Determination of the shortest path between roads in Jakarta with Dijkstra’s and Kruskal’s Algorithm

Introduction

As a person who enjoys exploring new countries, I've often relied on Google Maps especially when it comes to navigating unfamiliar places. Every time I use this application, I’m always in awe of the application’s ability to find the most efficient routes connecting between the destinations entered by its users, making its users traverse a shorter distance, saving time, and reducing fuel consumption. In addition to these benefits, this also translates to a positive impact on the environment which further elaborates how important and amazing this application is. As a student who is traversing far distances between home and school, this application is also extremely useful for reducing costs incurred for my family as we traverse the shortest path. According to the internet, there are many ways you can take to reduce this fuel consumption, like being mindful of how much weight you're carrying or adopting smoother driving techniques, however, in my opinion, these strategies don't quite match up to the sheer efficiency of simply taking the shortest route, which is something that Google Maps does.

During the COVID-19 pandemic, there has been a rise in the utilization of delivery services. This is attributed to the fact that people have limited mobility due to quarantine measures and health concerns. In light of this, certain delivery services have encountered inefficiencies in their operations as they struggle to optimize their routes for maximum efficiency and minimal fuel consumption, leading to a mismatch between their earnings and expenses. In this IA, I want to figure out how to find the shortest path from one destination to another destination using one of the algorithms Google Maps uses, Dijsktra’s algorithm.

I also want to figure out the minimum total cost given that there is a number of destinations to traverse so that these drivers can incur the least cost by reducing unnecessary “back and forth” routes. Through research, I figured it is possible to achieve this through a graph theory algorithm, a minimum spanning tree, namely Kruskal’s algorithm. Despite Kruskal’s algorithm having a minimum spanning tree objective, I want to know whether it does find the minimum distance between nodes as well as Dijkstra’s algorithm which we know to have a main focus of finding the shortest distance between two nodes.

These minimization algorithms fall under the domain of optimization as they try to find the least distance to traverse between points. In specific, these Greedy algorithms fall under this optimization domain. According to Programiz, a greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. Dijkstra’s algorithm is a good example of a greedy algorithm that focuses on making the locally optimal choice at each step as it finally gets the shortest path to traverse between destinations (nodes in a graph). Kruskal’s algorithm is also a greedy algorithm as it chooses the most efficient connections step by step until the full minimum spanning tree is formed which will be explained as we go on this exploration.

Aim and methodology

The main purpose of this IA is to figure out the minimum cost it takes for delivery services to deliver a packet to assigned destinations with an optimal amount of fuel by traversing the shortest distance between destinations. This IA will also investigate the efficiency of Dijkstra’s and Kruskal’s algorithms in finding the shortest cost path. For the graph itself, it will be based on real-world demographic data which will be plotted as nodes in a graph.

My Exploration Question

What is the shortest path for a delivery man to optimize the delivery of goods throughout Jakarta (Indonesia’s Capital City) to reduce cost and save time?

Dijkstra’s algorithm

Dijkstra’s algorithm is an algorithm that is used to find the shortest path between vertices in a graph. This algorithm is particularly applicable to weighted graphs where edges have specific costs or weights, which are considered non-negative. With only non-negative weights, once a node has been reached, any path that revisits this node would necessarily be longer. If negative weights are introduced, a subsequent path could loop back to a visited node and potentially reduce the total path cost, which violates the algorithm's assumptions. The steps of this algorithm is shown below:

1. All nodes in the graph, excluding the source node, are initially marked as unvisited and are stored in an unvisited set.
2. Each node is assigned a tentative distance value which is initially set to either 0 for the source node (as the distance from the node to itself is 0) or infinity for the other nodes which helps indicate that they have not yet been part of the path.
3. The source node is chosen as the current node to start the algorithm. From the current node, the algorithm iterates through the neighboring unvisited nodes.

4. It greedily calculates the distance required to reach each neighboring node and compares it with the current distance. If the calculated distance is smaller, the current node is updated to the neighboring node with the smallest tentative distance. After considering all possible unvisited nodes, the current node is marked as visited and removed from the unvisited set.

This process of greedily finding the shortest path continues indefinitely until one of two situations occurs, causing the algorithm to terminate:

- The destination node has been marked as visited, indicating that the shortest path to that node has been determined.
- The tentative distance between nodes is infinity, which occurs only when there is no possible connection between them.[1]

Example of using the algorithm

Consider the following weighted undirected graph in Figure 2 which I decided to draw for ease of understanding and better visualization.

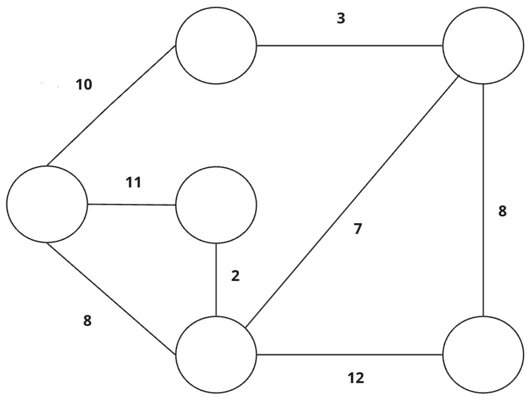


Figure 2. An example of a weighted undirected graph

In Figure 2, the first node is connected to other nodes through edges with costs of 8, 10, and 11. However, for all other nodes, their distances are initially set to infinity as there is no determined path yet. This is depicted accurately in the figure.

[1] "Greedy Algorithm". Programiz.Com, 2023, https://www.programiz.com/dsa/greedy-algorithm. Accessed 31 Aug 2023.

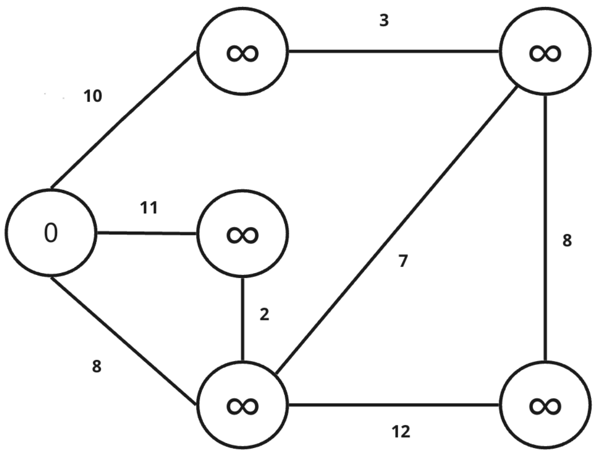


Figure 3. Graph with vertices named infinity and a source node of 0.

In Figure 3, node 0 is considered the current node, while the remaining nodes are part of the unvisited set. Starting from node 0, the algorithm calculates the tentative distance to each neighboring node and compares it with the current distance to greedily determine the shortest path. This process allows for the continuous refinement of the path.

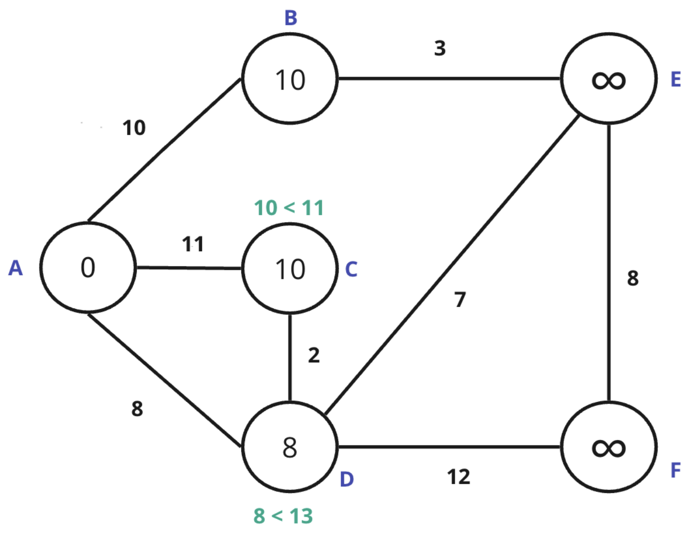


Figure 4. Updated graph with valued vertices indicating the shortest path possible to each of them

Here, it can be seen the comparison between the distance 10<11 at node 'C'. Similarly, when considering the node labeled 'C', the neighboring nodes are evaluated and assigned tentative distances. However, if the tentative distance of an adjacent vertex is already smaller than the new tentative distance, it is not updated. This process is illustrated in Figure 5 which I drew.

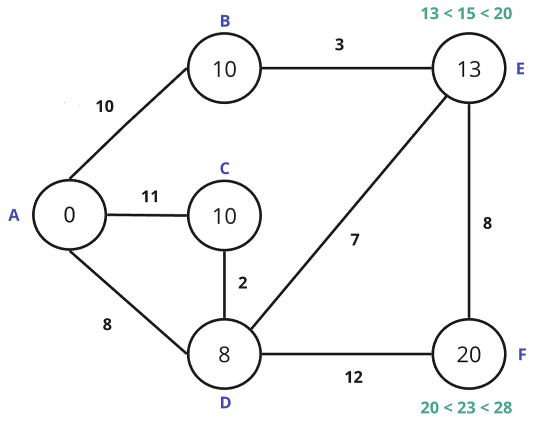


Figure 5. Final graph showing the shortest path possible to all nodes from the source node 'A'

Since Dijkstra's algorithm is proven to work, I googled through the web to find its proof. After reading and understanding the proof, I've compiled my own proof of Dijkstra's algorithm using contradiction. I chose to prove by contradiction because I found it more intuitive for me to prove the Dijkstra Algorithm's nature to not accept negative cycles.

<u>Proving Dijkstra's Algorithm by Contradiction</u>

Assumption for Contradiction

Assume that for a vertex u such that when it is added to the ShortestDistanceSet (the set of nodes for which the shortest distance is found), the calculated distance ($D(S,\ u)$) is not the actual shortest distance ($\delta(S,\ u)$) where S is the source vertex.

For this scenario, lets assume:

- Assume there's a case that there exists a first vertex x for which the incorrect scenario happens.
- Let z be the first vertex on this path not yet in the ShortestDistanceSet, and y be the immediate predecessor of z on the path.
- Due to the assumption, we can assume that $D(S,\ y) = \delta(S,\ y)$ because y is included in the DistanceSet before x where x is the first vertex for which the incorrect scenario happens.

Proving based on the set of assumptions:

- The distance $D(S, z)$ can be expressed as $D(S, y) + weight(y, z)$

- Dijkstra's algorithm selects the next node to include in the ShortestDistanceSet based on the smallest known distance. When it selects x, it means that $D(S, x)$ is the smallest among all nodes not yet in the ShortestDistanceSet.
- Hence, $D(S, x) \leq D(S, z)$ (Since if $D(S, z)$ were smaller, z would have been chosen first)
- In graph theory, it is shown that any sub-path of a shortest path is itself a shortest path. Hence, the path from S to x can be broken down from S to z and z to x
- Hence, we can deduce that $\delta(S,\ x) = \delta(S,\ z) + \delta(z,\ x)$
- From $D(S, x) \leq D(S, z)$ where $D(S, z) = \delta(S,\ y) + weight(y, z)$

  $\Rightarrow \delta(S,\ y) + weight(y, z) + \delta(z,\ x) \leq \delta(S,\ y) + weight(y, z)$

  which is false since $\delta(z,\ x)$ is positive (since Dijkstra's algorithm can only handles positive weights) which in turn would result in the LHS being larger than equal to RHS.

Hence, Dijkstra's algorithm is proven by contradiction

Extension of Proof (Proving that Dijkstra's Algorithm won't work with negative weights):

Now consider the scenario where negative weights are introduced:

Suppose there exists an edge (y, z) with a negative weight in the graph, such that y is already in the ShortestDistanceSet but z is not. According to Dijkstra's algorithm, the tentative distance to z through y denoted $D(S, z)$, would be updated as $D(S, y) + weight(y, z)$. Because $weight(y, z)$ is negative, $D(S, z)$ could potentially become less than $D(S, y)$, suggesting that a shorter path has been found after y was included in the ShortestDistanceSet. This stands in contrast to Dijkstra's algorithm, which does not revisit nodes after their shortest paths are found; instead, it chooses the node with the smallest tentative distance to include in the ShortestDistanceSet. The algorithm is not meant to update the distances of vertices that are already in the ShortestDistanceSet, which is what would happen if the weights were negative. This would cause Dijkstra's algorithm to malfunction, resulting in an inconsistent shortest path calculation. Hence, we can say that Dijkstra's algorithm cannot be applied to graphs where negative edge weights exist because they contradict the presumptions on which it is based.

As we've been taught in our Mathematics course to use counterexamples to further prove or disprove a mathematical assertion, I've decided to use counterexamples to further solidify that this algorithm is not meant to be used with negative weights.

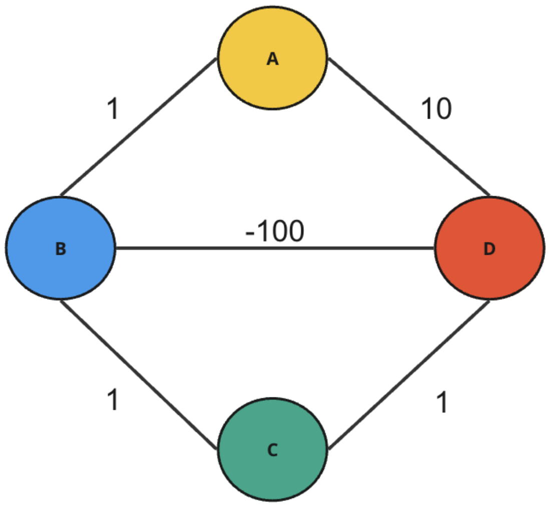


Figure 6. Counterexample of Dijkstra's Algorithm with negative weights

Dijkstra's algorithm assumes that once a node has been "visited", then this path is the final shortest path to that node. This assumption holds for graphs with non-negative edge weights because once a node has been visited, no shorter path to it can be found. Above, I've drawn a simple example of Dijkstra's algorithm with negative weights. This counter-example can be explained with the following:

1. Initialization: All nodes are marked as unvisited. The tentative distance is set to 0 for the source node A and infinity for the other nodes (B, C, and D)
2. Starting from A: The algorithm updates the distance to the node B to 1 $(0 + 1)$ and the distance to D to 10 $(0 + 10)$. B and D are still unvisited.
3. Visiting B: The algorithm considers B as the current node and updates the distance to C to 2 $(1 + 1)$. It also checks the distance to D through B, which would be -99 $(1 - 100)$. However, since Dijkstra's algorithm is a greedy algorithm and does not revisit nodes, it would not update the distance to D as it has already been set to 10.
4. Visiting C: The algorithm considers C as the current node and updates the distance to D to 3 $(2 + 1)$.

The algorithm would terminate after visiting all nodes, incorrectly finding the shortest path from $A \to C \to D$ with a distance of 3 instead of the correct path of $A \to B \to D$ with a distance of -99 for finding the shortest path between nodes A and D. This counterexample proves that Dijkstra's algorithm won't work with negative weights.

Kruskal’s Algorithm

Kruskal’s algorithm is a minimum spanning tree algorithm that focuses on finding a subset of edges in a graph that forms a tree that includes all vertices with a minimum sum of weights from all the edges in the new graph. Kruskal’s algorithm focuses on weighted undirected graphs[2]. In Kruskal's algorithm, the functionality is relatively straightforward, employing a greedy approach. Initially, each vertex is treated as a separate tree (a tree is an undirected graph in which any two vertices are connected by exactly one path which also means that it has no cycles[3]). The algorithm proceeds by iteratively selecting the edge with the minimum weight from the available edges and adding it to the growing forest. This edge is chosen only if it does not create a cycle with the edges already included in the forest. The process is repeated until all the vertices are part of a single tree, resulting in the minimum spanning tree. Unlike Dijkstra's algorithm, which focuses on finding the shortest path from a source vertex to all other vertices, Kruskal's algorithm concentrates on constructing a tree that spans all vertices with the least total weight. This makes Kruskal's algorithm suitable for solving problems involving network connectivity, such as determining the minimum cost to connect a set of locations.

Flowchart of Kruskal’s Algorithm

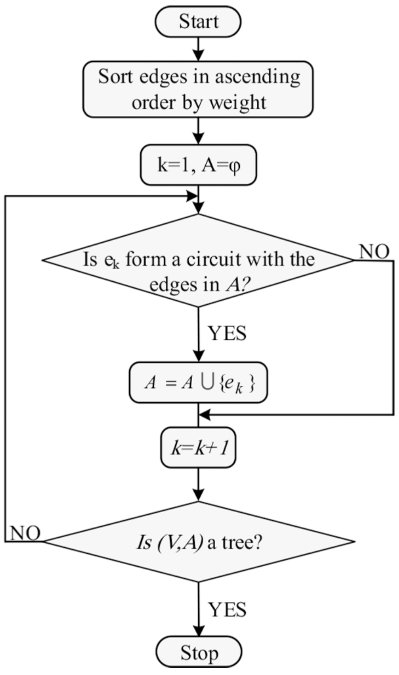

Start
Sort edges in ascending order by weight
k=1, A=φ
Is e_k form a circuit with the edges in *A?*
NO
YES
$A = A \cup \{e_k\}$
k=k+1
NO
Is (V,A) a tree?
YES
Stop

Figure 13. Flowchart of Kruskal’s algorithm[4]

[2] “Find Shortest Paths from Source to All Vertices Using Dijkstra’s Algorithm.” GeeksforGeeks, 28 Mar. 2023, www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/. Accessed 14 June 2023.
[3] romin_va. “Difference between Graph and Tree.” GeeksforGeeks, GeeksforGeeks, 28 Feb. 2023, www.geeksforgeeks.org/difference-between-graph-and-tree/.
[4] Iraji, Farzad, and Ebrahim Farjah. Figure 3. Visualization of Kruskal’s Algorithm., www.researchgate.net/figure/Visualization-of-Kruskals-algorithm_fig2_43952602. Accessed 11 Feb. 2024.

Example of using the algorithm

To illustrate the algorithm, consider the illustration shown in Figure 7-10 which I drew using Miro and assigning it random weights with a minimum amount of nodes for ease of understanding.

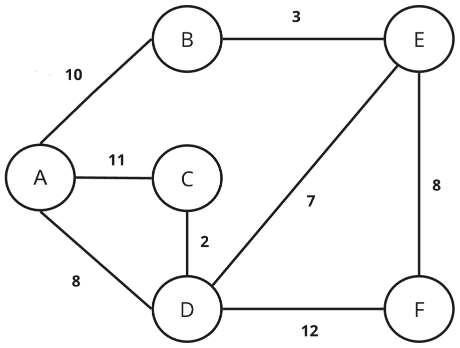


Figure 7. Example of a weighted undirected graph

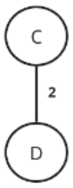


Figure 8. Since the edges are sorted, this edge (connecting nodes C and D) will be chosen as it has the smallest weight

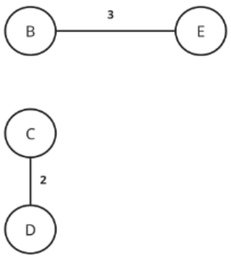


Figure 9. Since the edges are sorted, this edge will be chosen as this is the second-most minimum edge

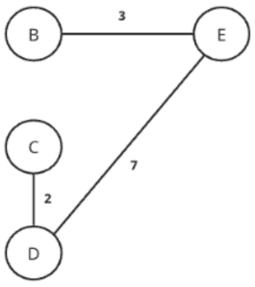


Figure 10. Third most minimum edge

Here, I've drawn Figures 8, 9, and 10, showing the process of selection of nodes in Kruskal's algorithm where Figure 8 → Figure 9 → Figure 10. This process is repeated and will result in a minimum spanning tree (MST).

Similar to Dijkstra's algorithm proof, I was searching the web regarding the proof of Kruskal's algorithm and concluded on my own way of proof. Although there was some proof using contradiction, I've decided to use induction. This is because of the algorithm's nature which is step-wise (incrementally constructing a solution). It also emphasizes that the algorithm maintains correctness at each stage which is a key aspect of ensuring the overall correctness of the MST produced. By proving that each incremental step preserves the MST properties, I figured it would display effectively that Kruskal's algorithm can't "go wrong" as it builds the MST.

Proof of Kruskal's Algorithm by Induction

Base Case

An empty set of edges, T, is an MST due to the fact that a set that doesn't have any edges can't possibly violate MST properties.

Inductive Hypothesis

Assume that after selecting k edges, the set T is part of some MST. These k edges do not form any cycles and each edge is the smallest that connects two different nodes.

Inductive Step

Adding the $(k + 1)^{th}$ edge (in this case let's say edge e), is the smallest edge that connects any two components that are currently not connected in T. Since the edge e connects two separate nodes, adding it to T does not create a cycle. The edge e is the smallest edge weight connecting two nodes which aligns with the greedy choice property. Hence, adding e to T maintain the MST property.

Hence, by mathematical induction, since the base case is true and the algorithm holds true for $k + 1$ edges assuming that the algorithm is true for k edges, Kruskal's algorithm successfully creates a minimum spanning tree since each edge is chosen based on the greedy method (smallest weights between two nodes).

Jakarta's Map

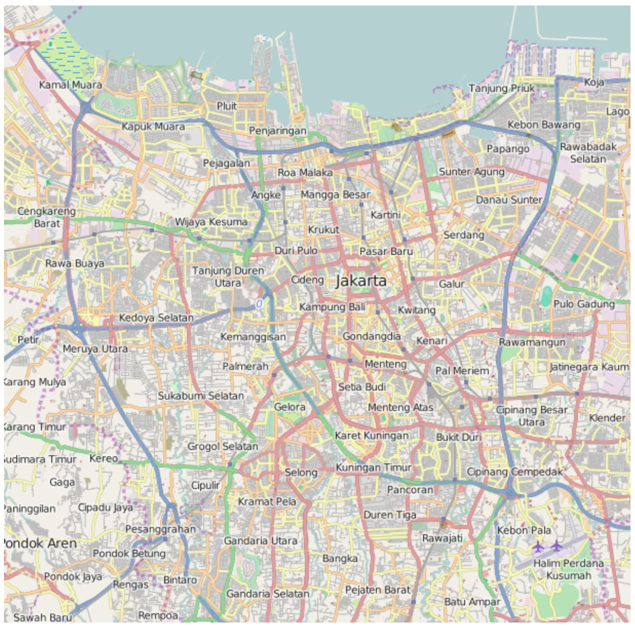


Figure 11. Map of Jakarta[5]

[5] Farras. "Berkas:Indonesia Jakarta Location Map.Svg." Wikipedia, Wikimedia Foundation, id.wikipedia.org/wiki/Berkas:Indonesia_Jakarta_location_map.svg. Accessed 11 Feb. 2024.

Plotting map into a graph manually. Distance between nodes are determined using Google Maps.

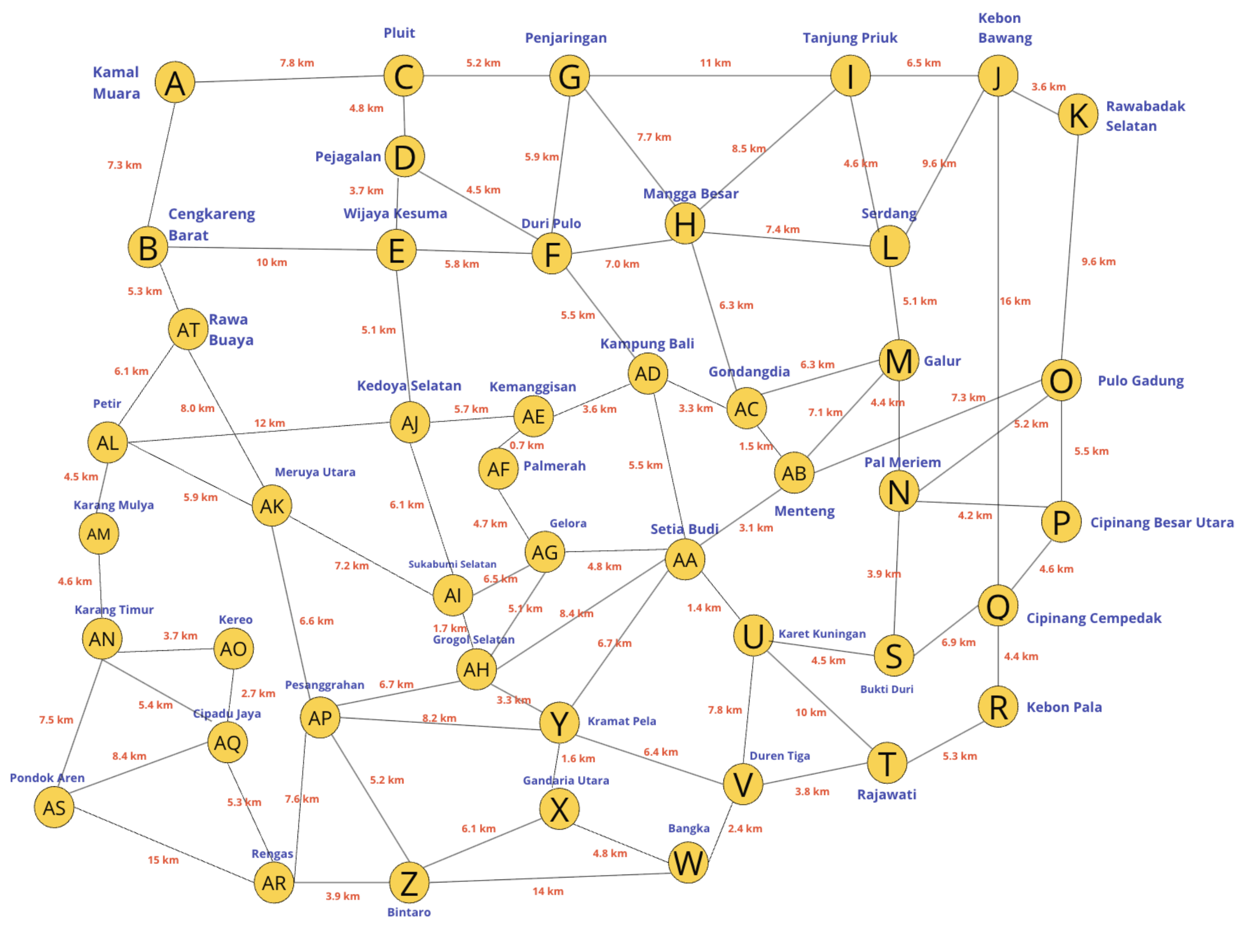


Figure 12. Plotting map into a graph manually by using Figure 11 as a reference (done by putting Figure 11 as a background image and overlapping it using circles and connection lines). I labeled each node as letters for simplification in representation for explanation throughout this exploration.

Due to the fact that I couldn't find any mapped-out Jakarta maps with nodes and their respective shortest distance between nodes, I've graphed the whole Jakarta map based on Figure 11 on a website called Miro. I graphed each street with a circle by using the map as a background. After graphing each node in Figure 11, I manually labeled each of the nodes with their names and variables. I also connected their respective connections while graphing the nodes and manually Google Mapped the distance between each node. Since I googled map the distance between each node, this distance between each node is already the shortest distance as Google Maps uses the Dijkstra algorithm too. Hence, it can be concluded that the system of nodes above is efficient and usable for this exploration. Due to the immense amount of nodes, this exploration will first explore the smaller nodes using mathematical approach and then proceed to use programming to find both Dijkstra's and Kruskal's algorithm for all 46 nodes in the graph.

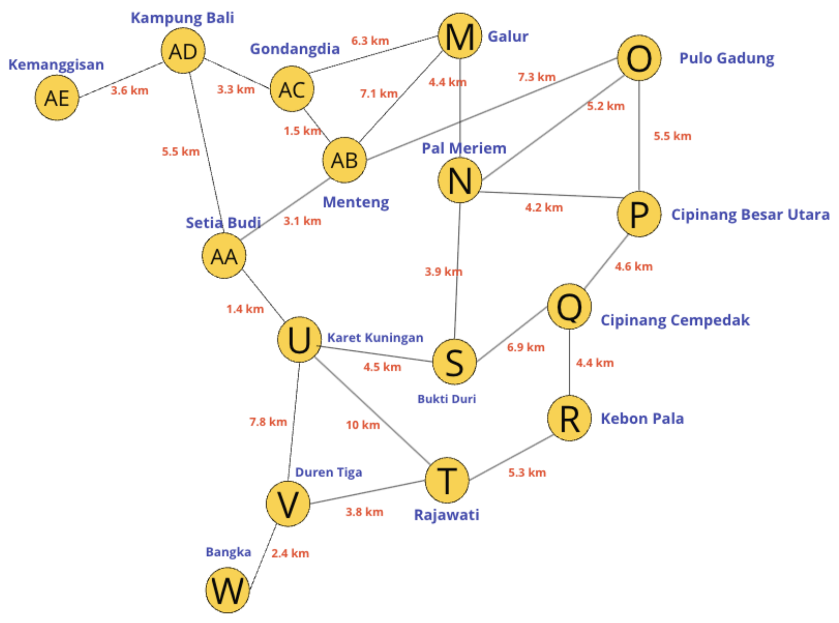


Figure 13. The simplified version of Jakarta's map (I didn't change the representation of letters for consistency purposes)

Here, I decided to reduce the number of nodes from approximately 46 to 16 nodes for ease of explanation and presentation of finding the shortest paths with Dijkstra's and Kruskal's Algorithms as shown in Figure 13. These 16 nodes represent some of the most frequented routes in Jakarta and will be used for the following sections.

Dijkstra's Algorithm

Visited	AE	AD	AA	AC	AB	M	N	O	S	P	Q	R	U	T	V	W
{AE}	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
{AE, AD}	0	3.6	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
{AE, AD, AA}	0	3.6	9.1	6.9	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
{AE, AD, AA, AC}	0	3.6	9.1	6.9	8.4	13.2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
{AE, AD, AA, AC, AB}	0	3.6	9.1	6.9	8.4	13.2	∞	15.7	∞	∞	∞	∞	∞	∞	∞	∞
{AE, AD, AA, AC, AB, M}	0	3.6	9.1	6.9	8.4	13.2	∞	15.7	∞	∞	∞	∞	10.5	∞	∞	∞
{AE, AD, AA, AC, AB, M, O}	0	3.6	9.1	6.9	8.4	13.2	∞	15.7	15.0	∞	∞	∞	10.5	20.5	18.3	∞
{AE, AD, AA, AC, AB, M, O, U}	0	3.6	9.1	6.9	8.4	13.2	17.6	15.7	15.0	∞	∞	∞	10.5	20.5	18.3	∞
{AE, AD, AA, AC, AB, M, O, U, S}	0	3.6	9.1	6.9	8.4	13.2	17.6	15.7	15.0	∞	21.9	∞	10.5	20.5	18.3	∞
{AE, AD, AA, AC, AB, M, O, U, S, T}	0	3.6	9.1	6.9	8.4	13.2	17.6	15.7	15.0	21.2	21.9	∞	10.5	20.5	18.3	∞
{AE, AD, AA, AC, AB, M, O, U, S, T, V}	0	3.6	9.1	6.9	8.4	13.2	17.6	15.7	15.0	21.2	21.9	∞	10.5	20.5	18.3	∞
{AE, AD, AA, AC, AB, M, O, U, S, T, V, N}	0	3.6	9.1	6.9	8.4	13.2	17.6	15.7	15.0	21.2	21.9	∞	10.5	20.5	18.3	20.7
{AE, AD, AA, AC, AB, M, O, U, S, T, V, N, Q}	0	3.6	9.1	6.9	8.4	13.2	17.6	15.7	15.0	21.2	21.9	25.8	10.5	20.5	18.3	20.7
{AE, AD, AA, AC, AB, M, O, U, S, T, V, N, Q, P}	0	3.6	9.1	6.9	8.4	13.2	17.6	15.7	15.0	21.2	21.9	25.8	10.5	20.5	18.3	20.7
{AE, AD, AA, AC, AB, M, O, U, S, T, V, N, Q, P, W}	0	3.6	9.1	6.9	8.4	13.2	17.6	15.7	15.0	21.2	21.9	25.8	10.5	20.5	18.3	20.7
{AE, AD, AA, AC, AB, M, O, U, S, T, V, N, Q, P, W, R}	0	3.6	9.1	6.9	8.4	13.2	17.6	15.7	15.0	21.2	21.9	25.8	10.5	20.5	18.3	20.7

Figure 14. Table of process of Dijkstra's Algorithm

Here, I've selected the node AE as the source node which means that every distance here that is not infinity represents the shortest distance between node AE and its respective nodes which is represented with the header of each column. This source node is represented by marking each row with the header column AE as 0. The 'Visited' header column, as its name implies, marks the process of Dijkstra's algorithm performing shortest path finding as it marks the neighboring unvisited nodes as visited. This process can be annotated as seen in the following:

- Given a weighted graph $G = (V, E)$ with a weight function $w: E \rightarrow \mathbb{R}$ (as the edges are real number weights), and a source vertex $s \in V$, Dijkstra's algorithm constructs a shortest-path three from the source, which includes the shortest path to all other vertices in the graph. To denote:
    - S: Set of vertices whose final shortest-path weight from the source, s, is already determined.
    - $d[v]$: The current shortest-path weight from s to vertex v.
    - $\pi[v]$: The predecessor vertex of v in the shortest-path tree.
- As explained in the earlier sections of this exploration, we set $d[v]$ to infinity for all $v \neq s$ and $d[s] = 0$ (shortest path from source to source is 0). The set S is empty and $\pi[v]$ is undefined for all v. The algorithm proceeds in rounds, where in each round it selects a vertex $u \notin S$ with the minimum $d[u]$ and adds u to S. For each vertex v adjacent to u (denoted as $(u, v) \in E$), it "relaxes" the edge (u, v), updating $d[v]$ and $\pi[v]$ if a shorter path from s to v is encountered through u. "Relaxation," which is used to "greedily" assign the shortest path to respective nodes by continuously updating values of $d[v]$ and $\pi[v]$ when a shorter path is found which can be given by:
    - If $d[u] + w(u, v) < d[v]$, then set $d[v] = d[u] + w(u, v)$ and $\pi[v] = u$
- The table (Figure 14) above shows this process, with each row representing the state of S and $d[v]$ after each vertex is added to S. For example, the row corresponding to the set $\{AE,\ AD,\ AA\}$ shows the distances from s (which is AE for this case) to all vertices after vertices $AE, AD,$ and AA have been added to S. Each cell $d[v]$ in the row is the best-known shortest-path weight from s to v after considering all vertices in S up to that point. The infinity symbol (∞) indicates that the vertex has not been reached yet or no shorter path can be found. The final row, where $S = V$, represents the competition of the algorithm, with $d[v]$ containing the weight of the shortest path from s to v for all vertices v.

Sample Calculation:

As explained, Figure 14 uses AE as the source node making the numbers on the table represent the shortest distance between AE and its respective nodes. For instance, the shortest distance between AE and AB is 8.4 km, as immediately detectable in the table (which is also the reason why I took this as an obvious example). This can be done by comparing the path $AE \rightarrow AD \rightarrow AA \rightarrow AB\ >\ AE \rightarrow AD \rightarrow AC \rightarrow AB$ in which can be calculated with $3.6 + 5.5 + 3.1\ >\ 3.6 + 3.3 + 1.5$ with $12.2\ > 8.4$ which is the reason why 8.4 km is correctly deduced and presented on the table. This further proves that the proof in the earlier section is correct and that the algorithm does find the shortest path between nodes. So, with the fact that this algorithm is working and the table displays the correct information, we can deduce that the farthest distances between two nodes is from AE to T on 25.8 km. This also means that Kemanggisan is 25.8 km from Kebon Pala and drivers might want to reconsider travelling through these two nodes just for one delivery.

However, the table in Figure 14 has inabilities in deducing distances between a node other than the source, AE, and another node that is also other than the source, AE. This is done on purpose because I wanted to show the process of finding shortest path of one node through all nodes instead of mapping all of them one by one (which I would also do in a later part of this investigation when using all 46 nodes).

Kruskal's Algorithm

Steps	Edge Added	Weight
1	AA-U	1.4
2	AC-AB	1.5
3	V-W	2.4
4	AB-AA	3.1
5	AD-AC	3.3
6	AE-AD	3.6
7	T-V	3.8
8	N-S	3.9
9	N-P	4.2
10	M-N	4.4
11	Q-R	4.4
12	U-S	4.5
13	P-Q	4.6
14	N-O	5.2
15	T-R	5.3
16	U-B	7.8

Figure 15. Table of process of Kruskal's Algorithm

Here, I've drawn the table of process of Kruskal's Algorithm of selection of edges with minimum weights. This process can be seen from below:

1. Firstly, we initialize the forest (which was defined in earlier sections). Each node starts as a disjoint set.

   $Forest = \{\{AA\}, \{U\}, \{AC\}, \{AB\}, \{V\}, \{W\}, \{AD\}, \{AE\}, \{T\}, \{N\}, \{S\}, \{P\}, \{Q\}, \{R\}, \{B\}, \{M\}, \{O\}\}$

2. Edges are then ordered from the smallest to the largest weight.

   $E = \{(AA, U), (AC, AB), (V, W), (AB, AA), (AD, AC), (AE, AD), \ldots\}$

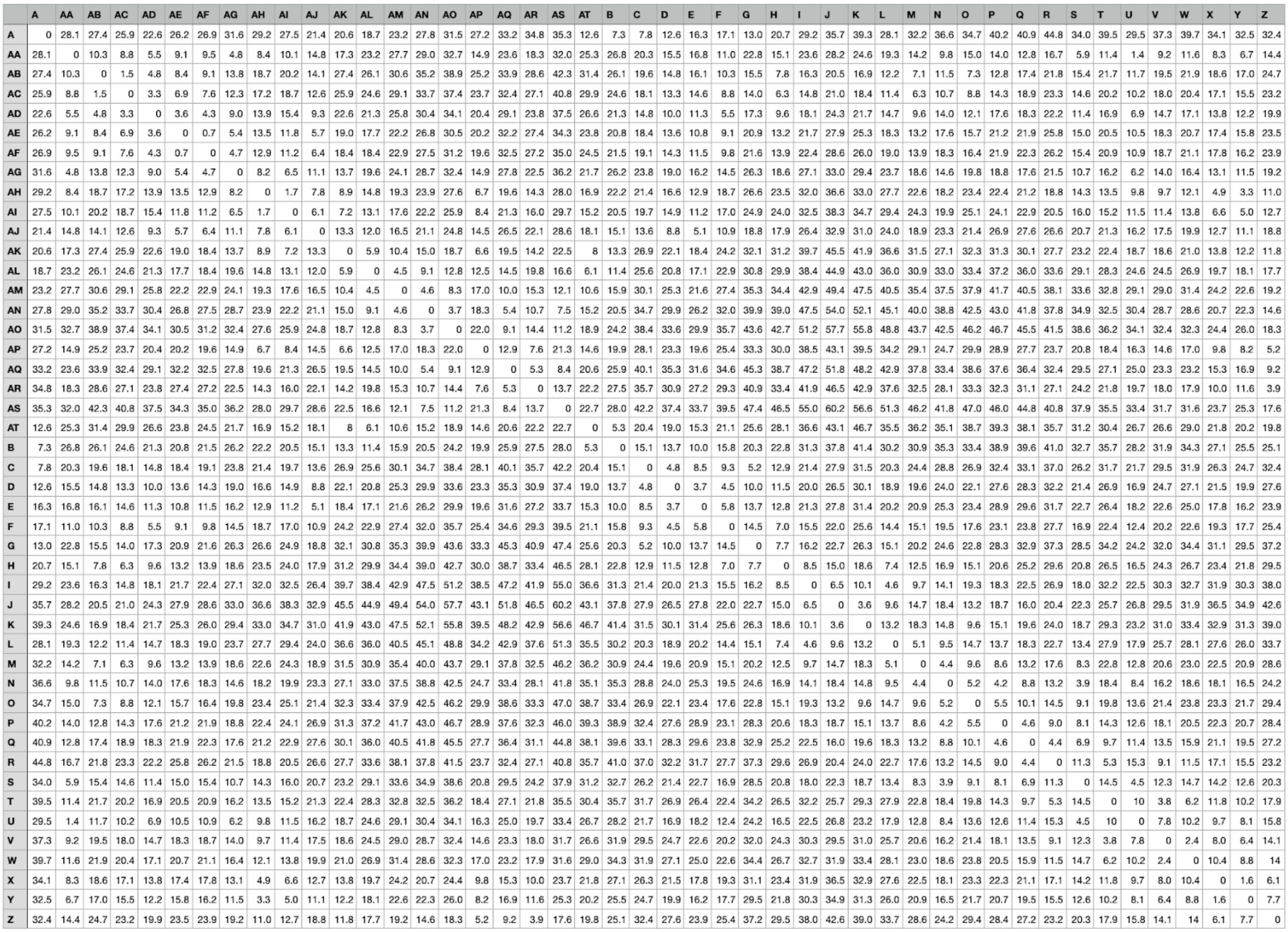
3. For each edge (u, v) with weight w the following condition applies:
   - If u and v belong to different subsets (no cycle is formed), the edge is added to the MST.

4. We iterate through the sorted edge list and apply union-find operations:
   - For edge (AA, U) with weight 1.4, since AA and U are in different subsets, we add it to the MST and combine these subsets.
   - We continue with (AC, AB) with weight 1.5, (V, W) with weight 2.4, and so on, only adding edges that do not close a cycle.

5. The algorithm concludes once all vertices are connected, which may occur before all edges are considered if a spanning tree is already formed.

Hence, Kruskal's algorithm would form the MST in the order of edges listed in the table (Figure 15), starting with the edge between nodes AA and U with a weight of 1.4 kilometers, followed by AC-AB with 1.5 kilometers, and so forth, always ensuring that each new edge connects two previously unconnected components of the graph, thus avoiding any cycles. The process continues sequentially, with the final MST including edges up to U-B with a weight of 7.8 kilometers.

<u>Dijkstra's Algorithm in Finding the Shortest Distance Path Between Nodes for all 46 Nodes</u>



	A	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM	AN	AO	AP	AQ	AR	AS	AT	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	0	28.1	27.4	25.9	22.6	26.2	26.9	31.6	29.2	27.5	21.4	20.6	18.7	23.2	27.8	31.5	27.2	33.2	34.8	35.3	12.6	7.3	7.8	12.6	16.3	17.1	13.0	20.7	29.2	35.7	39.3	28.1	32.2	36.6	34.7	40.2	40.9	44.8	34.0	39.5	29.5	37.3	39.7	34.1	32.5	32.4
AA	28.1	0	10.3	8.8	5.5	9.1	9.5	4.8	8.4	10.1	14.8	17.3	23.2	27.7	29.0	32.7	14.9	23.6	18.3	32.0	25.3	26.8	20.3	15.5	16.8	11.0	22.8	15.1	23.6	28.2	24.6	19.3	14.2	9.8	15.0	14.0	12.8	16.7	5.9	11.4	1.4	9.2	11.6	8.3	6.7	14.4
AB	27.4	10.3	0	1.5	4.8	8.4	9.1	13.8	18.7	20.2	14.1	27.4	26.1	30.6	35.2	38.9	25.2	33.9	28.6	42.3	31.4	26.1	19.6	14.8	16.1	10.3	15.5	7.8	16.3	20.5	16.9	12.2	7.1	11.5	7.3	12.8	17.4	21.8	15.4	21.7	11.7	19.5	21.9	18.6	17.0	24.7
AC	25.9	8.8	1.5	0	3.3	6.9	7.6	12.3	17.2	18.7	12.6	25.9	24.6	29.1	33.7	37.4	23.7	32.4	27.1	40.8	29.9	24.6	18.1	13.3	14.6	8.8	14.0	6.3	14.8	21.0	18.4	11.4	6.3	10.7	8.8	14.3	18.9	23.3	14.6	20.2	10.2	18.0	20.4	17.1	15.5	23.2
AD	22.6	5.5	4.8	3.3	0	3.6	4.3	9.0	13.9	15.4	9.3	22.6	21.3	25.8	30.4	34.1	20.4	29.1	23.8	37.5	26.6	21.3	14.8	10.0	11.3	5.5	17.3	9.6	18.1	24.3	21.7	14.7	9.6	14.0	12.1	17.6	18.3	22.2	11.4	16.9	6.9	14.7	17.1	13.8	12.2	19.9
AE	26.2	9.1	8.4	6.9	3.6	0	0.7	5.4	13.5	11.8	5.7	19.0	17.7	22.2	26.8	30.5	20.2	32.2	27.4	34.3	23.8	20.8	18.4	13.6	10.8	9.1	20.9	13.2	21.7	27.9	25.3	18.3	13.2	17.6	15.7	21.2	21.9	25.8	15.0	20.5	10.5	18.3	20.7	17.4	15.8	23.5
AF	26.9	9.5	9.1	7.6	4.3	0.7	0	4.7	12.9	11.2	6.4	18.4	18.4	22.9	27.5	31.2	19.6	32.5	27.2	35.0	24.5	21.5	19.1	14.3	11.5	9.8	21.6	13.9	22.4	28.6	26.0	19.0	13.9	18.3	16.4	21.9	22.3	26.2	15.4	20.9	10.9	18.7	21.1	17.8	16.2	23.9
AG	31.6	4.8	13.8	12.3	9.0	5.4	4.7	0	8.2	6.5	11.1	13.7	19.6	24.1	28.7	32.4	14.9	27.8	22.5	36.2	21.7	26.2	23.8	19.0	16.2	14.5	26.3	18.6	27.1	33.0	29.4	23.7	18.6	14.6	19.8	18.8	17.6	21.5	10.7	16.2	6.2	14.0	16.4	13.1	11.5	19.2
AH	29.2	8.4	18.7	17.2	13.9	13.5	12.9	8.2	0	1.7	7.8	8.9	14.8	19.3	23.9	27.6	6.7	19.6	14.3	28.0	16.9	22.2	21.4	16.6	12.9	18.7	26.6	23.5	32.0	36.6	33.0	27.7	22.6	18.2	23.4	22.4	21.2	18.8	14.3	13.5	9.8	9.7	12.1	4.9	3.3	11.0
AI	27.5	10.1	20.2	18.7	15.4	11.8	11.2	6.5	1.7	0	6.1	7.2	13.1	17.6	22.2	25.9	8.4	21.3	16.0	29.7	15.2	20.5	19.7	14.9	11.2	17.0	24.9	24.0	32.5	38.3	34.7	29.4	24.3	19.9	25.1	24.1	22.9	20.5	16.0	15.2	11.5	11.4	13.8	6.6	5.0	12.7
AJ	21.4	14.8	14.1	12.6	9.3	5.7	6.4	11.1	7.8	6.1	0	13.3	12.0	16.5	21.1	24.8	14.5	26.5	22.1	28.6	18.1	15.1	13.6	8.8	5.1	10.9	18.8	17.9	26.4	32.9	31.0	24.0	18.9	23.3	21.4	26.9	27.6	26.6	20.7	21.3	16.2	17.5	19.9	12.7	11.1	18.8
AK	20.6	17.3	27.4	25.9	22.6	19.0	18.4	13.7	8.9	7.2	13.3	0	5.9	10.4	15.0	18.7	6.6	19.5	14.2	22.5	8	13.3	26.9	22.1	18.4	24.2	32.1	31.2	39.7	45.5	41.9	36.6	31.5	27.1	32.3	31.3	30.1	27.7	23.2	22.4	18.7	18.6	21.0	13.8	12.2	11.8
AL	18.7	23.2	26.1	24.6	21.3	17.7	18.4	19.6	14.8	13.1	12.0	5.9	0	4.5	9.1	12.8	12.5	14.5	19.8	16.6	6.1	11.4	25.6	20.8	17.1	22.9	30.8	29.9	38.4	44.9	43.0	36.0	30.9	33.0	33.4	37.2	36.0	33.6	29.1	28.3	24.6	24.5	26.9	19.7	18.1	17.7
AM	23.2	27.7	30.6	29.1	25.8	22.2	22.9	24.1	19.3	17.6	16.5	10.4	4.5	0	4.6	8.3	17.0	10.0	15.3	12.1	10.6	15.9	30.1	25.3	21.6	27.4	35.3	34.4	42.9	49.4	47.5	40.5	35.4	37.5	37.9	41.7	40.5	38.1	33.6	32.8	29.1	29.0	31.4	24.2	22.6	19.2
AN	27.8	29.0	35.2	33.7	30.4	26.8	27.5	28.7	23.9	22.2	21.1	15.0	9.1	4.6	0	3.7	18.3	5.4	10.7	7.5	15.2	20.5	34.7	29.9	26.2	32.0	39.9	39.0	47.5	54.0	52.1	45.1	40.0	38.8	42.5	43.0	41.8	37.8	34.9	32.5	30.4	28.7	28.6	20.7	22.3	14.6
AO	31.5	32.7	38.9	37.4	34.1	30.5	31.2	32.4	27.6	25.9	24.8	18.7	12.8	8.3	3.7	0	22.0	9.1	14.4	11.2	18.9	24.2	38.4	33.6	29.9	35.7	43.6	42.7	51.2	57.7	55.8	48.8	43.7	42.5	46.2	46.7	45.5	41.5	38.6	36.2	34.1	32.4	32.3	24.4	26.0	18.3
AP	27.2	14.9	25.2	23.7	20.4	20.2	19.6	14.9	6.7	8.4	14.5	6.6	12.5	17.0	18.3	22.0	0	12.9	7.6	21.3	14.6	19.9	28.1	23.3	19.6	25.4	33.3	30.0	38.5	43.1	39.5	34.2	29.1	24.7	29.9	28.9	27.7	23.7	20.8	18.4	16.3	14.6	17.0	9.8	8.2	5.2
AQ	33.2	23.6	33.9	32.4	29.1	32.2	32.5	27.8	19.6	21.3	26.5	19.5	14.5	10.0	5.4	9.1	12.9	0	5.3	8.4	20.6	25.9	40.1	35.3	31.6	34.6	45.3	38.7	47.2	51.8	48.2	42.9	37.8	33.4	38.6	37.6	36.4	32.4	29.5	27.1	25.0	23.3	23.2	15.3	16.9	9.2
AR	34.8	18.3	28.6	27.1	23.8	27.4	27.2	22.5	14.3	16.0	22.1	14.2	19.8	15.3	10.7	14.4	7.6	5.3	0	13.7	22.2	27.5	35.7	30.9	27.2	29.3	40.9	33.4	41.9	46.5	42.9	37.6	32.5	28.1	33.3	32.3	31.1	27.1	24.2	21.8	19.7	18.0	17.9	10.0	11.6	3.9
AS	35.3	32.0	42.3	40.8	37.5	34.3	35.0	36.2	28.0	29.7	28.6	22.5	16.6	12.1	7.5	11.2	21.3	8.4	13.7	0	22.7	28.0	42.2	37.4	33.7	39.5	47.4	46.5	55.0	60.2	56.6	51.3	46.2	41.8	47.0	46.0	44.8	40.8	37.9	35.5	33.4	31.7	31.6	23.7	25.3	17.6
AT	12.6	25.3	31.4	29.9	26.6	23.8	24.5	21.7	16.9	15.2	18.1	8	6.1	10.6	15.2	18.9	14.6	20.6	22.2	22.7	0	5.3	20.4	19.0	15.3	21.1	25.6	28.1	36.6	43.1	46.7	35.5	36.2	35.1	38.7	39.3	38.1	35.7	31.2	30.4	26.7	26.6	29.0	21.8	20.2	19.8
B	7.3	26.8	26.1	24.6	21.3	20.8	21.5	26.2	22.2	20.5	15.1	13.3	11.4	15.9	20.5	24.2	19.9	25.9	27.5	28.0	5.3	0	15.1	13.7	10.0	15.8	20.3	22.8	31.3	37.8	41.4	30.2	30.9	35.3	33.4	38.9	39.6	41.0	32.7	35.7	28.2	31.9	34.3	27.1	25.5	25.1
C	7.8	20.3	19.6	18.1	14.8	18.4	19.1	23.8	21.4	19.7	13.6	26.9	25.6	30.1	34.7	38.4	28.1	40.1	35.7	42.2	20.4	15.1	0	4.8	8.5	9.3	5.2	12.9	21.4	27.9	31.5	20.3	24.4	28.8	26.9	32.4	33.1	37.0	26.2	31.7	21.7	29.5	31.9	26.3	24.7	32.4
D	12.6	15.5	14.8	13.3	10.0	13.6	14.3	19.0	16.6	14.9	8.8	22.1	20.8	25.3	29.9	33.6	23.3	35.3	30.9	37.4	19.0	13.7	4.8	0	3.7	4.5	10.0	11.5	20.0	26.5	30.1	18.9	19.6	24.0	22.1	27.6	28.3	32.2	21.4	26.9	16.9	24.7	27.1	21.5	19.9	27.6
E	16.3	16.8	16.1	14.6	11.3	10.8	11.5	16.2	12.9	11.2	5.1	18.4	17.1	21.6	26.2	29.9	19.6	31.6	27.2	33.7	15.3	10.0	8.5	3.7	0	5.8	13.7	12.8	21.3	27.8	31.4	20.2	20.9	25.3	23.4	28.9	29.6	31.7	22.7	26.4	18.2	22.6	25.0	17.8	16.2	23.9
F	17.1	11.0	10.3	8.8	5.5	9.1	9.8	14.5	18.7	17.0	10.9	24.2	22.9	27.4	32.0	35.7	25.4	34.6	29.3	39.5	21.1	15.8	9.3	4.5	5.8	0	14.5	7.0	15.5	22.0	25.6	14.4	15.1	19.5	17.6	23.1	23.8	27.7	16.9	22.4	12.4	20.2	22.6	19.3	17.7	25.4
G	13.0	22.8	15.5	14.0	17.3	20.9	21.6	26.3	26.6	24.9	18.8	32.1	30.8	35.3	39.9	43.6	33.3	45.3	40.9	47.4	25.6	20.3	5.2	10.0	13.7	14.5	0	7.7	16.2	22.7	26.3	15.1	20.2	24.6	22.8	28.3	32.9	37.3	28.5	34.2	24.2	32.0	34.4	31.1	29.5	37.2
H	20.7	15.1	7.8	6.3	9.6	13.2	13.9	18.6	23.5	24.0	17.9	31.2	29.9	34.4	39.0	42.7	30.0	38.7	33.4	46.5	28.1	22.8	12.9	11.5	12.8	7.0	7.7	0	8.5	15.0	18.6	7.4	12.5	16.9	15.1	20.6	25.2	29.6	20.8	26.5	16.5	24.3	26.7	23.4	21.8	29.5
I	29.2	23.6	16.3	14.8	18.1	21.7	22.4	27.1	32.0	32.5	26.4	39.7	38.4	42.9	47.5	51.2	38.5	47.2	41.9	55.0	36.6	31.3	21.4	20.0	21.3	15.5	16.2	8.5	0	6.5	10.1	4.6	9.7	14.1	19.3	18.3	22.5	26.9	18.0	32.2	22.5	30.3	32.7	31.9	30.3	38.0
J	35.7	28.2	20.5	21.0	24.3	27.9	28.6	33.0	36.6	38.3	32.9	45.5	44.9	49.4	54.0	57.7	43.1	51.8	46.5	60.2	43.1	37.8	27.9	26.5	27.8	22.0	22.7	15.0	6.5	0	3.6	9.6	14.7	18.4	13.2	18.7	16.0	20.4	22.3	25.7	26.8	29.5	31.9	36.5	34.9	42.6
K	39.3	24.6	16.9	18.4	21.7	25.3	26.0	29.4	33.0	34.7	31.0	41.9	43.0	47.5	52.1	55.8	39.5	48.2	42.9	56.6	46.7	41.4	31.5	30.1	31.4	25.6	26.3	18.6	10.1	3.6	0	13.2	18.3	14.8	9.6	15.1	19.6	24.0	18.7	29.3	23.2	31.0	33.4	32.9	31.3	39.0
L	28.1	19.3	12.2	11.4	14.7	18.3	19.0	23.7	27.7	29.4	24.0	36.6	36.0	40.5	45.1	48.8	34.2	42.9	37.6	51.3	35.5	30.2	20.3	18.9	20.2	14.4	15.1	7.4	4.6	9.6	13.2	0	5.1	9.5	14.7	13.7	18.3	22.7	13.4	27.9	17.9	25.7	28.1	27.6	26.0	33.7
M	32.2	14.2	7.1	6.3	9.6	13.2	13.9	18.6	22.6	24.3	18.9	31.5	30.9	35.4	40.0	43.7	29.1	37.8	32.5	46.2	36.2	30.9	24.4	19.6	20.9	15.1	20.2	12.5	9.7	14.7	18.3	5.1	0	4.4	9.6	8.6	13.2	17.6	8.3	22.8	12.8	20.6	23.0	22.5	20.9	28.6
N	36.6	9.8	11.5	10.7	14.0	17.6	18.3	14.6	18.2	19.9	23.3	27.1	33.0	37.5	38.8	42.5	24.7	33.4	28.1	41.8	35.1	35.3	28.8	24.0	25.3	19.5	24.6	16.9	14.1	18.4	14.8	9.5	4.4	0	5.2	4.2	8.8	13.2	3.9	18.4	8.4	16.2	18.6	18.1	16.5	24.2
O	34.7	15.0	7.3	8.8	12.1	15.7	16.4	19.8	23.4	25.1	21.4	32.3	33.4	37.9	42.5	46.2	29.9	38.6	33.3	47.0	38.7	33.4	26.9	22.1	23.4	17.6	22.8	15.1	19.3	13.2	9.6	14.7	9.6	5.2	0	5.5	10.1	14.5	9.1	19.8	13.6	21.4	23.8	23.3	21.7	29.4
P	40.2	14.0	12.8	14.3	17.6	21.2	21.9	18.8	22.4	24.1	26.9	31.3	37.2	41.7	43.0	46.7	28.9	37.6	32.3	46.0	39.3	38.9	32.4	27.6	28.9	23.1	28.3	20.6	18.3	18.7	15.1	13.7	8.6	4.2	5.5	0	4.6	9.0	8.1	14.3	12.6	18.1	20.5	22.3	20.7	28.4
Q	40.9	12.8	17.4	18.9	18.3	21.9	22.3	17.6	21.2	22.9	27.6	30.1	36.0	40.5	41.8	45.5	27.7	36.4	31.1	44.8	38.1	39.6	33.1	28.3	29.6	23.8	32.9	25.2	22.5	16.0	19.6	18.3	13.2	8.8	10.1	4.6	0	4.4	6.9	9.7	11.4	13.5	15.9	21.1	19.5	27.2
R	44.8	16.7	21.8	23.3	22.2	25.8	26.2	21.5	18.8	20.5	26.6	27.7	33.6	38.1	37.8	41.5	23.7	32.4	27.1	40.8	35.7	41.0	37.0	32.2	31.7	27.7	37.3	29.6	26.9	20.4	24.0	22.7	17.6	13.2	14.5	9.0	4.4	0	11.3	5.3	15.3	9.1	11.5	17.1	15.5	23.2
S	34.0	5.9	15.4	14.6	11.4	15.0	15.4	10.7	14.3	16.0	20.7	23.2	29.1	33.6	34.9	38.6	20.8	29.5	24.2	37.9	31.2	32.7	26.2	21.4	22.7	16.9	28.5	20.8	18.0	22.3	18.7	13.4	8.3	3.9	9.1	8.1	6.9	11.3	0	14.5	4.5	12.3	14.7	14.2	12.6	20.3
T	39.5	11.4	21.7	20.2	16.9	20.5	20.9	16.2	13.5	15.2	21.3	22.4	28.3	32.8	32.5	36.2	18.4	27.1	21.8	35.5	30.4	35.7	31.7	26.9	26.4	22.4	34.2	26.5	32.2	25.7	29.3	27.9	22.8	18.4	19.8	14.3	9.7	5.3	14.5	0	10	3.8	6.2	11.8	10.2	17.9
U	29.5	1.4	11.7	10.2	6.9	10.5	10.9	6.2	9.8	11.5	16.2	18.7	24.6	29.1	30.4	34.1	16.3	25.0	19.7	33.4	26.7	28.2	21.7	16.9	18.2	12.4	24.2	16.5	22.5	26.8	23.2	17.9	12.8	8.4	13.6	12.6	11.4	15.3	4.5	10	0	7.8	10.2	9.7	8.1	15.8
V	37.3	9.2	19.5	18.0	14.7	18.3	18.7	14.0	9.7	11.4	17.5	18.6	24.5	29.0	28.7	32.4	14.6	23.3	18.0	31.7	26.6	31.9	29.5	24.7	22.6	20.2	32.0	24.3	30.3	29.5	31.0	25.7	20.6	16.2	21.4	18.1	13.5	9.1	12.3	3.8	7.8	0	2.4	8.0	6.4	14.1
W	39.7	11.6	21.9	20.4	17.1	20.7	21.1	16.4	12.1	13.8	19.9	21.0	26.9	31.4	28.6	32.3	17.0	23.2	17.9	31.6	29.0	34.3	31.9	27.1	25.0	22.6	34.4	26.7	32.7	31.9	33.4	28.1	23.0	18.6	23.8	20.5	15.9	11.5	14.7	6.2	10.2	2.4	0	10.4	8.8	14
X	34.1	8.3	18.6	17.1	13.8	17.4	17.8	13.1	4.9	6.6	12.7	13.8	19.7	24.2	20.7	24.4	9.8	15.3	10.0	23.7	21.8	27.1	26.3	21.5	17.8	19.3	31.1	23.4	31.9	36.5	32.9	27.6	22.5	18.1	23.3	22.3	21.1	17.1	14.2	11.8	9.7	8.0	10.4	0	1.6	6.1
Y	32.5	6.7	17.0	15.5	12.2	15.8	16.2	11.5	3.3	5.0	11.1	12.2	18.1	22.6	22.3	26.0	8.2	16.9	11.6	25.3	20.2	25.5	24.7	19.9	16.2	17.7	29.5	21.8	30.3	34.9	31.3	26.0	20.9	16.5	21.7	20.7	19.5	15.5	12.6	10.2	8.1	6.4	8.8	1.6	0	7.7
Z	32.4	14.4	24.7	23.2	19.9	23.5	23.9	19.2	11.0	12.7	18.8	11.8	17.7	19.2	14.6	18.3	5.2	9.2	3.9	17.6	19.8	25.1	32.4	27.6	23.9	25.4	37.2	29.5	38.0	42.6	39.0	33.7	28.6	24.2	29.4	28.4	27.2	23.2	20.3	17.9	15.8	14.1	14	6.1	7.7	0

Figure 16. Results of running the Dijkstra Algorithm using code (Appendix 1 code)

Having a passion in Computer Science, I decided to use code to make this table because it includes a large amount of nodes (46 nodes) and would be too laborious to manually do it one by one. This table also generates the shortest distance between any two nodes of the graph, unlike the previous section which is reliant on one source node.

Sample Calculation

Since Dijkstra's Algorithm finds the shortest path between nodes, finding the intersection in the table above would result in the finding of the shortest distance between the two respective headers in a specific cell that is chosen. For instance, when finding node C (Pluit) to node Y (Kramat Pela), one should locate the cell where the row for node C and the column for node Y intersect to find the shortest distance between them which in this case would be. I chose these points as they are quite frequented streets.

Krukal's Algorithm in Finding the Minimum Spanning Tree

Node 1	Node 2	Weight between nodes (kilometers)
A	B	7.3
C	D	4.8
C	G	5.2
B	AT	5.3
AT	AL	6.1
AL	AM	4.5
AL	AK	5.9
AM	AN	4.6
AN	AO	3.7
AN	AQ	5.4
AN	AS	7.5
AQ	AR	5.3
AR	Z	3.9
AP	Z	5.2
Z	X	6.1
Y	X	1.6
Y	AH	3.3
AH	AI	1.7
W	V	2.4
V	T	3.8
T	R	5.3
U	AA	1.4
U	S	4.5
S	N	3.9
AA	AG	4.8
AI	AJ	6.1
AG	AF	4.7
AJ	E	5.1
AE	AF	0.7
AE	AD	3.6
AD	AC	3.3
AD	F	5.5
AC	AB	1.5
AC	H	6.3
M	N	4.4
M	L	5.1
N	P	4.2
N	O	5.2
P	Q	4.6
Q	R	4.4
K	J	3.6
J	I	6.5
I	L	4.6
F	D	4.5
E	D	3.7

Figure 17. Table of Node Relations and its Respective Weights for the Minimum Spanning Tree

This table is also generated with code for the same reasons as the previous section. This table above (Figure 14) shows the MST that would be formed from the highly dense graph. In real-life applications utilizing an MST, which can be made using Kruskal's algorithm as explained throughout this exploration, could lead to more predictable delivery schedules and increased customer satisfaction due to the reduction in delivery time variability.

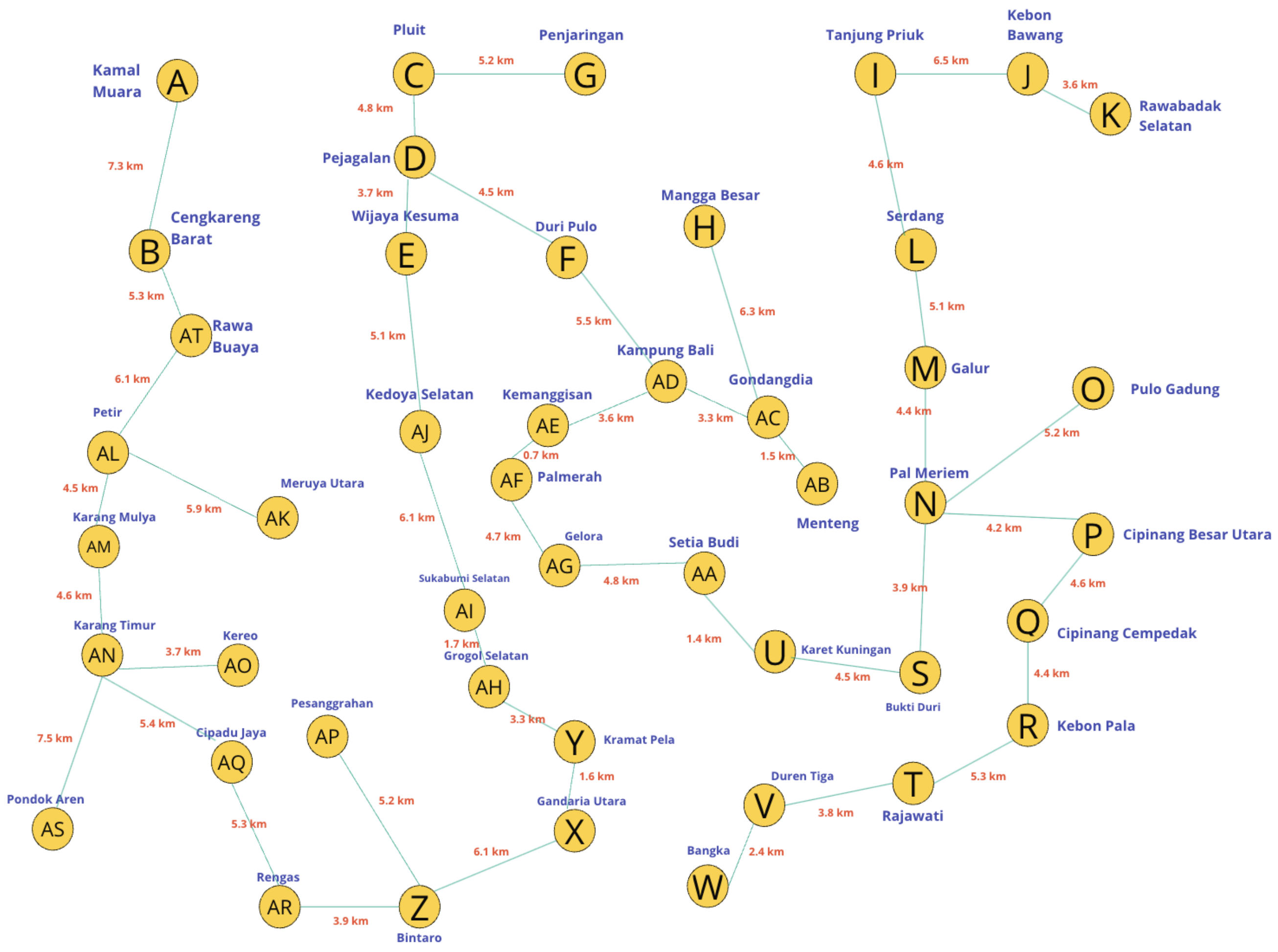


Figure 18. Minimum Spanning Tree using Kruskal's Algorithm

As seen in the above image (Figure 18) which I drew manually by deleting the parts of the main graph (Figure 12), Kruskal's Algorithm successfully makes a minimum spanning tree where all nodes are interconnected. This means that any node in the graph can traverse to any other node, which takes into account that the total weight of this graph is minimum. This interconnectedness ensures that one can travel from any node to another while collectively minimizing the total distance or weight traversed across the entire network. In practical terms, such an MST is invaluable for delivery services. It enables them to design routes that connect all delivery points while minimizing travel distance.

Sample Calculation

Since this Internal Assessment is revolving more around finding the shortest distance between nodes, this sample calculation for MST would be to find the shortest distance between nodes. For instance, taking node C (Pluit) to node Y (Kramat Pela), we would simply add all the weight resulting in:

$$4.8 + 3.7 + 5.1 + 6.1 + 1.7 + 3.3 = 24.7\ km$$

Conclusion

Although it is true that for the sample calculations, both algorithms result in the same shortest distance, it is presumptive to assume that both always result in the same shortest distance. Taking a very obvious example, the shortest distance between nodes 'A' and 'C' is visibly 7.8 km which Dijkstra's Algorithm successfully get. However, due to the minimum spanning tree having the tendency to cut as much connections between nodes to reduce the total weight the graph carries, the connection between node 'A' and 'C' in the MST is non-existent which means that in the MST, to get from node 'A' to 'C', one has to cycle through the graph, resulting in an inefficiency in calculation. To illustrate, it would be:

$7.3 + 5.3 + 6.1 + 4.5 + 4.6 + 5.4 + 5.3 + 3.9 + 6.1 + 1.6 + 3.3 + 1.7 + 6.1 + 5.1 + 3.7 + 4.8 = 74.8\,km$

which is much larger than $7.8\,km$ and hence, it is safe to assume that Dijkstra's algorithm is indeed the better method to find the shortest distance between 2 nodes (which represents locations in a map) compared to Kruskal's algorithm. Krukal's algorithm in the other hand, can be more beneficial in finding the total shortest distance a delivery man needs to traverse each node (although an additional algorithm is needed).

Reflecting on the aim and research question of this investigation, the results show that Dijkstra's algorithm effectively meets this objective by providing the shortest individual routes. However, it is important to acknowledge that real-life route optimization for delivery services may require a combination of both algorithms to handle different scenarios — such as the planning of single deliveries versus multiple deliveries on one route. While the investigation has been successful in applying these algorithms to the simulated road network of Jakarta, some limitations have arisen. I had a hard time mapping every single road in Jakarta for this investigation based on a map and was unsure which streets were connected. In addition, I had to Google Map each connection one by one which was very laborious. However, from an algorithm point of view, Kruskal's algorithm could not efficiently calculate individual shortest paths. To overcome this, future explorations could integrate additional algorithms that complement Kruskal's, such as using Dijkstra's algorithm post-MST creation to refine individual routes or exploring more sophisticated route optimization algorithms that can handle a broader range of constraints. In practice, achieving the ideal balance between total distance minimization and individual route efficiency is complex and may not be entirely captured by traditional graph theory algorithms alone. In the future, I could also consider multiple apps to deduce the distance between nodes like Waze, which would make the distances different. I didn't do it in this exploration due to the time constraint that needed me to compare and contrast between which one to trust and even doing one cycle of Google Mapping each connection of every 46 nodes took me a lot of time. Overall, I’m really happy with the results of this exploration as I reached my initial aim of finding Dijkstra VS. Kruskal’s algorithm for shortest path finding to reduce cost and save time for delivery men. This investigation also helped me to discover even more about graph theorems which are out of the syllabus and are in line with my interest in Computer Science too. It also helped me develop soft skills such as mathematical presentation, problem-solving, critical thinking, research skills, persistence, and analytical skills. I’m looking forward to doing this again in the future and learning and loving math even more.

APPENDIX

"Greedy Algorithm". Programiz.Com, 2023, https://www.programiz.com/dsa/greedy-algorithm. Accessed 31 Aug 2023.

“Find Shortest Paths from Source to All Vertices Using Dijkstra’s Algorithm.” *GeeksforGeeks*, 28 Mar. 2023, www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/. Accessed 14 June 2023.

romin_va. “Difference between Graph and Tree.” GeeksforGeeks, GeeksforGeeks, 28 Feb. 2023, www.geeksforgeeks.org/difference-between-graph-and-tree/.

Iraji, Farzad, and Ebrahim Farjah. Figure 3. Visualization of Kruskal’s Algorithm., www.researchgate.net/figure/Visualization-of-Kruskals-algorithm_fig2_43952602. Accessed 11 Feb. 2024.

Farras. “Berkas:Indonesia Jakarta Location Map.Svg.” Wikipedia, Wikimedia Foundation, id.wikipedia.org/wiki/Berkas:Indonesia_Jakarta_location_map.svg. Accessed 11 Feb. 2024.

“Kruskal’s Minimum Spanning Tree (MST) Algorithm.” *GeeksforGeeks*, 31 Mar. 2023, *www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/.* Accessed 15 June 2023.

	A	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM	AN	AO	AP	AQ	AR	AS	AT	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	0	28.1	27.4	25.9	22.6	26.2	26.9	31.6	29.2	27.5	21.4	20.6	18.7	23.2	27.8	31.5	27.2	33.2	34.8	35.3	12.6	7.3	7.8	12.6	16.3	17.1	13.0	20.7	29.2	35.7	39.3	28.1	32.2	36.6	34.7	40.2	40.9	44.8	34.0	39.5	29.5	37.3	39.7	34.1	32.5	32.4
AA	28.1	0	10.3	8.8	5.5	9.1	9.5	4.8	8.4	10.1	14.8	17.3	23.2	27.7	29.0	32.7	14.9	23.6	18.3	32.0	25.3	26.8	20.3	15.5	16.8	11.0	22.8	15.1	23.6	28.2	24.6	19.3	14.2	9.8	15.0	14.0	12.8	16.7	5.9	11.4	1.4	9.2	11.6	8.3	6.7	14.4
AB	27.4	10.3	0	1.5	4.8	8.4	9.1	13.8	18.7	20.2	14.1	27.4	26.1	30.6	35.2	38.9	25.2	33.9	28.6	42.3	31.4	26.1	19.6	14.8	16.1	10.3	15.5	7.8	16.3	20.5	16.9	12.2	7.1	11.5	7.3	12.8	17.4	21.8	15.4	21.7	11.7	19.5	21.9	18.6	17.0	24.7
AC	25.9	8.8	1.5	0	3.3	6.9	7.6	12.3	17.2	18.7	12.6	25.9	24.6	29.1	33.7	37.4	23.7	32.4	27.1	40.8	29.9	24.6	18.1	13.3	14.6	8.8	14.0	6.3	14.8	21.0	18.4	11.4	6.3	10.7	8.8	14.3	18.9	23.3	14.6	20.2	10.2	18.0	20.4	17.1	15.5	23.2
AD	22.6	5.5	4.8	3.3	0	3.6	4.3	9.0	13.9	15.4	9.3	22.6	21.3	25.8	30.4	34.1	20.4	29.1	23.8	37.5	26.6	21.3	14.8	10.0	11.3	5.5	17.3	9.6	18.1	24.3	21.7	14.7	9.6	14.0	12.1	17.6	18.3	22.2	11.4	16.9	6.9	14.7	17.1	13.8	12.2	19.9
AE	26.2	9.1	8.4	6.9	3.6	0	0.7	5.4	13.5	11.8	5.7	19.0	17.7	22.2	26.8	30.5	20.2	32.2	27.4	34.3	23.8	20.8	18.4	13.6	10.8	9.1	20.9	13.2	21.7	27.9	25.3	18.3	13.2	17.6	15.7	21.2	21.9	25.8	15.0	20.5	10.5	18.3	20.7	17.4	15.8	23.5
AF	26.9	9.5	9.1	7.6	4.3	0.7	0	4.7	12.9	11.2	6.4	18.4	18.4	22.9	27.5	31.2	19.6	32.5	27.2	35.0	24.5	21.5	19.1	14.3	11.5	9.8	21.6	13.9	22.4	28.6	26.0	19.0	13.9	18.3	16.4	21.9	22.3	26.2	15.4	20.9	10.9	18.7	21.1	17.8	16.2	23.9
AG	31.6	4.8	13.8	12.3	9.0	5.4	4.7	0	8.2	6.5	11.1	13.7	19.6	24.1	28.7	32.4	14.9	27.8	22.5	36.2	21.7	26.2	23.8	19.0	16.2	14.5	26.3	18.6	27.1	33.0	29.4	23.7	18.6	14.6	19.8	18.8	17.6	21.5	10.7	16.2	6.2	14.0	16.4	13.1	11.5	19.2
AH	29.2	8.4	18.7	17.2	13.9	13.5	12.9	8.2	0	1.7	7.8	8.9	14.8	19.3	23.9	27.6	6.7	19.6	14.3	28.0	16.9	22.2	21.4	16.6	12.9	18.7	26.6	23.5	32.0	36.6	33.0	27.7	22.6	18.2	23.4	22.4	21.2	18.8	14.3	13.5	9.8	9.7	12.1	4.9	3.3	11.0
AI	27.5	10.1	20.2	18.7	15.4	11.8	11.2	6.5	1.7	0	6.1	7.2	13.1	17.6	22.2	25.9	8.4	21.3	16.0	29.7	15.2	20.5	19.7	14.9	11.2	17.0	24.9	24.0	32.5	38.3	34.7	29.4	24.3	19.9	25.1	24.1	22.9	20.5	16.0	15.2	11.5	11.4	13.8	6.6	5.0	12.7
AJ	21.4	14.8	14.1	12.6	9.3	5.7	6.4	11.1	7.8	6.1	0	13.3	12.0	16.5	21.1	24.8	14.5	26.5	22.1	28.6	18.1	15.1	13.6	8.8	5.1	10.9	18.8	17.9	26.4	32.9	31.0	24.0	18.9	23.3	21.4	26.9	27.6	26.6	20.7	21.3	16.2	17.5	19.9	12.7	11.1	18.8
AK	20.6	17.3	27.4	25.9	22.6	19.0	18.4	13.7	8.9	7.2	13.3	0	5.9	10.4	15.0	18.7	6.6	19.5	14.2	22.5	8	13.3	26.9	22.1	18.4	24.2	32.1	31.2	39.7	45.5	41.9	36.6	31.5	27.1	32.3	31.3	30.1	27.7	23.2	22.4	18.7	18.6	21.0	13.8	12.2	11.8
AL	18.7	23.2	26.1	24.6	21.3	17.7	18.4	19.6	14.8	13.1	12.0	5.9	0	4.5	9.1	12.8	12.5	14.5	19.8	16.6	6.1	11.4	25.6	20.8	17.1	22.9	30.8	29.9	38.4	44.9	43.0	36.0	30.9	33.0	33.4	37.2	36.0	33.6	29.1	28.3	24.6	24.5	26.9	19.7	18.1	17.7
AM	23.2	27.7	30.6	29.1	25.8	22.2	22.9	24.1	19.3	17.6	16.5	10.4	4.5	0	4.6	8.3	17.0	10.0	15.3	12.1	10.6	15.9	30.1	25.3	21.6	27.4	35.3	34.4	42.9	49.4	47.5	40.5	35.4	37.5	37.9	41.7	40.5	38.1	33.6	32.8	29.1	29.0	31.4	24.2	22.6	19.2
AN	27.8	29.0	35.2	33.7	30.4	26.8	27.5	28.7	23.9	22.2	21.1	15.0	9.1	4.6	0	3.7	18.3	5.4	10.7	7.5	15.2	20.5	34.7	29.9	26.2	32.0	39.9	39.0	47.5	54.0	52.1	45.1	40.0	38.8	42.5	43.0	41.8	37.8	34.9	32.5	30.4	28.7	28.6	20.7	22.3	14.6
AO	31.5	32.7	38.9	37.4	34.1	30.5	31.2	32.4	27.6	25.9	24.8	18.7	12.8	8.3	3.7	0	22.0	9.1	14.4	11.2	18.9	24.2	38.4	33.6	29.9	35.7	43.6	42.7	51.2	57.7	55.8	48.8	43.7	42.5	46.2	46.7	45.5	41.5	38.6	36.2	34.1	32.4	32.3	24.4	26.0	18.3
AP	27.2	14.9	25.2	23.7	20.4	20.2	19.6	14.9	6.7	8.4	14.5	6.6	12.5	17.0	18.3	22.0	0	12.9	7.6	21.3	14.6	19.9	28.1	23.3	19.6	25.4	33.3	30.0	38.5	43.1	39.5	34.2	29.1	24.7	29.9	28.9	27.7	23.7	20.8	18.4	16.3	14.6	17.0	9.8	8.2	5.2
AQ	33.2	23.6	33.9	32.4	29.1	32.2	32.5	27.8	19.6	21.3	26.5	19.5	14.5	10.0	5.4	9.1	12.9	0	5.3	8.4	20.6	25.9	40.1	35.3	31.6	34.6	45.3	38.7	47.2	51.8	48.2	42.9	37.8	33.4	38.6	37.6	36.4	32.4	29.5	27.1	25.0	23.3	23.2	15.3	16.9	9.2
AR	34.8	18.3	28.6	27.1	23.8	27.4	27.2	22.5	14.3	16.0	22.1	14.2	19.8	15.3	10.7	14.4	7.6	5.3	0	13.7	22.2	27.5	35.7	30.9	27.2	29.3	40.9	33.4	41.9	46.5	42.9	37.6	32.5	28.1	33.3	32.3	31.1	27.1	24.2	21.8	19.7	18.0	17.9	10.0	11.6	3.9
AS	35.3	32.0	42.3	40.8	37.5	34.3	35.0	36.2	28.0	29.7	28.6	22.5	16.6	12.1	7.5	11.2	21.3	8.4	13.7	0	22.7	28.0	42.2	37.4	33.7	39.5	47.4	46.5	55.0	60.2	56.6	51.3	46.2	41.8	47.0	46.0	44.8	40.8	37.9	35.5	33.4	31.7	31.6	23.7	25.3	17.6
AT	12.6	25.3	31.4	29.9	26.6	23.8	24.5	21.7	16.9	15.2	18.1	8	6.1	10.6	15.2	18.9	14.6	20.6	22.2	22.7	0	5.3	20.4	19.0	15.3	21.1	25.6	28.1	36.6	43.1	46.7	35.5	36.2	35.1	38.7	39.3	38.1	35.7	31.2	30.4	26.7	26.6	29.0	21.8	20.2	19.8
B	7.3	26.8	26.1	24.6	21.3	20.8	21.5	26.2	22.2	20.5	15.1	13.3	11.4	15.9	20.5	24.2	19.9	25.9	27.5	28.0	5.3	0	15.1	13.7	10.0	15.8	20.3	22.8	31.3	37.8	41.4	30.2	30.9	35.3	33.4	38.9	39.6	41.0	32.7	35.7	28.2	31.9	34.3	27.1	25.5	25.1
C	7.8	20.3	19.6	18.1	14.8	18.4	19.1	23.8	21.4	19.7	13.6	26.9	25.6	30.1	34.7	38.4	28.1	40.1	35.7	42.2	20.4	15.1	0	4.8	8.5	9.3	5.2	12.9	21.4	27.9	31.5	20.3	24.4	28.8	26.9	32.4	33.1	37.0	26.2	31.7	21.7	29.5	31.9	26.3	$\boxed{24.7}$	32.4
D	12.6	15.5	14.8	13.3	10.0	13.6	14.3	19.0	16.6	14.9	8.8	22.1	20.8	25.3	29.9	33.6	23.3	35.3	30.9	37.4	19.0	13.7	4.8	0	3.7	4.5	10.0	11.5	20.0	26.5	30.1	18.9	19.6	24.0	22.1	27.6	28.3	32.2	21.4	26.9	16.9	24.7	27.1	21.5	19.9	27.6
E	16.3	16.8	16.1	14.6	11.3	10.8	11.5	16.2	12.9	11.2	5.1	18.4	17.1	21.6	26.2	29.9	19.6	31.6	27.2	33.7	15.3	10.0	8.5	3.7	0	5.8	13.7	12.8	21.3	27.8	31.4	20.2	20.9	25.3	23.4	28.9	29.6	31.7	22.7	26.4	18.2	22.6	25.0	17.8	16.2	23.9
F	17.1	11.0	10.3	8.8	5.5	9.1	9.8	14.5	18.7	17.0	10.9	24.2	22.9	27.4	32.0	35.7	25.4	34.6	29.3	39.5	21.1	15.8	9.3	4.5	5.8	0	14.5	7.0	15.5	22.0	25.6	14.4	15.1	19.5	17.6	23.1	23.8	27.7	16.9	22.4	12.4	20.2	22.6	19.3	17.7	25.4
G	13.0	22.8	15.5	14.0	17.3	20.9	21.6	26.3	26.6	24.9	18.8	32.1	30.8	35.3	39.9	43.6	33.3	45.3	40.9	47.4	25.6	20.3	5.2	10.0	13.7	14.5	0	7.7	16.2	22.7	26.3	15.1	20.2	24.6	22.8	28.3	32.9	37.3	28.5	34.2	24.2	32.0	34.4	31.1	29.5	37.2
H	20.7	15.1	7.8	6.3	9.6	13.2	13.9	18.6	23.5	24.0	17.9	31.2	29.9	34.4	39.0	42.7	30.0	38.7	33.4	46.5	28.1	22.8	12.9	11.5	12.8	7.0	7.7	0	8.5	15.0	18.6	7.4	12.5	16.9	15.1	20.6	25.2	29.6	20.8	26.5	16.5	24.3	26.7	23.4	21.8	29.5
I	29.2	23.6	16.3	14.8	18.1	21.7	22.4	27.1	32.0	32.5	26.4	39.7	38.4	42.9	47.5	51.2	38.5	47.2	41.9	55.0	36.6	31.3	21.4	20.0	21.3	15.5	16.2	8.5	0	6.5	10.1	4.6	9.7	14.1	19.3	18.3	22.5	26.9	18.0	32.2	22.5	30.3	32.7	31.9	30.3	38.0
J	35.7	28.2	20.5	21.0	24.3	27.9	28.6	33.0	36.6	38.3	32.9	45.5	44.9	49.4	54.0	57.7	43.1	51.8	46.5	60.2	43.1	37.8	27.9	26.5	27.8	22.0	22.7	15.0	6.5	0	3.6	9.6	14.7	18.4	13.2	18.7	16.0	20.4	22.3	25.7	26.8	29.5	31.9	36.5	34.9	42.6
K	39.3	24.6	16.9	18.4	21.7	25.3	26.0	29.4	33.0	34.7	31.0	41.9	43.0	47.5	52.1	55.8	39.5	48.2	42.9	56.6	46.7	41.4	31.5	30.1	31.4	25.6	26.3	18.6	10.1	3.6	0	13.2	18.3	14.8	9.6	15.1	19.6	24.0	18.7	29.3	23.2	31.0	33.4	32.9	31.3	39.0
L	28.1	19.3	12.2	11.4	14.7	18.3	19.0	23.7	27.7	29.4	24.0	36.6	36.0	40.5	45.1	48.8	34.2	42.9	37.6	51.3	35.5	30.2	20.3	18.9	20.2	14.4	15.1	7.4	4.6	9.6	13.2	0	5.1	9.5	14.7	13.7	18.3	22.7	13.4	27.9	17.9	25.7	28.1	27.6	26.0	33.7
M	32.2	14.2	7.1	6.3	9.6	13.2	13.9	18.6	22.6	24.3	18.9	31.5	30.9	35.4	40.0	43.7	29.1	37.8	32.5	46.2	36.2	30.9	24.4	19.6	20.9	15.1	20.2	12.5	9.7	14.7	18.3	5.1	0	4.4	9.6	8.6	13.2	17.6	8.3	22.8	12.8	20.6	23.0	22.5	20.9	28.6
N	36.6	9.8	11.5	10.7	14.0	17.6	18.3	14.6	18.2	19.9	23.3	27.1	33.0	37.5	38.8	42.5	24.7	33.4	28.1	41.8	35.1	35.3	28.8	24.0	25.3	19.5	24.6	16.9	14.1	18.4	14.8	9.5	4.4	0	5.2	4.2	8.8	13.2	3.9	18.4	8.4	16.2	18.6	18.1	16.5	24.2
O	34.7	15.0	7.3	8.8	12.1	15.7	16.4	19.8	23.4	25.1	21.4	32.3	33.4	37.9	42.5	46.2	29.9	38.6	33.3	47.0	38.7	33.4	26.9	22.1	23.4	17.6	22.8	15.1	19.3	13.2	9.6	14.7	9.6	5.2	0	5.5	10.1	14.5	9.1	19.8	13.6	21.4	23.8	23.3	21.7	29.4
P	40.2	14.0	12.8	14.3	17.6	21.2	21.9	18.8	22.4	24.1	26.9	31.3	37.2	41.7	43.0	46.7	28.9	37.6	32.3	46.0	39.3	38.9	32.4	27.6	28.9	23.1	28.3	20.6	18.3	18.7	15.1	13.7	8.6	4.2	5.5	0	4.6	9.0	8.1	14.3	12.6	18.1	20.5	22.3	20.7	28.4
Q	40.9	12.8	17.4	18.9	18.3	21.9	22.3	17.6	21.2	22.9	27.6	30.1	36.0	40.5	41.8	45.5	27.7	36.4	31.1	44.8	38.1	39.6	33.1	28.3	29.6	23.8	32.9	25.2	22.5	16.0	19.6	18.3	13.2	8.8	10.1	4.6	0	4.4	6.9	9.7	11.4	13.5	15.9	21.1	19.5	27.2
R	44.8	16.7	21.8	23.3	22.2	25.8	26.2	21.5	18.8	20.5	26.6	27.7	33.6	38.1	37.8	41.5	23.7	32.4	27.1	40.8	35.7	41.0	37.0	32.2	31.7	27.7	37.3	29.6	26.9	20.4	24.0	22.7	17.6	13.2	14.5	9.0	4.4	0	11.3	5.3	15.3	9.1	11.5	17.1	15.5	23.2
S	34.0	5.9	15.4	14.6	11.4	15.0	15.4	10.7	14.3	16.0	20.7	23.2	29.1	33.6	34.9	38.6	20.8	29.5	24.2	37.9	31.2	32.7	26.2	21.4	22.7	16.9	28.5	20.8	18.0	22.3	18.7	13.4	8.3	3.9	9.1	8.1	6.9	11.3	0	14.5	4.5	12.3	14.7	14.2	12.6	20.3
T	39.5	11.4	21.7	20.2	16.9	20.5	20.9	16.2	13.5	15.2	21.3	22.4	28.3	32.8	32.5	36.2	18.4	27.1	21.8	35.5	30.4	35.7	31.7	26.9	26.4	22.4	34.2	26.5	32.2	25.7	29.3	27.9	22.8	18.4	19.8	14.3	9.7	5.3	14.5	0	10	3.8	6.2	11.8	10.2	17.9
U	29.5	1.4	11.7	10.2	6.9	10.5	10.9	6.2	9.8	11.5	16.2	18.7	24.6	29.1	30.4	34.1	16.3	25.0	19.7	33.4	26.7	28.2	21.7	16.9	18.2	12.4	24.2	16.5	22.5	26.8	23.2	17.9	12.8	8.4	13.6	12.6	11.4	15.3	4.5	10	0	7.8	10.2	9.7	8.1	15.8
V	37.3	9.2	19.5	18.0	14.7	18.3	18.7	14.0	9.7	11.4	17.5	18.6	24.5	29.0	28.7	32.4	14.6	23.3	18.0	31.7	26.6	31.9	29.5	24.7	22.6	20.2	32.0	24.3	30.3	29.5	31.0	25.7	20.6	16.2	21.4	18.1	13.5	9.1	12.3	3.8	7.8	0	2.4	8.0	6.4	14.1
W	39.7	11.6	21.9	20.4	17.1	20.7	21.1	16.4	12.1	13.8	19.9	21.0	26.9	31.4	28.6	32.3	17.0	23.2	17.9	31.6	29.0	34.3	31.9	27.1	25.0	22.6	34.4	26.7	32.7	31.9	33.4	28.1	23.0	18.6	23.8	20.5	15.9	11.5	14.7	6.2	10.2	2.4	0	10.4	8.8	14
X	34.1	8.3	18.6	17.1	13.8	17.4	17.8	13.1	4.9	6.6	12.7	13.8	19.7	24.2	20.7	24.4	9.8	15.3	10.0	23.7	21.8	27.1	26.3	21.5	17.8	19.3	31.1	23.4	31.9	36.5	32.9	27.6	22.5	18.1	23.3	22.3	21.1	17.1	14.2	11.8	9.7	8.0	10.4	0	1.6	6.1
Y	32.5	6.7	17.0	15.5	12.2	15.8	16.2	11.5	3.3	5.0	11.1	12.2	18.1	22.6	22.3	26.0	8.2	16.9	11.6	25.3	20.2	25.5	24.7	19.9	16.2	17.7	29.5	21.8	30.3	34.9	31.3	26.0	20.9	16.5	21.7	20.7	19.5	15.5	12.6	10.2	8.1	6.4	8.8	1.6	0	7.7
Z	32.4	14.4	24.7	23.2	19.9	23.5	23.9	19.2	11.0	12.7	18.8	11.8	17.7	19.2	14.6	18.3	5.2	9.2	3.9	17.6	19.8	25.1	32.4	27.6	23.9	25.4	37.2	29.5	38.0	42.6	39.0	33.7	28.6	24.2	29.4	28.4	27.2	23.2	20.3	17.9	15.8	14.1	14	6.1	7.7	0

Sample Calculation

<u>Dijkstra’s Algorithm Code</u> **(APPENDIX 1 CODE)**

```python
import pandas as pd

import networkx as nx


edge_data = [

    ('A', 'C', 7.8), ('A', 'B', 7.3), ('B', 'AT', 5.3), ('AT', 'AL', 6.1), ('AT', 'AK', 8),

    ('AL', 'AK', 5.9), ('AL', 'AM', 4.5), ('AM', 'AN', 4.6), ('AN', 'AO', 3.7), ('AN', 'AQ', 5.4),

    ('AN', 'AS', 7.5), ('AS', 'AR', 15), ('AQ', 'AR', 5.3), ('AR', 'AP', 7.6), ('AP', 'Z', 5.2),

    ('AR', 'Z', 3.9), ('AP', 'Y', 8.2), ('AP', 'AH', 6.7), ('Z', 'X', 6.1), ('Z', 'W', 14),

    ('W', 'V', 2.4), ('X', 'Y', 1.6), ('Y', 'V', 6.4), ('V', 'T', 3.8), ('V', 'U', 7.8),

    ('U', 'T', 10), ('U', 'S', 4.5), ('U', 'AA', 1.4), ('Y', 'AA', 6.7), ('Y', 'AH', 3.3),

    ('AI', 'AH', 1.7), ('AI', 'AG', 6.5), ('AG', 'AA', 4.8), ('AH', 'AA', 8.4), ('AK', 'AI', 7.2),

    ('AK', 'AP', 6.6), ('AJ', 'AI', 6.1), ('AJ', 'AE', 5.7), ('AE', 'AF', 0.7), ('AF', 'AG', 4.7),

    ('AD', 'AA', 5.5), ('AD', 'AC', 3.3), ('AC', 'AB', 1.5), ('M', 'AC', 6.3), ('M', 'AB', 7.1),

    ('M', 'N', 4.4), ('N', 'S', 3.9), ('O', 'N', 5.2), ('O', 'AB', 7.3), ('O', 'P', 5.5),

    ('N', 'P', 4.2), ('P', 'Q', 4.6), ('Q', 'R', 4.4), ('Q', 'S', 6.9), ('R', 'T', 5.3),

    ('T', 'V', 3.8), ('K', 'O', 9.6), ('J', 'K', 3.6), ('I', 'J', 6.5), ('I', 'L', 4.6),

    ('L', 'M', 5.1), ('H', 'L', 7.4), ('H', 'AC', 6.3), ('H', 'F', 7.0), ('F', 'E', 5.8),

    ('F', 'D', 4.5), ('C', 'D', 4.8), ('D', 'E', 3.7), ('B', 'E', 10.0), ('D', 'F', 4.5),

    ('C', 'G', 5.2), ('G', 'H', 7.7), ('H', 'I', 8.5), ('J', 'L', 9.6), ('F', 'H', 7.0),

    ('E', 'F', 5.8), ('AL', 'AJ', 12.0), ('E', 'AJ', 5.1), ('F', 'AD', 5.5), ('AD', 'AE', 3.6),

    ('AS', 'AQ', 8.4), ('J', 'Q', 16.0)

]


G = nx.Graph()

G.add_weighted_edges_from(edge_data)

all_nodes = sorted(set(sum(([edge[0], edge[1]] for edge in edge_data), [])))

dijkstra_table = pd.DataFrame(index=all_nodes, columns=all_nodes)

for node in all_nodes:

    lengths = nx.single_source_dijkstra_path_length(G, node)

    for target_node in all_nodes:

        length = lengths.get(target_node, float('inf'))

        dijkstra_table.loc[node, target_node] = length if length != float('inf') else 'infinity'


for col in dijkstra_table.columns:

    dijkstra_table[col] = dijkstra_table[col].apply(lambda x: f'{x:.1f}' if isinstance(x, float) else x)


dijkstra_table.to_csv('dijkstra_results.csv')

print("Dijkstra's algorithm table saved to 'dijkstra_results.csv'")
```

Kruskal's Algorithm Code (APPENDIX 2 CODE)

```
import pandas as pd

import networkx as nx


edge_data = [

    ('A', 'C', 7.8), ('A', 'B', 7.3), ('B', 'AT', 5.3), ('AT', 'AL', 6.1), ('AT', 'AK', 8),

    ('AL', 'AK', 5.9), ('AL', 'AM', 4.5), ('AM', 'AN', 4.6), ('AN', 'AO', 3.7), ('AN', 'AQ', 5.4),

    ('AN', 'AS', 7.5), ('AS', 'AR', 15), ('AQ', 'AR', 5.3), ('AR', 'AP', 7.6), ('AP', 'Z', 5.2),

    ('AR', 'Z', 3.9), ('AP', 'Y', 8.2), ('AP', 'AH', 6.7), ('Z', 'X', 6.1), ('Z', 'W', 14),

    ('W', 'V', 2.4), ('X', 'Y', 1.6), ('Y', 'V', 6.4), ('V', 'T', 3.8), ('V', 'U', 7.8),

    ('U', 'T', 10), ('U', 'S', 4.5), ('U', 'AA', 1.4), ('Y', 'AA', 6.7), ('Y', 'AH', 3.3),

    ('AI', 'AH', 1.7), ('AI', 'AG', 6.5), ('AG', 'AA', 4.8), ('AH', 'AA', 8.4), ('AK', 'AI', 7.2),

    ('AK', 'AP', 6.6), ('AJ', 'AI', 6.1), ('AJ', 'AE', 5.7), ('AE', 'AF', 0.7), ('AF', 'AG', 4.7),

    ('AD', 'AA', 5.5), ('AD', 'AC', 3.3), ('AC', 'AB', 1.5), ('M', 'AC', 6.3), ('M', 'AB', 7.1),

    ('M', 'N', 4.4), ('N', 'S', 3.9), ('O', 'N', 5.2), ('O', 'AB', 7.3), ('O', 'P', 5.5),

    ('N', 'P', 4.2), ('P', 'Q', 4.6), ('Q', 'R', 4.4), ('Q', 'S', 6.9), ('R', 'T', 5.3),

    ('T', 'V', 3.8), ('K', 'O', 9.6), ('J', 'K', 3.6), ('I', 'J', 6.5), ('I', 'L', 4.6),

    ('L', 'M', 5.1), ('H', 'L', 7.4), ('H', 'AC', 6.3), ('H', 'F', 7.0), ('F', 'E', 5.8),

    ('F', 'D', 4.5), ('C', 'D', 4.8), ('D', 'E', 3.7), ('B', 'E', 10.0), ('D', 'F', 4.5),

    ('C', 'G', 5.2), ('G', 'H', 7.7), ('H', 'I', 8.5), ('J', 'L', 9.6), ('F', 'H', 7.0),

    ('E', 'F', 5.8), ('AL', 'AJ', 12.0), ('E', 'AJ', 5.1), ('F', 'AD', 5.5), ('AD', 'AE', 3.6),

    ('AS', 'AQ', 8.4), ('J', 'Q', 16.0)

]


G = nx.Graph()

G.add_weighted_edges_from(edge_data)

mst = nx.minimum_spanning_tree(G, algorithm='kruskal')

mst_edges = mst.edges(data=True)

mst_data = [(u, v, d['weight']) for u, v, d in mst_edges]

mst_df = pd.DataFrame(mst_data, columns=['Node1', 'Node2', 'Weight'])


mst_df.to_csv('kruskal_mst.csv', index=False)

print("Kruskal's MST saved to 'kruskal_mst.csv'")
```