The Optimal Route for Assurance Delivery

Sandy Auttelet

Executive Summary

This report is a presentation of my solution to the optimal route problem I was provided by your company, Assurance Delivery. Not only have I solved the optimal route problem for the provided data describing distance and delay probability, but I have coded an application for more general use by your company. I have built a rudimentary graphical user interface that can later be developed into a web application for your drivers to utilize, so they can find the optimal route on the go. I have also done some analysis for future work on this application in the event you desire to upgrade its performance capabilities. Because we are interested in both minimizing driving distance as well as probability of delay, I have introduced some adjustable parameters for users to input into the application to better suit the specific situation. These parameters include a delay importance value. For routes starting with city A, I recommend you set this value to 1500 based on the data provided. I have also introduced a maximum tolerance for each data set like time of delivery and max risk willing to acquire. I am writing my computational solution using Python and utilizing a module called Networkx. With the shortest path algorithm found in this module, I have computed solutions for the optimal path based on either distance or delay. For example, to minimize the distance traveled from city A to city G, the optimal route is $A \to I \to G$ with a total distance traveled of 70.0 units with 3.09% probability of delay. In the same example, the optimal path to minimize delay probability is $A \to H \to C \to G$ with a total probability of delay of 1.44% traveling 152.0 units as I reported in my latest status report. I have now successfully implemented the weighting parameter, and found an intermediate path, $A \to H \to G$ with a distance traveled of 91.0 units, and a probability of delay of 1.66%, which is more optimal considering both parameters.

Contents

List of Figures

3 Here you can see all possible, optimal paths for the route from $A \to K$. We start at city A colored pink, travel to the intermediate green colored cities, and end at the city K colored cyan. All cities not traveled to are colored red. 12

1 Introduction

Route optimization is crucial for any delivery company as it minimizes drive time, ultimately saving money on labor and fuel. Because you have the money-back-guarantee for orders delivered late, mitigating delay is especially important for your business model when factoring in the financial risk associated with each delivery. It was my goal to analyze the provided data to optimize your current routes for your startup company. In addition to this task, I also built you a general computational framework for ensuring optimal delivery for any set of data with two parameters like distance and delay probability. This addition means your company is able to add more cities to your route and obtain similar results to what I derived from this data set. As part of this general approach to the problem, I built a graphical user interface (GUI) to enable employees to find optimal routes on the go, providing more applications as your company grows.

To provide a deeper analysis of the data, I have added some input parameters to the system. Because we want to minimize both distance traveled and probability of delay, I've created an input parameter called delay importance that you may change based on the financial risk of an order being late. This parameter should be increased if you want to increase the importance of minimizing delay. I suggest you increase this value for high dollar orders as they will cost your company more when arrived late than low dollar orders. More about mitigating financial risk will be discussed in section 6 below and I will have more concrete recommendations in subsection 7.1. For this assessment, I have also added a cost of order parameter in the input data.

I have also introduced parameters like max time willing to travel, for the events where your drivers must be off at a certain time and thus cannot take long routes, or to mitigate late deliveries. I implemented a max risk willing to occur input in case you want to ensure financial risk is below a certain threshold. Additional parameters may be introduced as you develop this software further, and I will make some suggests for those in section 7.2 as well. You can find a full analysis of the provided data in subsection 5.2 as well as results for the current routes your company is taking in subsection 5.1. To provide an accurate financial risk assessment, I also gathered some demographic data to measure risk in dollars, found in section 6. Because this demographic data is highly volatile, I have implemented some code to make this adjustable to assess risk moving forward.

2 Methodology Overview

I wrote my computational framework and GUI in Python. I decided to model your problem like an undirected, irregular graph, thus utilized a module within Python called NetworkX. Graph based modeling has become increasingly popular due to its speed and adaptability. We begin the model by defining each city as a node. A node in a graph has connections, like intercity highways, called edges. Each edge is associated with a weight. The weights in our case are either the distance traveled, delay probability, or a weighted sum combining both data sets. Our problem relates to an undirected graph because your distance and delay probability do not depend on direction. This symmetry qualifies your graph as undirected.

The irregularity in your graph appears because some cities like city A have four connections, while other cities like city D have six connections. The number of connections for each city is called the degree, and the degree is not the same for all cities in your route, thus making the graphs irregular.

We are looking to minimize two separate graphs with different weight values, distance and probability of delay. Therefore, our weight, denoted by w_{ij}, must be some value that takes into account both variables. Because the distances traveled are much larger than the probability of delay, if we want the probability of delay to matter in our optimal path calculation, we must scale the delay probability up by multiplying by some number, call it u. Then u can increase the importance of factoring in the delay probability. We might care about delay probability more for an order with more financial risk. Since your distance data is not given to me with units, I have assume the unit of distance is the standard mile.

Suppose you have one order that would cost \$1000 to refund, and one order that costs only \$10 to refund. It is more important to avoid a delay for the first order, so u should be large to mitigate occurrence of delay that would require a \$1000 refund. However, if the distance traveled when factoring in the importance of delay differs from the distance traveled without factoring in delay is more than \$10 in fuel and labor costs, then it is not worth it to worry about the delay and thus we should set u to be 0. The financial risk assessment is discussed further below, but for now, we understand $u \in \mathbb{R}^+$, i.e. some non-negative number to make the delay probability at least as important as the distance traveled. If we denote the distance traveled from city to city as d_{ij} and the delay probability as p_{ij}, then our new weight for each edge becomes $w_{ij} = d_{ij} + u \cdot p_{ij}$. The u value is inputted in the GUI and named "Delay Importance".

I then found the shortest path utilizing the weight, w_{ij} for each edge. This path was found using the shortest path function found in NetworkX. This function utilizes an algorithm called dijkstra algorithm and is discussed further in section 4. I also used the all pairs dijkstra path function to find all paths from starting city A to all other cities in the route. Using these paths, I found a value for the delay importance that will take into account delay probability for all deliveries made when traveling out of A. Lastly, I included a plotting tool that will help the user visualize the route, color coding the starting city, ending city and all intermediate cities traveled to in the optimal path.

3 Data Collection and Preprocessing

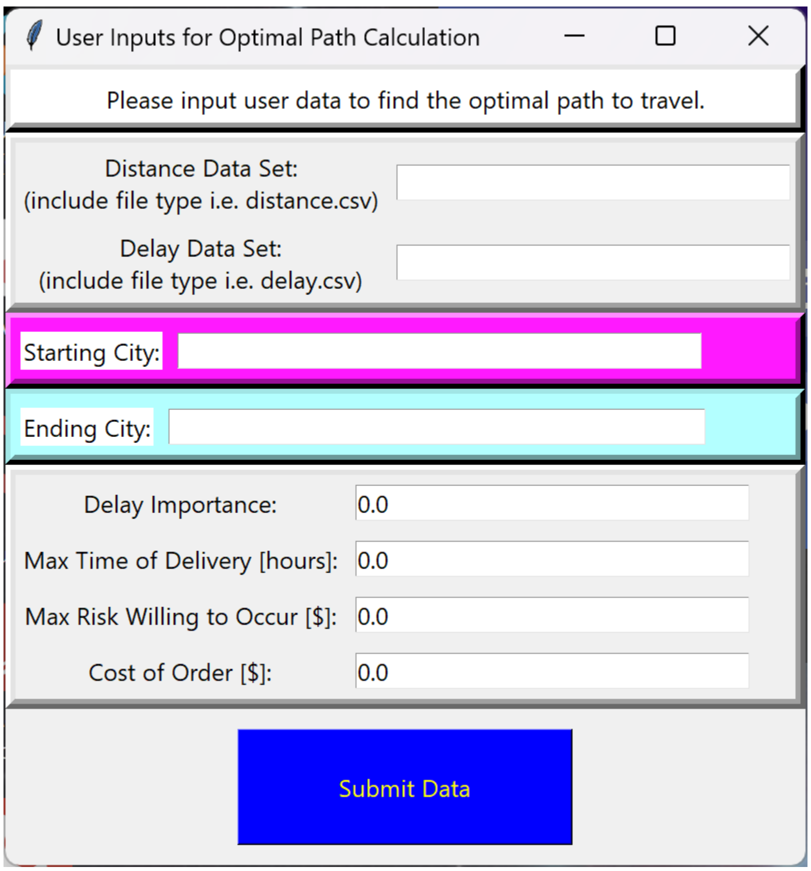
3.1 Computational Usage and Warnings

From the given data, the distance and delay probability must have the same edges and nodes to enable the weight w_{ij} to be found. For this reason, I have implemented a similarity test within the computation to ensure before attempting to build the weighted graph, we establish that both graphs have the same structure. If their structure is different, you will see an error and the rest of the computation will not run successfully. Because we are weighting our graph by only two different parameters, distance and delay, I have created a file called graph-2D in the computation that runs all functions except the GUI, plotting and tests.

The main file will also need to be run to execute any computations. At the end of the main file, there are print functions to execute a printed report and plotting from the provided inputs.

It is crucial when using the GUI that all files are saved in the same location and you have checked your directory appropriately. If you choose not to use the GUI, simply comment out the import from the main file and input your own default parameters. I have also created a tests file based on the example you provided me in the problem statement. If any of these tests fail, you should see an error message printed with a suggestion of where to look for the failure. These tests may fail if the code is manipulated in any way. I have not built tests for the GUI and thus it is a good idea to input default data manually if you suspect an error in input. If an input must be a number, you will see 0.0 in the GUI's input box by default. If the box is empty, the GUI is expecting words as input as shown in the image below.

Figure 1: Graphical User Interface Empty Demonstration



This GUI is easily modulated to add more input parameters, or find the optimal path results from other data, thus not restricted to the sets I was provided. Once data has been submitted, a report will be printed to the terminal and plots will be displayed depicting the optimal route to travel based on input parameters.

3.2 Input Parameters

From the load-data function within the GUI file, there are some items saved that are used within the computation and printed report. I will be explicit here what these items do and why they are important for the analysis below.

$$\text{file1-name} := \text{The first input file's explicit name including file type.} \tag{1}$$
$$\text{file2-name} := \text{The second input file's explicit name including file type.} \tag{2}$$
$$\text{source} := \text{Starting city for the route we are analyzing.} \tag{3}$$
$$\text{sink} := \text{Ending city for the route we are analyzing.} \tag{4}$$
$$\text{weight} := \text{Delay importance for the weighted graph.} \tag{5}$$
$$\text{tol} := \text{A list with two inputs, max time and max risk.} \tag{6}$$
$$\text{order-cost} := \text{Input cost of order, used to assess if a path is optimal.} \tag{7}$$
$$\text{file1-text} := \text{The name of the first input file (i.e. distance).} \tag{8}$$
$$\text{file2-text} := \text{The name of the second input file (i.e. delay).} \tag{9}$$
$$\text{file1-type} := \text{The type of the first input file (i.e. csv).} \tag{10}$$
$$\text{file2-type} := \text{The type of the second input file (i.e. csv).} \tag{11}$$

The first seven objects are used for the computation. The file1-text and file2-text are used for printing the report, while file1-type file2-type are unused but may be needed for building tests for the GUI and thus included in this load-data function return.

4 Shortest Path Optimization

4.1 Computational Framework

The NetworkX module's shortest path function by default utilizes the dijkstra algorithm. The dijkstra algorithm is quite general and thus useful for a variety of problems, with more information and examples found in [1]. Describing this algorithm using what I learned from [1], we begin with a weighted network, like the distance data provided to me to solve this problem. This network is constructed into the undirected, irregular graph, G. This algorithm utilizes a directed graph, so we must choose our starting city to find adequate directional reference points. Here, $G = (V, E, W)$ where V is a set of nodes, E is a set of edges and W is a set of edge weights. To be specific, each $w_{ij} \in W$ is associated with a specific edge $e_{ij} = (i, j) \in E$, where each node is either i or j for $i \neq j$ since it does not take any time to get from city A to city A.

To solve our problem, we must pick a source node, $s \in V$ to be our starting city. In my computation, you can specify the ending city called sink, t, or you can run the parallel problem which will find optimal routes for each ending city. This algorithm has a specific qualifier that $w_{ij} \geq 0\ \forall (i, j) \in E$. Therefore, we cannot have negative distances or negative probabilities of delay. In practical problems such as this, we will not run into this issue.

In algorithm execution, first we assign a flow (or total distance traveled) value, d_ℓ, to each node that is not our source. This value is initially infinity. We then assign a status label, p for permanent or t for temporary, akin to whether or not we have chosen this node to be in our optimal path. For example, our source node is given the label $(0, p)$ since it is 0 distance away from itself and this is where we start. Since we know this node must be in our optimal path, it is labeled p for permanent. All other intermediate nodes are marked with a label (∞, t) so we can check them all to find the optimal path. Our sink node is marked with a label (∞, p) since we know it must be in our path, but we don't know the minimum distance of the path yet. If the flow value is smaller than the current shortest path flow value, then that status label is changed from t to p, meaning the path is replaced as our new shortest path. We iterate through all paths in this way until we find the one with the smallest flow value, d_ℓ.

4.2 Mathematical Model

Let's denote J to be the set of nodes with temporary labels that can be reached from the current node i by an edge, (i, j). Then we want to minimize the new distance value, $d_j = \min\{d_j, d_i + w_{ij}\}$ where w_{ij} is the weight of the edge (i, j) given by the provided data. Looking at all the nodes in J, we can determine which one has the smallest d_j value, call that d_{j^*}, and we choose that one to become a member of our optimal path.

To truly find the smallest path, we are actually looking to minimize the sum of all d_{j^*} values in our path. Therefore, the problem becomes:

$$
\begin{aligned}
\min_{d_\ell} \quad & \sum_{i=1}^{N} \sum_{j=1}^{i} w_{ij} \cdot x_{ij} \\
s.t. \quad & \sum_{j=1}^{N} x_{ij} - 2y_i = 0, \\
& \sum_{j=1}^{N} x_{sj} = 1, \\
& \sum_{j=1}^{N} x_{tj} = 1, \\
& w_{ij} \in \mathbb{R}, \\
& w_{ij} > 0, \\
& x_{ij}, y_i \in \{0, 1\}.
\end{aligned}
$$

Here N denotes the number of nodes in the graph. We introduce the binary variables x_{ij} and y_i to ensure we have a valid path. Here, $x_{ij} = 1$ if edge (i, j) or (j, i) is in the path, and 0 otherwise. We introduce y_i to be a variable denoting that a node is in the path. Then $y_i = 1$ if the node is in an edge we cross in our path, and is 0 when that node is never in our path.

4.3 Example of Algorithm Process

Suppose we were trying to find an optimal path from A to B and found a path that looked like $A \to B \to A \to D \to E \to D \to B$. This path is not optimal because it violates all conditions in our minimization problem. First, we notice the edges in our path are $(A, B), (B, A), (A, D), (D, E), (E, D), (D, B)$. Then for our first condition, $\sum_{j=1}^{N} x_{Dj} - 2y_D = 2$ so we have traveled from D to E then back to D, which is logically not an optimal route as we should have just not gone to E at all. Likewise, the second constraint is not met since A is our source node, s and therefore $\sum_{j=1}^{N} x_{Aj} = 3$, meaning we traveled to our source twice, so we could have just traveled from A to B and stopped, but we kept traveling. We also break the last constraint for the same reason. Therefore, these constraints are introduced to ensure we do not travel to a node more than once, and we can always enter a city from one city, and leave toward a different city.

To find the minimal d_ℓ value for this example, we must approach with the numerical data. Looking at simply the distance data, we find the optimal path to be $A \to D \to B$. Since city A and B are not connected, we must look at all the connections of A, in this case D, H, I and L. To get from A to D, we must drive 42 miles. We must explore the other options where to get from A to L, it only takes 25 miles, and I only takes 29 miles, and H takes 59 miles. However, I, L, and H are not connected to B, so we would have to travel to another city before getting to B. Since D is connected to B and with a small distance of 8 miles, we can get from A to B in just 50 miles by traveling through D, marking D with a label $(50, t)$. If we chose to exit from $A \to L$ instead, we would find that we have path choices $A \to L \to D \to B$ with a total distance of 73 miles, and we label L as $(73, t)$. We could also travel from $A \to L \to E \to B$ with a total distance of 108 miles, marking L with another label $(108, t)$. Because $108 > 73$, we mark L with a label $(73, p)$ and D with a label $(73, t)$. However, the shortest path is clearly taken by not traveling to L at all, so we mark D with a label $(50, p)$. This iterative process is repeated for all possible paths from $A \to B$ and we then find the nodes with the smallest d_ℓ values to be in our path. Therefore, the optimal path for this example is $A \to D \to B$.

5 Results and Analysis

5.1 Results

As I informed you in my status report, I first computed the optimal paths for each data set independently. As an example, to minimize the distance traveled from city A to city G, the optimal route is $A \to I \to G$ with a total distance traveled of 70.0 miles and probability of delay is 3.09%. In the same example, the optimal path to minimize delay probability is $A \to H \to C \to G$ with a total probability of delay of 1.44% and distance traveled is 152 miles. The following results are depicted below.

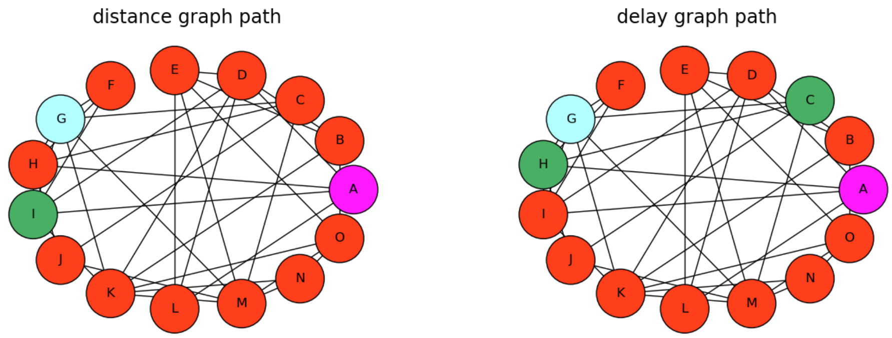


Figure 2: A circular graph of each data set for visualization. The starting point A is colored pink, and the ending point G is colored cyan. All intermediate cities in the path are colored green. Any cities not in the optimal path are colored red.

Because the optimal delay path increased the distance traveled by more than double, and the optimal distance path more than doubled the delay probability, the weight parameter is crucial for finding a good middle ground to minimize both flow values.

I also performed some experiments on this data set to determine an appropriate delay importance value for routes where the starting city was A. I determined all paths from A to all other cities and determined importance values by first determining the ratio between the delay probability and distance. This ratio is important because as a reminder we are trying to minimize $\sum_{i=1}^{N}\sum_{j=1}^{i} w_{ij} \cdot x_{ij}$. Excluding the binary variable for the moment, we are essentially trying to find an appropriate u where $w_{ij} = d_{ij} + u \cdot p_{ij} = 0$. Then we are specifically looking for when $u \approx \frac{d_{ij}}{p_{ij}}$. Therefore, I tracked this ratio in my computation and then found the maximum ratio within graphs. I then used this max ratio as an upper bound for a weight interval of ten thousand points. I iterated through these points using each point as my delay importance value. I tracked the paths and found every instance where the path shifted to find all intermediate paths. For routes starting with A, the only ending cities with intermediate paths are cities G and K found in Table 3 below. First, we look at the paths optimal for distance and delay separately:

Table 1: Optimal Distance Path with Starting City A

Route	Optimal Distance Path	Distance	Delay
$A \to B$	$A \to D \to B$	50.0	0.022468
$A \to C$	$A \to I \to G \to C$	101.0	0.036466
$A \to D$	$A \to D$	42.0	0.005126
$A \to E$	$A \to D \to E$	70.0	0.006883
$A \to F$	$A \to I \to F$	57.0	0.048414
$A \to G$	$A \to I \to G$	70.0	0.030866
$A \to H$	$A \to H$	59.0	0.000378
$A \to I$	$A \to I$	29.0	0.018658
$A \to J$	$A \to I \to G \to C \to J$	112.0	0.062145
$A \to K$	$A \to D \to K$	71.0	0.0306
$A \to L$	$A \to L$	25.0	0.009449
$A \to M$	$A \to D \to K \to M$	111.0	0.070582
$A \to N$	$A \to D \to K \to N$	80.0	0.061306
$A \to O$	$A \to D \to B \to O$	75.0	0.052575

Table 2: Optimal Delay Path with Starting City A

Route	Optimal Delay Path	Distance	Delay
$A \to B$	$A \to D \to E \to B$	91.0	0.011555
$A \to C$	$A \to H \to C$	121.0	0.008831
$A \to D$	$A \to D$	42.0	0.005126
$A \to E$	$A \to D \to E$	70.0	0.006883
$A \to F$	$A \to H \to F$	69.0	0.014319
$A \to G$	$A \to H \to C \to G$	152.0	0.014432
$A \to H$	$A \to H$	59.0	0.000378
$A \to I$	$A \to D \to I$	84.0	0.011671
$A \to J$	$A \to H \to J$	132.0	0.006121
$A \to K$	$A \to D \to I \to K$	130.0	0.020113
$A \to L$	$A \to L$	25.0	0.009449
$A \to M$	$A \to D \to E \to M$	135.0	0.012617
$A \to N$	$A \to D \to E \to O \to N$	105.0	0.025975
$A \to O$	$A \to D \to E \to O$	96.0	0.015015

Looking specifically at all intermediate paths, we can see that the delay and path distance have changed significantly when varying the delay importance value.

Table 3: Intermediate Paths with Starting City A

Route	Alternate Path	Ratio Slice	Distance	Delay
$A \rightarrow B$	NA			
$A \rightarrow C$	NA			
$A \rightarrow D$	NA			
$A \rightarrow E$	NA			
$A \rightarrow F$	NA			
$A \rightarrow G$	$A \rightarrow H \rightarrow G$	1483.5	91.0	0.016604
$A \rightarrow H$	NA			
$A \rightarrow I$	NA			
$A \rightarrow J$	NA			
$A \rightarrow K$	$A \rightarrow I \rightarrow K$	1155.54	75.0	0.0271
$A \rightarrow L$	NA			
$A \rightarrow M$	NA			
$A \rightarrow N$	NA			
$A \rightarrow O$	NA			

To visualize these intermediate paths, below are all paths found from the $A \rightarrow K$ route.

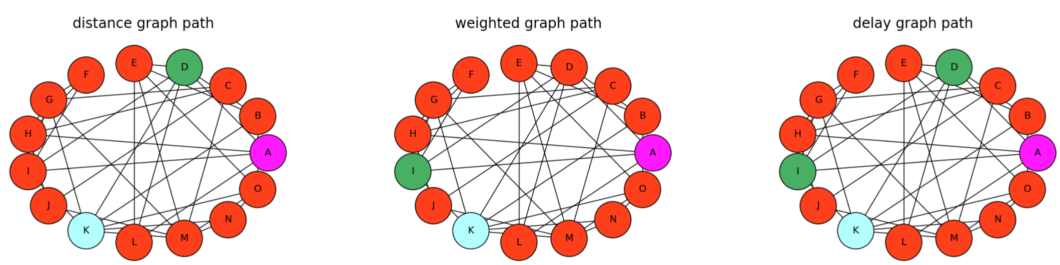


Figure 3: Here you can see all possible, optimal paths for the route from $A \rightarrow K$. We start at city A colored pink, travel to the intermediate green colored cities, and end at the city K colored cyan. All cities not traveled to are colored red.

5.2 Analysis

To find a logical delay importance value, when looking at Table 3, you can see that the ratio slice is close to 1500 for both paths. Therefore, choosing a delay importance value of 1500 will produce either intermediate path. However, if you go over 8000 on this value, delay will take over the edge weight for the $A \rightarrow K$ route and you will no longer obtain the intermediate path. To get delay to take over the edge weight on the $A \rightarrow G$ route, you will have to set the delay importance to almost 30,000 since the optimal delay path accrues a 152.0 distance and the difference in delay is small compared to the intermediate path and the optimal delay path. If you're interested in computing a similar analysis for other starting cities, the function find-optimal-param should be used to look for intermediate paths. The ratio slice will return with each path and a similar sort of analysis can be completed. I ran a similar analysis with all other starting cities and have determined values below for which you might find intermediate paths should you start from a different city. My analysis for the following values was not thorough and thus there could be missing intermediate paths if looking here. Therefore, I recommend checking these paths by looking at the printed report as it will tell you all possible paths that exist from the source to sink.

Table 4: Possible Delay Importance Values to Find Intermediate Paths

Starting City	Delay Importance Values
B	2600
C	1600
D	500
E	1000
F	2600
G	7000
H	1300
I	2700
J	7000
K	6100
L	1600
M	1100
N	1300
O	3000

6 Financial Impact Assessment

6.1 Demographics

To provide an accurate financial risk assessment, I must assess the demographic conditions of your delivery routes. Because your company is starting up in Washington State, I have pulled the local data from [2] where the average cost of diesel fuel per gallon is \$3.999 in Clarkston, WA. This city has the lowest reported average fuel cost in Washington currently, compared to \$4.708 per gallon statewide. Buying fuel in Idaho or Montana could prove to be financially efficient depending on delivery locations as their state average is much lower than Washington. I recommend trying to buy fuel in Clarkston, WA and will use this \$3.999 value by default in my computation. Additionally, the average speed in WA is reported in [3] to be 63MPH. This number is used by default to compute drive time to factor in labor costs. According to [4], on the high end because I know you pay your drivers well, a regional driver makes roughly \$0.55 per mile driven. I am also using the average miles per gallon of the trucks in your fleet, Freightliner Cascadias no more than 10 years old, to be 7MPG as reported by [5].

From this information, we can derive the cost of service to your company. First we define the variables as such:

$$\begin{aligned}
\text{distance traveled} &:= d\ [miles] \\
\text{wage of employee} &:= w\ [\$/mile] = 0.55 \\
\text{fuel efficiency} &:= f\ [miles/gallon] = 7 \\
\text{fuel cost} &:= g\ [\$/gallon] = 3.999 \\
\text{average speed} &:= v\ [miles/hour] = 63
\end{aligned}$$

From here we can derive the cost of delivery to be:

$$\text{labor cost} + \text{drive cost} := c\ [\$] = w \cdot d + \frac{g \cdot d}{f} = d(1.121).$$

I have not assessed some financial variables like vehicle maintenance, insurance costs, and others. This cost assessment can be improved later, by including these variables.

The total drive time is approximated by:

$$\text{time to deliver} := t\ [hours] = \frac{d}{v} = \frac{d}{63}.$$

I have assessed drive time due to the money-back-guarantee condition in your business model. For this reason, I have included a max time parameter in the user inputs to ensure the optimal route fits within this bound. Additionally, truck drivers are generally on a timed schedule and sometimes cannot take a route if it is going to take longer than the allocated time they have left on the road. This user inputted time parameter should consider both conditions and choose the condition that is shortest to ensure the order is not late and can be delivered by the driver.

6.2 Optimal Conditions Factoring in Risk

Lastly, I have included a parameter which asks for the cost of the order. This value is used to predict the risk of each delivery route. This risk model is rudimentary and should be improved upon by a financial manager. However, I wanted to include these parameters to show you how important it is to consider the financial risk. Based on my estimates, some orders may be worth the risk, while other orders may not. I have approximated risk by multiplying the probability of loss with the value of what you could lose. The probability of loss is equivalent to the probability of delay, as this factor is the risk you are taking in the delivery. The value of what you could lose is approximated below. First we define the cost of order:

$$\begin{aligned}
\text{price of order} &:= q\ [\$] \\
\text{max time} &:= m\ [hours] \\
\text{probability of delay} &:= p = P(t > m) \\
\text{risk} &:= r\ [\$] = q \cdot p \\
\text{cost of service} &:= V\ [\$] = r + c < q
\end{aligned}$$

If $V > q$ then the route assessed is not financially sustainable, since we do not want to accumulate more labor and drive cost than the order is worth. We also want to make sure that the delivery and warranty costs are less than the price of the order.

7 Recommendations and Next Steps

7.1 Recommendations

The most expensive routes are likely going to be traveling from city A to city M, as it generally has a long distance and high probability of delay with no intermediate paths. I would recommend, if M is not a high revenue city, removing M from your deliverable cities until you add more cities with more optimal routes. When looking at the most optimal routes for other cities, I recommend trying to keep your probability of delay below 3%. This number seems low enough to make good choices on total distance factoring in intermediate routes and minimizing delay probability.

Due to the financial assessments I've implemented, if you are not charging enough for your delivery service, this will become apparent based on the warnings printed. I suggest you continue to monitor the printed warnings to ensure you are making a large enough profit to satisfy your stakeholders. Considering there are some financial factors I did not assess, I also suggest you hire a financial manager to get a more thorough assessment of your profits to ensure the financial parameters are well identified and additional financial hurdles do not surprise you in your route deliveries.

7.2 Next Steps

As a next step, I would encourage you to hire a software engineer to complete a mobile application for calculating the optimal route. This mobile application can be used by your drivers to ensure they are taking optimal routes as often as possible. Some planning may occur, but sometimes traffic occurs randomly, and it is a good idea to be factoring in current conditions instead of relaying just on predicted conditions. The GUI I built for this project can be continued upon to build a more robust application. Your engineer should write test functions for this application and build error messages if inputs are incorrect. Due to lack of time, I did not write any tests for the GUI and thus have not provided these additional error messages, but they will not be hard to implement for a well trained engineer. There are also four additional tests that need to be written for functions in the graph-2D file. I have outlined which ones are not tested in the tests file providing a function showing the message "Need to write this test."

8 Conclusion

In this report, I have explained my computational and mathematical model for solving the optimal path problem. I have included financial risk parameters when assessing if the path is optimal, and built a graphical user interface to allow modular input parameters. This assessment can help you decide how much to charge, acquire information about when you are not charging enough, and help optimize your labor costs. Using the optimal route can also help mitigate refunds based on your money-back-guarantee, and ensure your customers are as satisfied as possible. Ideally, drivers will not have to work overtime for late deliveries and thus company morale will stay strong.

All code for this project was written in Python and optimization was conducted using a module called NetworkX. I have included test functions as well as other types of functions for analysis moving forward should you choose to conduct any more analysis. I assessed all routes for starting city A to all other cities, and recommend a delay importance value of 1500. I also computed a short report that will print in a python terminal if this code is run. In an attempt to visualize the data, I wrote plotting functions and included some images above to show a representation of some optimal route calculations. If you are interested in hiring a software engineer to build the mobile application I suggested, I am available for hire. You may contact my assistant Jarvis at (729)-342-6867 for more billing and scheduling information. Lastly, I have included all of my code in an appendix below if you would like to use this work in the future.

References

[1] Solving Shortest Path Problem: Dijkstra's Algorithm. 2009.

[2] gasprices.aaa.com. AAA Gas Prices. 2024.

[3] CNNMoney. (n.d.). CNNMoney. What is the average speed limit in your state? 2024.

[4] indeed.com. Truck Driver Salary in Washington State. 2024.

[5] fuelly.com. Freightliner Cascadia MPG - Actual MPG from 47 Freightliner Cascadia owners. 2024.

9 Appendix

9.1 main file

```
"""

@author: Sandy Auttelet

"""

import graph_2D as g2
import plotting_data as pd
import numpy as np
import networkx as nx
import tests as ts
import os
import csv
import saving_gui as GUI #Comment out this import if you do not want to
    execute GUI

#Check current directory's path
#os.getcwd()

data = np.genfromtxt('input_data.txt', dtype=None, delimiter=",", encoding
    =None)

file1_name,file2_name,source,sink,weight,tol,order_cost,file1_text,
    file2_text,file1_type,file2_type = GUI.load_data()

#else:
"""
Default User Input
======================================================================
"""
data1 = np.loadtxt(file1_name, delimiter=",")
data2 = np.loadtxt(file2_name, delimiter=",")

# source = 'A'
# sink = 'G'

# weight = 7000

# #Maximum tolerance for [distance, delay]
# tol = [100.0, 0.06]

# #Additional parameters for printing and financial risk assessment
# order_cost = 100
# file1_text = 'distance'
# file2_text = 'delay'
# file1_type = 'csv'
# file2_type = 'csv'

#=====================================================================

```

```python



"""
Building Graphs
====================================================================
"""
graph1 = g2.create_graph(data1)
graph2 = g2.create_graph(data2)
n = len(data1)
m = len(data2)

sim = ts.test_data_similarity(graph1,graph2)

graph_weighted = g2.weighted_sum_edge_graph_2d(graph1, graph2, weight,n,
    sim=sim)[0]
k = len(graph_weighted.nodes)

#====================================================================





"""
Optimal Paths
====================================================================
"""
path_g1 = nx.shortest_path(graph1, source=source, target=sink, weight='
    weight')
path_g2 = nx.shortest_path(graph2, source=source, target=sink, weight='
    weight')
path_weighted = nx.shortest_path(graph_weighted, source=source, target=
    sink, weight='weight')

#====================================================================





"""
Checking Tolerance
====================================================================
"""

path_with_weight = g2.find_optimal_param(graph1,graph2,source,sink,n,sim)
print(path_with_weight)

paths = g2.run_shortest_parallel(graph1, graph2,graph_weighted, source)

all_paths = []
for i in range(len(path_with_weight)):
    all_paths.append(path_with_weight[i][0])
```

```


# #The code below was used to find intermediate paths and the ratio slice.
# paths_with_weights = []
# for i in range(len(graph1.nodes)):
#     path_with_weight = g2.find_optimal_param(graph1,graph2,'A',chr(i+65),n,sim)
#     paths_with_weights.append(path_with_weight)


#====================================================================





"""
Finding Flows
=====================================================================
"""
#1D Flow
flow1_opt = g2.get_path_flow(graph1, path_g1)
flow1_g2 = g2.get_path_flow(graph1, path_g2)
flow2_opt = g2.get_path_flow(graph2, path_g2)
flow2_g1 = g2.get_path_flow(graph2, path_g1)


#2D Flow
flow1_w = g2.get_path_flow(graph1, path_weighted)
flow2_w = g2.get_path_flow(graph2, path_weighted)

#Irrelivant value numerically due to delay importance parameter
flow_weighted = g2.get_path_flow(graph_weighted, path_weighted)

#====================================================================




"""
=====================================================================
=====================================================================
Print Report
=====================================================================
=====================================================================
"""

print(f'Optimal {file1_text} path:', path_g1)
print(f'{file1_text} of optimal {file1_text} path:', flow1_opt)
print(f'{file2_text} of optimal {file1_text} path:', flow2_g1, "\n")

print(f'Optimal {file2_text} path:', path_g2)
print(f'{file1_text} of optimal {file2_text} path:', flow1_g2)
print(f'{file2_text} of optimal {file2_text} path:', flow2_opt, "\n")
```

```
print("Optimal weighted path:", path_weighted)
print(f'{file1_text} of weighted path:', flow1_w)
print(f'{file2_text} of weighted path:', flow2_w, "\n")


print(f'All paths from {source} to {sink}\n', all_paths, "\n")

optimal_paths = g2.optimal_path(path_with_weight, graph1, graph2, tol[0],
    tol[1],order_cost)
print(f'All optimal paths from {source} to {sink} with input parameters
    assessed:\n', optimal_paths,"\n")

print(f'All optimal weighted paths from {source} to all other cities with
    {file1_text} and {file2_text} respectively:\n', paths)

"""
Plotting Graphs
====================================================================
"""
pd.plot_graph(graph1,source,sink,n,path_g1, title=f'{file1_text} graph
    path')
pd.plot_graph(graph2,source,sink,m,path_g2, title=f'{file2_text} graph
    path')
pd.plot_graph(graph_weighted,source,sink,k,path_weighted, title='weighted
    graph path')

#===================================================================


```

9.2 graph-2D file

```
"""

@author: Sandy Auttelet

"""

import numpy as np
import networkx as nx

#For rounding if needed:
def find_mantissa(num):
    """
    Finds optimal rounded values for report printing.

    Parameters
    ----------
    num : float
        number you are trying to round.

    Returns
```

```
    -------
    rounded_num : float
        rounded number to three decimal places.

    """
    scale = int(round(np.log10(num),0))-7 #the 7 here defines order of accuracy in reporting
    mantissa = int(round(num/10**scale,0))
    rounded_num = mantissa*10**scale
    rounded_num = round(rounded_num,3)
    return rounded_num

def create_graph(data):
    """
    Builds a networkx graph object from a matrix of edge weights.

    Parameters
    ----------
    data : list of list of floats
        edge weights for graph

    Returns
    -------
    G : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html

    """
    G = nx.Graph(weight=0)
    n = len(data)
    for i in range(n):
        G.add_node(chr(i+65))
    for i in range(n):
        for j in range(n):
            if data[i][j] != 0:
                G.add_edge(chr(i+65),chr(j+65), weight=data[i][j])
    return G

def weighted_sum_edge_graph_2d(graph1, graph2, weights_p,n,sim=True):
    """
    Builds a networkx graph object with edge weights calculated from weighted sum of two input graphs.

    Parameters
    ----------
    graph1 : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    graph2 : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    weights_p : float
        adjustable parameter for weighted sum calculating manipulating importance of graph2 data compared to graph1
    n : int
        number of nodes in each graph
    sim : bool, optional
```

```
        Used to determine if input graphs have same structure for weighted sum calulation. The default is True.

    Returns
    -------
    TYPE
        DESCRIPTION.

    """
    if sim == False:
        print("Your two data sets are not the same size with the same connections and thus a weighted graph cannot be computed.")
        return None
    G = nx.Graph(weight=0)
    ratios = []
    for i in range(n):
        G.add_node(chr(i+65))
    for i in range(n):
        for j in range(n):
            if graph1.get_edge_data(chr(i+65),chr(j+65)) != None and graph2.get_edge_data(chr(i+65),chr(j+65)) != None:
                edge1_weight = graph1.get_edge_data(chr(i+65),chr(j+65),"weight")
                edge2_weight = graph2.get_edge_data(chr(i+65),chr(j+65),"weight")
                ratios.append(edge1_weight['weight']/edge2_weight['weight'])
                weighted_sum = edge1_weight['weight']+weights_p*edge2_weight['weight']
                G.add_edge(chr(i+65),chr(j+65), weight=weighted_sum)
    return G, max(ratios)


def get_path_flow(graph, path):
    """
    Determines the sum of the weight on each edge from a specified travel path.

    Parameters
    ----------
    graph : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    path : list of strings
        Strings denote nodes from graph, list of a specific path taken.

    Returns
    -------
    flow : float
        Sum of weights on edges traveled from input path.

    """
    flow = 0
    for i in range(1,len(path)):
        edge_weight = graph.get_edge_data(path[i-1],path[i],"weight")
```

```python
        if graph.get_edge_data(path[i-1],path[i]) != None:
            flow += edge_weight['weight']
    return round(flow,6)

def run_shortest_parallel(graph1, graph2, graph_weighted, source):
    """
    Returns all optimal paths for a source node to all other nodes in the graph.

    Parameters
    ----------
    graph1 : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    graph2 : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    graph_weighted : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html, generated from weighted sum.
    source : string
        node label for starting point in paths.

    Returns
    -------
    path_info : list of list of strings and two floats
        first part of list is optimal path with each node traveled to, second element
        is distance traveled in path, and third is probability of delay.

    """
    paths = dijkstra(graph_weighted)
    path_info = []
    for i in range(len(paths[source])):
        flow1_paths = get_path_flow(graph1, paths[source][chr(i+65)])
        flow2_paths = get_path_flow(graph2, paths[source][chr(i+65)])
        path_info.append([paths[source][chr(i+65)], flow1_paths, flow2_paths])
    return path_info

def dijkstra(graph):
    path = dict(nx.all_pairs_dijkstra_path(graph))
    return path

def find_optimal_param(graph1,graph2,source,sink,n,sim):
    """
    Determines what value of input weight parameter switches path aand saves those numbers with associated path.

    Parameters
    ----------
    graph1 : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    graph2 : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    source : String
```

```
        Starting point
    sink : String
        Ending point
    n : int
        number of nodes in graphs
    sim : bool
        ensures graphs have the same structure so weighted sum can be
    computed.

    Returns
    -------
    path_shifts : list of list of strings and one float
        list of path, strings denoting nodes traveled to, for shortest
    path with a weight parameter in each list.

    """
    graph0, ratio = weighted_sum_edge_graph_2d(graph1, graph2, 0,n,sim=sim
    )
    weights = np.linspace(0,int(ratio+100),10000)
    path_0 = nx.shortest_path(graph0, source=source, target=sink, weight='
    weight')
    path_shifts = [[path_0,0]]
    k = 1
    for i in range(len(weights)):
        path_w = path_0
        graph_weighted = weighted_sum_edge_graph_2d(graph1, graph2,
    weights[i],n,sim=sim)[0]
        path_0 = nx.shortest_path(graph_weighted, source=source, target=
    sink, weight='weight')
        if path_w != path_0:
            k+=1
            path_shifts.append([path_0,weights[i]])
    return path_shifts

def financial_risk(d,p,q,tol1,tol2,i, w=0.55,f=7.0,g=3.999,v=63.0):
    """
    Determines if route is financially viable based on risk assessment
    calculations.

    Parameters
    ----------
    d : flaot
        distance traveled
    p : float
        probability of delay
    q : float
        price of order
    m : float
        max time able to drive
    tol2 : float
        max risk willing to take
    w : float, optional
        wage of driver per mile. The default is 0.55.
    f : float, optional
```

```
        fuel efficiency of Freigthliner Cascadia truck. The default is 7.0.
    g : float, optional
        average cost of fuel per gallon. The default is 3.999.
    v : float, optional
        average speed in miles per hour. The default is 63.0.

    Returns
    -------
    bool
        returns true if cost of service is less than price of order.

    """
    c = w*d+(g*d/f)
    t = d/v
    r = q*p
    V = r+c
    if r > tol2:
        print(f'Current risk is larger than max risk input for path {i+1}.\n')
        return False
    if t > tol1:
        print(f'Current expected time of delivery is larger than max time input for path {i+1}.\n')
        return False
    if V > q:
        print(f'Current cost of service is larger than cost of order for path {i+1}.\n')
        return False
    if q < c:
        print(f'Current weighting parameter is not optimal for shortest distance for path {i+1}.\n')
        return False
    else:
        return True

def optimal_path(paths, graph1, graph2, tol1, tol2,cost):
    """
    Determines if a path is optimal based on input parameters and financial risk assessment.

    Parameters
    ----------
    paths : list of lists
        All possible paths from source to sink
    graph1 : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    graph2 : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    tol1 : float
        Max time able to travel
    tol2 : float
        max risk willing to accept
    cost : flaot
```

```
        price charged to customer for order delivery

    Returns
    -------
    optimal_paths : list of lists
        all possible paths that are considered optimal based on input
    parameters

    """
    optimal_paths = []
    for i in range(len(paths)):
        flow1 = get_path_flow(graph1, paths[i][0])
        flow2 = get_path_flow(graph2, paths[i][0])
        risk = financial_risk(flow1,flow2,cost,tol1,tol2, i)
        if risk == True:
            optimal_paths.append(paths[i][0])
    if len(optimal_paths) < 1:
        print("No optimal path was found based on input parameters.")
        print("Consider adjusting input parameters and recalculating the
    optimal path.\n")
    return optimal_paths



```

9.3 plotting file

```
"""

@author: Sandy Auttelet

"""

import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

def plot_graph(graph,source,sink,n,path,title=None,labels=None):
    """
    Plots arbitrary circular graph wuth optimal path labels

    Parameters
    ----------
    graph : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    source : string
        origin point node label
    sink : string
        ending point node label
    n : int
        number of nodes
    path : list of strings
        node labels for optimal path
```

```
    title : string, optional
        Title of plot. The default is None

    Returns
    -------
    None. Prints plot for graph. Node labeled source colored pink (#FF19FF),
    sink node colored cyan (#B3FFFF), and other nodes in optimal path colored
    green (#49AF64). All nodes not in path colored red (#FE3F1C).

    """
    fig, ax = plt.subplots()
    fig.suptitle(title, fontsize='xx-large')
    pos = nx.circular_layout(graph)
    V = list(pos)

    node_size = 2000
    edgecolors = 'black'

    node_color = []
    for i in range(n):
        node_color.append('#FE3F1C')
        for j in range(len(path)):
            if chr(i+65) == source:
                node_color[i] = '#FF19FF'
                break;
            if chr(i+65) == sink:
                node_color[i] = '#B3FFFF'
                break;
            if chr(i+65) == path[j] and chr(i+65) != source and chr(i+65) != sink:
                node_color[i] = '#49AF64'
                break;



    nx.draw_networkx(graph, ax=ax, pos=pos, nodelist=V, edgelist=[],
                     node_color=node_color, edgecolors=edgecolors,
                     node_size=node_size)
    nx.draw_networkx_edges(graph, ax=ax, pos=pos, edgelist=graph.edges,
                           node_size=node_size)

    if labels != None:
        nx.draw_networkx_edge_labels(graph, ax=ax, pos=pos,font_size='xx-large')
    ax.axis('off')
    #plt.savefig('optimal_path.png', dpi=400)
```

9.4 GUI file

```python
"""

@author: sandy

"""

import tkinter as tk
from tkinter import filedialog
import csv
import os
from ctypes import windll
import numpy as np

"""
===============================================================================
This file is for creating a GUI if user wants to input data into a text file named
input_data.txt.

Files will be overwritten so ensure this does not happen if you want to save
many data inputs by changing save name from input_data to another name, or save
data in a different location.

Errors can occur if directory is not corrected so ensure you have saved input_data
to desired directory with os.chdir() to change saving location.

This GUI has not been tested nor will it determine input errors. All errors will
present themselves when running the main file.

Window must be closed to proceed through the computation.
===============================================================================
"""

#os.getcwd() #Check your current directory path

#Change directory to save to another location
os.chdir('C:/Users/sandy/OneDrive/Documents/Classes/Math 464- Lin Opt/Final Project/data')

window = tk.Tk()

my_data = []

window.geometry("800x800")
window.title("User Inputs for Optimal Path Calculation")

```

```
title_frame = tk.Frame(master=window,relief=tk.RAISED, borderwidth=10,
    height=400, bg="white")
title_frame.pack(fill=tk.X)
title = tk.Label(title_frame, text="Please input user data to find the
    optimal path to travel.", bg="white")
title.pack(padx=(5, 5), pady=5)

frame1 = tk.Frame(master=window,relief=tk.RAISED, borderwidth=10, height
    =100)
frame1.pack(fill=tk.X)

sourceframe = tk.Frame(master=window,relief=tk.RAISED, borderwidth=10,
    height=50, bg='#FF19FF')
sourceframe.pack(fill=tk.X)

sinkframe = tk.Frame(master=window,relief=tk.RAISED, borderwidth=10,
    height=25, bg='#B3FFFF')
sinkframe.pack(fill=tk.X)

frame2 = tk.Frame(master=window,relief=tk.RAISED, borderwidth=10, height
    =100)
frame2.pack(fill=tk.X)

graph1_val = tk.StringVar()
graph2_val = tk.StringVar()
source_val = tk.StringVar()
sink_val = tk.StringVar()
weight_val = tk.DoubleVar()
tol1_val = tk.DoubleVar()
tol2_val = tk.DoubleVar()
cost_val = tk.DoubleVar()

graph1_entry = tk.Entry(frame1, textvariable=graph1_val, fg="black", bg="
    white", width=30)
graph1_label = tk.Label(frame1, text="Distance Data Set:\n(include file
    type i.e. distance.csv)")
graph1_label.grid(row=0, column=0, padx=(5, 5), pady=5)

graph2_entry = tk.Entry(frame1, textvariable=graph2_val, fg="black", bg="
    white", width=30)
graph2_label = tk.Label(frame1, text="Delay Data Set:\n(include file type
    i.e. delay.csv)")
graph2_label.grid(row=1, column=0, padx=(5, 5), pady=5)

source_entry = tk.Entry(sourceframe, textvariable=source_val, fg="black",
    bg="white", width=40)
source_label = tk.Label(sourceframe, text="Starting City:", bg="white")
source_label.grid(row=0, column=0, padx=(5, 5), pady=5)

sink_entry = tk.Entry(sinkframe, textvariable=sink_val, fg="black", bg="
    white", width=41)
sink_label = tk.Label(sinkframe, text="Ending City:",bg="white")
sink_label.grid(row=0, column=0, padx=(5, 5), pady=5)

```

```
weight_entry = tk.Entry(frame2, textvariable=weight_val, fg="black", bg="white", width=30)
weight_label = tk.Label(frame2, text="Delay Importance:")
weight_label.grid(row=0, column=0, padx=(5, 5), pady=5)

tol1_entry = tk.Entry(frame2, textvariable=tol1_val, fg="black", bg="white", width=30)
tol1_label = tk.Label(frame2, text="Max Time of Delivery [hours]:")
tol1_label.grid(row=1, column=0, padx=(5, 5), pady=5)

tol2_entry = tk.Entry(frame2, textvariable=tol2_val, fg="black", bg="white", width=30)
tol2_label = tk.Label(frame2, text="Max Risk Willing to Occur [$]:")
tol2_label.grid(row=2, column=0, padx=(5, 5), pady=5)

cost_entry = tk.Entry(frame2, textvariable=cost_val, fg="black", bg="white", width=30)
cost_label = tk.Label(frame2, text="Cost of Order [$]:")
cost_label.grid(row=3, column=0, padx=(5, 5), pady=5)

graph1_entry.grid(row=0, column=1,padx=10,pady=10)
graph2_entry.grid(row=1, column=1,padx=10,pady=10)
source_entry.grid(row=0,column=1,padx=10,pady=10)
sink_entry.grid(row=0,column=1,padx=10,pady=10)
weight_entry.grid(row=0,column=1,padx=10,pady=10)
tol1_entry.grid(row=1,column=1,padx=10,pady=10)
tol2_entry.grid(row=2,column=1,padx=10,pady=10)
cost_entry.grid(row=3,column=1,padx=10,pady=10)


def save_info():
    tk.messagebox.showinfo(title="Submit successful", message="Data submitted succesfully. Report and plots will print in terminal.")
    graph1 = graph1_entry.get()
    graph2 = graph2_entry.get()
    source = source_entry.get()
    sink = sink_entry.get()
    weight = weight_entry.get()
    tol1 = tol1_entry.get()
    tol2 = tol2_entry.get()
    cost = cost_entry.get()

    file = open("input_data.txt", "w")
    file.write(graph1)
    file.write(",")
    file.write(graph2)
    file.write(",")
    file.write(source)
    file.write(",")
    file.write(sink)
    file.write(",")
    file.write(weight)
    file.write(",")
    file.write(tol1)
```

```
    file.write(",")
    file.write(tol2)
    file.write(",")
    file.write(cost)
    file.close()
    window.destroy()



file_path = None
submit_button = tk.Button(window,
    text="Submit Data",
    command=save_info,
    width=25,
    height=5,
    bg="blue",
    fg="yellow",
)

submit_button.pack(padx=20,pady=20)


def on_closing():
    window.destroy()

window.protocol("WM_DELETE_WINDOW", on_closing)

window.mainloop()

def load_data():
    data = np.genfromtxt('input_data.txt', dtype=None, delimiter=",",
   encoding=None)

    file1_name = str(data['f0'])
    file2_name = str(data['f1'])
    source = str(data['f2'])
    sink = str(data['f3'])
    weight = data['f4']
    tol = []
    tol.append(data['f5'])
    tol.append(data['f6'])
    order_cost = data['f7']
    file1_text = file1_name.split(".")
    file2_text = file2_name.split(".")
    return file1_name,file2_name, source, sink, weight, tol, order_cost,
   file1_text[0], file2_text[0], file1_text[1], file2_text[1]


```

9.5 tests file

```
"""

```

```python
@author: Sandy Auttelet

"""


import graph_2D as g2
import networkx as nx
import plotting_data as pd


test_distance_data = [[0.,  2.9, 3.2, 4.2, 0.,  0.,  0. ],\
 [2.9, 0.,  0.,  2.3, 0.,  4.1, 0. ],\
 [3.2, 0.,  0.,  2.1, 3.2, 0.,  0. ],\
 [4.2, 2.3, 2.1, 0.,  3.7, 2.4, 4.2],\
 [0.,  0.,  3.2, 3.7, 0.,  0.,  2.9],\
 [0.,  4.1, 0.,  2.4, 0.,  0.,  3. ],\
 [0.,  0.,  0.,  4.2, 2.9, 3.,  0. ]]

test_delay_data = [[0., 0.021, 0.042, 0.033, 0., 0., 0.],\
 [0.021, 0., 0., 0.03, 0., 0.011, 0.],\
 [0.042, 0., 0., 0.042, 0.013, 0., 0.],\
 [0.033, 0.03, 0.042, 0., 0.019, 0.008, 0.057],\
 [0., 0., 0.013, 0.019, 0., 0., 0.01],\
 [0., 0.011, 0., 0.008, 0., 0., 0.028],\
 [0., 0., 0., 0.057, 0.01, 0.028, 0.]]


def test_data_similarity(graph1,graph2):
    """
    Tests if two graphs have the same structure.

    Parameters
    ----------
    graph1 : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html
    graph2 : networkx graph object
        https://networkx.org/documentation/stable/tutorial.html

    Returns
    -------
    v : bool
        True if two graphs have same structure else false with printed
    warnings.

    """
    v = True
    nodes2 = list(graph2.nodes())
    nodes1 = list(graph1.nodes())
    if len(nodes1) != len(nodes2):
        v = False
        print("The two graphs do not have the same number of nodes and
    thus a weighted graph cannot be created.")
        return v
    edges2 = list(graph2.edges())
```

```python
    edges1 = list(graph1.edges())
    for i in range(len(nodes1)):
        if nodes1[i] != nodes2[i]:
            print("Warning: The two graphs do not have the same ordered node labels.")
            if edges1[i] != edges2[i]:
                print("Warning: The two graphs do not have the same ordered edge labels.")
        if len(edges1) != len(edges2):
            v = False
            print("The two graphs do not have the same number of edges and thus a weighted graph cannot be created.")
            if edges1[i] != edges2[i]:
                print("Warning: The two graphs do not have the same ordered edge labels")
            return v
        if edges1[i] != edges2[i]:
            print("Warning: The two graphs do not have the same ordered edge labels.")

def test_dist_image():
    """
    Generates image for specific test data to ensure accurate plotting.

    Returns
    -------
    None.

    """
    test_graph = g2.create_graph(test_distance_data)
    short_path = ['A', 'D', 'G']
    pd.plot_graph(test_graph,'A','G',7,short_path, title='Distance Graph Path')

def test_delay_image():
    """
    Generates image for specific test data to ensure accurate plotting.

    Returns
    -------
    None.

    """
    test_graph = g2.create_graph(test_delay_data)
    short_path = ['A', 'B', 'F', 'G']
    pd.plot_graph(test_graph,'A','G',7,short_path, title='Delay Graph Path')

def test_weighted_image():
    """
    Generates image for specific test data to ensure accurate plotting.

    Returns
    -------
```

```
    None.

    """
    test_graph1 = g2.create_graph(test_distance_data)
    test_graph2 = g2.create_graph(test_delay_data)
    test_weighted_graph = g2.weighted_sum_edge_graph_2d(test_graph1,
    test_graph2, [1.0,1.0],7)
    short_path = ['A', 'B', 'F', 'G']
    pd.plot_graph(test_weighted_graph,'A','G',7,short_path, title='
    Weighted Graph Path')

# test_dist_image()
# test_delay_image()
# test_weighted_image()

def test_graph_creation_dist():
    """
    Tests if distance graph built as expected from test data.

    Returns
    -------
    None. Prints warning for graph_2d() in g2

    """
    test_graph = g2.create_graph(test_distance_data)
    nodes = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
    edges = [('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'D'), ('B', 'F'), (
    'C', 'D'), ('C', 'E'), ('D', 'E'), ('D', 'F'), ('D', 'G'), ('E', 'G'),
    ('F', 'G')]
    i = 0
    for node in test_graph.nodes():
        if node != nodes[i]:
            print("Warning: graph_2d() in g2 did not create the nodes of
    your distance graph as expected.")
        i += 1
    i = 0
    for edge in test_graph.edges():
        if edge != edges[i]:
            print("Warning: graph_2d() in g2 did not create the edges of
    your distance graph as expected.")
        i += 1
test_graph_creation_dist()

def test_graph_creation_delay():
    """
    Tests if delay graph built as expected from test data.

    Returns
    -------
    None. Prints warning for graph_2d() in g2

    """
    test_graph = g2.create_graph(test_delay_data)
    nodes = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

```
    edges = [('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'D'), ('B', 'F'), ('C', 'D'), ('C', 'E'), ('D', 'E'), ('D', 'F'), ('D', 'G'), ('E', 'G'), ('F', 'G')]
    i = 0
    for node in test_graph.nodes():
        if node != nodes[i]:
            print("Warning: graph_2d() in g2 did not create the nodes of your delay graph as expected.")
        i += 1
    i = 0
    for edge in test_graph.edges():
        if edge != edges[i]:
            print("Warning: graph_2d() in g2 did not create the edges of your delay graph as expected.")
        i += 1

test_graph_creation_delay()

def test_weighted_graph_creation():
    """
    Tests if weighted graph built as expected from test data.

    Returns
    -------
    None. Prints warning for weighted_sum_edge_graph_2d() in g2

    """
    test_graph1 = g2.create_graph(test_distance_data)
    test_graph2 = g2.create_graph(test_delay_data)
    test_weighted_graph = g2.weighted_sum_edge_graph_2d(test_graph1, test_graph2, [1.0,1.0],7)
    nodes = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
    if len(nodes) != len(test_weighted_graph.nodes):
        print("Warning: weighted_sum_edge_graph_2d() in g2 did not create the nodes of your weighted graph as expected.")
    edges = [('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'D'), ('B', 'F'), ('C', 'D'), ('C', 'E'), ('D', 'E'), ('D', 'F'), ('D', 'G'), ('E', 'G'), ('F', 'G')]
    if len(edges) != len(test_weighted_graph.edges):
        print("Length Warning: weighted_sum_edge_graph_2d() in g2 did not create the edges of your weighted graph as expected.")
    print(test_weighted_graph.edges)
    i = 0
    for node in test_weighted_graph.nodes():
        if node != nodes[i]:
            print("Warning: weighted_sum_edge_graph_2d() in g2 did not create the nodes of your weighted graph as expected.")
        i += 1
    i = 0
    for edge in test_weighted_graph.edges():
        if edge != edges[i]:
            print("Warning: weighted_sum_edge_graph_2d() in g2 did not create the edges of your weighted graph as expected.")
        i += 1
```

```python

# test_weighted_graph_creation()

def test_shortest_distance_path():
    """
    Tests networkx shortest path for distance data is computed properly with printed warnings if fails.

    Returns
    -------
    None.

    """
    test_graph_dist = g2.create_graph(test_distance_data)
    test_path_dist = nx.shortest_path(test_graph_dist, source='A', target='G', weight='weight')
    short_path = ['A', 'D', 'G']
    i = 0
    for node in test_path_dist:
        if len(short_path) != len(test_path_dist):
            print("Warning: nx.shortest_path() did not compute as expected. Likely weight assignment error.")
            break;
        if node != short_path[i]:
            print("Warning: nx.shortest_path() did not compute as expected. Likely edge label assignment error.")
        i += 1

test_shortest_distance_path()

def test_shortest_delay_path():
    """
    Tests networkx shortest path for delay data is computed properly with printed warnings if fails.

    Returns
    -------
    None.

    """
    test_graph_delay = g2.create_graph(test_delay_data)
    test_path_delay = nx.shortest_path(test_graph_delay, source='A', target='G', weight='weight')
    short_path = ['A', 'B', 'F', 'G']
    i = 0
    for node in test_path_delay:
        if len(short_path) != len(test_path_delay):
            print("You have not found the accurate shortest path.")
            break;
        if node != short_path[i]:
            print("You have not found the accurate shortest path.")
        i += 1

test_shortest_delay_path()
```

```python

def test_shortest_weighted_path():
    """
    Tests if networkx shortest path for weighted data is computed properly
     with printed warnings if fails.

    Returns
    -------
    None.

    """
    print("Need to write this test.")

def test_flow_dist():
    """
    Tests g2.get_path_flow() if accurate flow for a specific path is
   computed correctly for distance data.

    Returns
    -------
    None.

    """
    short_path = ['A', 'D', 'G']
    test_graph_dist = g2.create_graph(test_distance_data)
    test_flow_dist = g2.get_path_flow(test_graph_dist, short_path)
    real_flow = 8.4
    if test_flow_dist != real_flow:
        print("You have not found the accurate shortest path flow.")

test_flow_dist()

def test_flow_delay():
    """
    Tests g2.get_path_flow() if accurate flow for a specific path is
   computed correctly for delay data.

    Returns
    -------
    None.

    """
    short_path = ['A', 'B', 'F', 'G']
    test_graph_delay = g2.create_graph(test_delay_data)
    test_flow_delay = g2.get_path_flow(test_graph_delay, short_path)
    real_flow = 0.06
    if test_flow_delay != real_flow:
        print("You have not found the accurate shortest path flow.")

test_flow_delay()


def test_shortest_parallel_path():
    print("Need to write this test.")
```

```python

def test_optimal_path():
    print("Need to write this test.")

def test_find_optimal_param():
    print("Need to write this test.")

def test_financial_risk():
    print("Need to write this test.")


```