Developing A Web-Based Application for Finding Meeting Places

Aedan D. Pettit
The College of Wooster, apettit20@wooster.edu

Developing a Web-Based Application for Finding Meeting Places

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
Department of Mathematical and Computational Sciences
at The College of Wooster

by
Aedan Pettit
The College of Wooster
2020

Advised by:

Dr. Nathan Fox

THE COLLEGE OF
WOOSTER

Abstract

Midway: Meeting Place Finder is a web application which allows users to supply three or more locations and provides them with a place to meet that minimizes their total driving time. Using techniques from graph theory, an algorithm is developed in order to make this service possible. Then, using Python, this algorithm is implemented into the backend of the web application along with a simple, user-friendly interface. This application has exciting potential to be continually expanded and improved in the future beyond this initial version.

Acknowledgments

First and foremost I would like to thank my parents for their unending support throughout my life and academic career. Without their love and support I would certainly have been unable to attain the success I have had both inside and outside the classroom. I would also like to thank my advisor, Dr. Nathan Fox, for his ongoing guidance throughout the IS process; his advice and organization have been extremely helpful to me. Additionally, I would like to thank my wonderful girlfriend, Carly Hughes, for her love, support, and patience this semester while I have often been stressed and consumed by my work. Finally, I have to thank my housemates: Alex, Grant, Jackson, Joe, Nick, Noah, Matt, and Miki, as well as the entire Wooster Cross Country family, for their constant support and incredible ability to always put a smile on my face. The support of all of these people has been incredibly helpful and uplifting during the many struggles that come with IS and I am extremely grateful to each of them.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

When meeting up with groups of friends or family, it can often be challenging to decide where a meeting should take place. Ultimately, the goal is to minimize the total amount of time it takes for everybody to reach the meeting place, but currently there are few, if any, readily available tools capable of finding such places. Current applications are sufficient for pairs of individuals but not for groups. For example, one could easily estimate a halfway point using a routing service such as Google Maps or a dedicated meeting-place-finding service such as `meetways.com`. Attempting to accomplish this with three or more points would not be easy; it would require guessing and checking with Google Maps and is not even offered by `meetways.com`. Thus, the goal of this thesis is to create a web application capable of providing this service.

There are two goals that must be accomplished in order to provide a meeting-place-finding service: an algorithm that finds the meeting place and a web application that can take inputs for the algorithm and communicate its results. The most mathematically intensive part of this problem is creating the algorithm, which requires knowledge of graph theory. In particular, this project utilizes Dijkstra's single-source shortest path algorithm. Our meeting-place-finding algorithm is able to take a minimum of three input addresses and successfully output a meeting location that minimizes the total driving time from the input addresses. On the flip side, creating a web application requires significant amounts of programming in order to create a front-end user interface as well as a back-end which is connected to the algorithm itself. The front-end is able to provide users with a way to give input addresses and display the results of the algorithm in a way that is easy for users to interpret. The responsibility of the back-end is to process the user input, run the algorithm, and provide the result of the algorithm to the front-end. Midway: Meeting Place Finder is an application which successfully completes each of these goals and serves as the primary product of this thesis.

First we discuss important mathematical details which are relevant to the creation of this application. Specifically we discuss topics related to graph theory, quasimetrics, Dijkstra's algorithm, and the Floyd-Warshall algorithm. Then we go into detail about the software tools necessary for the development stage of the project. This includes the programming language and web framework used, as well as necessary packages and supplementary resources. After that, we give a detailed account of several algorithms that are capable of finding meeting places, including step-by-step descriptions and analysis of their runtime and accuracy. Next, we describe the process of developing the application itself and some challenges that were faced. We also include the final product of the development process. Finally, we conclude with a discussion on future ways that this application could be expanded or improved.

CHAPTER 2

Mathematical Background

2.1 Graph Theory

At the most basic level, a graph is a collection of nodes and edges, where edges represent connections between two nodes. More formally, a graph G has a set V containing nodes, also known as vertices, and a set E containing pairs of nodes which represent edges. For example, in Figure 2.1, the vertices are A, B, C, D, E, and F, while the edges are U, V, W, X, Y, and Z. Elements of the set E can be either ordered or unordered pairs. If the pairs are unordered, then the edges are considered to be undirected, which means that the edges can be travelled in either direction. For example, if the edges are undirected, then the edges (u, v) and (v, u) both represent the edge between the vertices v and u so only one of them needs to be included in E. Alternatively, the elements of E can be ordered pairs, in which case the edges are considered to be directed, meaning they can only be travelled in one direction. If a graph has directed edges, the element $(u, v) \in E$ represents an edge going from u to v, while (v, u) is an edge going from v to u. A graph with directed edges is known as a directed graph or a digraph [25]. An example of this type of graph is shown in Figure 2.2.

Another relevant type of graph is a *planar graph*. A planar graph is a graph that

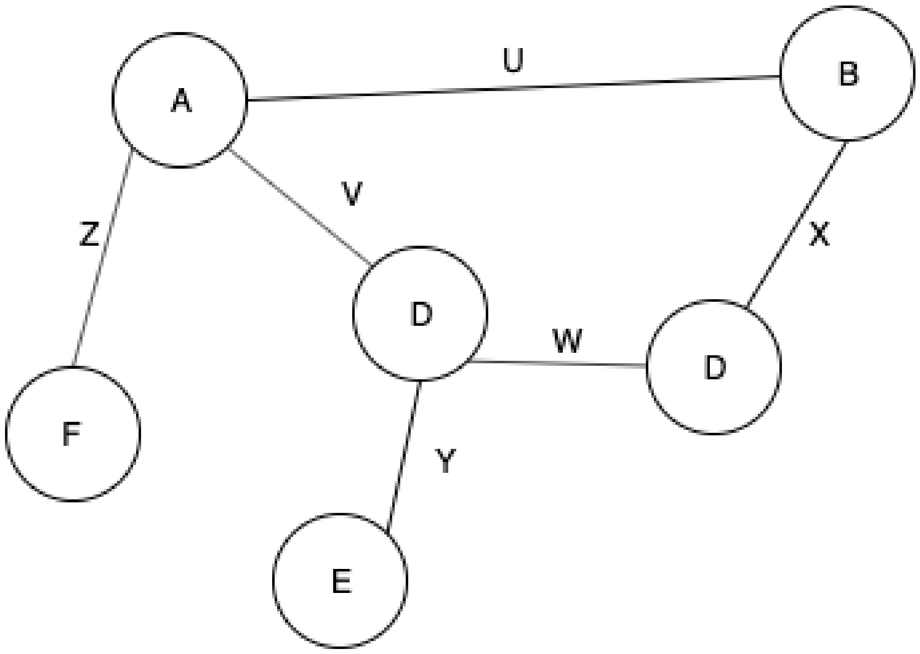


Figure 2.1: An unweighted, undirected graph

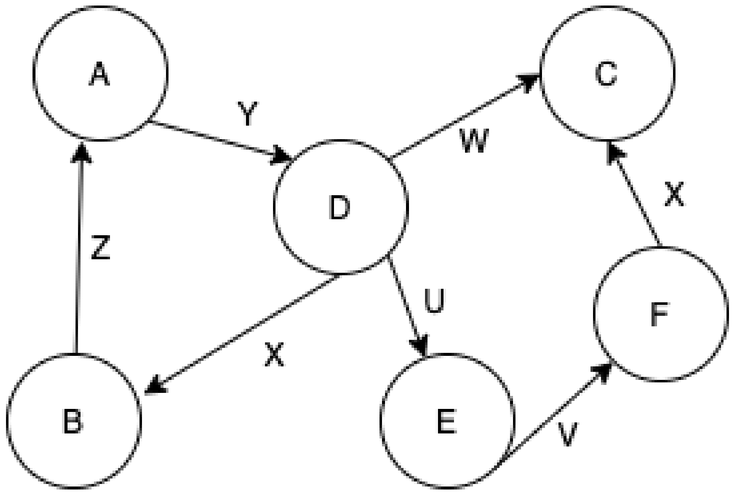


Figure 2.2: An unweighted, directed graph

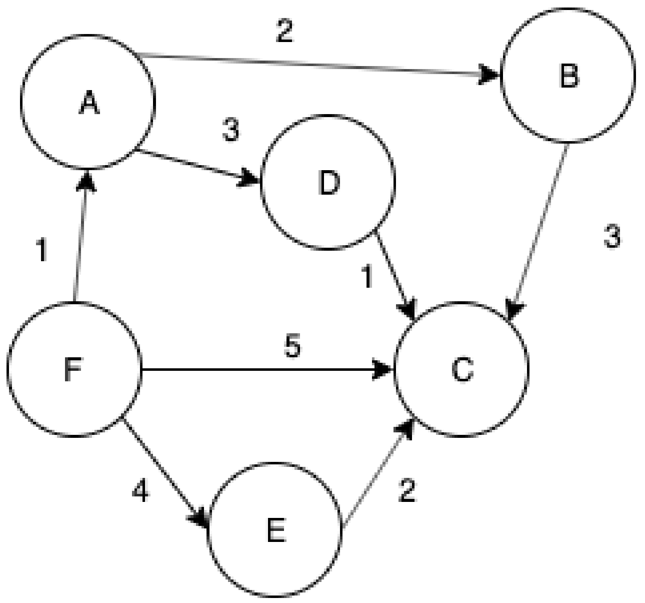


Figure 2.3: A weighted, directed graph

"has the property that no two edges cross except at vertices of the graph" [25]. This means that all of the graphs we have seen so far are also planar graphs. One useful attribute of planar graphs is that they cannot have more than $3n - 6$ edges, where n is the number of nodes in the graph [9].

The primary way this project uses graphs is by traversing along the edges, which is done by creating paths. A *path* is defined as a sequence of alternating vertices and edges that represent a way to travel from a source vertex to a destination vertex without repeating any edges or vertices [10]. For a graph as described above, the *distance* of a path is the number of edges on the path. However, for our problem weighted graphs are more relevant. A weighted graph is a graph where every edge has a number associated with it, called a *weight*. Figure 2.3 provides an example of a weighted graph. These weights can represent different things depending on what the graph represents, but one common use for weights is to represent distances. When working with a weighted graph the distance, or length, of a path is the sum of weights on the edges of the path.

2.2 Metrics and Quasimetrics

In Chapter 4, the problem we introduce is given a more formal definition in order to express it in mathematical terms. To do this, we use a concept with strong roots in real analysis and topology: quasimetrics. This section first introduces metrics in general before outlining what a quasimetric is and why this metric variant is necessary in the context of this thesis.

Given a set X and a function d, the pair (X, d) is a *metric space* if:

1. $d(x, y) \geq 0$
2. $d(x, y) = 0$ if and only if $x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) + d(y, z)$.

Here, the function d is called a *metric* or *distance function*. The general idea is that d represents distance, which makes the properties easy to interpret. Distances are nonnegative, a point is only zero distance away from itself, and the distance from point x to point y is the same as the distance from y to x. The last property is the triangle inequality, which dictates that the sum of the length of any two sides of a triangle must be greater than the length of the third side.

A simple example of a metric space is the set of real numbers, $\mathbb{R}$. In this case, $\mathbb{R}$ has a metric of:

$$d(x, y) := |x - y|.$$

From here, the first three properties of metric spaces are trivial to verify. The triangle inequality can also be verified without much trouble:

$$\begin{aligned} d(x,z) &= |x-z| \\ &= |x-y+y-z| \\ &\leq |x-y|+|y-z| \\ &= d(x,y)+d(y,z), \end{aligned}$$

thus showing that $d(x,z) \leq d(x,y)+d(y,z)$, which satisfies the triangle inequality [18].

This basic definition of a metric is a necessary starting point for the development of a more formal definition of our problem, but it represents a more ideal scenario than what one would find in the real world. To better encapsulate our problem, we need to work with a *quasimetric*. A quasimetric follows the same properties as a metric with one key exception: In order for a pair (X,d) to be a quasimetric [27], it does not necessarily need to be true that $d(x,y) = d(y,x)$. That is, any metric is also a quasimetric, but quasimetrics also describe cases where $d(x,y) \neq d(y,x)$. This definition is useful in the context of our problem because road networks include one-way streets. In the case of one way streets, the distance from a point x to a point y might not be the same as from y to x because they might be required to take different roads due to the existence of one-way streets.

Given that we are working with graphs which represent road networks, it is important for us to verify that these definitions still apply when working with graphs. Now, given an undirected graph, G, we define distance $d(u,v)$ as the length of the shortest path from u to v on G. This information allows us to trivially conclude the first three criteria for a metric: $d(u,v) \geq 0$, $d(u,v) = 0$ if and only if $u = v$, and $d(u,v) = d(v,u)$ for all $u,v \in V(G)$, where $V(G)$ is the set of all vertices in G. The last step to showing that d is a metric is verifying that it satisfies the triangle inequality.

If we consider P_1 to be the shortest path from u to v and P_2 to be the shortest path from v to w, then the path P_1 followed by P_2 produces path from u to w of length $d(u,v) + d(v,w)$. Since we can find a path such that $d(u,w) = d(u,v) + d(v,w)$, we can guarantee that $d(u,w) \leq d(u,v) + d(v,w)$, which satisfies the triangle inequality [7]. If G is directed, as real-life road networks are, the only one of these properties that is violated is symmetry, which means that for a directed graph our definition of a quasimetric is satisfied.

2.3 Important Algorithms

Here, we introduce some important graph algorithms. These algorithms solve problems similar to what we intend to solve and can be used as starting points for determining how to solve our problem. Note that these algorithms use weighted graphs that can be either directed or undirected.

2.3.1 Dijkstra's Algorithm

The single-source shortest path (SSSP) problem is the problem of finding the shortest path from a single vertex to all the remaining vertices in the graph. Dijkstra's algorithm is an iterative algorithm that successfully solves the SSSP.

The algorithm begins by selecting a source vertex s. Then, the neighbors of s are labeled with the weights of the edges from s, s is labeled with 0, and the rest of the vertices are labeled as infinity. The neighboring vertex with the smallest distance to s is added to the set of visited vertices. After that, the algorithm iteratively visits the vertex with the smallest distance that has not yet been visited and repeats the process. Once the algorithm concludes, it has found the shortest path from s to each of the other vertices in the graph [10]. Dijkstra's algorithm has a runtime complexity of $O(V \log V + E \log V)$, where V is the number of vertices and E represents the

```
Input: G(V, E, w)  // A weighted graph with source vertex s
Output: D[n] // Distances from source to vertices

// Initialize distance and predecessor (seen) arrays
p[s] <- 0
S <- $\emptyset$
foreach $v$ in $V - s$:
  p[v] <- $\infty$

Q <- V // List of vertices to observe
while Q is $\neq \emptyset$:
  u <- min(Q) // u is the vertex in Q with the smallest distance
  S <- S $\cup$ {u} // add u to seen vertices
  foreach $v$ in Neighbors($u$):
    // w(u, v) = weight on edge from u to v
    if p[v] > p[u] + w(u, v):
      p[v] <- p[u] + w(u, v)

return p
```

Listing 2.1: Dijkstra's algorithm

number of edges in the graph. A pseudocode implementation of this algorithm can be seen in Listing 2.1.

We can prove the correctness of Dijkstra's algorithm by proving that for each vertex v in the graph S, at any time during the execution of the algorithm, the path $P_{s,v}$ found by the algorithm is the shortest path between the source s and the vertex v. This proof is done using induction. There is a graph G with non-negative edge weights, a set of vertices V, a set of visited vertices S, and a source vertex s. We denote the distance to a vertex v found by Dijkstra's algorithm to be $d(v)$ and we denote the weight of the shortest path from s to v to be $\delta(v)$.

Proposition 1. *$d(v) = \delta(v)$ for all v at the end of the algorithm.*

Proof. The base case is when $S = \{s\}$. The claim is trivially true because the trivial path from s to itself has length 0, which must be the shortest path. Thus, $d(v) = \delta(v) = 0$.

For our inductive hypothesis, assume that for all vertices $v \in S$ that $d(v) = \delta(v)$.

Now, we need to show that for an arbitrary vertex $a \notin S$, $d(a) = \delta(a)$. For the sake of contradiction, assume the shortest path from s to a is $P_{s,a}$ and that:

$$\text{length}(P_{s,a}) < d(a).$$

Since $a \notin S$, we know that $P_{s,a}$ must start in S but eventually leave the set in order to get to a. We pick an edge bc to be the first edge on $P_{s,a}$ that leaves S. Since $b \in S$, we know that $d(b) = \delta(b)$, which means the path $P_{s,b}$ is the shortest path between the vertices. Thus, we know:

$$\text{length}(P_{s,b}) + \text{length}(bc) \leq \text{length}(P_{s,a}),$$

$$d(b) + \text{length}(bc) \leq \text{length}(P_{s,a}).$$

Since c is adjacent to b, $d(b)$ must have been updated by the algorithm, so we know:

$$d(c) \leq d(b) + \text{length}(bc).$$

The vertex a was picked by the algorithm which means it must have the smallest distance label, which implies:

$$d(a) \leq d(c).$$

Translating these inequalities in reverse gives us:

$$d(a) \leq d(b) + \text{length}(bc)$$

$$d(a) \leq \text{length}(P_{s,a})$$

Earlier, we assumed:

$$\text{length}(P_{s,a}) < d(a),$$

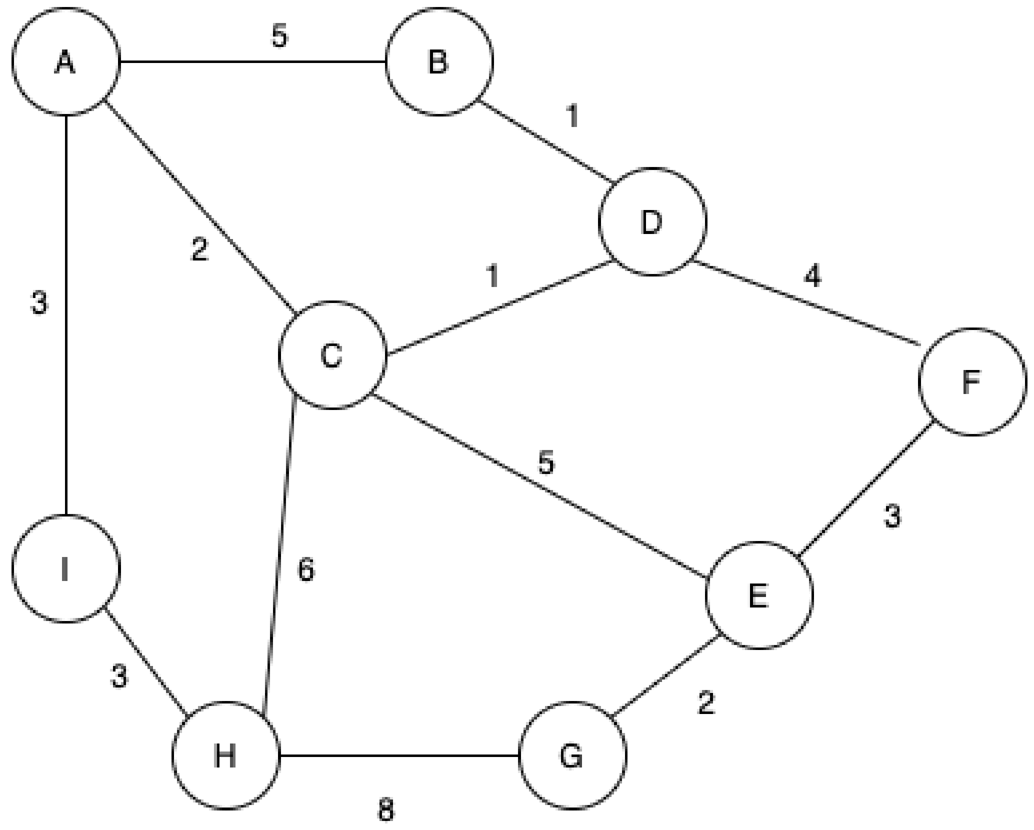


Figure 2.4: Graph for Dijkstra's algorithm example

giving us,

$$d(a) < d(a),$$

which is clearly a contradiction.

Thus, it must be true that for all vertices $v \in G$, $d(v) = \delta(v)$. This means that Dijkstra's algorithm successfully finds the shortest path from the source vertex s to all other vertices in the graph. □

Now, to demonstrate this algorithm, we use the graph from Figure 2.4 with source vertex A. Consider our list of distances to be p and our list of seen vertices to be S. Initially, the first entry in p, which represents the vertex A, is zero and the remaining entries, representing vertices B through I, are infinity. Here, S contains only the vertex A. Now, we find the distances from A to its neighbors, which are B,

C, and I. The distances to each of these vertices are 5, 2, and 3 respectively; these distances are now added to p. Since the shortest distance is to vertex C, we go there next. Next, we add to p the distances to C's neighbors: 3 to D, 7 to E, and 8 to H. Now we are finished with C, so it is added to S and we move on to I. The only neighbor of I is H, which, when traveling through I, is a distance of 6 from A. Since 6 is less than 8, the current value for H, the value stored in p becomes 6. The next step is to add I to S and then go to vertex B. Its only neighbor is D, and in this case the distance is 6, which is less than the distance currently stored for D, so we add B to S and do not change p. Now we go to D because it has the shortest distance in p among vertices not in s. Here we add 7 to S for vertex F and add D to s. Next we go to H where we add a value of 9 to G's position in p while also adding H to s. After that we go to E; we cannot find a shorter path to E's neighbors so we do not change p, and we add E to S. Last, we visit F and then G, finding in each case that there is no change in p. We have now completely run the algorithm, our final value for S is:

$$S = [A, C, I, B, D, H, E, F, G],$$

which represents the order we have visited the vertices in the graph. We have also calculated the distances from the source vertex to each other vertex:

$$d(S) = p = [0, 2, 3, 5, 3, 6, 7, 7, 9].$$

This means that we started at the node A with distance $d(A) = 0$, travelled to C with distance $d(C) = 5$, then I, with $d(I) = 2$ and so on. Table 2.1 shows the intermediate values of S and p throughout the algorithm.

Step	S	p
0	$[A]$	$[0,\infty,\infty,\infty,\infty,\infty,\infty,\infty,\infty]$
1	$[A]$	$[0,2,3,5,\infty,\infty,\infty,\infty,\infty]$
2	$[A,C]$	$[0,2,3,5,3,7,8,\infty,\infty]$
3	$[A,C,I]$	$[0,2,3,5,3,6,7,\infty,\infty]$
4	$[A,C,I,B]$	$[0,2,3,5,3,6,7,\infty,\infty]$
5	$[A,C,I,B,D]$	$[0,2,3,5,3,6,7,7,\infty]$
6	$[A,C,I,B,D,H]$	$[0,2,3,5,3,6,7,7,9]$
7	$[A,C,I,B,D,H,E]$	$[0,2,3,5,3,6,7,7,9]$
8	$[A,C,I,B,D,H,E,F]$	$[0,2,3,5,3,6,7,7,9]$
9	$[A,C,I,B,D,H,E,F,G]$	$[0,2,3,5,3,6,7,7,9]$

Table 2.1: Step-by-step values for Dijkstra example

2.3.2 Floyd-Warshall Algorithm

Similar to the SSSP problem, there also exists the all-pairs shortest path (APSP) problem. This is the problem of finding the shortest distance between every pair of vertices in the graph. While this can be done by performing Dijkstra's algorithm on every vertex in the graph, that would result in occasions where the same paths are evaluated multiple times; the Floyd-Warshall algorithm is a more efficient method for solving this problem.

At the beginning of the algorithm we define $D^{(0)}$ to be an $n \times n$ matrix representing the initial paths between every pair of vertices. Similar to Dijkstra's algorithm, if there is no direct path between two vertices, their entry in $D^{(0)}$ is infinity. Then, from $k = 1$ to n we calculate $D^{(k)}$ where k represents the new vertex that paths may travel through. Here, every ordered pair of vertices (i, j) is observed in order to calculate $d_{ij}^{(k)}$, which represents the distance of the shortest path from i to j. We can define $d_{ij}^{(k)}$ as:

$$d_{ij} = \begin{cases} w_{ij} & k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & k \geq 1. \end{cases}$$

Once we reach $k = n$, the algorithm outputs the matrix $D^{(n)}$, which represents the lengths of the shortest path between each pair of nodes in the graph. This algorithm

```
Input: G(V, E, w)   // A weighted graph
Output: D[n, n] // Distances between pairs of vertices

// Initialize D
for $i = 1$ to $n$:
    for $j = 1$ to $n$:
        if $(i, j)$ in $E$:
            // w(i, j) = weight on edge from i to j
            D[i, j] <- w(i, j)
        else:
    D[i, j] <- ∞

for $k = 1$ to $n$:
    for $i = 1$ to $n$:
        for $j = 1$ to $n$:
            if D[i, k] + D[k, j] < D[i, j]:
                D[i, j] <- D[i, k] + D[k, j]

return D
```

Listing 2.2: Floyd-Warshall Algorithm

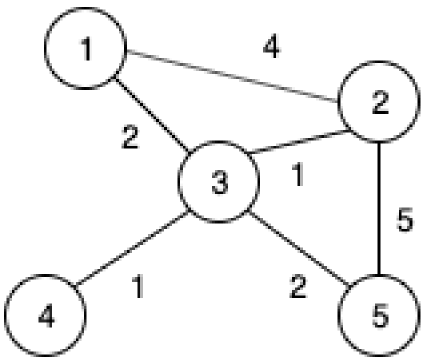


Figure 2.5: Graph for Floyd-Warshall algorithm example

is an example of dynamic programming because it takes advantage of disjoint subproblems to find a solution [8]. It has a run-time complexity of $\mathcal{O}(V^3)$, where V is the number of vertices in the graph. A pseudocode implementation of this algorithm is shown in Listing 2.2.

Next, we demonstrate this algorithm using the graph shown in Figure 2.5. First, we create our initial matrix:

$$D^{(0)} = \begin{pmatrix} 0 & 4 & 2 & \infty & \infty \\ 4 & 0 & 1 & \infty & 5 \\ 2 & 1 & 0 & 1 & 2 \\ \infty & \infty & 1 & 0 & \infty \\ \infty & 5 & 2 & \infty & 0 \end{pmatrix}.$$

Now, we let $k = 1$ and calculate $D^{(1)}$. In the case of this example, $D^{(1)} = D^{(0)}$ because vertex 1 cannot be used to reach any vertices except 2 and 3, which are already connected to it. Next we need to calculate $D^{(2)}$ by observing paths that go through vertex 2. Here, we see that by going through vertex 2 we can create a path from 1 to 5 with a weight of 9. This is the only new path we can create that is shorter than paths that have already been found, so

$$D^{(2)} = \begin{pmatrix} 0 & 4 & 2 & \infty & 9 \\ 4 & 0 & 1 & \infty & 5 \\ 2 & 1 & 0 & 1 & 2 \\ \infty & \infty & 1 & 0 & \infty \\ 9 & 5 & 2 & \infty & 0 \end{pmatrix}.$$

Continuing through the problem, we now look at $k = 3$. By incorporating vertex 3 we can create several shorter paths than we could before. We can now create paths from vertex 1 to vertices 4 and 5 and from vertex 2 to vertex 4. Additionally, we find shorter paths between vertices 1 and 2 and vertices 2 and 5, which gives us a new D matrix of

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 2 & 3 & 4 \\ 3 & 0 & 1 & 2 & 3 \\ 2 & 1 & 0 & 1 & 2 \\ 3 & 2 & 1 & 0 & 3 \\ 4 & 3 & 2 & 3 & 0 \end{pmatrix}.$$

Looking ahead to $k = 4$ and $k = 5$, we can see that they do not create any new or shorter paths between vertices. Thus, the algorithm exists with an output matrix of $D = D^{(3)}$.

CHAPTER 3

Application Development Environment

Choosing an application development environment is one of the first steps in the software development process. The application development environment is the collection of software tools that are used to create the application. In the world of web development, there are many languages and frameworks that can be used, each with their own positives and negatives. This chapter introduces the language, framework, and libraries that are used in this project and why they are chosen.

3.1 Python

Choosing a programming language is a critical first step in any software engineering project. There are many languages out there, and there is not always an objectively correct choice for a particular project. Several factors are considered in this choice, including applicability to the problem at hand, ability to interact with other tools, and developer familiarity. Deciding to develop a web-based application narrows down the options significantly, as only a few languages are widely used in web development: JavaScript, Python, Java, Ruby, and PHP. Among these, JavaScript and Python are by far the most loved by developers; conversely, PHP and Java are among the least liked, according to a 2019 survey by `hired.com` [1]. This, combined with the extensive documentation and resources for both Python and JavaScript narrows the choice down to these two languages. At this point, either language is likely to be equally good, so Python is used because of developer familiarity.

Python is a modern programming language that is one of the most popular programming languages in the world. It is known for being one of the easiest languages to learn as well as one of the most versatile. Additionally, it has extensive documentation and an extensive library of packages that are easy to install and can help save time and energy. There are two versions of Python currently used by developers: Python 2 and Python 3. However, as of January 1, 2020, Python 2 is no longer officially supported, which means that it is no longer suitable for new projects. As a result, this project uses Python 3. Python also has several web frameworks to choose from and is known for being a great back-end programming language [29].

3.2 Django

Along with choosing a programming language, it is important to pick a web framework to use when developing a web-based application. Web frameworks are software "that provides a way to build and run web applications" [26]. This allows the programmer to worry only about code that is unique to their application, instead of dealing with the hassle of writing code has already been written for countless other applications. Web frameworks are typically associated with a specific programming language, and Python has several options including Django, Flask, and Pyramid. Among these options, Django stands out as the best choice for this project.

Django is the most popular Python web framework [22], and it is used in many popular websites including Pinterest, *The Onion*, and BitBucket [2]. It is a full-stack framework, which means that it "supports the development of back-end services, front-end interfaces, and databases” [22]. Additionally, Django is free and very easy to install; it is available on the Python package manager, pip, and initializing a project requires only one terminal command [13]. The main draws to using this framework are its speed, scalability, and popularity. As mentioned, it is incredibly easy to start a project using Django, and the framework provides extensive starter code to help get development rolling. Django applications also scale easily. This is an important feature for any web application, because this ensures that it will be able to handle any number of users that may want to use it. While popularity is not and should not be the main factor when determining what tools to use, Django’s popularity means that is has extensive documentation and tutorials across the web, making it easy to learn and get help when necessary. Additionally, its popularity makes it a useful tool to learn.

3.3 Python Packages

When developing software, it is usually best practice to avoid writing code that has already been written. Doing so makes development take longer and can take away from the overall quality of the software product. Packages are pieces of software that have already been written to provide some sort of functionality that might be needed for many different projects. Similar to using a web framework, using packages allows developers to focus on what really makes their project unique. This project utilizes several Python packages, and this section gives some details on what they are and what functionality they provide.

3.3.1 NetworkX

In this project, it is necessary to be able to analyze road systems in order to find meeting places. These networks are best interpreted as graphs, and to work with them it is necessary to have an implementation of graphs in Python. NetworkX is a package that provides this functionality. It is designed for the "creation, manipulation, and study of the structure, dynamics, and function of complex networks" [20]. While this project does not use NetworkX to manually create graphs, that is done in OSMnx (mentioned below), NetworkX is used to traverse through and analyze the graphs that are created. More specifically, NetworkX is used to store graphs, analyze node and edge attributes, and access the connections stored within graphs.

3.3.2 OSMnx

As mentioned before, this project uses NetworkX to interact with graphs, but in order to do so there needs to be a way to acquire the necessary graphs representing street networks. This is done using the OSMnx package. Created by urban planning professor Geoff Boeing, "OSMnx is a Python package for downloading administrative boundary shapes and street networks from OpenStreetMap" [5]. OpenStreetMap is an open-source database of global map data that is freely available and used by many developers. OSMnx was originally created in the context of analyzing street networks in the context of urban planning and analysis, but that has a common requirement with this project: the ability to create a graph representing the road network in some geographical area. It allows this to be done using several different ways of defining the desired area. A user may ask for a location by name, by address, by defining a bounding box of latitude and longitude values, or by providing a specific latitude and longitude. The package is able to do this by querying OpenStreetMap. Once the data is acquired from OpenStreetMap, OSMnx uses NetworkX to create graphs that contain all the relevant data including road names, road types, and speed limits. Figure 3.1 is a visualization of a graph created by OSMnx that represents the street network of Wooster, Ohio.

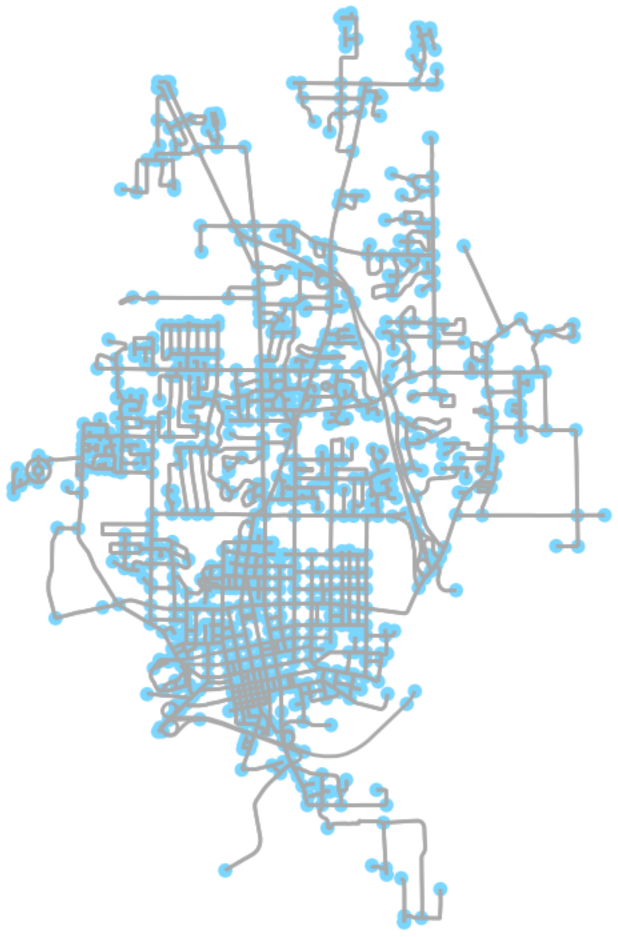


Figure 3.1: OSMnx graph of Wooster, Ohio road network

3.3.3 Geopy

Geopy is a Python package that is used for geocoding. It allows developers to find the geographical coordinates of text addresses, which is useful in this application for creating graphs and locating nodes on a graph closest to the user-given locations. This package is a client for several geocoding services, but the particular one used by this application is Nominatim, which is also used by OSMnx. Additionally, this package enables reverse geocoding, which allows conversion from coordinates to an address, which is useful for describing where a meeting place is, since most users would presumably not want to receive geographic coordinates [12].

3.4 Classic Web Development Tools

Web frameworks help to do most of the heavy-lifting when it comes to web development, but they cannot quite do everything. There are a few web development tools that are essentially inescapable when creating web applications. Here, we briefly introduce the ones that are necessary for this project.

3.4.1 JavaScript

JavaScript is a programming language that was created specifically to be used on the web. It follows a Java-like expression syntax, but it does not have static typing and tends to be a much more free-form programming language than Java. JavaScript is particularly useful for allowing developers to make webpages interactive [15]. The language provides convenient access to elements on a webpage, and it enables developers to define functionality for different events that may be given by the user. This functionality cannot be done using Python, which is why this project must utilize some JavaScript. Listing 3.1 shows a hello world program in JavaScript.

```
// Log "Hello world!" to the console
console.log("Hello world!")

// Display "Hello world!" in a popup on a webpage
alert("Hello world!")
```

Listing 3.1: JavaScript Hello World

```
<!DOCTYPE html>
<html>
<body>

  <h1>Hello World This is my title</h1>
  <p> And this a some content.</p>

</body>
</html>
```

Listing 3.2: HTML Hello World

3.4.2 HTML and CSS

Hyper-text markup language, or HTML, is the base for most websites on the internet. It is a formatting language that provides structuring for website content. HTML consists of a series of elements, which enable developers to define how things should look [14]. Django interacts well with HTML files and even gives the developer the ability to pass information from their Python code to HTML files. Essentially, HTML is used to visualize the application and give users the ability to interact with it, but it plays no role in the actual inner-workings of the application. A hello world HTML page can be seen in Listing 3.2.

Cascading style sheets (CSS) go hand-in-hand with HTML and are a method of further refining the visual aspect of a web page. CSS allows developers to define specific attributes for types of HTML elements and the individual elements themselves. These attributes include color, font, alignment, and size, among other things [30]. While this project does not involve much development with CSS, it is important to note that there is CSS under the hood in order to make the application look aesthetically pleasing. The CSS used to format pop-ups in our application (which will be discussed later) can be found in Listing 3.3.

```css
.mapboxgl-popup {
  max-width: 200px;
}

.mapboxgl-popup-content {
  text-align: center;
  font-family: 'Open Sans', sans-serif;
}
```

Listing 3.3: CSS for formatting pop-ups

3.4.3 Bootstrap

HTML and CSS are great resources, but creating a professional looking website using these tools alone is difficult and time consuming. Since the focus of this project is the back-end component and not the front-end interface, it is necessary to utilize tools that streamline the process of creating a nice user interface. This is where Bootstrap, "one of the most popular front-end frameworks . . . in the world" [21], comes in. As a front-end framework, Bootstrap has created a system of CSS and JavaScript files that allows developers to quickly assemble nice-looking web pages. The ability to quickly create quality interfaces combined with extensive documentation makes Bootstrap a valuable resource for this project.

CHAPTER 4

Meeting-place Algorithm

4.1 Problem in Context

In the Introduction, we discussed the problem in a broad sense. Here, we discuss it in a more formal and mathematical context. To do this we recall quasimetrics, which we discussed in Section 2.2. We use a quasimetric because, for road networks it is not necessarily true that $d(x, y) = d(y, x)$. For example, in a city with one way streets there may be an edge beginning at x and ending at y but no such edge from y to x. In this case, traveling from y to x may require traveling along several edges, which would likely not take the same time as traveling along the edge from x to y.

To create a quasimetric that describes our problem, we must decide how to quantify the distance between two nodes. Here, there are two logical options: the distance traveled on the path from x to y, or the amount of time it takes to travel the path from x to y. Using the distance of the path from x to y would be useful in finding a meeting place that is geographically central, but it is more practical for users to receive the point that takes the least collective amount of time to reach. Hence, we quantify the distance from x to y as

$$d(x, y) = \text{the amount of time to travel from } x \text{ to } y.$$

Quantifying the distance between nodes is necessary for formalizing our problem, however this value alone does not sufficiently describe the problem at hand. To do this, we need to create a function describing the cumulative distances between our starting points x_i and an arbitrary node y. If we let $n \geq 3$ be the number of places the user wishes to find a meeting place between, then we can define our function D to be:

$$D(y) = \sum_{i=1}^{n} d(x_i, y),$$

where y is an arbitrary node. Our problem then, is to minimize this function. We can define our ideal meeting place m to be a node such that

$$D(m) \leq D(y) \text{ for all } y \in V,$$

where V is the set of all nodes in our graph. An alternative measure that could be used would be to sum the squares of the distances. This would penalize nodes that are farther away from one or more of the initial locations, and would find meeting places that take a roughly equal amount of time to reach from each of the initial locations. We do not use the square of the distances because our objective is to minimize the total time without attempting to make the times to the meeting place equal. However, this alternative measure could be worth implementing in future research.

4.2 Brute Force Method

A common first step in solving a problem algorithmically is to first come up with a brute force solution. This type of solution is meant to solve the problem without accounting for efficiency. It is typically the simplest way to solve the problem and helps to provide a starting point for future algorithmic optimization.

```
# Input: G, start_nodes  Graph and list of starting nodes
# Output: Node representing meeting place

dijkstra_results = []
for node in initial_nodes:
    dijkstra_output = nx.single_source_dijkstra(graph, node, weight=get_travel_time)
    dijkstra_results.append(dijkstra_output[0])

minimum_node = (None, math.inf)
for node in graph.nodes():
    node_sum = 0
    for result in dijkstra_results:
        if node in result:
            node_sum += result[node]
        else:
            node_sum = math.inf
    if node_sum < minimum_node[1]:
        minimum_node = (node, node_sum)

return minimum_node[0]
```

Listing 4.1: Brute force algorithm

For this particular project, the brute force method utilizes Dijkstra's algorithm. Dijkstra's algorithm is performed on each of the given starting nodes, with the distances to each node being stored. Then, the algorithm iterates through every node of the graph and calculates the sum of the values found by Dijkstra's algorithm for each node. The node with the minimum sum has the shortest combined distance from the starting nodes. The Python code for this algorithm is shown in Listing 4.1.

4.3 Optimization Techniques

Clearly, using a brute force technique is not an ideal solution because it requires going through the entire set of nodes multiple times. Here, we discuss some methods that may be used to improve the performance of the algorithm.

4.3.1 Contraction Hierarchies

One method of optimizing graph algorithms is by creating a contraction hierarchy. This technique takes an already created graph and does some preprocessing of it in order to enable algorithms to run faster. It takes advantage of the hierarchical nature

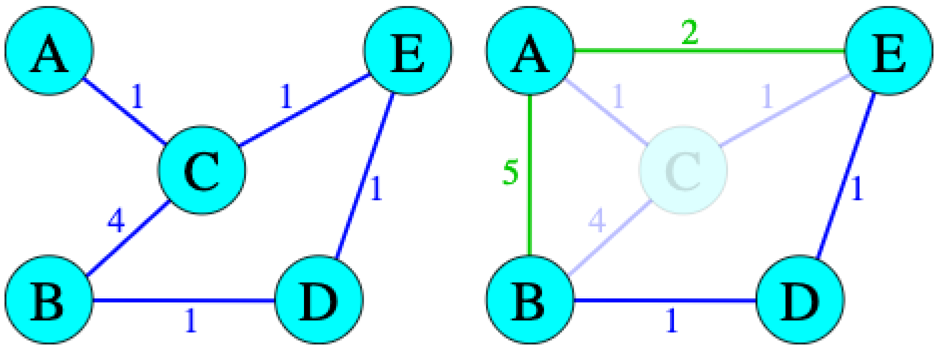


Figure 4.1: Simple node contraction example [28]

of road networks to create shortcuts between nodes, which allows algorithms to check fewer nodes while still obtaining an accurate result.

The process of creating a contraction hierarchy centers around contracting nodes from the graph. When a node is contracted, we remove it from the graph in such a way that all shortest paths are preserved. To ensure this, whenever a node v is being contracted, we run Dijkstra's algorithm on every predecessor u_i of v while ignoring v. This allows us to check if there exists a path from u_i to some successor, w, of v that bypasses v and has a length less then $\text{length}(u_i, v) + \text{length}(v, w)$; such paths are referred to as witness paths. If this is the case, then there does not need to be a shortcut from u_i to w because the shortest path between those nodes does not go through v, otherwise we construct a shortcut from u_i to w before we remove v from the graph. It is crucial that we do this, otherwise too many shortcuts will be added to the contraction hierarchy, which will result in slower queries.

Figure 4.1 shows a simple example of one step of the node contraction process. On the right we see the original graph and on the left we see the graph when the node C is contracted. In order to do this, shortcuts need to be added from A to E and from A to B because the shortest paths between those pairs of nodes travel through C. No shortcut needs to be added from B to E because the shortest path between those nodes does not include C [28].

While we need to check for witness paths, running Dijkstra's algorithm multiple times on every node of a large graph would be too cumbersome to implement in reality, which means we need to find ways to limit the number or depth of searches that are performed. There are two common ways of doing this: stopping Dijkstra's algorithm early and limiting the number of hops. When performing Dijkstra's algorithm, if we encounter a node x such that length$(u_i, x) >$ length(u, v) + length(v, w), then we can stop the Dijkstra search because it is not possible for a witness path to exist assuming there are no negative edge weights. This is an assumption that must be made for Dijkstra's algorithm to work in the first place, and, given the nature of road networks, this assumption is always true in the context of our problem. A hop in this context is the action of traveling through an edge. Limiting the search to k hops means that we only consider shortest paths that go through at most k edges. Choosing a value for k is a tradeoff between the amount of preprocessing time and the size of the final contracted graph. A smaller k value results in faster preprocessing but an increased number of edges in the graph. Conversely, a larger k makes the preprocessing take longer, but there are fewer edges in the final graph. An unbounded k value would result in the smallest possible graph but would not provide any optimization for the preprocessing phase [19].

Better contraction hierarchies can be created if node ordering is used. While not necessary for correctness, good node ordering improves both preprocessing and query time. Node ordering is implemented by keeping nodes in a priority queue with decreasing importance, where importance is a value calculated using a heuristic. On each iteration, the least important node is contracted; however, prior to being contracted its importance value is updated and if the node is no longer the least important, it is put back into the queue with an updated importance value. Once the least important node is identified, it is contracted.

The importance metric is a heuristic function may consider several criteria. We discuss four popular criteria, but developers could experiment with any number of criteria to improve their heuristic. The first criterion is called *edge difference*. If we consider $s(v)$ to be the number of shortcuts added when contracting a node v, $in(v)$ to be the incoming degree of v and $out(v)$ to be the outgoing degree of v, then edge difference is

$$ed(v) = s(v) - in(v) - out(v).$$

It is ideal to contract nodes that have a small edge difference because that helps to minimize the number of edges in the final graph. Another criterion is simply the number of contracted neighbors; it is preferable to contract nodes with fewer contracted neighbors. Next, we consider a metric called the shortcut cover. The shortcut cover of a node v, denoted $sc(v)$ is defined as the number of neighbors, w, of v such that we must create a shortcut to or from w when we contract v. A larger shortcut cover means that more nodes depend on v, which makes v an important node. We want to contract important nodes later, so we start by contracting nodes with a smaller shortcut cover. Finally, the node level criterion, $L(v)$ is an upper bound on the number of edges in the shortest path from any s to v in the contracted graph. Initially, $L(v) = 0$; once a node v is contracted, for any neighbor u of v, $L(u) = \max(L(u), L(v) + 1)$. Again, it is preferable to contract nodes with a small node level. Now that we have observed some different criteria, we can define the importance of a node to be [19]:

$$I(v) = ed(v) + cn(v) + sc(v) + L(v).$$

The weights on each of the criterion can be adjusted to improve the metric.

An example of an "important" node in a road network would be a point where one highway merges into another [19]. This type of node is important because many paths must travel through it and to get from one highway to the other, traveling through this node is unavoidable. Our importance criteria would consider this type of node to be important because contracting a node like this would require many additional shortcuts because it is the only way to travel between these two highways. The high number of additional shortcuts causes the edge difference, shortcut cover, and node level to increase, resulting in the node being contracted later.

Unfortunately, it is not possible to implement this system within the timeframe of this project. Performing the contraction hierarchy algorithm requires obtaining the entire graph for the problem. Attempting to collect this data is unable to be done due to the size of the data representing the road network for the entire United States. There must be a way to collect this data in a manageable way, but no such method was found during the timeframe of this project.

4.3.2 Dijkstra Optimizations

As is the case with many algorithms, choices in data structures can help Dijkstra's algorithm run faster. In our earlier discussion on this algorithm, we used lists to describe how we stored the vertices of the graph. Using a priority queue can speed things up immensely. NetworkX, the package used for working with graphs, implements Dijkstra's algorithm using the a package called `heapq` which implements this data structure. Figure 4.2 shows the time it takes to run the brute force algorithm using a non-optimized Dijkstra implementation and the time it takes using NetworkX's optimized version under eight different test inputs. Based on this graph, we see that for most test cases the optimized Dijkstra algorithm resulted in the entire brute force algorithm running $\sim 20 - 30$ times faster than with the non-optimized version. For test cases 3 and 8, which take longer to run due to increased distance in between starting points, this improvement was magnified to be $\sim 100 - 200$ times faster than with a non-optimized Dijkstra implementation.

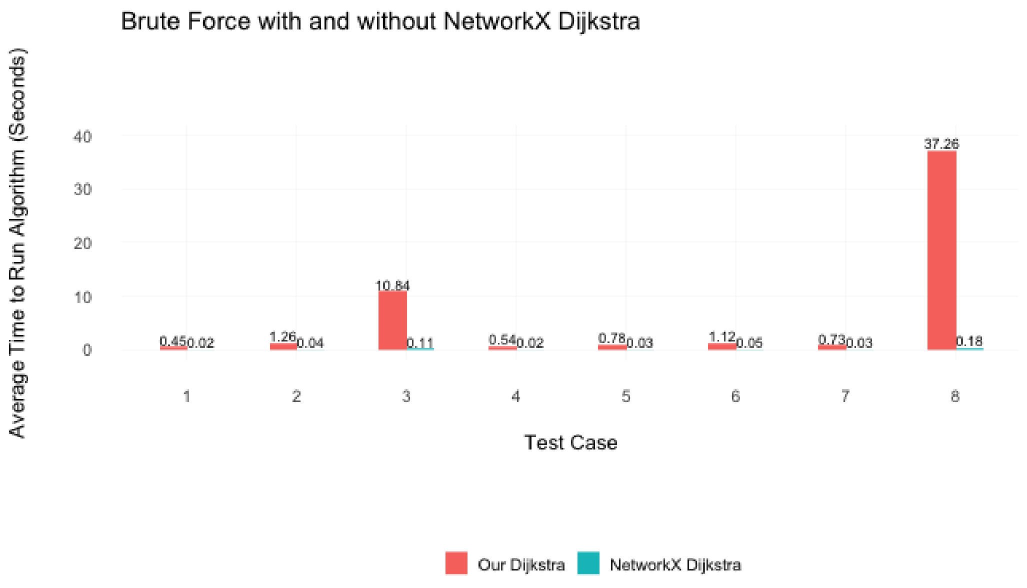


Figure 4.2: Comparison of optimized and non-optimized Dijkstra's algorithm run-time

The reason behind this speedup is the use of a priority queue to store the nodes that need to be observed. A priority queue, also known as a heap queue, implements a binary tree data structure where every parent has a value less than or equal to the values of its children. This method of storing nodes means that the node with the smallest value is always at the top of the tree. Having the smallest node at the top is a huge advantage for Dijkstra's algorithm because at every iteration the smallest remaining node is required. By using a priority queue, this node becomes readily available which makes finding it a constant-time operation. The cost of being able to do this is that inserting nodes into the queue takes $O(\log V)$ time; however, the query speedup far outweighs the additional time to insert elements. In contrast, when using a simple list to store nodes, finding the minimum node is a $O(V)$ operation because it is possible that the minimum node is the last element of this list. When working with graphs that have tens or hundreds of thousands of nodes, the ability to immediately find the smallest node at every iteration is a huge advantage, resulting in the incredibly faster run-time, as seen through our test cases.

4.3.3 Heuristic Algorithms

It is possible that our brute force algorithm is the best way to guarantee finding the meeting place that takes the minimum amount of time to reach. However, its accuracy alone does not guarantee that it is the best algorithm to use for our application. We must also consider the potential time benefits of using a *heuristic* algorithm. By definition, a heuristic is a "set of rules which, if followed, may achieve a solution but cannot guarantee doing so" [3]. The reason to use heuristics is that they have the potential to be much faster while producing a result that is good enough. Here we introduce some potential heuristic algorithms for this problem. For this section consider n to be the number of initial locations provided by the user.

Geographic Mean

The simplest algorithm that could be used to solve this problem is finding the geographic mean of the user-given locations. Finding the geographic mean only requires finding the average latitude and longitude of the given locations, which is an $O(n)$ operation, where n is the number of initial locations. Then, it finds the node in the graph closes to this coordinate and determines that node to be the meeting point. Clearly, this is the fastest conceivable algorithm for solving this problem; however, it is likely to have poor accuracy since it does not consider how long it takes to reach this point, which is the overall goal of the algorithm. Also note that we do not consider the curvature of the Earth, as we do not expect its effect to be significant enough to be worth the time it would take to account for it. Figure 4.3 shows the geographic mean point (marked in green) between The College of Wooster, Walmart Supercenter in Wooster, and the Wayne County Schools Career Center (marked in blue). The Python code for this algorithm is shown in Listing 4.2.

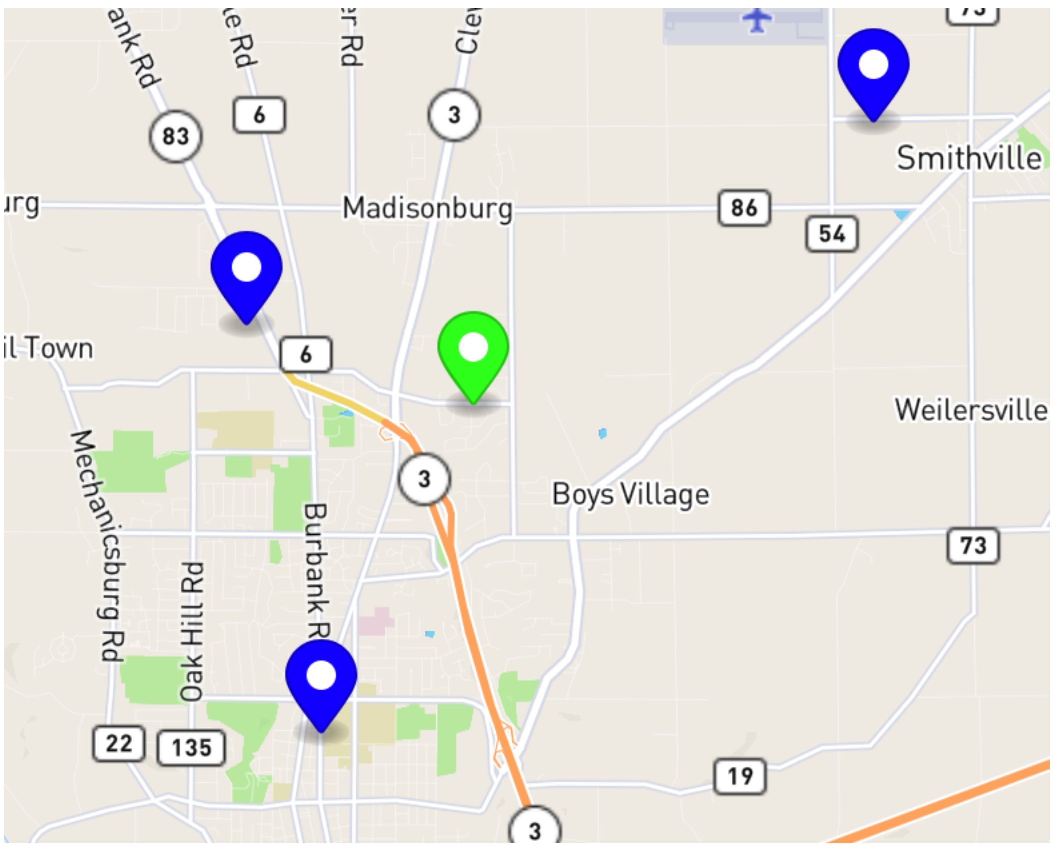


Figure 4.3: Example of a geographic mean point

```
# Input: graph, list of initial (lat, long) coordinates, list of initial nodes
# Output: Node representing meeting place
import osmnx as ox

lat_sum = 0
long_sum = 0
num_locations = len(initial_locations)
for location in initial_locations:
  lat_sum += location[0]
  long_sum += location[1]

mean_coordinate = (lat_sum / num_locations, long_sum / num_locations)
middle_node = ox.get_nearest_node(graph, mean_coordinate)

return middle_node
```

Listing 4.2: Geographic mean

Geographic Mean Path Traversal

Beginning by following the steps of the geographic mean algorithm, this heuristic takes advantage of the structure of the graph to improve the accuracy of the meeting point found. It does this by performing Dijkstra's algorithm on the geographic mean point. Then, it determines the minimum number of nodes on a path from the geographic mean to one of the initial locations. Using the paths found by Dijkstra's algorithm, it determines the next node on each path. If a majority of the paths have the same next node, then the algorithm slides down to that node and checks if the paths still match. In cases where there is no majority, the algorithm uses the geographic mean as the meeting point. The algorithm runs until it reaches the point where there is no similar node contained by a majority of paths or the end of one of the paths is reached. At this point, the last node reached by the algorithm is determined to be the meeting node. The code for this algorithm can be seen in Listing 4.3.

To demonstrate why this algorithm is effective, observe the graph in Figure 4.4. In this example, the vertices *A*, *B*, and *C* are our initial starting locations. The geographic mean point is labelled accordingly and is the point where the algorithm begins. Here, we see that the path from the geographic mean to *A* travels to *E* and then to *A*. The path to *B* travels to *F*, then *D*, then *B*; and the path to *C* goes to *F* and *D* before reaching *C*. Since the paths to both *B* and *C* first require going to *F*, and two points represents a majority where there are three total initial points, the algorithm travels to *F*. Then, since the paths to *B* and *C* both travel to *D* next, the algorithm also goes to vertex *D*. At this point the paths to *B* and *C* diverge, so the meeting point found by the path traversal algorithm is *D*.

We know this algorithm has better accuracy than the geographic mean algorithm, because it prevents similar paths from being traveled multiple times when it is not necessary. If we left the geographic mean point as the meeting point, then the path

```python
# Input: graph, list of initial (lat, long) coordinates, list of initial nodes
# Output: Node representing meeting place
import osmnx as ox
import networkx as nx
from collections import Counter

lat_sum = 0
long_sum = 0
num_locations = len(initial_locations)
for location in initial_locations:
  lat_sum += location[0]
  long_sum += location[1]

mean_coordinate = (lat_sum / num_locations, long_sum / num_locations)
middle_node = ox.get_nearest_node(graph, mean_coordinate)

distances, paths = nx.single_source_dijkstra(graph, middle_node, weight=
    ↪ get_travel_time)

paths_to_start = []
shortest_path_length = math.inf
for node in initial_nodes:
    path_to_node = paths[node]
    paths_to_start.append(paths[node])
    if len(path_to_node) < shortest_path_length:
        shortest_path_length = len(path_to_node)

if shortest_path_length < 2:
    meeting_node = middle_node
else:
    first_steps = [path[1] for path in paths_to_start]
    most_frequent_step = Counter(first_steps).most_common(1)
    previous_step = most_frequent_step[0]
    most_frequent_step = most_frequent_step[0]
    current_step = 1

    # while a majority of the paths follow the same next step
    # and the end of the path with the fewest edges hasn't been reached
    half = (len(initial_nodes)/2)
    while most_frequent_step[1] >  half and current_step + 1 < shortest_path_length:
        next_steps = [path[current_step + 1] for path in paths_to_start]
        previous_step = most_frequent_step
        most_frequent_step = Counter(next_steps).most_common(1)[0]
        current_step += 1

    meeting_node = previous_step[0]

return meeting_node
```

Listing 4.3: Geographic mean path traversal

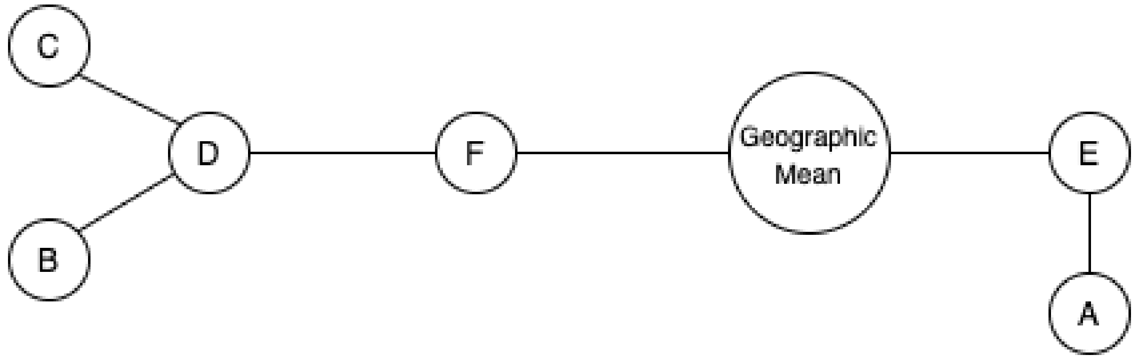


Figure 4.4: Path traversal algorithm example graph

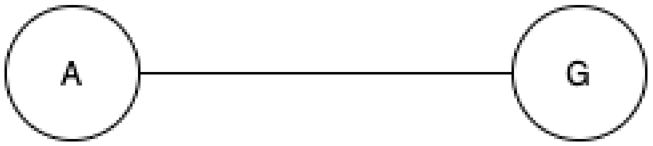


Figure 4.5: Example for path traversal proof

D to F to the geographic mean would be traveled twice: once by the person coming from B and again by the person coming from C. By sliding the meeting point over to D, this path is only traveled once. In this case it is traveled by the person coming from A, who travels that path in reverse.

Using Figure 4.5 as a reference, we can prove a result that supports our conclusion that path traversal is more accurate than geographic mean.

Proposition 2. *Assuming an underlying undirected graph, the majority of paths from the initial nodes to an optimal meeting point will never travel along the same edge incident to the meeting point.*

Proof. Let the edge AG be traveled by a majority of paths from starting nodes to node G. This means that if there are k starting nodes, there must be $m > \frac{k}{2}$ starting nodes whose paths to G go through A. Similarly, there must be $n < \frac{k}{2}$ starting nodes whose paths to G bypass the node A. Note that $n < m$ and $n + m = k$. Now, let $|M|$ be the sum of all path lengths from the m majority nodes to A. Let $|N|$ be the sum of path lengths from the n minority nodes to G. Finally, let $|AG|$ be the length of the path between A and G. Using these values, we can calculate the total sum of the path lengths from the starting nodes to both A and G:

$$\text{Sum(Paths to } G) = |M| + |N| + m \cdot |AG|$$

$$\text{Sum(Paths to } A) = |M| + |N| + n \cdot |AG|.$$

If G is the meeting point, then each of the m nodes which first travel to A must traverse the path AG. If A is the meeting point, then the n nodes which travel to G first must traverse the path AG. Since $m > n$ we can conclude that Sum(Paths to G) > Sum(Paths to A). This means that the majority of paths from the initial nodes to an optimal meeting point will never travel along the same edge incident to the meeting point. $\square$

Geographic Mean Neighbor Walk

This heuristic algorithm idea also builds on the geographic mean algorithm. Starting at the geographic mean node, it searches through all the neighbors of this node to see if any of them have a smaller cumulative time. If a better node is found, then the neighbors of that node are searched. This process repeats until either a node is reached which does not have any neighbors that are better or a maximum depth is reached. If this algorithm is performed on the graph in Figure 4.6, where the geographic mean in marked in blue, it first searches its neighbors, which are marked in green. Then it would search the appropriate green or white nodes corresponding to the neighbors of the minimal neighbor of the geographic mean node. Unfortunately, Dijkstra's algorithm must be run on each of the points in order to find the cumulative time it takes to reach them. For this reason, this heuristic algorithm is actually slower than brute force because it requires so many Dijkstra calls. A potential way to make this algorithm run faster than brute force would be to

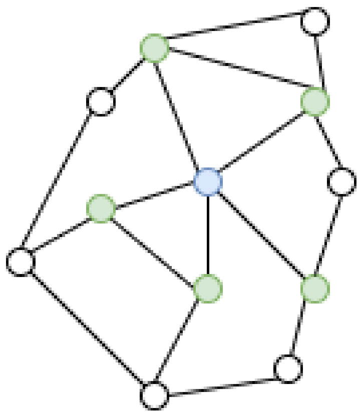


Figure 4.6: Graph for neighbor walk example

use the A* searching algorithm. The A* search algorithm might make this algorithm usable because it has a maximum time complexity of $O(E)$ where E is the number of edges in the graph. This is a much faster runtime than Dijkstra's algorithm, so running this many times may be better than running Dijkstra's algorithm only a few times. Unfortunately, NetworkX does not implement this algorithm on graphs that can have multiple edges between two nodes, so this is not currently a viable option.

Midpoint Intersection

The first step performed by this heuristic is to find the geographic means of subsets of the starting coordinates, which we call *midpoints*. For example, if there are four coordinates, then for each one its corresponding midpoint is the mean of the other three coordinates. For each of the initial locations, the midpoint of all other initial locations is found. Then, the path from the initial location to the midpoint is found. Figure 4.7 is an example of a midpoint and a path from its corresponding initial node. In this example we find the midpoint (colored red) corresponding to the node $(1, -5)$ and draw the appropriate path. Note that this is a simplification, and that each of these nodes would be part of a much larger graph.

Once this process is performed for each initial location, the heuristic checks if there are any points that are common between at least two of the paths and finds

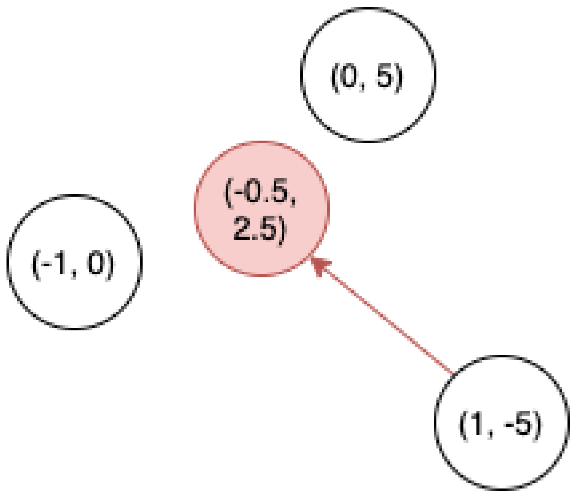


Figure 4.7: Example of a midpoint in the midpoint intersection algorithm

the cumulative time it takes to reach each of these points. If no such point exists, then the cumulative time to each midpoint and the geographic mean is calculated. In either case, the point with the minimum cumulative sum is considered to be the meeting point. As was the case in the neighbor walk algorithm, this heuristic relies on Dijkstra's algorithm to find paths to midpoints, resulting in more Dijkstra calls than the brute force algorithm, making it slower. Again, using A* could enable this algorithm to be faster than brute force, but that is not currently possible. Ultimately, this means that the neighbor walk and midpoint intersection algorithms are not beneficial since they are both slower and less accurate than the brute force method.

4.4 Algorithm Analysis and Comparison

Here we discuss the time complexity of the brute force, geographic mean, and path traversal algorithms as well as the accuracy and actual runtime of each as determined by data gathered by running the algorithms over a set of test cases. This analysis enables us to make more informed decisions about the pros and cons of each algorithm and helps us to determine which of them is the best choice for use in our application.

4.4.1 Complexity Analysis

Recall from Chapter 2 that the runtime of Dijkstra's algorithm is $O(V \log V + E \log V)$ where V is the number of vertices and E is the number of edges. We can further refine this runtime by taking advantage of the fact that planar graphs have no more than $3V - 6$ edges. This fact can be used because our road network is approximately a planar graph, since we define our vertices to be the intersection between two roads. This means that for the most part no two edges cross over each other except for at a node. Using this allows us to determine the runtime of Dijkstra's algorithm to be $O(V \log V + (3V - 6) \log V) = O((4V - 6) \log V) = O(V \log V)$. Now, we use this to help analyze the runtime complexities of our algorithms. Recall that n is the number of initial locations.

Brute Force

Determining the complexity of this algorithm is fairly simple. We break this algorithm down into two steps: obtaining distances and calculating sums of distances. To obtain distances from the input locations to all other nodes on the graph we perform Dijkstra's algorithm on each of the input locations. Thus, the complexity of this portion of the algorithm is $n * V \log V$ because we run Dijkstra's algorithm n times. Next, the algorithm finds the sum of distances for every node in the graph. This requires finding V sums; finding each of these sums requires n addition operations, resulting in a time complexity of $n * V$ for this portion of the algorithm. Since we can keep track of the minimum sum as we calculate the sums, finding the minimum is a constant time operation which adds nothing to our time complexity. With this information, we determine that the time complexity of the brute force algorithm is:

$$O(nV + n((4V - 6) \log V)) = O(nV \log V).$$

Geographic Mean

We briefly mentioned the complexity of this algorithm when it was introduced; here we reiterate its complexity and justify how we came to that conclusion. The step of finding the geographic mean coordinate has a complexity of $O(n)$, where n is the number of user-given locations. Doing this requires traversing through the list of initial locations exactly one time. Since this list has length n, the time complexity of this step of the algorithm must be $O(n)$. The complexity of finding the node in the graph is $O(V)$, because coordinates of each of the nodes in the graph must be found in order to determine which is the closest. This means that the overall complexity of the algorithm is $O(V + n)$. For very large graphs, the number of initial locations is insignificant compared to the number of nodes in the graph, so in these cases the runtime complexity is $O(V)$.

Geographic Mean Path Traversal

This algorithm begins with the geographic mean algorithm, so its first step has a complexity of $O(V)$. Then, it performs Dijkstra's algorithm one time, which has a time complexity of $O(V \log V)$. Finding the number of nodes in the path with the fewest edges is another $O(n)$ operation, because it only requires checking the lengths of n paths. The process of traversing paths to find improved meeting places is iterative. At each iteration, finding the next steps for each path is an $O(n)$ operation, and finding which of the steps is the most common is also $O(n)$. This means that each iteration has a complexity of $O(n)$. Further analysis could be done to determine how many iterations we would expect to need on average; however, it is clear that the dominant part of the runtime complexity of this algorithm is running Dijkstra's algorithm. This means that we can consider the overall complexity of this algorithm to be $O(V \log V)$.

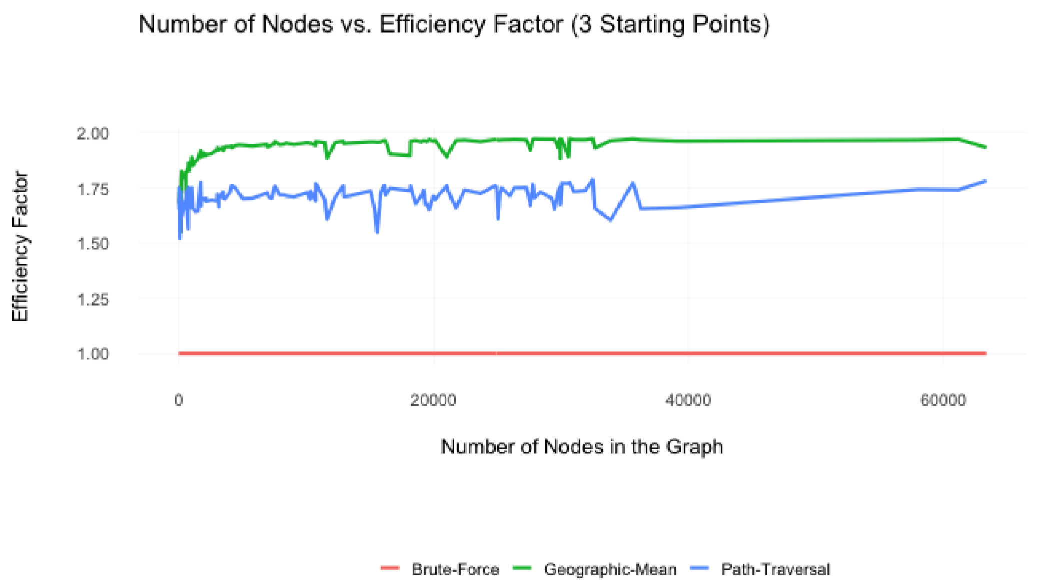


Figure 4.8: Algorithm efficiency factor comparison: 3 starting points

4.4.2 Accuracy and Runtime

In order to more concretely determine which of these algorithms performs best, we need to perform some data analysis. To do this we use a program which randomly generates test cases with varying numbers of starting points, locations, and distances between starting locations. This program then tests each algorithm by running it twice on each test case and collecting data on how long it takes to run. The first factor we look at is how fast each algorithm runs. We know that geographic mean and path traversal should run faster than brute force, but we need to determine *how much* faster they arre. As a way to provide some normalization between test cases, we call our "speed" metric `efficiency factor` and set it to be one plus the difference of the the runtime of the algorithm and the runtime of the brute force algorithm divided by the runtime of the brute force algorithm for the same test case. This means that the brute force algorithm always has an efficiency factor of 1. Any algorithm faster than it has an efficiency factor of greater than 1, and the maximum possible efficiency factor is 2. A higher efficiency factor corresponds to a faster runtime.

Figure 4.8 shows the graph of the efficiency values for each algorithm as they relate to the number of nodes in the graph. We use the number of nodes for the x-axis because it provides a better description of the size of the problem than a measurement such as the distance between the initial locations. This is because the algorithms must traverse through many nodes and edges on the graph, and the number of these that exist on the graph does not solely depend on the distance between the initial locations. Here we see that the geographic mean algorithm approaches the maximum efficiency factor of two. Additionally, the efficiency factor of the path traversal algorithm is consistently hovering around 1.75. It is slower than geographic mean, which is expected, but is still a significant improvement on brute force.

Next we must analyze the accuracy of these algorithms. The first step towards this is determining what it means to be accurate. In this case, we define accuracy of an algorithm as the difference between cumulative time from the initial locations to the meeting point found by the brute force method and the same time found by the algorithm being analyzed. To provide normalization, we calculate accuracy as one plus the difference of the time found by the algorithm and the time found by the brute force method divided by the time found by the brute force algorithm. Again, this results in the brute force algorithm always having an accuracy of 1 and a maximum possible value of 2. In this case, a value closer to one corresponds to a more accurate algorithm while, values greater from one correspond to less accuracy.

The graph in Figure 4.9 shows the accuracy of each algorithm as a function of the number of nodes in the graph. By definition, the accuracy of the brute force algorithm is always 1, which is expected because brute force finds the absolute best meeting point. This graph shows us that the geographic mean algorithm is far less accurate and its accuracy also varies wildly. The accuracy of the path traversal algorithm is similarly variable. However, the path traversal algorithm shows a

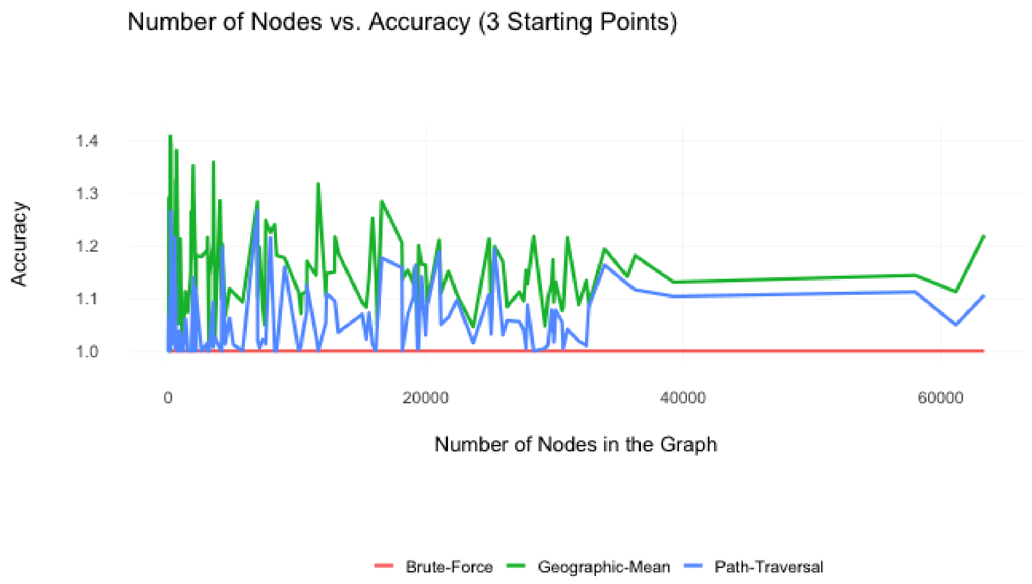


Figure 4.9: Algorithm accuracy comparison: 3 starting points

consistent improvement on the geographic mean algorithm and even finds the same meeting point as the brute force algorithm on some occasions.

While both speed and accuracy are necessary metrics to analyze these algorithms, in order to truly clarify which is "best" we must define a metric that encapsulates both measurements. Since we have already normalized the efficiency factor and accuracy, it is fairly simple to come up with a metric that uses both. In this case, we multiply the efficiency factor of the algorithm by two minus its accuracy. This formula is used because it maintains a baseline value of one for the brute force algorithm, it rewards efficiency and accuracy equally, and it accounts for the fact that higher accuracy is bad and higher efficiency is good. Values greater than one correspond to an algorithm having better overall performance than the brute force algorithm while values less than one correspond to an algorithm having a worse overall performance.

In Figure 4.10, we can see the performance of each algorithm. This graph indicates that both the geographic mean algorithm and the path traversal algorithm consistently outperform the brute force algorithm. The degree to which they

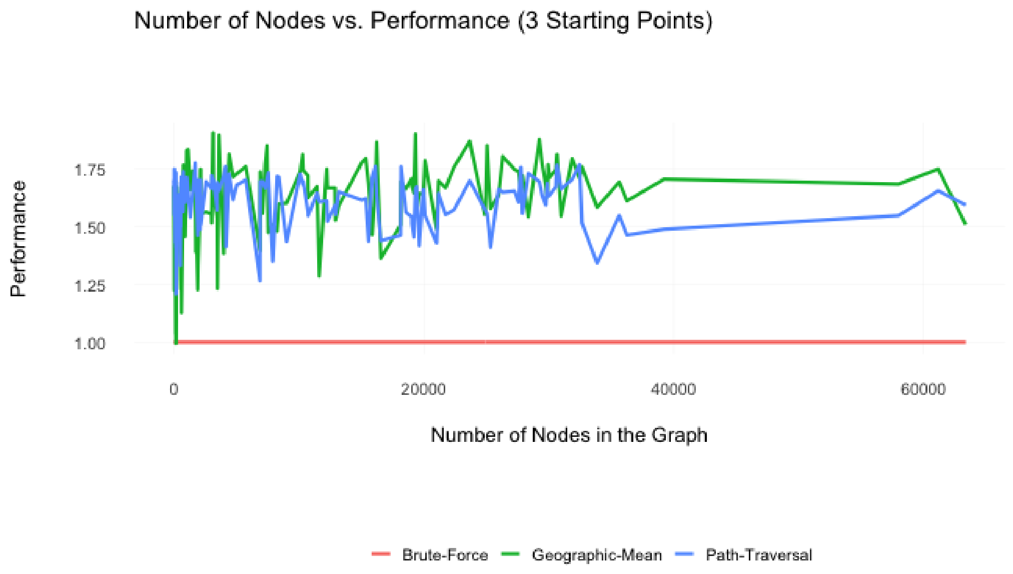


Figure 4.10: Algorithm performance comparison: 3 starting points

outperform brute force varies and there is no clear indication as to which cases enable these algorithms to perform better. Additionally, it is unclear which algorithm performs better between geographic mean and path traversal.

Before we can conclude our analysis, we must acknowledge that, while our current measurement of performance values speed and accuracy equally, users may value accuracy over efficiency. To account for this, we can change our formula for performance by taking the two minus the accuracy term of the algorithm and raising it to the third power. Raising to the third power is an arbitrary number, and further studies can be done to determine what the best adjustment would be. This does not change the baseline performance of the brute force algorithm, but it penalizes poor accuracy more than our previous method. We can see the result up this update performance formula in Figure 4.11.

When we make this change to our measurement of performance there is a noticeable change in results. The geographic mean and path traversal algorithms still perform better on average, but there are now cases where brute force outperforms them. Additionally, it now appears as though the path traversal algorithm

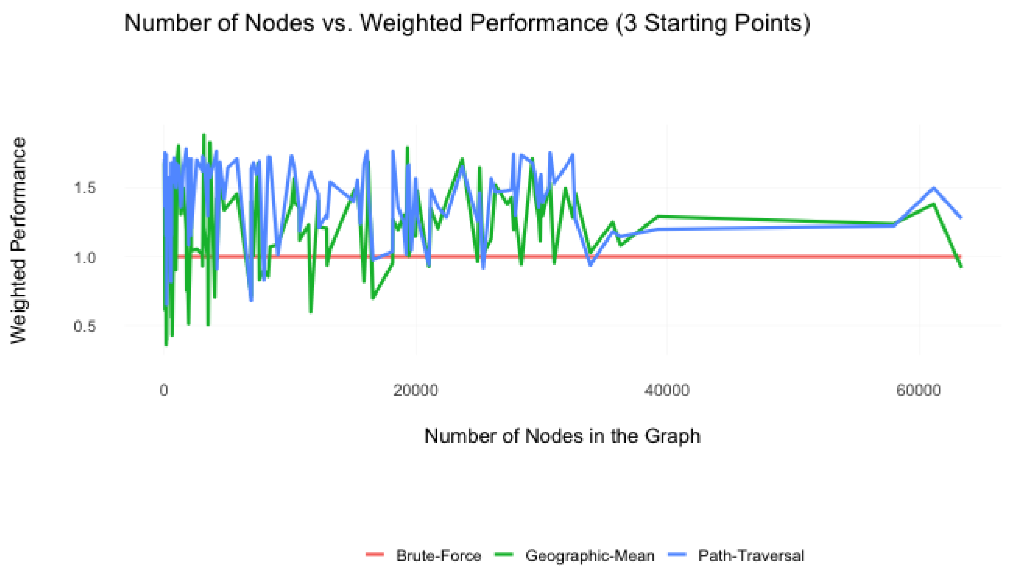


Figure 4.11: Algorithm performance comparison with additional weight on accuracy: 3 starting points

outperforms the geographic mean algorithm in most cases, and there are fewer cases of the path traversal algorithm performing worse than brute force.

Before making a final conclusion, we analyze these algorithms with both four and five initial points, to see if the results change. Figures 4.12 through 4.19 show the relevant data for doing this analysis. In the future, it would be useful to analyze the effects of adding even more initial points.

These figures show that the performance of the geographic mean algorithm remains steady with an increase in the number of starting points. However, the performance of the path traversal algorithm varies slightly. In particular, it appears that path traversal performs better with five initial locations than it does with four when we observe the weighted performance of the algorithms. This is likely due to a decrease in accuracy with four starting points. It makes some sense that this algorithm would perform worse with an even number of starting points than it does with an odd number of starting points. When there is an even number of starting points, it is more difficult for there to be a true majority of paths going in the same

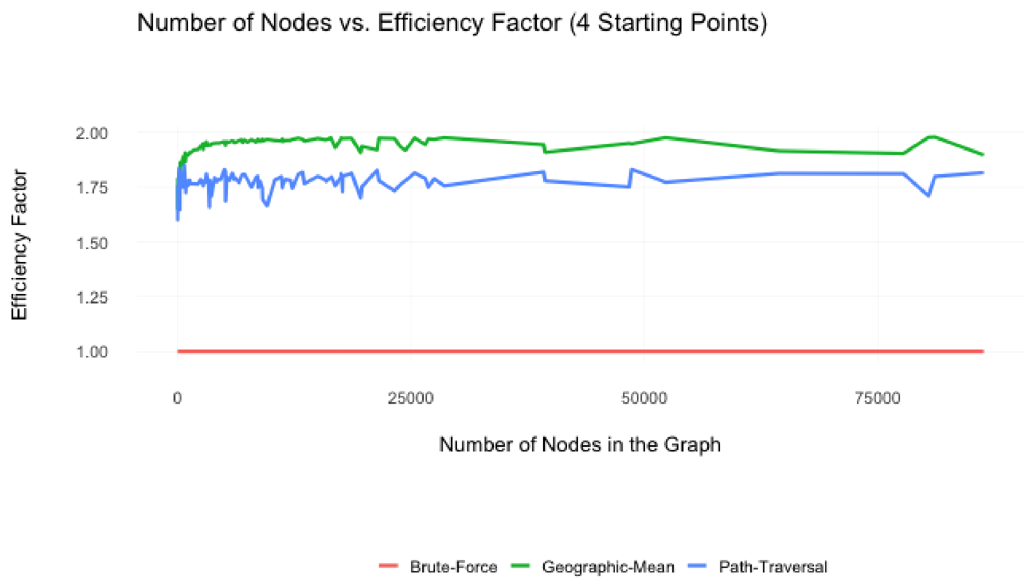


Figure 4.12: Algorithm efficiency factor comparison: 4 starting points

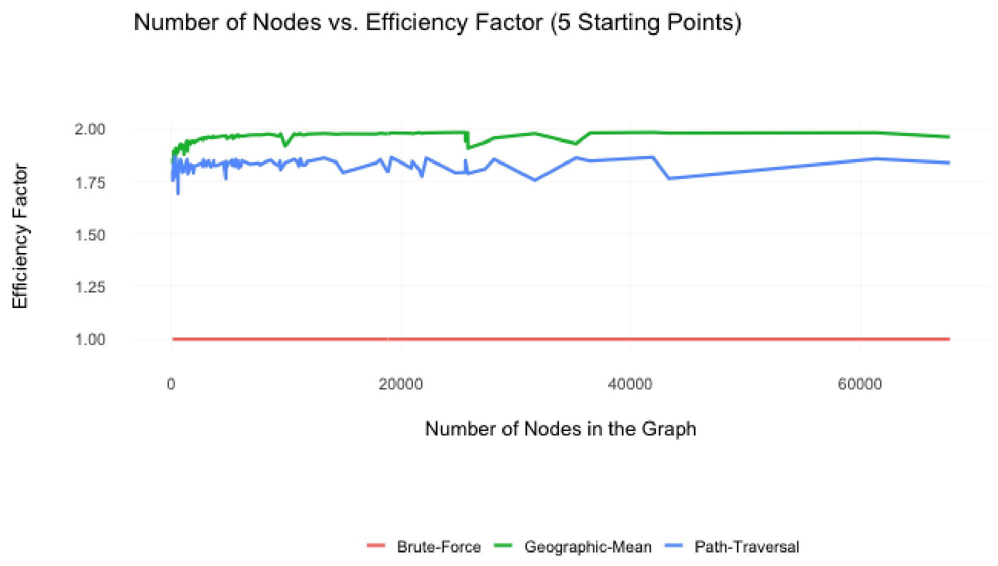


Figure 4.13: Algorithm efficiency factor comparison: 5 starting points

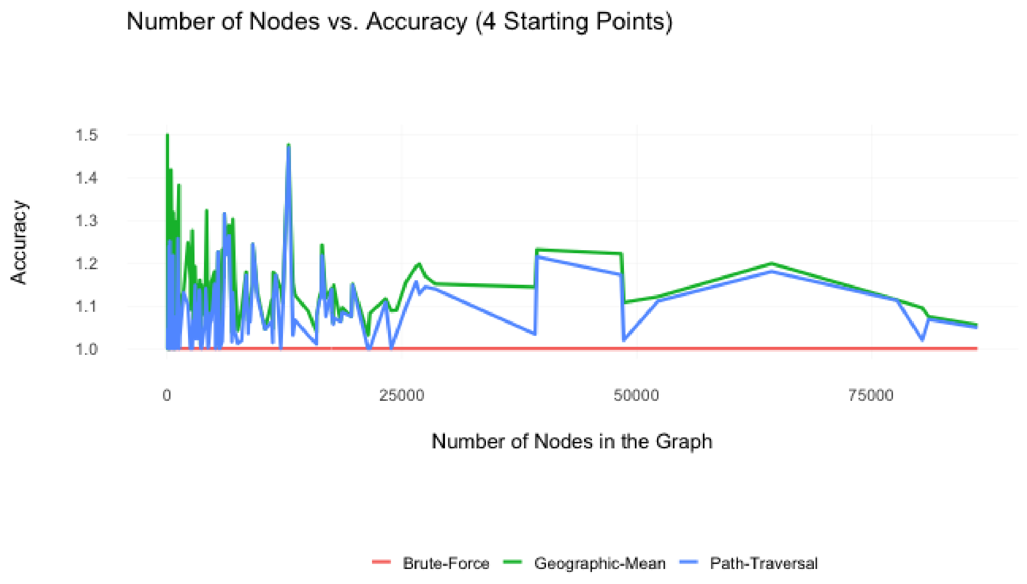


Figure 4.14: Algorithm accuracy comparison: 4 starting points

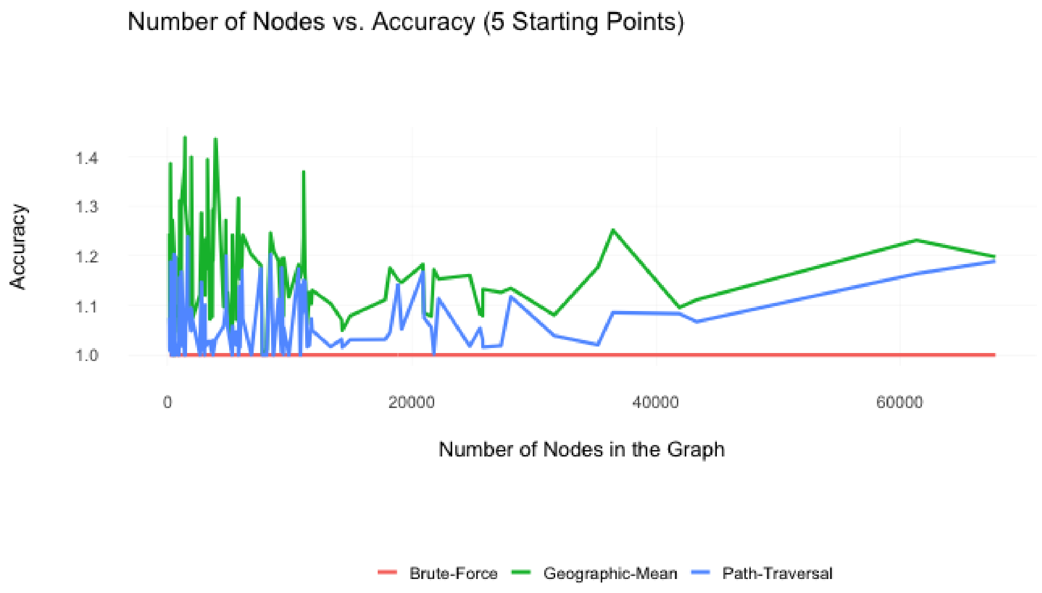


Figure 4.15: Algorithm accuracy comparison: 5 starting points

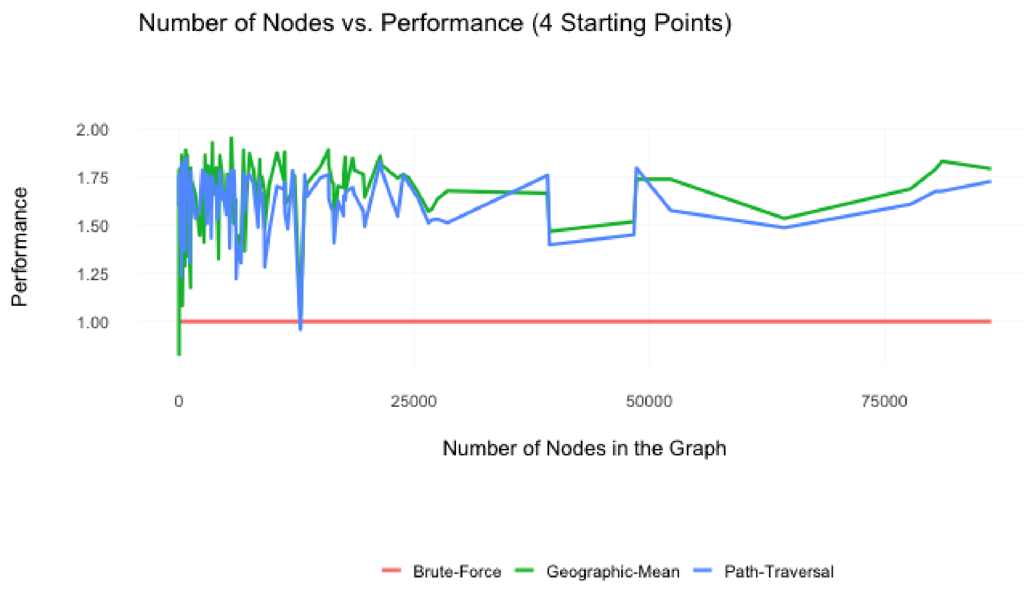


Figure 4.16: Algorithm performance comparison: 4 starting points

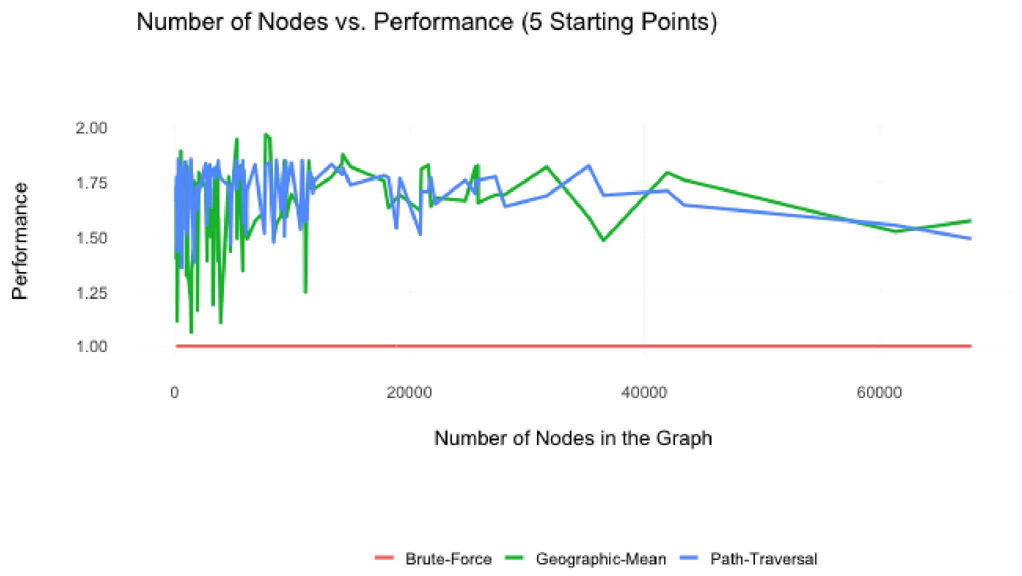


Figure 4.17: Algorithm performance comparison: 5 starting points

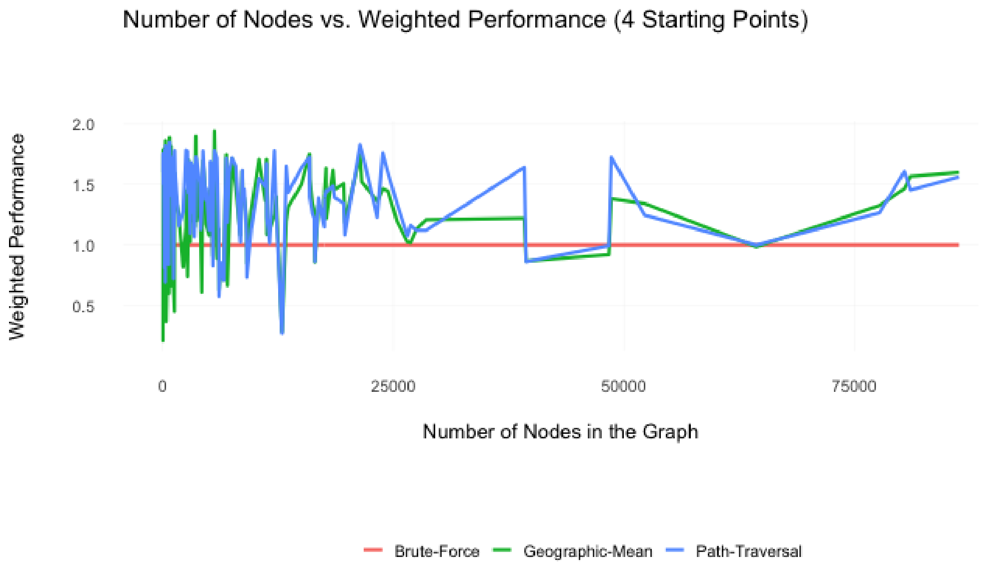


Figure 4.18: Algorithm performance comparison with additional weight on accuracy: 4 starting points

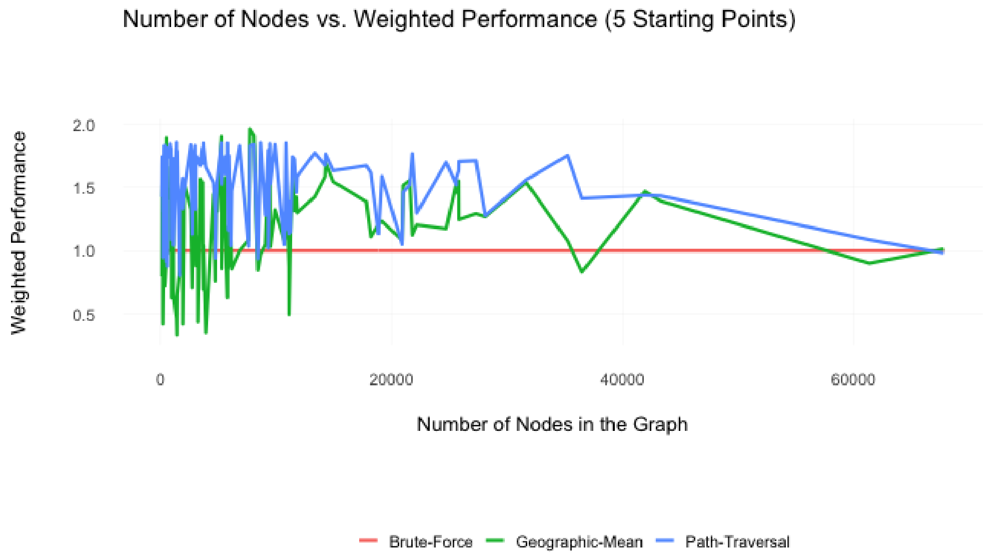


Figure 4.19: Algorithm performance comparison with additional weight on accuracy: 5 starting points

direction. Thus, with an even number of starting points the path traversal algorithm is more likely to determine the geographic mean point as the meeting point, which decreases the accuracy of the algorithm. We can also see that with five starting points, path traversal outperforms geographic mean more than it did before. This indicates that the geographic mean algorithm likely becomes less accurate as the number of initial locations increases, although further testing is needed to confirm this hypothesis. Overall, there is no runaway best choice among the three algorithms we observe. However, given that it appears to become better than geographic mean as the number of starting locations increases, the path traversal algorithm is likely the best choice if one algorithm must be chosen. Fortunately, integrating multiple algorithms into the web application is possible, so we can allow users to choose whether to prioritize speed or accuracy.

CHAPTER 5

MIDWAY: MEETING PLACE FINDER

5.1 IMPLEMENTATION

This section discusses the process of developing the web application, Midway: Meeting Place Finder. In particular, it highlights the setup of the development environment and the development of the back-end of the application.

5.1.1 CREATING THE DEVELOPMENT ENVIRONMENT

The development for this application is done using the PyCharm Community Edition Integrated Development Environment (IDE), which is chosen for developer familiarity as well as being free to use. Before doing any coding, a Django project needs to be created using the command line. To do this, Django needs to be installed which is done simply using pip: `python -m pip install Django` [13]. Once Django is installed, creating a project is able to be done with a single line in the terminal, once the developer navigates to the directory where the code needs to be stored: `django-admin startproject {project name}`. This does an initial set-up of a Django project, creating several files that provide necessary functionality for a Django application. However, this does not yet allow the developer to do very much; in order to start building something useful a Django app needs to be created within the project. This structure is used so that Django can allow developers to include multiple apps within a single project. Again, creating an app is a simple, one-line process: `python manage.py startapp {app name}`. The `manage.py` file is created by the `startproject` command and is used for all project management tasks such as creating apps and running the development server.

Once an application is created, the developer can get started with the coding part of developing the web app. To do this, necessary packages need to be installed. For most packages, this is a simple process using pip or done through the PyCharm IDE. However, the OSMnx package cannot be installed properly using these methods; there is a bug that causes the package to not be installed correctly when using pip. Instead, it must be installed using Anaconda, a Python platform for data science which can also be used to create virtual environments. Anaconda has its own website, where the software can be obtained. Once Anaconda is installed, the terminal must be restarted; then, a virtual environment may be created. Creating an environment that supports OSMnx takes two lines in the terminal:

```
conda config --prepend channels conda-forge
conda create -n ox --strict-channel-priority osmnx.
```

This creates a virtual environment called `ox` that supports OSMnx. All other packages can be installed easily using `conda install {package name}`. This environment can be activated using `conda activate ox`. At this point, the program can be run like any other Python program and the virtual environment provides it access to necessary packages.

5.1.2 Working with Django

When Django creates an application, it creates several files which serve as the core of the application; a few of these files are in the outer, project directory, and the others are in the application directory, which is contained in the project directory. In the project directory, the only changes that need to be made are small adjustments to the `settings` and `URL` files. The `settings` file contains information that Django uses to configure the website. Django initializes this file with default settings, so there are not many changes that need to be made. For this project, the changes made to the settings file helped to set up logging, form templates, and email. The `urls` file determines what URLs to use on the website; in the project directory this only specifies the path to the URLs of a specific application. Since this project only includes a single application, the only URL is an empty URL that directs Django to the URLs of our application.

The application directory is where most of the functionality of the website is created. This directory also contains a `urls` file where we specify all of the URLs for our application. Django makes this simple by creating a function called `path` that allows the developer to pass the URL, a *view* for the URL, and a name. Views are what actually dictate what happens in the application. When the URL is entered, the view given by the path dictates a function that will be called whenever that URL is accessed. These views are created in their own Python file named `views`. All view functions take a parameter called `request`, which is an instance of Django's `HttpRequest` class. When a page is requested, Django creates an instance of this class that contains metadata storing information such the path or method of the request [13].

Depending on the purpose of the view, it can be as simple as one line of code or much more complex. For example, this application has an `about` page, which is just a static webpage with some information on the website. The function for this view is trivial as it requires just a single line of code, as seen in Listing 5.1. All it needs to do is return a call to the `render` function, which takes the request and a filename and displays the given HTML file.

```python
def about(request):
  return render(request, 'about.html')
```

Listing 5.1: "About" view

Our application has three view functions that are not trivial: `index`, `results`, and `contact`. The index view is the front page of the website, and its view function needs to handle the form for users to enter locations. There are two cases that need to be handled; the first is the case where the user submits the form, and the other where the blank form is being displayed. When the user submits the form, the request has a method attribute with the value `POST`, so we can check for this case using an if statement. In this case, the data from the form needs to be collected, then checked to ensure it is valid. Once the data is validated, the view redirects to the results view, which handles what to do with the data. If the view needs to display the blank form, it creates an instance of the `Form` class, which is discussed later in this section, and returns a call to the `render` function, passing it the HTML file for the view and the necessary form. The view also needs to check if there are any errors because, if any error occurs when trying to find a meeting place, the user will be redirected back to this view. To do this, we check to see if there is a value in `request.session['error']`. The `request.session` object is a dictionary that gets passed along with requests that allows variables to be used across views. This is how the locations are made accessible to the results view, and it is also how error messages are made accessible to the index view. If there is no key `'error'` in this dictionary, the view returns the blank form with no error message. Otherwise, a blank form is returned and an error message is displayed at the top of the page. This view function, omitting import statements, is shown in Listing 5.2.

The results view handles what the user sees once they submit their locations. It is responsible for calling the function that runs the meeting-place-finding algorithm as well as compiling the names and coordinates of all relevant locations and passing

```python
def index(request):
  LocationFormSet = formset_factory(SingleLocationForm,
                                    formset=BaseLocationsFormsSet)

  if request.method == 'POST':
    enter_locations_form = EnterLocationsForm(request.POST)
    location_formset  = LocationFormSet(request.POST)
    if enter_locations_form.is_valid() and location_formset.is_valid():
      locations = [address['location'] for address in location_formset.cleaned_data]
      request.session['location_input'] = locations

      return HttpResponseRedirect('/results', {'locations': locations})
  else:
    enter_locations_form = EnterLocationsForm()
    location_formset = LocationFormSet()
  if 'error' in request.session:
    return render(request,
                  'mpf/index.html',
                  {'enter_locations_form': enter_locations_form,
                   'location_formset': location_formset,
                   'error': request.session['error']})
  else:
    return render(request,
                  'index.html',
                  {'enter_locations_form': enter_locations_form,
                   'location_formset': location_formset})
```

Listing 5.2: “Index” view

them to the HTML page. First, it goes through all of the user-given locations and attempts to find their geographic coordinates; if it is unable to do so then it redirects the user back to the home page with an error message telling them which address was invalid. Then, it runs the meeting-place-finding algorithm. Once that returns, it does a reverse geocode on the coordinates of the meeting place in order to give the user an address. Once all of this information is found, the view displays a page with a map that has markers at each of the user-given locations as well as the meeting place. These markers each have labels with their address, and are created using JavaScript. The code for this view is shown in Listing 5.3.

5.1.3 Algorithm Implementation

While pseudocode for Dijkstra’s algorithm and the meeting-place finding algorithms were provided in previous chapters, there were several challenges to actually implementing them within the application. The first of these challenges was to determine how to weight the edges in order to maximize the effectiveness of

```python
def results(request):

  locations = request.session['location_input']

      locator = geopy.geocoders.MapBox(mapbox_access_token)
        location_geocodes = []
        for location in locations:
            try:
                location_code = locator.geocode(location, timeout=60)
            except exc.GeocoderTimedOut:
                try:
                    location_code = locator.geocode(location, timeout=60)
                except exc.GeocoderTimedOut as e:
                    request.session['error'] = e.message
                    return HttpResponseRedirect('/')
            if location_code is None:
                logger.warning("Invalid location specified: {}".format(location))
                request.session['error'] = "Invalid location: {}".format(location)
                return HttpResponseRedirect('/')

            location_geocodes.append(location_code[1])

        try:
            meeting_place = find_meeting_place(location_geocodes)
        except InvalidLocationError as e:
            return render(request,
                          'mpf/index.html',
                          {'form': EnterLocationsForm(),
                           'error': e.message})

        location_geocodes.append((meeting_place['y'], meeting_place['x']))

        meeting_place_text = locator.reverse((meeting_place['y'], meeting_place['x'])
 ↪ ).address
        locations.append(meeting_place_text)

        mapbox_private_token = # This is associated with your account with the mapbox
 ↪ api

        return render(request,
                      'mpf/results.html',
                      {'mapbox_private_token': mapbox_private_token,
                       'location_list': locations,
                       'location_geocodes': location_geocodes})
```

Listing 5.3: "Results" view

Road Type	Default Speed Limit (mph)
Motorway	55
Trunk	55
Primary	45
Secondary	45
Tertiary	35
Unclassified	30
Residential	25
Motorway Link	45
Trunk Link	45
Primary Link	35
Secondary Link	35
Tertiary Link	25
Living Street	15

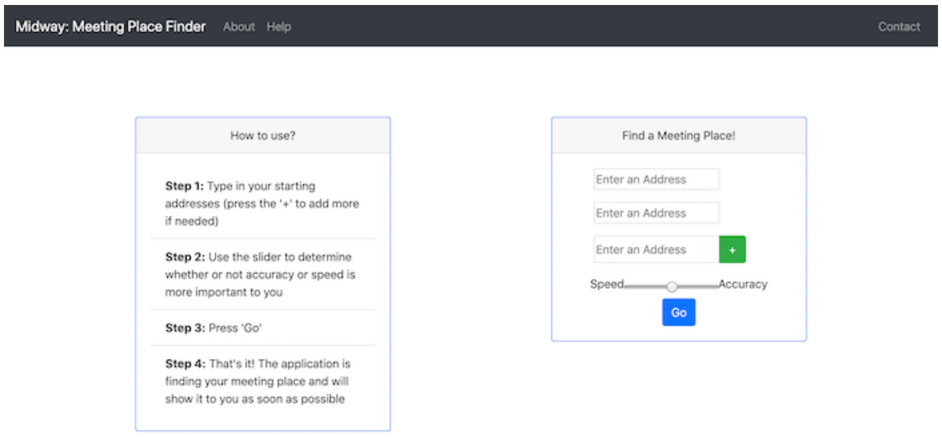
Table 5.1: Table of default speed limits

the application. Our objective is to minimize total driving time, but the edges provided by OSMnx do not contain an approximated time to drive along an edge. Conceptually, the estimation of driving time is simple; multiplying the length of the road segment and the speed the car is traveling and converting to the proper units gives us a good approximation of the amount of time it takes to drive along a particular road. Luckily, OSMnx has data for the length of every edge in the graph, which can be easily accessed and subsequently used for calculations. However, estimating the speed of travel along the edge is not quite as simple. The easiest way to estimate the speed is to assume that the car is traveling at exactly the speed limit. The problem with this is that speed limit data is not available for every edge. To combat this problem, we take advantage of the fact that OSMnx does define an edge type for every edge. Additionally, there are descriptions online as to what type of road these edge types refer to. For example, we assume that a *motorway* has a speed limit of 55 miles per hour, and a *residential* has a speed limit of 25. A complete list of road types and descriptions can be found at `https://wiki.openstreetmap.org/wiki/Key:highway` and a table of the road types used for reference within the application can be found in Table 5.1. In cases where the road type is not one of the ones listed, the speed limit is assumed to be 25 miles per hour. This was chosen because roads with higher speed limits are typically more significant and thus are more likely to already have a defined road type. Using this information, it is possible to write a function that assigns speed limits based on edge types to each edge that does not already have one. This solution is implemented by writing a larger function that estimates the travel time of any given edge and passing every edge to this function.

5.2 Results

The implementation described succeeds in building a functioning web application. This application accomplishes the goals of the project, with its ability to successfully find meeting places between 3+ locations and display these results in a manor that is easy for the user to understand. In cases where the distances between initial locations are small, the application displays results in approximately 2-3 seconds. For larger test cases, it could take up to 30 seconds to find a meeting point, and if meeting places are too far apart, the application may not be able to find a meeting place. "Too far apart" refers to cases where initial locations are several states apart, in which case the amount of data required to find a meeting place takes too long to download.

Now, we go through an example usage of this application step-by-step to demonstrate its capability. Note that this application is not deployed on the internet, and this example is using a server hosted on a personal computer and is only accessible via this computer. The first thing a user sees is the index page, which is shown in Figure 5.1; this page corresponds to the code from Listing 5.2. This page includes instructions for the user, and the interface is fairly self-explanatory by design. Following the instructions given, users enter their starting locations,



Step 1: Type in your starting addresses (press the '+' to add more if needed)

Step 2: Use the slider to determine whether or not accuracy or speed is more important to you

Step 3: Press 'Go'

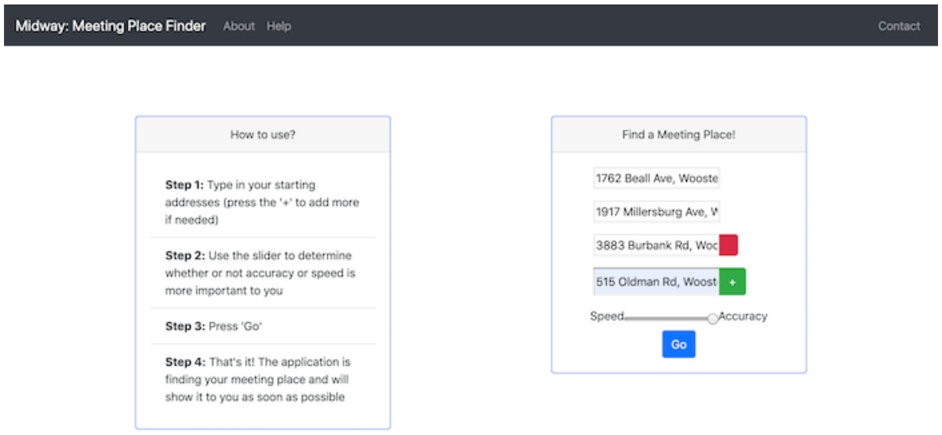
Step 4: That's it! The application is finding your meeting place and will show it to you as soon as possible

Figure 5.1: Initial page of Midway: Meeting Place Finder

using the + button to add additional locations as needed. They also use a slider to indicate how much they want to prioritize speed versus accuracy. Depending on the user's selection, the application uses either the brute force, geographic mean, or path traversal algorithm. An example of a user input is in Figure 5.2. In this example the brute force algorithm is used to find the meeting point because the user chose to put the maximum possible weight on accuracy. The full addresses in the input are: 1761 Beall Ave, Wooster, Ohio, 1917 Millersburg Rd, Wooster, Ohio, 3883 Burbank Rd, Wooster, Ohio, and 515 Oldman Rd, Wooster, Ohio.

At this point, the user presses the blue *Go* button, which initiates the algorithm and sends the user to the results page. This results page is dictated by the code from Listing 5.3 and can be seen in Figure 5.3. Each of the user-given locations is indicated by a dark blue marker, while the meeting location has a green marker. Additionally, the address of the meeting place is written out for the user, and the address of each location can be seen by clicking on its marker. An example of this feature can be seen in Figure 5.4.

In addition to this primary functionality, the application has a few other noteworthy features. First is the *About* page, which is a static page including some



How to use?

Step 1: Type in your starting addresses (press the '+' to add more if needed)

Step 2: Use the slider to determine whether or not accuracy or speed is more important to you

Step 3: Press 'Go'

Step 4: That's it! The application is finding your meeting place and will show it to you as soon as possible

Find a Meeting Place!

1762 Beall Ave, Wooste

1917 Millersburg Ave, W

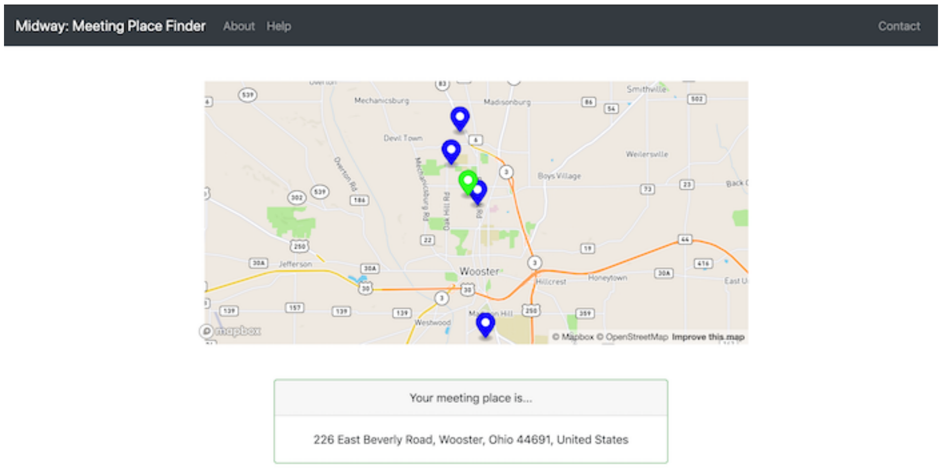
3883 Burbank Rd, Woo

515 Oldman Rd, Woost

Speed — Accuracy

Go

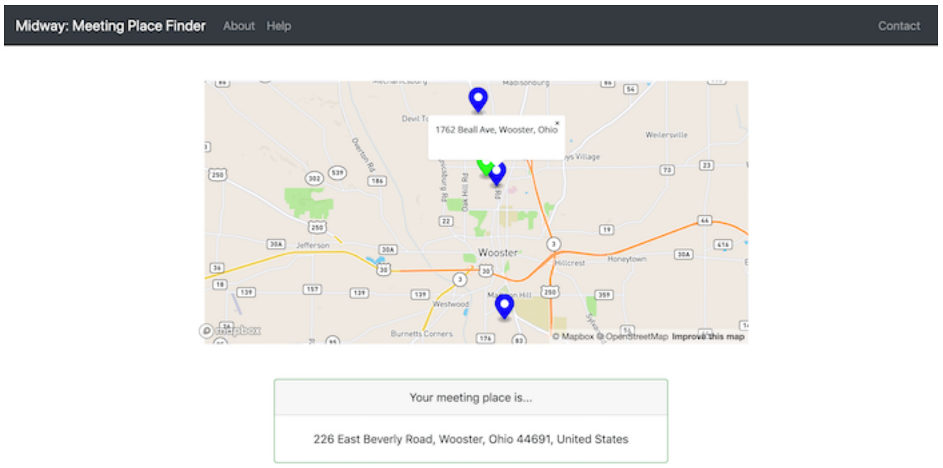
Figure 5.2: Example user input



Your meeting place is...

226 East Beverly Road, Wooster, Ohio 44691, United States

Figure 5.3: Results page

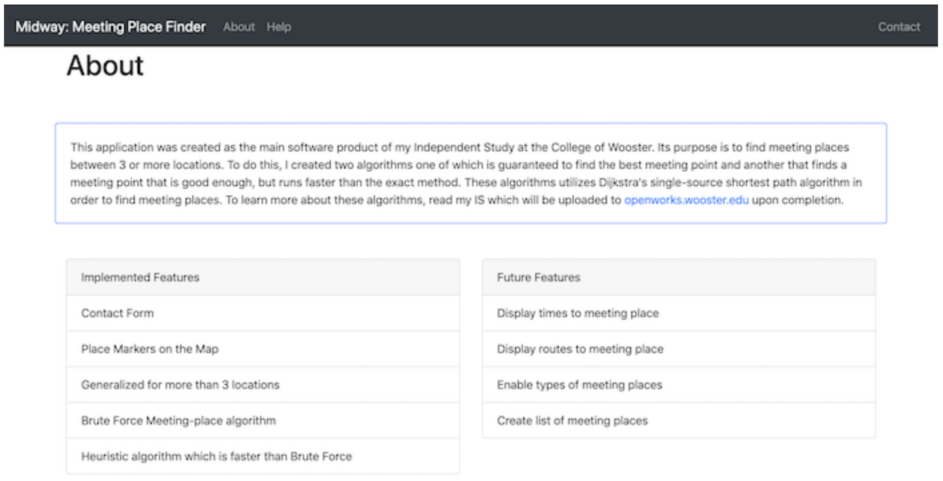


Your meeting place is...

226 East Beverly Road, Wooster, Ohio 44691, United States

Figure 5.4: Results page with marker clicked

additional information about the app. This page can be seen in Figure 5.5. There is also a *Contact* page, which allows users to send emails with questions or feature requests via the app. Figure 5.6 shows a screenshot of the contact form. Last, there is a *Help* page, which is used as an FAQ page to assist users if they encounter any common problems; this can be seen in Figure 5.7.



About

This application was created as the main software product of my Independent Study at the College of Wooster. Its purpose is to find meeting places between 3 or more locations. To do this, I created two algorithms one of which is guaranteed to find the best meeting point and another that finds a meeting point that is good enough, but runs faster than the exact method. These algorithms utilizes Dijkstra's single-source shortest path algorithm in order to find meeting places. To learn more about these algorithms, read my IS which will be uploaded to openworks.wooster.edu upon completion.

Implemented Features
Contact Form
Place Markers on the Map
Generalized for more than 3 locations
Brute Force Meeting-place algorithm
Heuristic algorithm which is faster than Brute Force

Future Features
Display times to meeting place
Display routes to meeting place
Enable types of meeting places
Create list of meeting places

Figure 5.5: About page

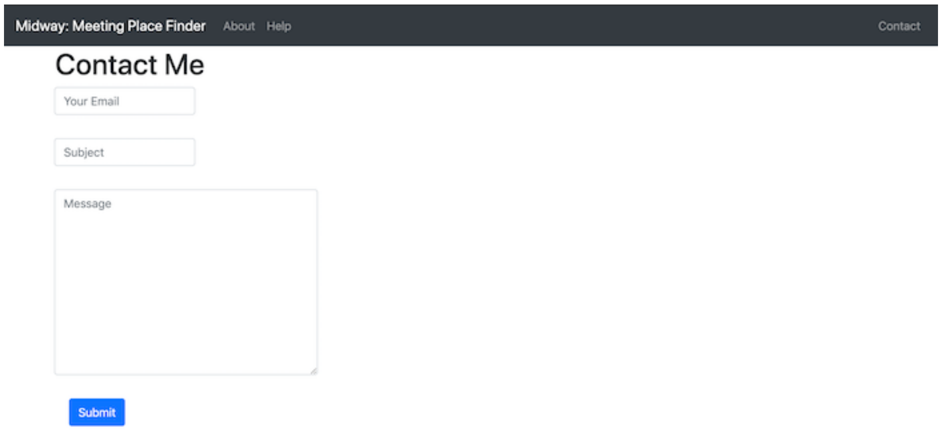
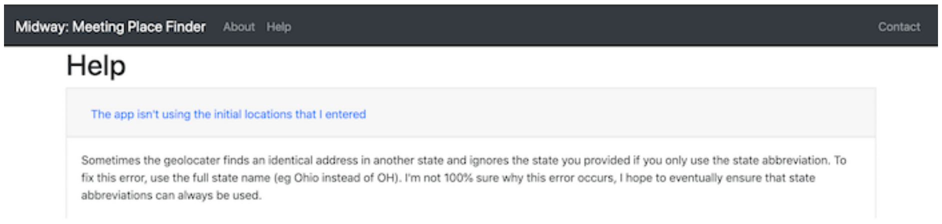


Figure 5.6: Contact page



Help

The app isn't using the initial locations that I entered

Sometimes the geolocater finds an identical address in another state and ignores the state you provided if you only use the state abbreviation. To fix this error, use the full state name (eg Ohio instead of OH). I'm not 100% sure why this error occurs, I hope to eventually ensure that state abbreviations can always be used.

Figure 5.7: Help page

CHAPTER 6

Conclusions and Future Work

This project resulted in the successful implementation of Midway: Meeting Place Finder, a web application capable of solving the problem of finding a meeting place between three or more user-given locations. Within the timeframe available we accomplished a set of minimal goals necessary for success. While this thesis ends with a positive result, there are also many ways in which it could be improved or extended. To complete our discussion, we review the necessary goals which were accomplished in addition to areas where further research can be done.

6.1 Completed Goals

The following goals were requirements for a successful application, all of which were completed:

- The application is capable of finding meeting locations between at least three locations specified by the user.
- The user interface is clean and simple, making it easy for the user to interact with the application.
- All locations relevant to the user's search are able to be displayed on a map.

Clearly, the ability to find meeting locations is the minimal requirement for the application, because this functionality is the purpose driving the development of the application. However, if this functionality is not implemented with a clean user interface then it is rendered useless, thus necessitating a quality and easy-to-use interface. Given that the application deals with location and map data, it is only logical that a user would like to view the locations on a map. While this feature is not technically necessary for a functioning application, it serves to help users better contextualize the meeting place specified by the program.

In addition to these primary goals, Midway: Meeting Place Finder successfully implements the secondary objective of generalizing for use with an arbitrary number of input locations. Naturally, this is beneficial because the ability to handle more input locations creates additional use cases for the application which is beneficial to users. When beginning development the difficulty of generalizing was unknown; however, the algorithm does not require any changes for generalization, and the most difficult step towards generalizing is receiving a variable number of inputs from the front-end of the web application.

6.2 Future Work

6.2.1 Optimizations

Future optimizations will center around determining ways to speed up the process of creating the graphs necessary for running the algorithm. The algorithm created is capable of finding a meeting place in under a minute even on test cases where the input locations are hundreds of miles apart. However, this is overshadowed by the fact that creating a graph can take minutes. Ideally, the solution to this problem is to find a way to download the graph of the entire continental United States. This proved to be too much data to download during this project. In the future, finding a way to download this data, or retrieve the data faster than the application currently does, would significantly improve the performance of the application.

In addition to reducing the amount of time it takes to create a graph, downloading the data and storing it locally would enable the implementation of a contraction hierarchy as described in Chapter 4. After completing the preprocessing required to create the contraction hierarchy, it is likely that it would enable the algorithm to run in an amount of time on the order of a few milliseconds. This would be incredibly beneficial to the app because users have come to expect quick response times, which the app currently does not provide in all scenarios.

Another optimization worth considering is implementing the geographic mean neighbor walk and midpoint intersection algorithms using the A* search algorithm. When using Dijkstra's algorithm, these two heuristics are rendered useless because they are slower than the brute force algorithm. If a good enough heuristic could be found, the A* search algorithm could enable these two algorithms to become useful in approximating a meeting place.

6.2.2 Features

While Midway: Meeting Place Finder is capable of solving the problem at hand, there are still several ways it could be improved in the future. Some potential ideas for new features are:

- Providing users with estimated times to the meeting point
- Allowing users to specify a type of meeting place
- Listing multiple potential meeting places for the user to choose from
- Enabling users to export directions to a meeting place
- Including alternative forms of transport such as walking and bicycling.

The ability to provide estimated times is helpful to the user because it is an additional piece of information that can help them make plans more effectively.

Specifying a type of meeting place means allowing users to choose to meet at specific locations such as restaurants or gas stations. Currently, if a user wants to meet at a specific type of place, they need to manually search for places around the meeting place given by the app. Listing different meeting places is helpful because it provides users with multiple options that they can easily search through to find the one that best suits their needs. Creating a way for users to export directions is helpful because if they want to go to the meeting place they need to find out how to get there, and it is convenient for them to do it through this application instead of going to another one. Alternative forms of transport can be useful for people who intend to use them; this feature serves to enable additional use cases which makes the app more useful for everybody.

In order to allow users to specify types of meeting places, additional constrains need to be programmed into the algorithm to ensure that the meeting place selected is of the proper type. Additionally, this requires the dataset to include information on the types of places. Listing multiple meeting places requires searching the map for additional nodes within some sort of radius of the meeting place found by the original algorithm. This likely would not be particularly difficult, although it would be important to ensure that the application supports this while still maintaining a clean and easy-to-use interface. Exporting directions would likely be an easier extension, as the directions should be found in the process of finding the meeting place itself. Then, we only need to translate the directions into a human-readable format. Alternative forms of transport would require a shift in datasets being used. The same algorithm will be usable across transportation methods, but the actual paths that are available would be different. It is unknown how difficult it is be to handle different modes of transport within the application.

As a whole, these features are non-essential for a functioning application, which is why they are not implemented yet. However, they all add additional benefits to users and are worth implementing in the future.

References

1. 2019 State of Software Engineers. URL `https://hired.com/page/state-of-software-engineers/hottest-coding-languages/#most-loved-languages`.

2. 25 of the Most Popular Python and Django Websites. `https://www.shuup.com/django/25-of-the-most-popular-python-and-django-websites/`, Nov 2019.

3. Michael John. Apter. *The Computer Simulation of Behaviour*. Hutchinson, 1970.

4. Geoff Boeing. OSMnx: A Python package to work with graph-theoretic OpenStreetMap street networks. *The Journal of Open Source Software*, 2(12):215, 2017. doi: 10.21105/joss.00215. URL `https://www.researchgate.net/publication/309738462_OSMnx_New_Methods_for_Acquiring_Constructing_Analyzing_and_Visualizing_Complex_Street_Networks`.

5. Geoff Boeing. OSMnx: Python for street networks. *Geoff Boeing*, May 2018. URL `https://geoffboeing.com/2016/11/osmnx-python-street-networks/`.

6. Andrej Brodnik and Marko Grgurovič. *Practical Algorithms for the All-Pairs Shortest Path Problem*, pages 163–180. Springer International Publishing, Cham, 2018. ISBN 978-3-319-77510-4. doi: 10.1007/978-3-319-77510-4_6. URL `"https://doi.org/10.1007/978-3-319-77510-4_6"`.

7. Gary Chartrand and Ping Zhang. *A First Course in Graph Theory*. Dover, 2012.

8. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

9. Reinhard Diestel. *Graph theory*. Springer, 2018.

10. Kayhan Erciyes. *Guide to Graph Algorithms: Sequential, Parallel and Distributed*. Springer Nature, 2019.

11. Dominique Feillet, Pierre Dejax, Michel Gendreau, and Cyrille Gueguen. An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems, Aug 2004. URL `https://onlinelibrary.wiley.com/doi/epdf/10.1002/net.20033`.

12. Geopy. URL https://geopy.readthedocs.io/en/stable/#.

13. Getting Started With Django. https://www.djangoproject.com/start/.

14. HTML basics. URL https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics.

15. JavaScript Documentation. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction.

16. Jake Kronika and Aidas Bendoraitis. *Django 2 web development cookbook: 100 practical recipes on building scalable Python web apps with Django 2*. Packt Publishing, 2018.

17. Nacima Labadie. *Metaheuristics for Vehicle Routing Problems*. Wiley, 2016.

18. Jiri Lebl. *Basic Analysis I: Introduction to Real Analysis, Volume I*. CREATESPACE, 2018.

19. Michael Levin. Contraction hierarchies. URL https://www.coursera.org/learn/algorithms-on-graphs/lecture/HV35U/highway-hierarchies-and-node-importance.

20. NetworkX Documention - Overview. URL https://networkx.github.io/documentation/networkx-1.10/overview.html.

21. Mark Otto and Jacob Thornton. Bootstrap documentation. URL https://getbootstrap.com/docs/4.3/about/overview/.

22. Yana Petlovana. Top 13 Python web frameworks to learn in 2020, Oct 2019. URL https://steelkiwi.com/blog/top-10-python-web-frameworks-to-learn/.

23. G. Ramalingam and Thomas Reps. An incremental algorithm for a generalization of the shortest-path problem, May 1996. URL https://www.sciencedirect.com/science/article/pii/S0196677496900462.

24. Arun Ravindran. *Django design patterns and best practices: industry-standard web development techniques and solutions using Python*. Packt Publishing, 2018.

25. Fred S Roberts and Barry Tesman. Applied Combinatorics. Mar 2009. doi: 10.1201/b12335.

26. Alexander Ryabtsev. Web frameworks: How to get started, Oct 2019. URL https://djangostars.com/blog/what-is-a-web-framework/.

27. Lynn Arthur. Steen and J. Arthur Jr. Seebach. *Counterexamples in Topology*. Holt, Rinehart, and Winston, 1970.

28. Michael Tandy. Contraction hierarchies path finding algorithm, illustrated using three.js, 2015. URL `https://www.mjt.me.uk/posts/contraction-hierarchies/`.

29. Paragyte Technologies. How good is Python for web development? - an analysis, Jun 2017. URL `https://medium.com/@paragyte2/how-good-is-python-for-web-development-an-analysis-41a5cb4e88fc`.

30. What is CSS? URL `https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS`.