ............................... 50
17. Graphical representation of roadway built with solely raw mapping data. .......... 51
18. Roadway constructed using shape nodes. ............................ 51
19  A graph with unnecessary nodes. ........................................ 52
20  A graph with the unnecessary nodes removed. ........................... 53
21  Class diagrams for the Node and Edge classes. .......................... 55
22  Algorithm timings scaled against the Brute Force algorithm performance. 68

List of Tables

1  When to update a label in the Bi-Directional heuristic algorithm. ............. 32
2  Characteristics of each test case. ........................................... 59
3  Short description, illustration, and special set size for each test-case. .......... 65
4  Table of algorithm timings for test-cases. .................................... 67
5  Number of nodes finalized as "special" for select test-cases. ...................... 69
1 Acknowledgments Page

This paper would not have been possible without the guidance and help of Dr. David Bernstein of James Madison University.

This paper was informed by the invaluable advice of Dr. Nathan Sprague and Dr. Michael Lam of James Madison University.

The author would also like to thank the James Madison University Honors Program for creating the opportunity for this research.
2 Abstract

Shortest-path problems have seen a huge amount of study over the course of the last 50 years. Not surprisingly, this means the original problem of finding the shortest path from some starting location to some destination location has been thoroughly studied. However a large number of slight variations on the original problem still have yet to be thoroughly examined. In this paper we examine one of those variations, namely the problem of determining the optimal route from some source location to some destination location such that at least one location in an intermediate set is included in the route. Three algorithms/heuristics for solving this problem were developed and then tested against modified Virginia roadway data. The Brute Force Algorithm, Bi-Directional Heuristic and Multi-Label Heuristic each solve the problem in slightly different ways. Using a variety of representative test-cases on the Virginia roadways we found that the Brute Force algorithm greatly outperformed both the Bi-Directional and Multi-Label Heuristics. We also consider why this might be the case and discuss several promising directions for future work.
1 Introduction

In this chapter we provide a brief introduction to the work covered in this thesis. First, we provide a discussion of the context of navigation and routing research. Second, we cover the motivation behind our particular research and finally, we provide an overview of the organization of this paper.

1.1 Context

Roadway navigation is the problem of determining the shortest route from some starting location to some destination using a set of known roads. Roadway navigation is such a common and pervasive problem that a number of organizations have developed effective tools to solve it.

For instance, when you want to make a cross-country road trip to visit your relatives, you visit a mapping site such as Google Maps or Mapquest. Using a tool like this, you need only enter your starting point and desired destination and you are nearly instantly supplied with the fastest route containing every road, turn, and distance down to a tenth of a mile.

Now, there are also portable, dynamic in-car navigation systems and other portable navigation systems (e.g. on mobile phones). These units constantly evaluate your position and the desired destination to provide you with the shortest route. Today they even incorporate real-time, crowd-sourced, traffic information. Some navigation systems even use other dynamic variables such as road work, weather, and emergency situations to help plot the safest and most efficient route possible. Regardless of the additional information used in the decision making process, these systems all share the common goal of providing users with efficient routes quickly.

Similar systems exist for pedestrians and transit users. Stuck in New York and need the
quickest route to Times Square? No problem, modern navigation software can calculate the most efficient combination of mass transit and walking to reach your destination.

Routing problems are not limited to roadways and travel. The development of the internet has led to another version of the routing problem in packet and information routing. The internet is nothing more than a complex network of interconnected users. As such whenever we use the internet to look up a web-page or use the phone to call a friend about lunch, the information requested is transmitted across a massive network of phone lines and fiber-optic wire. Although we usually take it for granted, navigating such a large network is complex and time consuming. Furthermore, when you consider the massive amount of information traveling over these networks at any given time, if we are careless about the routes this information travels, it could result in network congestion and bad performance for everyone. In other words, without efficient information routing, your webpage may just hang. Because of this, it is hugely important that information on the internet travels as efficiently as possible.

The problem of network routing, in the most basic sense, is no different from vehicle navigation. There is a packet of information (car) that needs to get from a start location to a destination using a network of wires (roadways) in the smallest amount of time possible. Network routing also takes into account the amount of other information on the network (traffic) and unexpected down network connections (weather) in evaluating the various routes.

Most common algorithms for solving these problems work by expanding out from the starting point and gradually discovering the best way to get to locations farther and farther from the start location. Eventually these algorithms expand out to the destination and the algorithms get the most efficient route from the start location to the destination location.
This concept can be difficult to understand, but it can be thought of as considering every route connecting two locations as a rain gutter. In a car-navigation problem this would essentially recreate a road-map where all the roads all interconnected gutters. Then to start finding the routes we start to pour water into the gutters at the start location and the water will gradually spread out through the various gutters at the same speed in all directions. If we time at what point the water reaches each point in the gutter system for the first time, then we will know the "distance" from the start location to each point in the system of gutters. In these gutters, the points that can be reached more quickly will always fill with water before the points that require a longer path. Eventually, the water will travel through the gutters and reach the destination that we care about.

We can often find the optimal path faster if we search from both the starting point and destination simultaneously. We can also do better by "directing" our search and not continuing to search in a direction where we know the best path to the destination cannot lie.

To understand how the optimal path could be found faster if we started from both directions simultaneously, consider the example of the rain gutter map from above. Imagine pouring the water from both the starting location and destination location simultaneously. It turns out that we can essentially time the point where the water from the starting location meets the water from the destination location. Because the water is essentially expanding in a circle from each location, stopping when the circles meet can be much faster than waiting for one big circle to reach from one end to the other. For some mathematical intuition consider the distance from start to destination $D$ this would essentially form the radius of the big circle that forms if we only proceed in one direction. To approximate the number of locations that have been visited when the circle is this big, we can use the area of the circle $\pi R^2$. Now imagine that instead of one big circle, we have
two smaller circles that meet at the halfway point. Then we have two circles with radius $\frac{R}{2}$. The cumulative area of these two circles is: $\pi \left( \left( \frac{R}{2} \right)^2 \right) + \pi \left( \left( \frac{R}{2} \right)^2 \right) = 2\pi \left( \left( \frac{R}{2} \right)^2 \right) = 2\pi \frac{R^2}{4} = \pi \frac{R^2}{2}$ which is half of the area of the larger circle formed by proceeding in a single direction. Theoretically this smaller area would correspond to a smaller number of locations that need to be visited which in turn would correspond to finding the optimal result in a shorter amount of time.

To understand how "directing" the search could be helpful, remember that each time we expand in a particular direction it takes time. So if we can ignore some direction we know will not be useful, we could expand more quickly in a direction that may be useful. For example, if a user was traveling from Alabama to New York, there is likely little use in expanding the route searching into Florida. The time spent expanding in the Florida direction would likely better be spent expanding farther north. So the algorithm may realize this and stop searching in the Florida direction.

Through tremendous amounts of research, several techniques like those listed above have been developed to increase the efficiency of route finding. The attention that has been given to this field reflects the importance of routing applications.

Of course, all of this assumes that the algorithms have the necessary data. Fortunately, these data have been collected by both a variety of government agencies and companies. Originally, government organizations played a pivotal role in data collection, compiling huge map books compiled in the past into digitized systems. Now some private companies collect the data themselves for commercial products like online mapping systems and in-car navigation systems. Moreover, several online mapping services, like Open Street Map, have used crowd sourcing in conjunction with government data to plot out roadways.

Also important are the user’s objectives A number of metrics may be used to determine what
route is the "best" one. The measurement could be a clever combination of a large number of factors including geographic distance, speed limit, and even the number of left turns versus right turns. Usage has shown that these factors can have a large effect on the routes actual efficiency. Furthermore, in reality a large number of other preferences likely contribute as well. Most people do not like to get on and off major highways multiple times and such a route may be considered "less optimal" than a route that remains on the highway for a majority of the trip.

Modern routing software is not limited to finding the quickest route from point A to point B, or from A to all other points. Some in-car navigation systems can add "waypoints," which are intermediate locations to visit on the way. For example, if a family were going on a cross-country trip from Virginia to California, they may want to stop for sight-seeing at the Mount Rushmore and then the Grand Canyon.

While the waypoint routing problem may seem drastically different at first glance, it is actually very similar to the general problem. It can be solved by finding the shortest route from the starting point to the first waypoint and then the shortest route from the first waypoint to the second, and so on until you have found the shortest route between all sequential points. Then you stitch the routes together and you have the answer. In the cross-country example above, you would first find the shortest route from Virginia to Mount Rushmore, then concatenate that route onto the shortest path from the Mount Rushmore to the Grand Canyon, and finally concatenate that path with the shortest path from the Grand Canyon to California. The route with all the smaller routes stitched together is the shortest overall route that includes all waypoints. This is illustrated in Figures 1 and 2.

It is important to notice that this sort of technique is only effective if the order of the waypoints
is specified. If the user needs to determine the fastest route that goes through some number of waypoints and also needs to determine what order to visit the waypoints, the problem is much more complicated. To gain intuition on why this is the case. Imagine a such a problem with 15 waypoints. To solve this problem with the aforementioned stitching strategy, the algorithm would need to stitch a route for each possible ordering of the waypoints. This would result in $15!$ stitchings or $13,000,000,000,000$ possible stitchings. Such a process would be hugely costly and not reasonable without some sort of trick to reduce the number of possibilities.

Not only can routing software find the fastest route from a single starting location to a single
destination, and the shortest path that passes through "waypoints," but it can also find the fastest route from a single node to a large number of destinations. For example, if someone was attempting to determine which of several hardware stores in their area to visit, they would want to simultaneously evaluate the distance to all of the hardware stores and then pick the one with the hardware store shortest distance. In network routing, a packet of information may need to be delivered to a large number of recipients. In such a case the routing software may calculate the optimal path to all of the destinations simultaneously. Solving this problem is another simple extension of the original algorithm; simply continue to extend outward until you have found the shortest route to all of the destinations.

1.2 Motivation for this Research

A number of powerful techniques exist to find the routes from a single location to some number of destinations with the added constraint that the route contains a specific waypoint. In this paper we ask a slightly different question. What if we are not concerned with passing through a single specific waypoint but are only interested in passing through a general area? For example, instead of traveling through a specific street on your way from Virginia to New York, you merely want to make sure you pass through a particular town in New Jersey. You don’t care what part of the town you pass through specifically, only that you pass through some part of the town.

This is a different problem from those discussed above and it cannot be solved by a trivial extension of existing algorithms. Hence, the purpose of this research is to develop several algorithms/heuristics for solving this problem and then compare their performance on actual road networks.
1.3 Organization of the Document

This thesis is organized as follows: in chapter 2 we discuss the formal approaches used to solve basic routing problems. In chapter 3, we provide a more formal description of the problem we are exploring. Then, in chapter 4 we discuss the experimental algorithms that we studied to solve our intermediate set problem. In chapter 5 we discuss the methods used to obtain and prepare the data for testing these algorithms. In chapter 6 we present the empirical results of testing our algorithms. Finally, in chapter 7 we conclude with a discussion of future research.
2 Background

In this chapter we both describe the notation we will use, define some important terms, and provide a summary of past research relevant to this thesis. In order to discuss this material in an efficient manner, we begin by covering relevant definitions for terms used in the remainder of this paper.

2.1 Definitions

Throughout this thesis we use the following notation and definitions (taken from [4]).

- Given an ordered graph $G(V, E)$ composed of the pair $(V, E)$ where $V$ is a non-empty finite set of distinct nodes (or vertices) and $E$ is a finite set of distinct ordered pairs of elements of $V$ called edges.

As shown in Figure 3, a node can be visualized as a dot and as shown in Figure 4, an edge can be visualized as a line connecting two dots (nodes).

![Figure 3: Node visualization.](image)

- An edge may be directed or undirected. A directed edge only allows travel in a single direction. An undirected edge may go in either direction. In all of the figures in this document.
the edges are undirected (or, in another sense, each undirected edge represents two directed edges with opposite directions).

- In a **weighted graph**, each edge has an associated value known as a "weight." A weight can represent any number of things depending on the context of the graph. For instance, in the context of the navigational problem, a weight could represent a driving distance or a travel cost.

- On a graph, a **walk** is a finite sequence of connected edges of the form \((v_0, v_1) \rightarrow (v_1, v_2) \rightarrow \ldots (v_{k-1}, v_k)\) where \(v_0\) is referred to as the **initial** vertex and \(v_k\) is referred to as the **final vertex**. [4]

- A walk where every edge is distinct is called a **trail**.

- A trail where every vertex is distinct is called a **path**.

- A graph where for any vertices \(x\) and \(y\) there exists a path from \(x\) to \(y\) is said to be **connected**.

- We use the word **route** to informally refer to walks, trails, paths and other related forms.

  It is essential to include this notion, because, given the nature of our problem it is possible for the solution to the problem to "double back" on itself resulting in a route that looks like the example in Figure 5. That is, it is possible for the same edge (or edges) to be included in the route multiple times if the route uses those edges to "reach out" and include the intermediate set.

- In a directed graph, an edge is said to be **outgoing** with respect to a node, if the edge originates at that node (i.e. if the node is the first element of the ordered pair).
• Likewise, an edge is said to be incoming with respect to a node, if said edge terminates at the node (i.e. if the node is the second element of the ordered pair).

• A cycle is a walk that begins and ends with the same node.

• A cycle in which the sum of the weights of the edges is less than zero is a negative weight cycle.

Throughout the remainder of this thesis we will classify problems in terms of $O^xD^y$ notation. Such a problem involves finding the shortest route from each of $x$ origins to each of $y$ destinations. For example, an $O^1D^1$ problem involves finding the shortest route from a single origin node to a single destination node, and a $O^nD^1$ problem involves finding the shortest route from each of $n$ origin nodes to a single destination node.

Furthermore, some of the problems in this thesis are expressed as $O^xI^zD^y$. These problems involve finding the optimal route from each of $x$ origin nodes to each of $y$ destination nodes, such that at least one node from $I$ is included in the path (where $I$ contains $z$ nodes). For example, an $O^1I^nD^1$ problem involves finding the shortest route from a single origin node to a single destination node such that some node $j \in I$ is on the route where $|I| = n$. 
2.2 $O^1D^1$: Single-Source Shortest Path Problems

The first class of problems we consider is the $O^1D^1$, better known as single-source single-destination shortest path problems. A class of problems has been the target of an enormous amount of research over the last 50 years.[1]

The most well-known algorithm for solving this problem was developed by Edgar Dijkstra in 1956.[2] Dijkstra’s algorithm, which is an example of a greedy algorithm, starts by labeling all nodes except for the starting vertex with a value of $\infty$. The starting node is given a label of zero, and a working queue is created containing only the starting vertex. The algorithm repeatedly selects the node in the queue with the smallest label (called $v$ here) and removes it from the queue, finalizing its label. Then, for each outgoing edge from $v$, if the label of the vertex on the other end of the edge is less than the sum of $v$’s label and the length of the edge, the other vertex’s label is updated to the smaller of those two values. If the other vertex is not already finalized then it is added to the queue. The algorithm ends when the destination node is finalized.[2][1] This process is summarized in Pseudocode 2.1.
Pseudocode 2.1: Dijkstras.

```plaintext
function Dijkstras(Graph, source_node, destination_node):

   for Node n in Graph:
      Labels[n] = infinity
   Labels[source_node] = 0
   Queue.add(source_node)

   while Queue is not empty:
      current = Queue.pop_smallest()
      if current == destination_node:
         return Labels[destination_node]
      for Edge e in V.edges:
         temp = Labels[V] + e.length()
         if temp < Labels[e.destination]:
            Labels[e.destination] = temp
            if (e.destination is not finalized)
               Queue.add(e.destination)

   return infinity
```

It is important to notice that the algorithm presented in Pseudocode 2.1 does not actually return the shortest path, just its length. However it can be easily extended to provide to order of the nodes in the shortest path by making each node keep track of its current predecessor. A node’s predecessor is the node which most recently altered this nodes label.

Dijkstra’s algorithm as described in his original paper has a time complexity of $O(V^2)$. Dial achieved some degree of asymptotic improvement using more efficient storage structures for the storage of nodes. To see how, consider the effect that the node data structure has on the efficiency of the algorithm. At every iteration, the algorithm must query the structure to obtain the node with the smallest label. Because of the frequency of this operation, there exists a need for a data structure that can efficiently maintain the node with the smallest label and can support quick
insertion. Therefore, using a heap data structure leads to the best performance. For example, using a Fibonacci heap the complexity becomes $O(|E| + |V|\log|V|)$. This represents a considerable improvement in sparse graphs where $E << |V|^2$.

Because Dijkstra’s algorithm always finalizes the node with the smallest label, negative weight cycles present an issue. Consider the graph shown in Figure 6:

![Figure 6: A simple graph with a negative weight cycle.](image)

Notice that going from node B to node C back to node B results in a total distance of -1. Therefore, an infinitely short path could be constructed by repeated traveling through this cycle. Hence, if you apply Dijkstra’s algorithm on this graph it will not terminate. It will repeatedly finalize nodes B and C decreasing the label each iteration. Because of this problem, Dijkstra’s algorithm does not work on a graph with a negative weight cycle. However, in the case of navigation research, this is likely not an issue because there is no real concept of a “negative” distance or travel time.

While Dijkstra’s algorithm performs well, it is possible to do even better with some simple modifications[5]. One such modification is involves working from both the start and destination vertices. This algorithm, is called "Pohl’s algorithm."

In Pohl’s algorithm two queues of nodes are maintained, one for the nodes reached from the
start vertex and one for the nodes reached from the destination vertex. Every node will also now have two labels, one representing the best known distance from the start vertex and another representing the best known distance from the destination vertex. The algorithm takes turns, alternating between iterations from the start vertex and from the destination vertex. The optimal path can be determined as soon as there is a single vertex that has been finalized by both the start vertex and the destination vertex.

Though counter-intuitive at first glance, the shortest path does not necessarily involve the vertex that has been finalized in both directions. This is, the algorithm needs to "clean-up", checking for edge cases. In order to provide some intuition as to why this is necessary, consider the moment when a common node is finalized. Call the distance from the source node to the common node $D_s$ and the distance from the destination node to the common node $D_d$. Now, observe that it is impossible for another path to exist where for some node on said path, both the distance to the source and the distance to the destination is shorter than $D_s$ and $D_d$ respectively. However it is possible for a shorter path to exist where some node has a shorter distance to either the source or destination nodes. The cleanup step checks for this condition. The cleanup is done by iterating through all of the nodes finalized by the source node and checking to see if a path can be made from the node, to the destination node by looking at the nodes pointed to by edges coming out of these nodes.[6] The complete algorithm is summarized in Pseudocode 2.2.
function BiDirection(Graph, source_node, destination_node):
    for Node n in Graph:
        source_labels[n] = dest_labels[n] = infinity
        source_labels[source_node] = 0
        dest_labels[destination_node] = 0
        source_queue.add(source_node)
        dest_queue.add(destination_node)
    while source_queue or dest_queue is not empty:
        // Forward iteration
        source_current = source_queue.pop_smallest()
        source_finalized.add(source_current)
        for Edge e in current.edges:
            temp = source_labels[source_current] + e.length()
            if temp < source_labels[e.destination]:
                source_labels[e.destination] = temp
                source_queue.add(e.destination)

        // Backwards iteration
        dest_current = dest_queue.pop_smallest()
        dest_finalized.add(dest_current)
        for Edge e in current.edges:
            temp = dest_labels[dest_current] + e.length()
            if temp < dest_labels[e.destination]:
                dest_labels[e.destination] = temp
                dest_queue.add(e.destination)

    // Check for end and clean up
    if source_finalized.contains(dest_current) or dest_finalized.contains(source_current):
        // Clean
        if source_finalized.contains(dest_current):
            final_node = dest_current
        else:
            final_node

Pseudocode 2.2: Pohls Algorithm.
final_node = source_current

best_length = source_labels[final_node] + dest_labels[final_node]

for Node f in source_finalized:
    for Edge e in f.edges:
        if source_labels[f] + e.length() + dest_labels[e.destination] < best_length:
            best_length = source_labels[f] + e.length() +
                          dest_labels[e.destination]

return best_length
Another class of problems involves finding the optimal path from a single source node to all other nodes in the graph. Dijkstra’s algorithm (and other so-called label-setting algorithms) may be extended to accomplish this by allowing the labeling process to continue until all nodes in the graph are finalized. Additionally, the Bellman-Ford algorithm, which is summarized in Pseudocode 2.3, identifies the optimal path from a source node to all other nodes. While the Bellman-Ford algorithm is asymptotically slower than Dijkstra’s algorithm: $O(|E| \times |V|)$, it has the additional benefit of being able to identify a negative weight cycle.

Pseudocode 2.3: Bellman Ford Algorithm.

```python
function Bellman_ford(Graph, source_node):
    for Node n in Graph:
        Labels[v] = infinity
    Labels[source_node] = 0
    for k in 1...n-1:
        for Edge e in Graph.edges:
            Labels[e.destination] =
            min(Labels[e.destination], Labels[e.source] + e.length())
    // After the function returns, Labels will contain the length of the shortest path
    // to all Nodes from source_node
```
2.4 $O^1I^1D^1$: Shortest Route through an Intermediate Node

Travelers do not always simply want to travel from some origin to a destination. It is often desirable to travel through some given intermediate location. Luckily, the aforementioned algorithms may be easily extended to find the optimal route from a source node to a destination node such that the path contains some intermediate node. To see the intuition behind the extension, consider traveling from your home to work and stopping at the local coffee shop on the way as shown below in Figure 7. This problem can be thought of as two separate problems. First, find the shortest route from your home to the coffee shop and second, find the shortest route from the coffee shop to work. As shown in Figure 8, after you find both of these routes, then concatenating them will result in a route that solves the overall. Furthermore, this combined route must be the optimal one.

Figure 7: A route with an intermediate destination thought of as a single route.
Figure 8: A route with an intermediate destination thought of as two distinct routes.

It is important to point out one detail of the problem that is easy to overlook. We have carefully called the solution a route and not a path because the resulting structure may not, in fact, be a path. To see why, consider the example in Figure 9. The shortest route from A to E through D must pass through the vertex C twice. Hence there is no path A to E through D. Yet, from a practical standpoint, we certainly would not want to say that one cannot travel from A to E through D.

Figure 9: Route versus path distinction.
3 The Problem

While the intermediate node problem \((O^1I^1D^1)\) allows for the identification of a route that passes through a single given node, it is extremely limited. In reality, there are many situations where a user does not really want to pass through a single, specific address, but really just cares about being routed through a particular region. For example consider a typical long commute. Suppose you hear that there is a huge amount of congestion on your usual route. Although you do not know any specific road or address in the region, you know that there is a pleasant alternative route through uptown. So in this situation you want to find the fastest route through this region (uptown) but you do not care about passing through any specific address.

More formally, we define the \(O^1I^1D^1\) to be the problem of finding the shortest route from some initial vertex to some destination vertex on a graph \(G(N,A)\) such that a single node from a specified subset of the set of nodes, \(S\), is included in the route. More formally, we intend to find a route from the source node to the destination node where there exists some node \(n_i\) on \(P\) such that \(n_i \in S\). The solution to the \(O^1I^1D^1\) problem is the route with the smallest sum of subsequent edge lengths.

There are a number of specific "real-world" situations that motivate this research. One such problem is that of a commuter who hears there is a large amount on congestion in a specific region \(A\) they normally pass through on their commute. The commuter may have heard that the route through region \(B\) is an excellent detour and desires the shortest route from their home through region \(B\) to their work as illustrated in Figure 10.
Another "real-world" problem is that of the "long-trip." In this scenario a traveler wishes to travel from their home to a distant city. The traveler desires to pass through a specific city. One such example, as illustrated in Figure 11, is that of a traveler desiring to go from Danville, Virginia to Alexandria, Virginia through Richmond, Virginia.

Figure 11: Long Trip problem.
In Figure 12 we see a simple example graph with a two node intermediate set. There are three routes from A, the starting node to E, the destination node. The route through B is of length 2, the route through C is of length 3 and the path through D is of length 4. The path through C is the solution to the problem because it is the shortest route that contains a node that lies within the intermediate set. We seek an algorithm that can successfully identify this route.

Figure 12: Simple intermediate set example.
4 Algorithm Descriptions

In this thesis, we look at three strategies for solving this problem. The first method is fairly straightforward and has therefore been dubbed the "Brute Force" Algorithm. This approach uses traditional Bi-Directional Dijkstra’s to identify the optimal routes from both the origin and destination nodes to every node in $S$. Then it iterates through every node in $S$ concatenating the optimal origin and optimal destination routes together and finding the shortest summed route.

The second approach applies a heuristic to the Brute Force Algorithm. Dubbed the "Bi-Directional Heuristic," this algorithm proceeds the same way as the Brute Force Algorithm, but stops finalizing once both directions have finalized a some common node in the $S$. This algorithm then iterates through all nodes that have been finalized through $S$ and checks to see if they can form a better route than the one created by the common node.

The final approach, dubbed the "Multi-Label Heuristic" only proceeds from a single direction. It performs Dijkstra’s algorithm from the origin node as normal, but all nodes also have a "special" label. The special label is changed when a node is labeled in the intermediate set, or by a node with a special label. The algorithm terminates when the destination node’s special label is finalized.

4.1 Brute Force Algorithm

One solution to the $O^{1}I^{n}D^{1}$ problem involves examining a large number of possible solutions and then determining which solution has the least cost. Begin by performing Dijkstra’s algorithm simultaneously from both the initial and destination vertices. Once both of the algorithms have finalized all nodes in $S$ we then find the minimum of the sums of source and destination labels for
each node in $S$. This minimum is the shortest route such that at least one node is contained in $S$ and is therefore a solution to the $O^1I^nD^1$ problem.

To formalize this idea, we first define the notion of a Dijkstra Iteration in Pseudocode 4.1. These steps will be used in the algorithm and heuristics that follow.

Pseudocode 4.1: Dijkstra Iteration.

```
Dijkstra Iteration:
1  current = Queue.pop_smallest()
2      if current == destination_node:
3          return Labels[destination_node]
4  for Edge e in V.edges:
5      temp = Labels[V] + e.length()
6      if temp < Labels[e.destination]:
7          Labels[e.destination] = temp
8          if (e.destination is not finalized)
9              Queue.add(e.destination)
```
Pseudocode 4.2: Brute Force Algorithm.

```python
function BruteForce(Graph, source, destination, S):
    source_count = S.size
    dest_count = S.size
    while a node remains in source or destination queues:
        Dijkstra Iteration from source
        Dijkstra Iteration from destination
        if node finalized by source is in S:
            source_count = source_count - 1
        if node finalized by destination is in S:
            dest_count = dest_count - 1
        if dest_count <= 0 and source_count <= 0:
            best_dist = infinity
        for Node n in S:
            best_dist = min(best_dist, source_labels[n] + dest_labels[n])
        return best_dist
    return infinity
```

Essentially the Brute Force algorithm approach solves the $O^1D^n$ problem from both the start and destination nodes using the nodes from the set $S$ as the destinations. Then it iterates through all of the nodes in $S$ concatenating the optimal path from the source node with the optimal path from the destination node. The best of all these paths is the solution to the $O^1I^mD^1$ problem. This is detailed in Pseudocode 4.2.

This algorithm will likely perform very well when $S$ is small and roughly equidistant from the source and destination nodes. This is because if $S$ is roughly equidistant from the source and destination nodes, the algorithm from both sides will only need to expand to half of the distance between the two nodes. If $S$ is small than iterating through all nodes in $S$ will be less costly. Conversely, the algorithm will certainly perform poorly if $S$ contains both the source and destination.
nodes and then contains a large number of nodes spreading out from them.

The best way to understand this algorithm is to consider a small example.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Destination</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>

To begin, all nodes are given labels of infinity from both the source node (A) and destination node (F). The source and destination nodes are then given a label of zero for their respective directions.
<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>∞</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Destination</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

In this iteration A is finalized by the source directions and F is finalized by the destination. The label for B is updated to 2 and the label of D is updated to 1 for the source direction. The label of C is updated to 2 and the label of E is updated to 1 for the destination direction.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>∞</td>
<td>1</td>
<td>2</td>
<td>∞</td>
</tr>
<tr>
<td>Destination</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Now D and E are finalized by the source and destination directions respectively because they have the smallest remaining labels. The labels of neighboring nodes are updated accordingly.
For this iteration we have a tie from the source node because both B and E have a label of 2 and similarly, there is a tie between C and D from the destination node. We break the tie arbitrarily and finalize B and C in the source and destination directions respectively and update the labels accordingly.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td>∞</td>
</tr>
<tr>
<td>Destination</td>
<td>∞</td>
<td>12</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

For this iteration we finalize E and D in the source and destination directions respectively and update the labels accordingly.
<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Destination</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

For this iteration we finalize F and A in the source and destination directions respectively and update the labels accordingly.
Figure 13: Brute Force algorithm example.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Destination</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

For this iteration we finalize C and B in the source and destination directions respectively and update the labels accordingly. All nodes in the intermediate set have now been finalized by both the source and destination directions so we stop iterating.

Finally, we iterate through each node in the intermediate set, summing their source and destination labels and find that the optimal route length in this case is 6 (either the sum on B or the sum on C because in this example the sums are the same; this is not true in general).
4.2 Bi-Directional Heuristic

Another solution to the $O^{1/p}D^1$ problem, referred to here as the Bi-Directional Heuristic algorithm, proceeds in much the same way as the Brute Force Algorithm but takes advantage of some heuristics to improve the running-time. Instead of waiting for all nodes in $S$ to be finalized in both the source and destination directions, we can terminate as soon as a single node in $S$ has been finalized in both the source and destination directions. However, we need to take some extra precautions because the optimal path could enter and exit the intermediate set at several points. As soon as we finalize a node in the intermediate set, it is finalized as a special node and all of the nodes finalized from a special node are special. All nodes now have both a special and normal label from both the source and destination node.

This concept is summarized in Table 1, which explains when a label is updated; in this table the row is the type of label that was just finalized, and the column is whether or not the neighbor node being examined is in the special set.

<table>
<thead>
<tr>
<th></th>
<th>In $S$</th>
<th>Outside $S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Label</td>
<td>Special and Normal</td>
<td>Normal</td>
</tr>
<tr>
<td>Special Label</td>
<td>Special and Normal</td>
<td>Special and Normal</td>
</tr>
</tbody>
</table>

Table 1: When to update a label in the Bi-Directional heuristic algorithm.

In the clean-up step we iterate through all nodes that have been finalized as special and attempt to construct a better route using the special labels. After we finish iterating, we hopefully have the optimal route. Note that though the intuition for this clean-up process is based on that for Pohl’s
Algorithm, we have not proven that the solution is optimal. Hence, this process is heuristic. The complete heuristic is summarized in Pseudocode 4.3.

Pseudocode 4.3: Bi-Directional Heuristic Algorithm.

```python
function Bi_Directional(Graph, source, destination, S):
    while a node remains in source or destination queues:
        Dijkstra Iteration from source
        if node with changed label is in special:
            update special label
        if finalized node has a special label:
            all nodes pointed to have their special labels updated

        Dijkstra Iteration from destination
        if node with changed label is in special:
            update special label
        if finalized node has a special label:
            all nodes pointed to have their special labels updated

        if node is special and
        finalized by source of destination has been finalized by other:
            best_dist = infinity
            for Node n in nodes with special labels from source:
                for each Edge e in outgoing from n:
                    best_dist = min(best_dist,
                                    source_special_labels[n] +
                                    e.length + dest_special_labels[n])
            return best_dist
    return infinity
```

This heuristic will likely perform well when $S$ is equidistant from the source and destination nodes. This will keep the number of nodes with special labels from the source node to a minimum, decreasing the size of the workload. Conversely, we expect this heuristic will perform poorly when
$S$ lies significantly closer to one of the nodes because it results in a costly cleanup step.
Again, the easiest way to understand this heuristic is with a simple example.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Destination</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>Source Special</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Destination Special</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

The source and destination labels represent the optimal known route from the source or destination node to a given node. The corresponding special labels represent the optimal route from the source or destination node that passes through the intermediate set.

In the first iteration, we give all labels a value of infinity except for the source and destination nodes which receive a label of zero for their respective directions.
In this iteration we finalize A and F from the source and destination respectively. Then we update all labels accordingly. Because B and C lie inside the special set, their special labels are also updated.

In this iteration we finalize D and E from the source and destination respectively. Then we update all labels accordingly.
<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td>∞</td>
</tr>
<tr>
<td>Destination</td>
<td>∞</td>
<td>12</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Source Special</td>
<td>4</td>
<td>2</td>
<td>12</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Destination Special</td>
<td>∞</td>
<td>12</td>
<td>2</td>
<td>∞</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

In this iteration, there is a tie between the B normal, B special, and E normal labels in the source direction as well as between the C normal, C special and D normal labels in the destination direction.

We break the tie arbitrarily and choose to finalize B in the source direction and C in the destination direction. Although it technically takes two iterations to finalize B normal and B special, and C normal, C special, in this diagram we show both finalizations in the same iteration for the sake of brevity.

All neighboring labels are updated. If a special label is finalized, neighboring special labels are effected.
In this iteration we finalize E and D from the source and destination directions respectively.

The labels are updated accordingly.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Destination</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Source Special</td>
<td>4</td>
<td>2</td>
<td>12</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Destination Special</td>
<td>∞</td>
<td>12</td>
<td>2</td>
<td>∞</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

In this iteration we finalize F and A from the source and destination directions respectively.

The labels are updated accordingly.
In this iteration there is a tie between C normal, D special and A special in the source direction and B normal, E special and F special in the destination direction. We break it arbitrarily and finalize the D special and E special labels, updating neighboring labels accordingly.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Destination</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Source Special</td>
<td>4</td>
<td>2</td>
<td>12</td>
<td>4</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>Destination Special</td>
<td>∞</td>
<td>12</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

In this iteration there is a tie between C normal and A special in the source direction and B normal and F special in the destination direction. We break it arbitrarily and finalize the A special and F special labels, updating neighboring labels accordingly.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Destination</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Source Special</td>
<td>4</td>
<td>2</td>
<td>12</td>
<td>4</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>Destination Special</td>
<td>∞</td>
<td>12</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
In this iteration C normal and B special are finalized in the source and destination directions respectively and the neighboring labels are updated accordingly.
In this iteration, E special and D special are finalized in the source and destination directions respectively. Because we have a common node with a special label from both directions, we are done. Next we iterate through each of the nodes with finalized special labels, checking the destination normal labels of their neighbors. We create a path by summing the source special label of the node finalized from the source and the destination label, and edge length of the edge connecting them. The lowest source label/destination label sum is the optimal path. In this case that number is 6, which is the sum of the special label of D from source (4) and the normal label of E from the destination (1), and the length of the edge connecting them (1).
4.3 Multi-Label Heuristic

The final technique studied here is a modification of Dijkstra’s original algorithm. The heuristic begins by giving each node two labels. The first label will be called the normal label and the second label will be referred to as the special label. As the algorithm proceeds, the normal label is used just as the standard label in Dijkstra’s algorithm. As soon as a node in $S$ is labeled, its special label will also be updated. Both the special labels and normal labels are treated equivalently by the heuristic. At each iteration, the smallest label out of both the special labels and normal labels will be selected to be finalized. If a normal label is finalized, all normal and special labels of adjacent nodes will be updated as described above. If a special label is finalized, then only adjacent special labels will be updated.

This heuristic uses the same rules for label updates as the Bi-Directional Heuristic. While it shares many of the same steps as the Bi-Directional Heuristic algorithm, it is fundamentally different in that it has no clean-up steps and only proceeds in a single direction. The heuristic terminates when the special label of the destination node is finalized. The value of that label hopefully is the solution to the $O^1 I^p D^1$ problem. Again in this paper, we do not prove that this heuristic does produce the optimal solution. The complete heuristic is summarized in Pseudocode 4.4.
Pseudocode 4.4: Multi-Label Heuristic.

1 function Multi_Label(Graph, source, destination, S):
2 while a node remains in source or destination queues:
3     Dijkstra Iteration from source
4     if the node smallest label is special:
5         label updates will be of the special nodes
6     if the node being updated is in the special set:
7         the special label will be updates as well as the normal label
8     if finalized node is the destination and special:
9         return special_labels[destination]
10 return infinity

This heuristic will likely perform well when the special set lies close to the destination node. In this case the number of nodes that need to be finalized twice will be relatively small. In the "worst" case, $S$ will lie in the opposite direction of the destination with respect to the source node. In this case a huge number of nodes will need to be finalized twice before the algorithm ends.
Now we present a simple example of the algorithm running:

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>∞</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Source Special</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

To begin, we initialize all labels to infinity except for the label of the source node (A) which receives a label of 0. Here we also finalize A and update the labels of B and D accordingly. Because B is in the intermediate set, its special label is also updated.
Now we finalize D and update the values of neighboring nodes appropriately.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>∞</td>
<td>1</td>
<td>2</td>
<td>∞</td>
</tr>
<tr>
<td>Source Special</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

In this case there is a tie between B and E. It is broken arbitrarily and we choose to finalize E.

The neighboring labels are updated accordingly and because C is in the intermediate set, its special label is updated.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Source Special</td>
<td>∞</td>
<td>2</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Here, B is finalized. Because it’s in the intermediate set, all of its neighbors have their special labels updated instead of their regular labels.
Now we finalize F and update labels of neighbor nodes accordingly.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Source Special</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Now there is a three way tie between A, C, and D, all in special labels. We arbitrarily choose to finalize the D special label and update all neighboring special labels appropriately.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Source Special</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>∞</td>
</tr>
</tbody>
</table>

Now there is a two-way tie. We choose to finalize the special label of A, and update all neighboring special labels accordingly. In this case no labels will change.
<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Source Special</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Now we finalize the special label of C and update all neighboring special labels accordingly.

<table>
<thead>
<tr>
<th>Label Type</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Source Special</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Now we finalize the special label of E and update all neighboring special labels accordingly.

In this case no labels will change.
Now we finalize the special label of F, the destination node. Because we have finalized the special label of the destination node we are finished. The length of the optimal route is the value of that finalized label: 6.
5 Design and Implementation of the Empirical Analysis

We now consider various issues related to the design and implementation of the empirical analysis of the algorithms and heuristics discussed in the previous chapter. We begin with a discussion of the data then discuss the code, and finally discuss the test cases.

5.1 Obtaining the Data

Our concern is with the "real-world" empirical performance of the algorithms and heuristics. Hence, we used a "real-world" road network.

The raw data was obtained from the Open Street Map database on September 22nd, 2015. The file was an XML file that contained all of the map data for the state of Virginia. In its original form the file was over 5 GigaBytes in size. It contained many unwanted elements, most of which were for drawing buildings and other features for OSM’s graphical interface. First we parsed the file using the OSM Filter program to remove all entries that were not vehicle roadways.

We then ran the file through a series of Java programs to convert the file to a usable form. The first program parsed the XML file and removed all XML tags that were not either nodes (geographic positions with a latitude and longitude) and ways (roads that are made up of an ordered series of nodes). Next we parsed the reduced XML file and converted the way and node tags to create a file that lists the nodes of our final graph and a file that lists all of the edges. The nodes were listed as an ID number, a latitude and a longitude. The edges were listed as the ID of the start node, the ID of the end node, and the geometric distance between the two points as a function of the geographic coordinates of the two nodes. While the size of the file was greatly reduced at this
point, it was still far too large to work efficiently. The major issue with the graph at this point was that the original data came with a number of "shape nodes". These nodes are used by the OSM GUI to draw the curves of the graph. Because raw graph data for roadways is essentially a starting coordinate, an ending coordinate, and a distance, in order for mapping software to generate the graphical depictions that actually look like roadways, a large number of shaping nodes must be added to the roadway data. Keeping in mind that map data breaks a real roadway into chunks that do not have any roads coming off of them (i.e. bounded by intersections with other roads) consider the fake roadway shown here in Figure 16: While this may be what the actual roadway looks like,

![Figure 16: Depiction of some imaginary roadway.](image)

for the purposes of electronic storage, the roadway is likely stored essentially like as in Figure 17.

Where the only information is the coordinate of the beginning intersection, the coordinate of the ending intersection, and then the actual distance of the road. Although addition meta-data is also stored (such as if the road is a highway, residential street etc.) for the purposes of this example only consider the three factors above.
While this representation is sufficient for routing software, it is not very appealing to the eye and cannot be used to build user-friendly GUI. Therefore, the roadway is divided into a large number of smaller, straight roadways such that they can be stitched together to form a more accurate visual representation. Such a representation may look like the illustration in Figure 18.

The creation of the additional smaller roadways results in a large number of extra shaping nodes. These shaping nodes are entirely cosmetic and serve no useful purpose to a routing algo-
rithm because they of the way the mapping data represents roadways. A roadway must be atomic, it cannot have any point where another roadway intersects with it. Therefore, because all of these smaller roadways constructed for shaping purposes cannot possibly branch onto another roadway, once you start on a roadway, you cannot change to another at any of the shaping nodes. This means that for routing purposes, where we are only concerned with possible routing choices, all shape nodes on a roadway effectively represent the same routing choice and are therefore redundant to us.

Thus, we removed the shape nodes from the map data by stitching together the nodes around them. A shape node can be identified because it only has one incoming edge and one outgoing edge. For example consider the graph in Figure 19: The nodes in between the source and destination nodes in this graph are not necessary, because there is only a single edge coming into and going out of these nodes. They can be removed from the graph by directly connecting the nodes at
the ends of the sequence of redundant nodes as in Figure 20.

![Figure 20: A graph with the unnecessary nodes removed.](image)

It is important to remove these nodes because the redundant nodes have a drastic impact on runtime. In the example graphs above, for example, Dijkstra’s algorithm will take five more iterations to identify the optimal path if the shape nodes are present.

The final data preparation program constructed the graph in memory using the node and edge files. The program then performed a depth-first search on the graph, looking for nodes with edges to only two other nodes. If this was the case, the node was a shape node and was removed, making the two neighbor nodes directly connected. After this step, the data had a reasonable number of nodes (~1,000,000). For the purposes of our testing, we made all the edges undirected and gave each edge the same distance in both directions.
5.2 Design of Code

The algorithms and heuristics were all implemented in the Java programming language. All of the code is included in Appendix A. This section summarizes some of the important aspects of the code.

5.2.1 Graph Structure

The graphs themselves consisted of Node and Edge objects. The Node objects have ID (long integer), latitude (double float), longitude (double float), and neighbor attributes. The ID, latitude, and longitude attributes are pulled directly from the node file. The neighbor attribute was represented as an ArrayList of Edge objects. The UML class diagrams of these two class can be found in Figure 21.

The ID attributes were unique and primarily used for debugging the routes. Most of the time routes produced by the different experimental algorithms were stored as a list of IDs and compared manually for correctness.

The latitude and longitude attributes were important for constructing $S$ for each test cases. Because each test-case specified $S$ as a rectangular region defined by a lower-left hand point and an upper-right hand point, the geographic coordinates of each node were compared to these constraints in order to determine if it should go in $S$.

An Edge object consisted of a distance (represented by a long integer) and a destination attribute, which was a Node object.

We chose to represent the distance as an integer in order to avoid floating point errors in the
summation of the route lengths. The Node attribute of the Edge object represented the destination of the Edge.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>+distance : long</td>
<td>+id : long</td>
</tr>
<tr>
<td>+destination : Node</td>
<td>+neighbors : ArrayList&lt;Edge&gt;</td>
</tr>
<tr>
<td>+Edge(long dist, Node dest)</td>
<td>+lat : double</td>
</tr>
<tr>
<td></td>
<td>+lon : double</td>
</tr>
<tr>
<td></td>
<td>+equals(Node other) : boolean</td>
</tr>
<tr>
<td></td>
<td>+compareTo(Node other) : int</td>
</tr>
<tr>
<td></td>
<td>+Node(long id)</td>
</tr>
<tr>
<td></td>
<td>+Node(long id, double lat, double lon)</td>
</tr>
</tbody>
</table>

Figure 21: Class diagrams for the Node and Edge classes.

5.2.2 Pre-Processing

Because the raw node and edge files were stored on disk, constructing the graph in its entirety each time the program ran was time consuming. To remedy this the graph was constructed once and then stored as a serialized object file. The main method for the Driver class read in a Java object file that contained a serialized version of the pre-constructed graph each time it ran.

After reading in the graph, the Driver method executed all the test-cases. For each case, it constructed $S$ by moving through the graph via breadth-first search and then timed the execution of the three experimental algorithms on each case.
5.2.3 The Algorithms and Heuristics

Each algorithm and heuristic was coded in a separate class. Each class contained a custom comparator class, a static method to run the algorithm, and several Maps to maintain the labels for each node. The comparators were used by the D-Heap to determine which of two nodes had the smallest current label. The comparators were essential because they interacted with the specific label Map objects for each file. The static method was the implementation of the corresponding algorithm for section 4 taking in a Node object for the source Node, a Node object for the destination Node, and a HashSet of Nodes to represent S.

5.3 Utilities

In addition to the experimental algorithms, a basic breadth-first search was implemented. This algorithm is used to collect data about the connectivity of the graph, namely the number of reachable nodes. The algorithm is also used during the course of test-cases construction to find a node with a given ID and to find all nodes that lie within the constraints of the S.

5.4 Test Cases

Because this research was intended to be applied to real-world navigational problems, we chose to construct a representative test-suite as opposed to a randomized one. Each test case was designed to assess the performance of the algorithms on a specific type of navigation problem.

The test-cases for the algorithms and heuristics were constructed with respect to a number of constraints including: distance between origin and destination nodes, distance between S and the
optimal $O^1D^1$ solution, size of $S$, and the whether or not the origin and destination nodes resided near the edges of the graph. The test-cases were created by obtaining geographical coordinates using an online map system. For each test case, an ideal latitude and longitude for the source and destination nodes, and the lower-left and upper-right corners of $S$ were recorded. These were considered the "raw" test cases. To save time during the execution of the program, the test-cases were pre-processed. For each test case, the ID of the closest node to the coordinates of the initial and destination vertices were determined via Euclidean distance and a set was filled with all nodes that lay within the bounds of $S$.

Because each $S$ is a rectangular region denoted only by the lower-left and upper-right corners, all test-cases have $S$ sets that are in the same geographic proximity and no $S$ sets are scattered throughout the map.

The following charts lists the characteristics for each test-case, a simple description of the navigation scenario the test represents, and a illustration of the problem:

The characteristics describe the nature of the particular test-case:

- $S$ close to $O^1D^1$ Solution: the set $S$ lies close to the path that is the solution to the $O^1D^1$.
- $S$ Larger: For each test case there is one version with a significantly larger $S$
- Source/Dest Close: the source and destination nodes are relatively close geometrically
- Length $O^1I^nD^1 =\text{Length } O^1D^1$: the special set lies on the $O^1D^1$ solution
- Middle vs Edge: whether or not the source and destination nodes lie near the edges of the graph. The $x$ indicates that they lie near the edge.
For each illustration, the intermediate set was drawn with respect to some minimum size to ensure visibility.
<table>
<thead>
<tr>
<th>Test-Case</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$S$ close to $O^1D^1$ Solution</td>
</tr>
<tr>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>e</td>
</tr>
<tr>
<td>3</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
</tr>
<tr>
<td>8</td>
<td>e</td>
</tr>
<tr>
<td>9</td>
<td>e</td>
</tr>
<tr>
<td>10</td>
<td>e</td>
</tr>
<tr>
<td>11</td>
<td>e</td>
</tr>
<tr>
<td>12</td>
<td>e</td>
</tr>
<tr>
<td>13</td>
<td>e</td>
</tr>
<tr>
<td>14</td>
<td>e</td>
</tr>
<tr>
<td>15</td>
<td>e</td>
</tr>
<tr>
<td>16</td>
<td>e</td>
</tr>
<tr>
<td>17</td>
<td>e</td>
</tr>
<tr>
<td>18</td>
<td>e</td>
</tr>
<tr>
<td>19</td>
<td>e</td>
</tr>
<tr>
<td>20</td>
<td>e</td>
</tr>
<tr>
<td>21</td>
<td>e</td>
</tr>
<tr>
<td>22</td>
<td>e</td>
</tr>
<tr>
<td>23</td>
<td>e</td>
</tr>
<tr>
<td>24</td>
<td>e</td>
</tr>
<tr>
<td>25</td>
<td>e</td>
</tr>
<tr>
<td>26</td>
<td>e</td>
</tr>
<tr>
<td>27</td>
<td>e</td>
</tr>
<tr>
<td>28</td>
<td>e</td>
</tr>
<tr>
<td>29</td>
<td>e</td>
</tr>
<tr>
<td>30</td>
<td>e</td>
</tr>
<tr>
<td>31</td>
<td>e</td>
</tr>
<tr>
<td>32</td>
<td>e</td>
</tr>
</tbody>
</table>

Table 2: Characteristics of each test case.
<table>
<thead>
<tr>
<th>Test-Case</th>
<th>Real-World Example</th>
<th>Illustration</th>
<th>Nodes in the Intermediate Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Harrisonburg to Staunton through Newmarket</td>
<td><img src="image1.png" alt="Illustration" /></td>
<td>397</td>
</tr>
<tr>
<td>2</td>
<td>Danville to Martinsville through South-Boston</td>
<td><img src="image2.png" alt="Illustration" /></td>
<td>1492</td>
</tr>
<tr>
<td>3</td>
<td>Harrisonburg to Staunton through Charlottesville</td>
<td><img src="image3.png" alt="Illustration" /></td>
<td>3970</td>
</tr>
<tr>
<td>4</td>
<td>Danville to Martinsville through Chatham</td>
<td><img src="image4.png" alt="Illustration" /></td>
<td>621</td>
</tr>
<tr>
<td>5</td>
<td>Richmond to Blacksburg through Charlottesville</td>
<td><img src="image5.png" alt="Illustration" /></td>
<td>3050</td>
</tr>
<tr>
<td>6</td>
<td>Danville to Alexandria through Lynchburg</td>
<td><img src="image6.png" alt="Illustration" /></td>
<td>19683</td>
</tr>
<tr>
<td></td>
<td>Route Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>-------------------------------------------</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Richmond to Blacksburg through Harrisonburg</td>
<td>2440</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Danville to Alexandria through Richmond</td>
<td>26822</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Harrisonburg to Staunton through Newmarket</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Danville to Martinsville through South-Boston</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Harrisonburg to Staunton through Charlottesville</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Danville to Martinsville through Chatham</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Route Description</td>
<td>Distance</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>--------------------------------------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Richmond to Blacksburg through Charlottesville</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Danville to Alexandria through Lynchburg</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Richmond to Blacksburg through Harrisonburg</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Danville to Alexandria through Richmond</td>
<td>264</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Harrisonburg to Staunton through Winchester</td>
<td>4300</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Danville to Martinsville through Franklin</td>
<td>886</td>
<td></td>
</tr>
</tbody>
</table>
19  Harrisonburg to Staunton through Fredricksburg       7692
20  Danville to Martinsville through Harrisonburg       3510
21  Richmond to Blacksburg through Williamsburg        2834
22  Danville to Alexandria through Franklin            795  
23  Richmond to Blacksburg through Alexandria           26217
24  Danville to Alexandria through Franklin             922
25  Harrisonburg to Staunton through Winchester  539

26  Danville to Martinsville through Franklin  263

27  Harrisonburg to Staunton through Fredricksburg  135

28  Danville to Martinsville through Harrisonburg  108

29  Richmond to Blacksburg through Williamsburg  408

30  Danville to Alexandria through Franklin  114
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Richmond to Blacksburg through Alexandria</td>
<td>838</td>
</tr>
<tr>
<td>32</td>
<td>Danville to Alexandria through Franklin</td>
<td>158</td>
</tr>
</tbody>
</table>

Table 3: Short description, illustration, and special set size for each test-case.
6 Results of the Empirical Analysis

We ran the Brute Force, Bi-Directional Heuristic, and Multi-Label Heuristic algorithms on a variety of test-cases derived from real world navigation scenarios on the Virginia road system using the data set described above.

6.1 Run Timing

Each test case was run on the same computer with a 3.40 GHz Intel Core i7-4770 CPU, with 15.6 GB of RAM running Linux Mint 17.3 Cinnamon 64-bit. In all cases, the algorithms yielded identical paths. All algorithms were measured using Java’s built in nanotime() method. The beginning time was recorded at the invocation of the algorithm method and the end time was measured as the moment the method returned. The average time is calculated over the average times of three trials performed sequentially. The average results of the test-cases are presented in the Table 4.

In all cases, the three methods found the same routes (increasing our confidence that the two heuristics can, in fact, be proven correct.
<table>
<thead>
<tr>
<th>Test Case</th>
<th>Brute Force Average Time (s)</th>
<th>Bi-Directional Heuristic Average Time (s)</th>
<th>Multi-Label Heuristic Average Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.17477</td>
<td>0.44409</td>
<td>0.52679</td>
</tr>
<tr>
<td>2</td>
<td>0.22347</td>
<td>0.61735</td>
<td>0.98886</td>
</tr>
<tr>
<td>3</td>
<td>0.20939</td>
<td>0.41965</td>
<td>0.66088</td>
</tr>
<tr>
<td>4</td>
<td>0.06303</td>
<td>0.14704</td>
<td>0.35740</td>
</tr>
<tr>
<td>5</td>
<td>1.07705</td>
<td>2.70973</td>
<td>5.71397</td>
</tr>
<tr>
<td>6</td>
<td>1.18419</td>
<td>4.59397</td>
<td>6.98301</td>
</tr>
<tr>
<td>7</td>
<td>1.82474</td>
<td>4.68963</td>
<td>5.05426</td>
</tr>
<tr>
<td>8</td>
<td>1.36348</td>
<td>2.51305</td>
<td>5.99824</td>
</tr>
<tr>
<td>9</td>
<td>0.10185</td>
<td>0.37872</td>
<td>0.46904</td>
</tr>
<tr>
<td>10</td>
<td>0.17206</td>
<td>0.61575</td>
<td>1.06207</td>
</tr>
<tr>
<td>11</td>
<td>0.15433</td>
<td>0.42879</td>
<td>0.94675</td>
</tr>
<tr>
<td>12</td>
<td>0.05187</td>
<td>0.18718</td>
<td>0.30810</td>
</tr>
<tr>
<td>13</td>
<td>1.01959</td>
<td>2.26545</td>
<td>6.06560</td>
</tr>
<tr>
<td>14</td>
<td>0.97254</td>
<td>4.52932</td>
<td>6.36686</td>
</tr>
<tr>
<td>15</td>
<td>1.70548</td>
<td>4.31310</td>
<td>5.52279</td>
</tr>
<tr>
<td>16</td>
<td>1.09279</td>
<td>2.39304</td>
<td>5.00164</td>
</tr>
<tr>
<td>17</td>
<td>0.63906</td>
<td>1.54265</td>
<td>3.15256</td>
</tr>
<tr>
<td>18</td>
<td>1.58386</td>
<td>3.12824</td>
<td>5.71822</td>
</tr>
<tr>
<td>19</td>
<td>1.06343</td>
<td>1.80844</td>
<td>4.57802</td>
</tr>
<tr>
<td>20</td>
<td>2.01548</td>
<td>5.03194</td>
<td>6.33807</td>
</tr>
<tr>
<td>21</td>
<td>1.58145</td>
<td>5.57890</td>
<td>6.03588</td>
</tr>
<tr>
<td>22</td>
<td>1.77758</td>
<td>4.14047</td>
<td>5.30674</td>
</tr>
<tr>
<td>23</td>
<td>3.03972</td>
<td>4.90248</td>
<td>6.91386</td>
</tr>
<tr>
<td>24</td>
<td>1.73377</td>
<td>3.68151</td>
<td>5.23870</td>
</tr>
<tr>
<td>25</td>
<td>0.55075</td>
<td>1.63989</td>
<td>3.25362</td>
</tr>
<tr>
<td>26</td>
<td>1.58711</td>
<td>3.21529</td>
<td>5.78379</td>
</tr>
<tr>
<td>27</td>
<td>1.37814</td>
<td>3.05423</td>
<td>5.20880</td>
</tr>
<tr>
<td>28</td>
<td>2.78002</td>
<td>5.68108</td>
<td>6.02398</td>
</tr>
<tr>
<td>29</td>
<td>1.59319</td>
<td>5.75933</td>
<td>6.17115</td>
</tr>
<tr>
<td>30</td>
<td>1.63281</td>
<td>4.22314</td>
<td>5.57775</td>
</tr>
<tr>
<td>31</td>
<td>2.34350</td>
<td>5.60663</td>
<td>6.48243</td>
</tr>
<tr>
<td>32</td>
<td>1.28426</td>
<td>6.06816</td>
<td>6.29057</td>
</tr>
</tbody>
</table>

Table 4: Table of algorithm timings for test-cases.
Because the test cases varied greatly in the magnitude of their timing, we adjusted the timing data by presenting it as a ratio between the running-time and the running-time of the Brute Force algorithm for the same test-case. This is illustrated in Figure 22.

Figure 22: Algorithm timings scaled against the Brute Force algorithm performance.
6.2 Discussion

From the plot of the algorithm timings in Figure 22, we see that in all cases, the Brute Force algorithm found the optimal solution in less time than the Bi-Directional Heuristic. In most cases, the Brute Force algorithm finished in approximately half the time as the Bi-Directional Heuristic. The Multi-Label Heuristic performed consistently slower than either of the other two.

Further analysis of the results of the Bi-Directional Heuristic algorithm on the test suite indicated that in almost all test-cases one of the Dijkstra algorithms had already labeled and finalized a large number of nodes with "special" labels. Because the clean-up step required iteration over every node finalized with a "special" label, the clean-up step was generally very costly. Table 5 contains several test cases and the number of nodes in the finalized special sets from both the source and destination Dijkstra algorithms at the conclusion of the algorithm:

<table>
<thead>
<tr>
<th>Test-case</th>
<th>Number of Nodes in Special Finalized Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Source Set</td>
</tr>
<tr>
<td>1</td>
<td>19576</td>
</tr>
<tr>
<td>5</td>
<td>13377</td>
</tr>
<tr>
<td>15</td>
<td>20256</td>
</tr>
<tr>
<td>20</td>
<td>311987</td>
</tr>
<tr>
<td>28</td>
<td>43065</td>
</tr>
</tbody>
</table>

Table 5: Number of nodes finalized as "special" for select test-cases.
For a more visible example, imagine traveling from Florida to New York with the condition that you must pass through Alabama. By the time the algorithm that originated in New York finalizes a node in Alabama, the algorithm that originated in Florida will likely have finalized most of the nodes on the east coast. Therefore, in order to complete the clean-up step, the Bi-Directional Heuristic must work through almost the entirety of the east coast, while the Brute Force algorithm need only work through the nodes in Alabama.

This leads us to wonder why the same problem does not occur with Pohl’s algorithm. We can speculate that it is because with the standard $O^1D^1$ problem, there is no innate concept of the "middle." That is to say, in the traditional problem, the meeting point of the two directions is always at the earliest possible instance. In our case, we force the meeting place to be closer to either the source or destination node, which is an additional constraint. This prevents the extra clean-up needed in the Bi-Directional Heuristic algorithm.
7 Conclusion and Directions for Future Research

We studied three possible solutions to the Shortest Path Intermediate Set Problem. The Brute Force algorithm runs Dijkstra’s algorithm, finalizing all nodes in the intermediate set from both the source and destination nodes and then iterating through all the nodes in the set to find the one with the shortest total path length. The Bi-Directional Heuristic algorithm involved running Dijkstra’s algorithm from both the source and destination nodes until they both finalize a common node in the intermediate set, then iterates through all nodes that had been finalized from the special set to find the best path. The Multi-Label Heuristic algorithm proceeded by running Dijkstra’s algorithm from the source node until it finalizes the destination node through the special set.

After measuring the time that each algorithm took on a variety of representative test cases, in all test-cases the Brute Force algorithm outperformed the other two algorithms in terms of time to solution.

In future work done on this problem we could lessen the effect of the problem of the intermediate set being closer to one node by implementing a "balancing" heuristic. As it is currently implemented, all Bi-directional algorithms that we studied take turns with each iteration. If it were known beforehand that the intermediate set was closer to either the source or destination node, as measured by some A* style metric, we could balance the iterations of the source and destination iterations accordingly. For instance, if it were known that the intermediate set was 10 times closer to the source node than to the destination node, we could allow the source node iterations to iterate once for every ten iterations of the destination iterations. This should greatly reduce the number of "special" finalized nodes that need to be cleaned up. If such a heuristic could be calculated efficiently, a more dynamic balancing system could be employed to give each of the algorithms more or less iterations at any time in the execution depending on their current distance from the intermediate set.
Another possible solution for the problem of an “unbalanced” middle set could involve halting the iterations from the first node to finalize all nodes in the intermediate set. Then allowing the other Dijkstra to continue uninterrupted until it has finalized a node in the intermediate set. After this event, the halted algorithm would be allowed to continue to execute normally, alternating iterations with the other algorithm until we knew that we had enough information to stop.

A different approach to improving the performance of the Bi-Directional Heuristic algorithm could involve implementing a more intelligent clean-up routine. As is, the clean-up iterates through all nodes that have been finalized through the intermediate set and checks all of them to find an optimal path. This is likely unnecessary, many of these nodes could likely be pruned out by using information from other nodes or other heuristics.

Another area of future work could be testing the parallelizability of each of these solutions. The Bi-Directional Heuristic and Brute Force algorithms seem to lend themselves towards parallelism and could likely achieve a considerable improvement in runtime.

An interesting question to ask is how to actually find the shortest path through intermediate nodes or even determining if such a path exists. In this research we took care to distinguish the "routes" we are finding from true "paths." A path would not allow for a repeated node in the sequence of edges. This could be a useful question to ask because it is unlikely that a user would want to turn around and traverse the same road twice. It is even possible that such a path does not exist such as in the case where there is only a single edge that enters the intermediate set from outside the intermediate set.

In the future, formal proofs must also be developed for the correctness of the algorithms and heuristics presented in this thesis. These proofs will likely draw heavily from the logic employed by the proof of correctness of Pohl’s algorithm.
8 Appendix

8.1 Code Listings

8.1.1 Brute Force Algorithm

```java
import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

/**
 * Contains an implementation of a brute force special set path algorithm
 *
 * @author Steven Young
 * @version February 20, 2016
 */
public class Brute {

    public static class DijsktraComp implements Comparator<Node> {
        @Override
        public int compare(Node x, Node y) {
            return Long.compare(source_distances.get(x), source_distances.get(y));
        }
    }

    public static class DijsktraCompdest implements Comparator<Node> {
        @Override
        public int compare(Node x, Node y) {
            return Long.compare(dest_distances.get(x), dest_distances.get(y));
        }
    }

    static Map<Node, Long> source_distances;
    static Map<Node, Long> dest_distances;

    /************
    * Runs the Brute Force algorithm on from the source node
    * to the destination node using special as the intermediate set
    *
    * @param source - node where the route must start
    */
```
* @param destination - node where the route must end
* @param special - intermediate set
* @return - the length of the optimal path from
*     the source node to the destination node that passes
*     through the intermediate set
*/

public static long bruteForce(Node source, Node destination, Set<Node> special) {

    long source_curDistance, dest_curDistance;
    Set<Node> source_finalized = new HashSet<Node>();
    Set<Node> dest_finalized = new HashSet<Node>();
    Comparator<Node> comp = new Brute.DijsktraComp();
    Comparator<Node> comp2 = new Brute.DijsktraCompdest();
    source_distances = new HashMap<Node, Long>();
    dest_distances = new HashMap<Node, Long>();
    DHeap<Node> source_q = new DHeap<Node>(comp, 3);
    DHeap<Node> dest_q = new DHeap<Node>(comp2, 3);
    boolean done = false;
    Node source_curNode;
    Node dest_curNode;
    long bestDist = Long.MAX_VALUE / 2 - 1;
    long source_count = special.size(), dest_count = special.size();

    source_q.add(source);
    dest_q.add(destination);
    boolean source_first = true, dest_first = true;

    while (!done && (source_q.size() != 0 || dest_q.size() != 0)) {

        if (source_count != 0) {
            source_curNode = source_q.remove();

            source_finalized.add(source_curNode);

            if (source_first) {
                source_curDistance = 0;
                source_first = false;
                source_distances.put(source_curNode, (long) 0);
            } else {
                source_curDistance = source_distances.get(source_curNode);
            }
        }

        // Now we have the label of curNode in curDistance
        for (Edge e : source_curNode.neighbors) {

            long tempDistance;
Node tempNode;
tempNode = e.destination;
tempDistance = e.distance;
if (!source_finalized.contains(tempNode)) {
  if (source_distances.get(tempNode) == null) {
    source_distances.put(tempNode,
    source_curDistance + tempDistance);
  } else {
    if (source_distances
      .get(tempNode) > source_curDistance
      + tempDistance) {

      source_distances.put(tempNode,
      source_curDistance + tempDistance);
    }
  }
  source_q.add(tempNode);
}
if (special.contains(source_curNode) && source_count > 0) {
  source_count --;
}

// Destination node iteration
if (dest_count != 0) {
  dest_curNode = dest_q.remove();
dest_finalized.add(dest_curNode);
if (dest_first) {
  dest_curDistance = 0;
dest_first = false;
dest_distances.put(dest_curNode, (long) 0);
} else {
  dest_curDistance = dest_distances.get(dest_curNode);
}
// Now we have the label of curNode in curDistance
for (Edge e : dest_curNode.neighbors) {

  long tempDistance;
  Node tempNode;
tempNode = e.destination;
tempDistance = e.distance;

  if (!dest_finalized.contains(tempNode)) {
    if (dest_distances.get(tempNode) == null) {

```java
dest_distances.put(tempNode,
   dest_curDistance + tempDistance);
} else {
   if (dest_distances.get(tempNode) > dest_curDistance
       + tempDistance) {
      dest_distances.put(tempNode,
         dest_curDistance + tempDistance);
   }
   dest_q.add(tempNode);
}
if (special.contains(dest_curNode) && dest_count > 0) {
   dest_count--;
}

// Check end condition
if (dest_count == 0 && source_count == 0) {
   done = true;
   // Perform the cleanup
   for (Node n : special) {
      if (source_distances.get(n) == null) {
         source_distances.put(n, Long.MAX_VALUE / 2 - 1);
      }
      if (dest_distances.get(n) == null) {
         dest_distances.put(n, Long.MAX_VALUE / 2 - 1);
      }
      if (source_distances.get(n) + dest_distances.get(n) < bestDist) {
         bestDist = source_distances.get(n) + dest_distances.get(n);
      }
   }
}
if (!done) {
   bestDist = 0;
}
return bestDist;
```
8.1.2 Bi-Directional Heuristic Algorithm

```java
import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

/**
 * Contains an implementation of our experimental special set \(O^1\) \(I^1\) algorithm
 *
 * @author Steven Young
 * @version February 20, 2016
 */
public class Better {

    public static Map<Node, Long> source_distances;
    public static Map<Node, Long> source_special_distances;
    public static Map<Node, Long> dest_distances;
    public static Map<Node, Long> dest_special_distances;

    public static class DijkstraComp implements Comparator<TwoWayNode> {
        @Override
        public int compare(TwoWayNode x, TwoWayNode y) {
            Long val1, val2;
            if (x.type) {
                val1 = source_special_distances.get(x.innerNode);
            } else {
                val1 = source_distances.get(x.innerNode);
            }
            if (y.type) {
                val2 = source_special_distances.get(y.innerNode);
            } else {
                val2 = source_distances.get(y.innerNode);
            }
            if (val1 == null) {
                val1 = Long.MAX_VALUE;
            }
            if (val2 == null) {
                val2 = Long.MAX_VALUE;
            }
            return Long.compare(val1, val2);
        }
    }
}
```
/**
 * Inner wrapper class for nodes that allows for the
differentiation between
 * special and non-special nodes
 */
public static class TwoWayNode {

    @SuppressWarnings("unused")
    private static final long serialVersionUID = 1L;
    public boolean type;
    public Node innerNode;

    public TwoWayNode(Node c, boolean type) {
        this.innerNode = c;
        this.type = type;
    }

    @Override
    public int hashCode() {
        return ("" + innerNode.hashCode() + type).hashCode();
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof TwoWayNode) {
            TwoWayNode oNode = (TwoWayNode) o;
            return (this.innerNode.equals(oNode.innerNode)
                    && this.type == oNode.type);
        }
        return false;
    }

    public static class DijkstraCompdest implements Comparator<TwoWayNode> {
        @Override
        public int compare(TwoWayNode x, TwoWayNode y) {
            Long val1, val2;
            if (x.type) {
                val1 = dest_special_distances.get(x.innerNode);
            } else {
                val1 = dest_distances.get(x.innerNode);
            }
            if (y.type) {
                val2 = dest_special_distances.get(y.innerNode);
            } else {

val2 = dest_distances.get(y.innerNode);
}
if (val1 == null) {
    val1 = Long.MAX_VALUE;
}
if (val2 == null) {
    val2 = Long.MAX_VALUE;
}
return Long.compare(val1, val2);
}

/**
* Performs the Better algorithm starting from source, to
* destination
* through the intermediate set (special)
*
* @param source
*   - source node for algorithm
* @param destination
*   - destination node for algorithm
* @param special
*   - intermediate set
* @return - length of the optimal path from source to destination
* through
*   special
*/
public static long better(Node source, Node destination,
        Set<Node> special) {

    long source_curDistance, dest_curDistance;

    Set<Node> source_finalized = new HashSet<>();
    Set<Node> source_spec_finalized = new HashSet<>();
    Set<Node> dest_finalized = new HashSet<>();
    Set<Node> dest_spec_finalized = new HashSet<>();

    Comparator<TwoWayNode> comp = new Better.DijkstraComp();
    Comparator<TwoWayNode> comp2 = new Better.DijkstraCompdest();
    Map<Node, Node> source_pred = new HashMap<Node, Node>();
    Map<Node, Node> dest_pred = new HashMap<Node, Node>();

    Set<Node> source_finalized_special = new HashSet<Node>();
    Set<Node> dest_finalized_special = new HashSet<Node>();
    source_distances = new HashMap<Node, Long>();
    dest_distances = new HashMap<Node, Long>();
    source_special_distances = new HashMap<Node, Long>();
dest_special_distances = new HashMap<Node, Long>();

DHeap<TwoWayNode> source_q = new DHeap<TwoWayNode>(comp, 3);
DHeap<TwoWayNode> dest_q = new DHeap<TwoWayNode>(comp2, 3);
boolean done = false;
TwoWayNode source_curNode;
TwoWayNode dest_curNode;
long bestDist = Long.MAX_VALUE / 2 - 1;

if (special.contains(source))
    source_q.add(new TwoWayNode(source, true));
else
    source_q.add(new TwoWayNode(source, false));

if (special.contains(destination))
    dest_q.add(new TwoWayNode(destination, true));
else
    dest_q.add(new TwoWayNode(destination, false));

boolean source_first = true, dest_first = true;

while (!done && (source_q.size() != 0 || dest_q.size() != 0)) {
    source_curNode = source_q.remove();

    if (source_curNode.type) {
        source_spec_finalized.add(source_curNode.innerNode);
    } else {
        source_finalized.add(source_curNode.innerNode);
    }

    if (source_first) {
        source_curDistance = 0;
        source_first = false;
        if (source_curNode.type) {
            source_special_distances.put(source_curNode.innerNode, 0L);
        } else {
            source_distances.put(source_curNode.innerNode, 0L);
        }
    } else {
        if (source_curNode.type) {
            if (source_special_distances
                .get(source_curNode.innerNode) == null) {
                source_curDistance = Long.MAX_VALUE / 2;
            } else {
                source_curDistance = source_special_distances
                    .get(source_curNode.innerNode);
            }
        }
if (source_curNode.type) {
    for (Edge e : source_curNode.innerNode.neighbors) {
        long tempDistance;
        Node tempNode;

        tempNode = e.destination;
        tempDistance = e.distance;

        if (!source_spec_finalized.contains(tempNode)) {
            if (source_special_distances.get(tempNode) == null) {
                source_special_distances.put(tempNode, 
                source_curDistance + tempDistance);
            } else {
                if (source_special_distances.get(tempNode) > source_curDistance + tempDistance) {
                    source_special_distances.put(tempNode, 
                    source_curDistance + tempDistance);
                }
            }
            source_q.add(new TwoWayNode(tempNode, true));
        }
    }
} else {
    for (Edge e : source_curNode.innerNode.neighbors) {
        long tempDistance;
        Node tempNode;

        tempNode = e.destination;
        tempDistance = e.distance;

        if (!source_finalized.contains(tempNode)) {
        } else {
            if (source_distances.get(source_curNode.innerNode) == null) {
                source_curDistance = Long.MAX_VALUE / 2;
            } else {
                source_curDistance = source_distances.get(source_curNode.innerNode);
            }
        }
    }

    // Now we have the label of curNode in curDistance
    if (source_curNode.type) {
        for (Edge e : source_curNode.innerNode.neighbors) {
            long tempDistance;
            Node tempNode;

            tempNode = e.destination;
            tempDistance = e.distance;

            if (!source_spec_finalized.contains(tempNode)) {
                if (source_special_distances.get(tempNode) == null) {
                    source_special_distances.put(tempNode, 
                    source_curDistance + tempDistance);
                } else {
                    if (source_special_distances.get(tempNode) > source_curDistance + tempDistance) {
                        source_special_distances.put(tempNode, 
                        source_curDistance + tempDistance);
                    }
                }
                source_q.add(new TwoWayNode(tempNode, true));
            }
        }
    }
} else {
    for (Edge e : source_curNode.innerNode.neighbors) {
        long tempDistance;
        Node tempNode;

        tempNode = e.destination;
        tempDistance = e.distance;

        if (!source_finalized.contains(tempNode)) {
        } else {
            if (source_distances.get(source_curNode.innerNode) == null) {
                source_curDistance = Long.MAX_VALUE / 2;
            } else {
                source_curDistance = source_distances.get(source_curNode.innerNode);
            }
        }
    }
}
if (source_distances.get(tempNode) == null) {
    source_distances.put(tempNode, 
    source_curDistance + tempDistance);
} else {
    if (source_distances
        .get(tempNode) > source_curDistance
        + tempDistance) {
        source_distances.put(tempNode, 
        source_curDistance + tempDistance);
    }
}
TwoWayNode n = new TwoWayNode(tempNode, false);
source_q.add(n);
if (special.contains(tempNode)
    && !source_spec_finalized.contains(tempNode)) {
    if (source_special_distances
        .get(tempNode) == null) {
        source_special_distances.put(tempNode, 
        source_curDistance + tempDistance);
    } else {
        if (source_special_distances
            .get(tempNode) > source_curDistance
            + tempDistance) {
            source_special_distances.put(tempNode, 
            source_curDistance + tempDistance);
        }
    }
source_q.add(new TwoWayNode(tempNode, true));
}
}

// Destination Node iteration
dest_curNode = dest_q.remove();
if (dest_curNode.type) {
    dest_spec_finalized.add(dest_curNode.innerNode);
} else {
    dest_finalized.add(dest_curNode.innerNode);
}
if (dest_first) {
    dest_curDistance = 0;
    dest_first = false;
    if (dest_curNode.type) {
dest_special_distances.put(dest_curNode.innerNode, 0l);

} else {
    dest_distances.put(dest_curNode.innerNode, 0l);
}
}
}

if (dest_curNode.type) {
    if (dest_special_distances.get(dest_curNode.innerNode) == null) {
        dest_curDistance = Long.MAX_VALUE / 2;
    } else {
        dest_curDistance = dest_special_distances.get(dest_curNode.innerNode);
    }
}

if (dest_distances.get(dest_curNode.innerNode) == null) {
    dest_curDistance = Long.MAX_VALUE / 2;
} else {
    dest_curDistance = dest_distances.get(dest_curNode.innerNode);
}

} // Now we have the label of curNode in curDistance

if (dest_curNode.type) {
    for (Edge e : dest_curNode.innerNode.neighbors) {
        long tempDistance;
        Node tempNode;

        tempNode = e.destination;
        tempDistance = e.distance;
        if (!dest_spec_finalized.contains(tempNode)) {
            if (dest_special_distances.get(tempNode) == null) {
                dest_special_distances.put(tempNode, dest_curDistance + tempDistance);
            } else {
                if (dest_special_distances.get(tempNode) > dest_curDistance + tempDistance) {
                    dest_special_distances.put(tempNode, dest_curDistance + tempDistance);
                }
            }
        }
        dest_q.add(new TwoWayNode(tempNode, true));
    }
}
for (Edge e : dest_curNode.innerNode.neighbors) {
    long tempDistance;
    Node tempNode;

    tempNode = e.destination;
    tempDistance = e.distance;
    if (!dest_finalized.contains(tempNode)) {
        if (dest_distances.get(tempNode) == null) {
            dest_distances.put(tempNode,  
            dest_curDistance + tempDistance);
        } else {
            if (dest_distances.get(tempNode) > dest_curDistance  
            + tempDistance) {
                dest_distances.put(tempNode,  
                dest_curDistance + tempDistance);
            }
        }
    }

    TwoWayNode n = new TwoWayNode(tempNode, false);
    dest_q.add(n);
    if (special.contains(tempNode)  
    && !dest_spec_finalized.contains(tempNode)) {
        if (dest_special_distances.get(tempNode) == null) {
            dest_special_distances.put(tempNode,  
            dest_curDistance + tempDistance);
        } else {
            if (dest_special_distances.get(tempNode) > dest_curDistance  
            + tempDistance) {
                dest_special_distances.put(tempNode,  
                dest_curDistance + tempDistance);
            }
        }
    }
    dest_q.add(new TwoWayNode(tempNode, true));
}

if (special.contains(source_curNode.innerNode)) {
    source_finalized_special.add(source_curNode.innerNode);
}

if (special.contains(dest_curNode.innerNode)) {
    dest_finalized_special.add(dest_curNode.innerNode);
}

84
if ((source_spec_finalized.contains(source_curNode.innerNode) && dest_spec_finalized.contains(source_curNode.innerNode)) || (source_spec_finalized.contains(dest_curNode.innerNode) && dest_spec_finalized.contains(dest_curNode.innerNode))) {

done = true;

// Perform the cleanup
if (source_spec_finalized.contains(source_curNode.innerNode) && dest_spec_finalized.contains(source_curNode.innerNode)) {
    bestDist = source_special_distances.get(source_curNode.innerNode) + dest_special_distances.get(source_curNode.innerNode);
} else {
    bestDist = source_special_distances.get(dest_curNode.innerNode) + dest_special_distances.get(dest_curNode.innerNode);
}

for (Node n : source_spec_finalized) {
    for (Edge e : n.neighbors) {
        Node dest = e.destination;
        long dist = e.distance;

        if (dest_distances.get(dest) == null) {
            dest_distances.put(dest, Long.MAX_VALUE / 2 - 1);
        }
        if (dest_distances.get(dest) + source_special_distances.get(n) + dist < bestDist) {
            bestDist = dest_distances.get(dest) + source_special_distances.get(n) + dist;
            dest_pred.put(n, dest);
        }
    }
}

for (Node n : dest_spec_finalized) {
    for (Edge e : n.neighbors) {

Node dest = e.destination;
long dist = e.distance;

if (source_distances.get(dest) == null) {
    source_distances.put(dest, Long.MAX_VALUE / 2 - 1);
}

if (source_distances.get(dest) + dest_special_distances.get(n) + dist < bestDist) {
    bestDist = source_distances.get(dest) + dest_special_distances.get(n) + dist;
    source_pred.put(n, dest);
}
if (!done)
{
    bestDist = 0;
}

return bestDist;
8.1.3 Multi-Label Heuristic

```java
import java.util.Comparator;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

/**
 * Class that contains a method to perform the
 * "Multi-Label Heuristic Algorithm"
 *
 * @author Steven Young
 * @version February 22, 2016
 */
public class SingleSpecial {

    public static Map<Node, Long> distances;
    public static Map<Node, Long> special_distances;

    public static class DijkstraComp implements Comparator<TwoWayNode> {
        @Override
        public int compare(TwoWayNode x, TwoWayNode y) {
            Long val1, val2;
            if (x.type) {
                val1 = special_distances.get(x.innerNode);
            } else {
                val1 = distances.get(x.innerNode);
            }
            if (y.type) {
                val2 = special_distances.get(y.innerNode);
            } else {
                val2 = distances.get(y.innerNode);
            }
            if (val1 == null) {
                val1 = Long.MAX_VALUE;
            }
            if (val2 == null) {
                val2 = Long.MAX_VALUE;
            }
            return Long.compare(val1, val2);
        }
    }
}

/***
 * Inner wrapper class for nodes that allows for the
 */
```
differentiation between
* special and non-special nodes
*/

public static class TwoWayNode {

@SuppressWarnings("unused")
private static final long serialVersionUID = 1L;
public boolean type;
public Node innerNode;

public TwoWayNode(Node c, boolean type) {
    this.innerNode = c;
    this.type = type;
}

@Override
public int hashCode() {
    return ("" + innerNode.hashCode() + type).hashCode();
}

@Override
public boolean equals(Object o) {
    if (o instanceof TwoWayNode) {
        TwoWayNode oNode = (TwoWayNode) o;
        return (this.innerNode.equals(oNode.innerNode)
                && this.type == oNode.type);
    }
    return false;
}

}/**
 * Performs "Single Special Algorithm" For details of the algorithm
 * first
 * tell me if it works for not
 *
 * @param source - source node
 * @param dest - destination node
 * @param special - intermediate set
 * @return - the optimal route from source to dest through the
 * intermediate
 */
static long SingleDijSpecial(Node source, Node dest, Set<Node> special) {
}
long distance = 0, curDistance;
Set<Node> finalized = new HashSet<Node>();
Set<Node> finalizedTwo = new HashSet<Node>();
Comparator<TwoWayNode> comp = new SingleSpecial.DijkstraComp();
DHeap<TwoWayNode> q = new DHeap<TwoWayNode>(comp, 3);
distances = new HashMap<Node, Long>();
special_distances = new HashMap<Node, Long>();
boolean found = false;
TwoWayNode curNode = null;

if (special.containsKey(source)) {
    q.add(new TwoWayNode(source, true));
} else {
    q.add(new TwoWayNode(source, false));
}
boolean first = true;

while (!found && q.size() != 0) {
    curNode = q.remove();
    if (curNode.innerNode.equals(dest) && curNode.type) {
        break;
    }
    if (curNode.type) {
        finalizedTwo.add(curNode.innerNode);
    } else {
        finalized.add(curNode.innerNode);
    }
    if (first) {
        curDistance = 0;
        if (curNode.type) {
            special_distances.put(curNode.innerNode, 0L);
        } else {
            distances.put(curNode.innerNode, 0L);
        }
        first = false;
    } else {
        if (curNode.type) {
            if (special_distances.get(curNode.innerNode) == null) {
                curDistance = Long.MAX_VALUE / 2;
            } else {
                curDistance = special_distances.get(curNode.innerNode);
            }
        } else {
            if (distances.get(curNode.innerNode) == null) {
                curDistance = Long.MAX_VALUE / 2;
            } else {

        }
curDistance = distances.get(curNode.innerNode);
}
}
// Now we have the label of curNode in curDistance
if (curNode.type) {
    for (Edge e : curNode.innerNode.neighbors) {
        long tempDistance;
        Node tempNode;

        tempNode = e.destination;
        tempDistance = e.distance;
        if (!finalizedTwo.contains(tempNode)) {
            if (special_distances.get(tempNode) == null) {
                special_distances.put(tempNode, curDistance + tempDistance);
            } else {
                if (special_distances.get(tempNode) > curDistance + tempDistance) {
                    special_distances.put(tempNode, curDistance + tempDistance);
                }
            }
            q.add(new TwoWayNode(tempNode, true));
        }
    }
} else {
    for (Edge e : curNode.innerNode.neighbors) {
        long tempDistance;
        Node tempNode;

        tempNode = e.destination;
        tempDistance = e.distance;
        if (!finalized.contains(tempNode)) {
            if (distances.get(tempNode) == null) {
                distances.put(tempNode, curDistance + tempDistance);
            } else {
                if (distances.get(tempNode) > curDistance + tempDistance) {
                    distances.put(tempNode, curDistance + tempDistance);
                }
            }
            TwoWayNode n = new TwoWayNode(tempNode, false);
            q.add(n);
        }
    }
}
if (special.contains(tempNode) && !finalizedTwo.contains(tempNode)) {
    if (special_distances.get(tempNode) == null) {
        special_distances.put(tempNode, curDistance + tempDistance);
    } else {
        if (special_distances.get(tempNode) > curDistance + tempDistance) {
            special_distances.put(tempNode, curDistance + tempDistance);
        }
    }
    q.add(new TwoWayNode(tempNode, true));
}

return distance;
8.1.4  DHeap

import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Map;

/**
 * Implements a priority queue with the
 * property that inserting a new item when the item is
 * already contained in the set will result in the
 * value of the item in the set being changed
 *
 * @author Steven Young
 */
public class DHeap<T> {

    private Comparator<T> comp;
    private Map<T, Integer> inQueue;
    private ArrayList<T> struct;
    private int degree;

    public DHeap(Comparator<T> comp, int degree) {
        this.comp = comp;
        this.degree = degree;
        inQueue = new HashMap<T, Integer>();
        struct = new ArrayList<T>();
    }

    public boolean contains(T obj) {
        return inQueue.get(obj) != null;
    }

    /**
     * Compares the item at the index to each of its
     * parents and swaps the values if the parent is
     * of greater value
     *
     * @param indx
     */
    public void upsift(int indx) {
        T element = this.struct.get(indx);
        boolean done = false;
        while(indx != 0 && !done) {
            if(this.comp.compare(element, this.struct.get((indx - 1)/
                degree)) == -1) {
                swap(indx, (indx-1)/degree);
            }
        }
    
}
index = (index-1)/degree;
} else {
    done = true;
}
}

/**
* Compares the item at the parameter
* index to its smallest child and swaps their values
* if the child is smaller than this element
*
* @param indx
*/
public void downshift(int indx) {
    T element = this.struct.get(indx);
    boolean done = false;
    while(indx * degree + 1 < this.struct.size() && !done) {
        //Find smallest child
        int smallestIndx = indx * degree + 1;
        for(int i = indx * degree + 2; i <= (indx + 1) * degree && i <
            this.struct.size(); i++) {
            if(this.comp.compare(this.struct.get(i), this.struct.get( 
                smallestIndx)) == -1) {
                smallestIndx = i;
            }
        }
        if(this.comp.compare(this.struct.get(smallestIndx), element) == 
            -1) {
            swap(indx, smallestIndx);
            indx = smallestIndx;
        } else {
            done = true;
        }
    }
}

public void add(T obj) {
    if(this.contains(obj)) {
        //Already in the queue
        int pos = this.inQueue.get(obj);
        upsift(pos);
    } else {
        //must perform normal insertion
        this.struct.add(this.struct.size(), obj);
        this.inQueue.put(obj, this.struct.size() - 1);
        upsift(this.struct.size()-1);
/**
 * Removes and returns the smallest item
 * currently in the queue
 */
public T remove() {
    swap(0, this.struct.size()-1);
    T returnVal = this.struct.remove(this.struct.size()-1);
    this.inQueue.remove(returnVal); //remove from map
    if(this.struct.size() > 0)
        downshift(0);
    return returnVal;
}

/****
 * Helper method that swaps the items in the
 * underlying struct at indices a and b
 *
 * @param a
 * @param b
 */
public void swap(int a, int b) {
    T itemA = this.struct.get(a);
    this.struct.set(a, this.struct.get(b));
    this.struct.set(b, itemA);
    //Update records
    this.inQueue.put(this.struct.get(a), a);
    this.inQueue.put(this.struct.get(b), b);
}

/****
 * @return
 */
public long size() {
    return this.struct.size();
}
9 References


