MINIMUM PATH LINKING KUALA LUMPUR AND SINGAPORE BY HIGH-SPEED RAIL

A REPORT SUBMITTED IN FULFILLMENT OF THE REQUIREMENT FOR BIC 10103 (DISCRETE STRUCTURE) COURSE

SEM 2021/2022

GROUP MEMBERS (GROUP 3):
TAN KAY LI (AI200298)
TAN PEI YU (AI200286)
TEE EE KEE (AI200245)
WONG MANN CHYI (AI200276)
YEE SUZAN (AI200251)

LECTURER:
DR. NURUL ASWA BINTI OMAR
PUAN AZWANIZA BINTI POHARAN @ BUNARI

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITI TUN HUSSEIN ONN MALAYSIA

TABLE OF CONTENTS

Case Study

Case study focuses on the explanation for a question or a phenomenon. Hence, the explanation of the given question will be discussed in this case study.

For the given task, the minimum path linking Kuala Lumpur and Singapore for a High-Speed Rail is required to be determined. First of all, the path of the High-Speed Rail is required to be determined using tools such as Google Maps. Besides, the distance between each and every stop is required to be found out as well using tools such as DISTANCESTO.COM. In this case, several paths and stops from Kuala Lumpur to Singapore will be found out. Hence, in order to identify the minimum path linking Kuala Lumpur and Singapore, Dijkstra's Algorithm is required to be applied in the calculation to figure out the shortest path and its length.

Mathematical Modelling

Graph theory is considered a mathematical model in this project. It is a helpful tool that consists of vertices and edges to model pairwise relationships between objects. The graph is represented as G = (V, E). Vertices are also referred to as nodes. Additionally, the edges can be referred to as arcs, as they are related to an ordered pair of vertices, as illustrated in Figure 1.

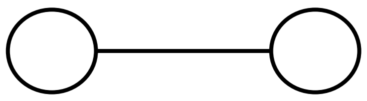
Vertices (Nodes)	Edges (Arcs)
	

Figure 1: Examples of vertices(nodes) and edges(arcs)

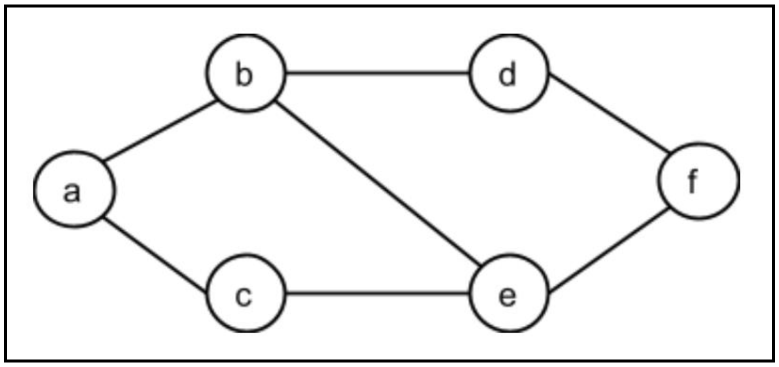


Figure 2: Example of graph

Moreover, Dijkstra’s algorithm is an algorithm that can be applied on a weighted graph to find the shortest path from a beginning node to a destination node. In this project, Dijkstra’s algorithm can be used to calculate the minimum path linking Kuala Lumpur and Singapore by high-speed rail. There are four Dijkstra’s Rules. The first rule is that there is a positive weight on every edge and the distance to the source vertex is set to zero and the rest is set to infinity. The second rule is that all the vertices adjacent to the current vertex are relaxed and followed by selecting the nearest vertex as the next current vertex. Finally, repeat rules 2 and 3 until the destination is achieved.

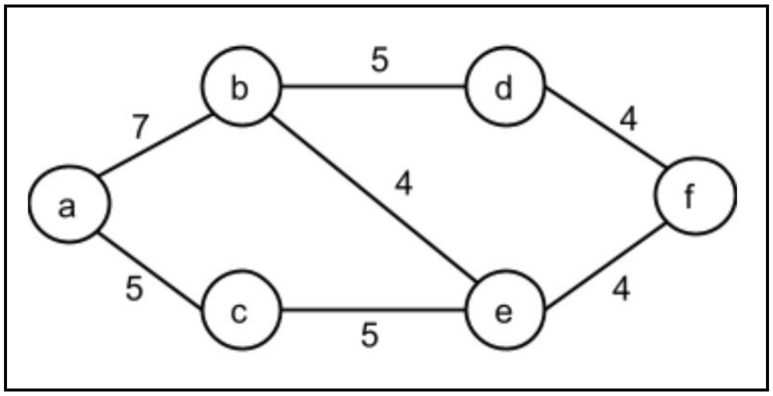


Figure 3: Example of weighted graph

Based on the Figure 3 above, the shortest path from beginning node, *a* to destination node, *f* can be found using Dijkstra's Algorithm. By following the Dijkstra's Rules, the result is shown in Figure 4 below where the shortest path from *a* to *f* is *a, c, e, f* with the length of 14.

v	a	b	c	d	e	f
a	0	7	5	INF	INF	INF
c	0	7	5	INF	10	INF
b	0	7	5	12	10	INF
e	0	7	5	12	10	14
d	0	7	5	12	10	14

Table 1: Calculating the example of weighted graph using Dijkstra's Algorithm Table

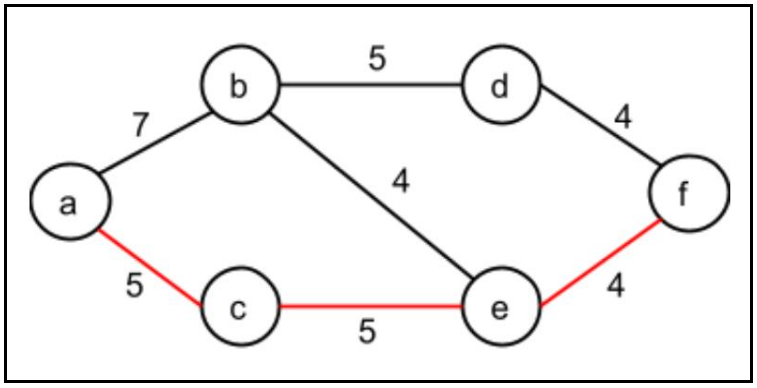


Figure 4: The shortest path from a to f using Dijkstra's Algorithm Table

Construction of The Mathematical Model

A graph in terms of Graph Theory is the collection of points, which are known as vertices joined by lines named edges. In this project, an undirected graph will be produced in order to show the path linking Kuala Lumpur and Singapore by High-speed Rail. To identify each stop of the High-speed Rail, Google Maps will be used, then, the distances between stops will also be measured by using DISTANCESTO.COM, every distance will be shown in kilometer (km). As mentioned above, a graph should be contained with vertices and edges. In this project, vertices of the graph represented each stop involved Kuala Lumpur, Putrajaya, Semenyih, Senawang, Port Dickson, Tampin, Tanjung Kling, Muar, Pagoh, Yong Peng, Batu Pahat, Simpang Renggam, Benut, Pontian District, Gelang Patah and Tuas Checkpoint. Edges between vertices represent the path linking each stop and the weights of edges will all be shown in km.

Figure 5 shows the distance and link between Kuala Lumpur, Malaysia and Tuas Checkpoint, Singapore on the map. Two black circles represent the starting or ending destination of the High-speed Rail, which are Kuala Lumpur and Tuas Checkpoint of Singapore. The red colored straight line shows the straight path between both destinations, hence, every stop to be chosen should not be far away from the red colored straight line.

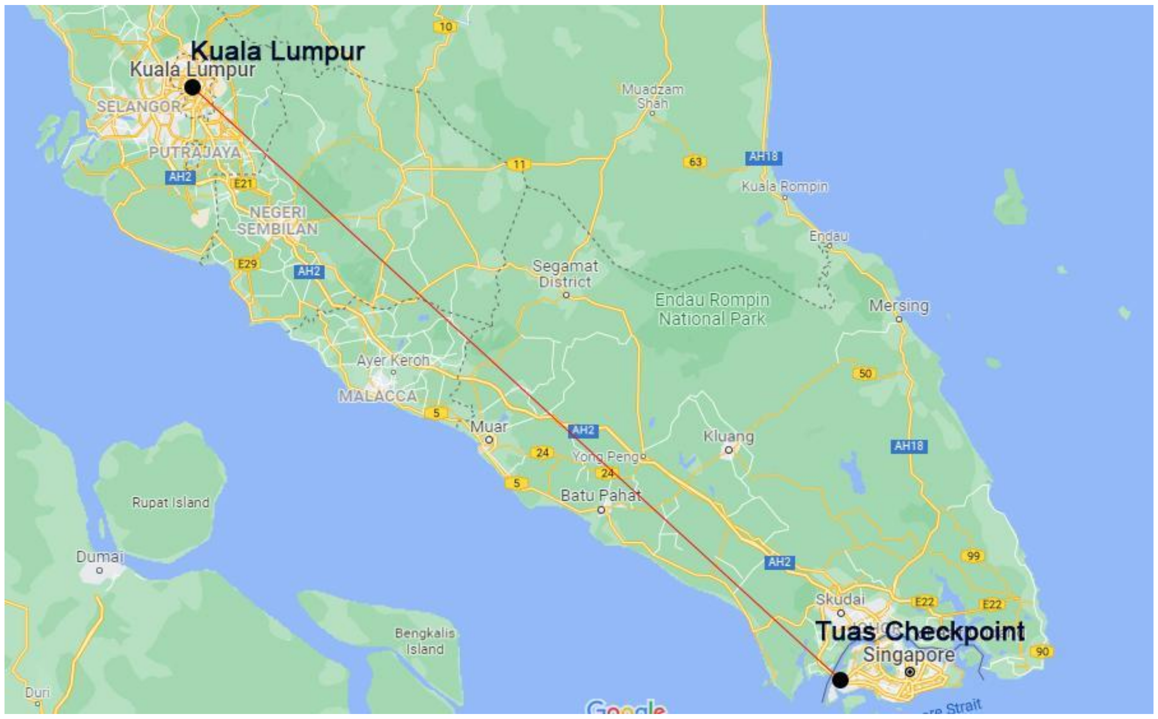


Figure 5: Distance and link between Kuala Lumpur, Malaysia and Tuas Checkpoint, Singapore on map

Figure 6 shows every stop point chosen for this project in order to create a High-speed Rail. There are 15 destinations chosen as the stop point, thus, there will be a total of 17 vertices in this project.

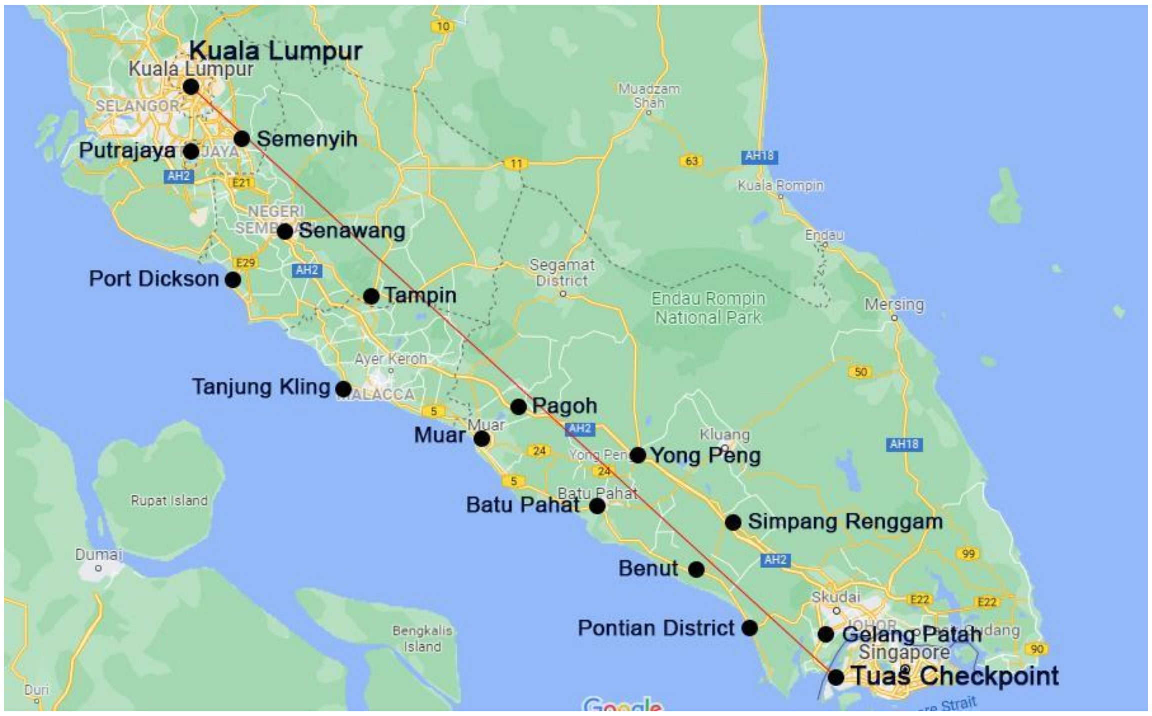


Figure 6: Stops of High-speed Rail chosen shown on map

Figure 7 shows edges between each stop, those are the paths planned for High-speed Rail. According to Figure 7, Tampin, Ayer Keroh, Muar and Pagoh may have more than one edge linked to various vertices. Furthermore, Figure 8 shows the clearer graph drawn with complete edges and vertices.

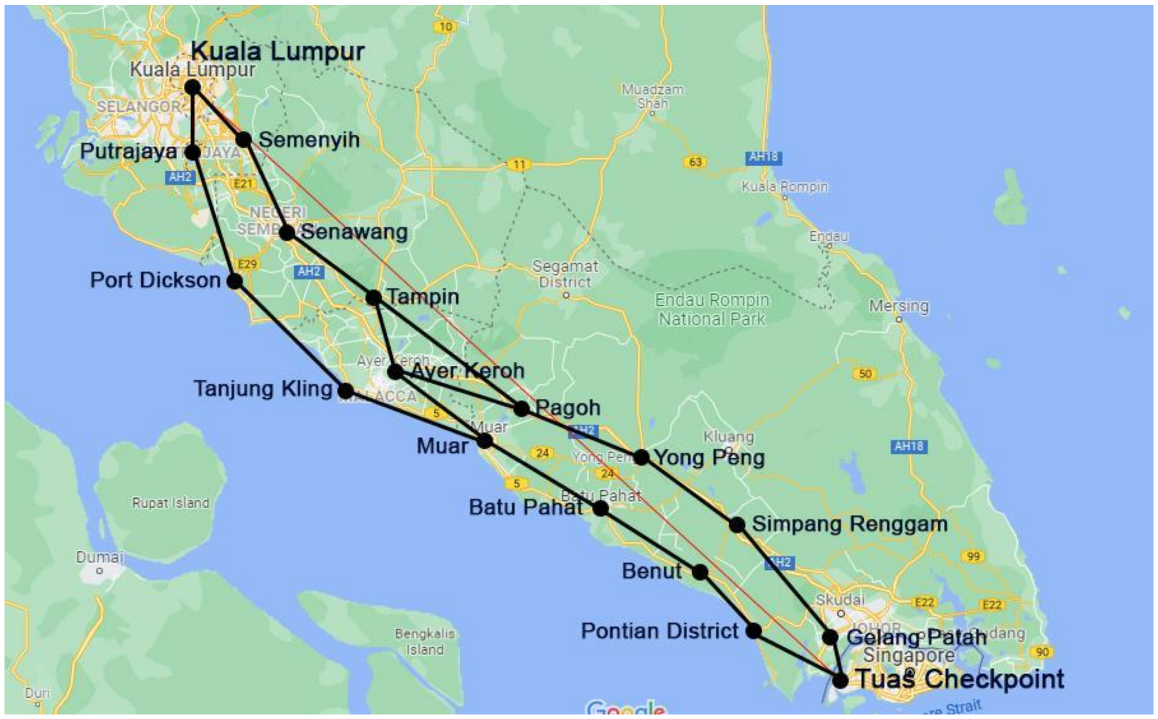


Figure 7: Edges between each stop show on map

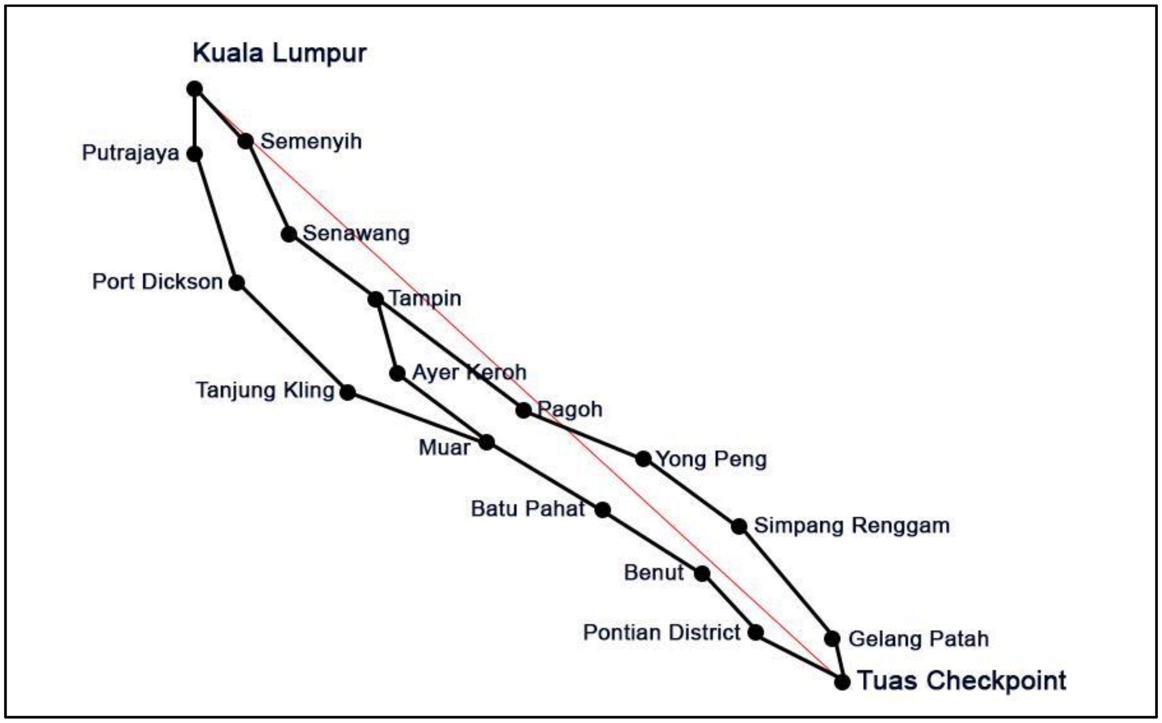


Figure 8: Edges between each stop

In order to create a complete graph, each vertex should be joined with edges. Thus, in Figure 9, the weight of each edge represents the distance of each stop, and are measured and shown in the unit of km.

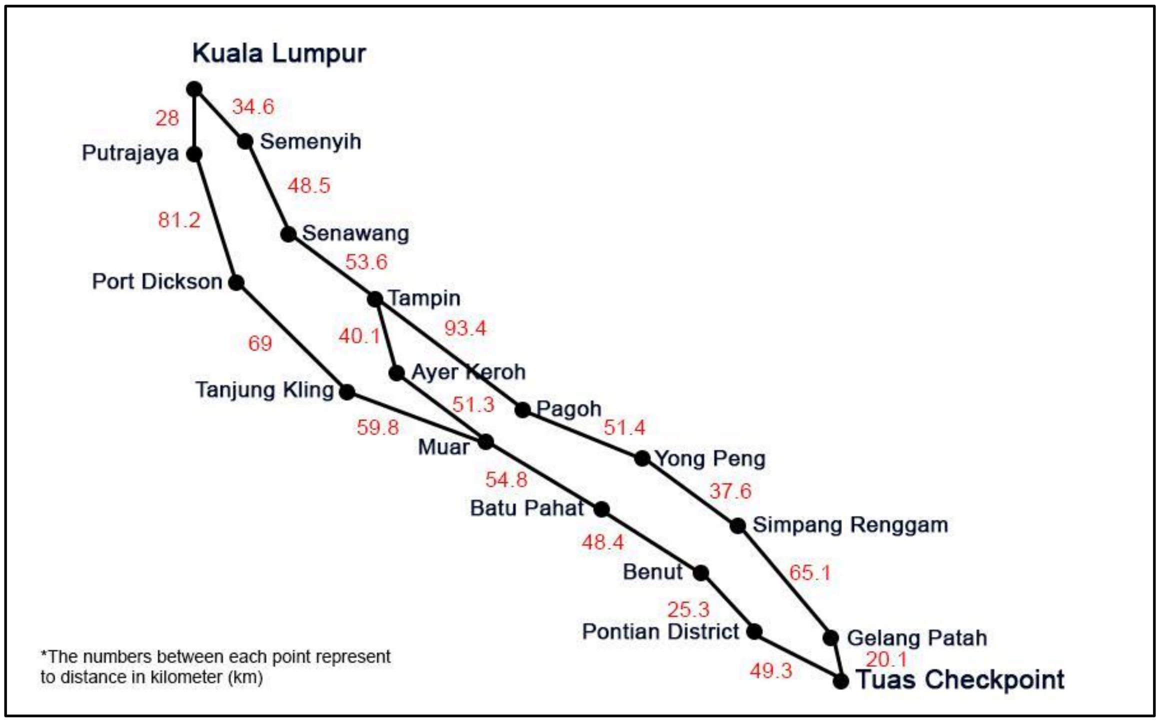


Figure 9: Weight of each edge

Shortest path refers to the minimum length between two locations or destinations. An individual who visits a website such as Google Maps, they usually look for the shortest path between the two locations. The High-speed Rail should also be linked to a minimum path, thus the shortest distance theory of Discrete Mathematic should be considered in this project.

By using Dijkstra's Algorithm, a set of vertices which have shortest paths from source will be maintained. The graph's cost adjacency matrix, where cost is the edge's weight, represents the graph. All of the diagonal values in the graph's cost adjacency matrix are 0. There are few steps needed to be included in Dijkstra's Algorithm and are shown as below:

1. Initially, no vertex placed in sets.
2. Include the source vertex, V_s in the table. Then, determine all the paths from V_s to all other linked vertices.
3. Include all the vertices in the table which nearest to V_s and find the shortest paths to all the vertices, update the values at the same time.
4. Assume that there are n vertices, repeat the steps $n - 1$ vertices are not included in the table.

The completion of this process will lead us to the shortest paths to all the vertices from the source vertex.

Now, by following the process and all the rules of Dijkstra's Algorithm, the shortest path of the High-speed Rail will be calculated. Figure 10 shows the abbreviation of all the vertices of the graph while Table 2 is a table to show the workings of Dijkstra’s Algorithm.

Kuala Lumpur - KL	Semenyih – SM
Putrajaya - PJ	Senawang – SN
Port Dickson - PD	Tampin – TP
Tanjung Kling - TK	Ayer Keroh – AK
Muar - MU	Pagoh – PG
Batu Pahat - BP	Yong Peng – YP
Benut - BN	Simpang Renggam – SR
Pontian District – PT	Gelang Patah - GP
Tuas Checkpoint - TC	

Figure 10: Abbreviation for vertices in the table

v	KL	PJ	SM	PD	SN	TK	TP	MU	AK	BP	PG	BN	YP	PT	SR	GP	TC
KL	0	28	34.6	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF
PJ	0	28	34.6	109.2	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF
SM	0	28	34.6	109.2	83.1	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF
SN	0	28	34.6	109.2	83.1	INF	136.7	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF
PD	0	28	34.6	109.2	83.1	178.2	136.7	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF
TP	0	28	34.6	109.2	83.1	178.2	136.7	INF	176.8	INF	230.1	INF	INF	INF	INF	INF	INF
AK	0	28	34.6	109.2	83.1	178.2	136.7	228.1	176.8	INF	230.1	INF	INF	INF	INF	INF	INF
TK	0	28	34.6	109.2	83.1	178.2	136.7	228.1	176.8	INF	230.1	INF	INF	INF	INF	INF	INF
MU	0	28	34.6	109.2	83.1	178.2	136.7	228.1	176.8	282.9	230.1	INF	INF	INF	INF	INF	INF
PG	0	28	34.6	109.2	83.1	178.2	136.7	228.1	176.8	282.9	230.1	INF	281.5	INF	INF	INF	INF
YP	0	28	34.6	109.2	83.1	178.2	136.7	228.1	176.8	282.9	230.1	INF	281.5	INF	319.1	INF	INF
BP	0	28	34.6	109.2	83.1	178.2	136.7	228.1	176.8	282.9	230.1	331.3	281.5	INF	319.1	INF	INF
SR	0	28	34.6	109.2	83.1	178.2	136.7	228.1	176.8	282.9	230.1	331.3	281.5	INF	319.1	384.2	INF
BN	0	28	34.6	109.2	83.1	178.2	136.7	228.1	176.8	282.9	230.1	331.3	281.5	356.6	319.1	384.2	INF
PT	0	28	34.6	109.2	83.1	178.2	136.7	228.1	176.7	282.9	230.1	331.3	281.5	356.6	319.1	384.2	405.9
GP	0	28	34.6	109.2	83.1	178.2	136.7	228.1	176.7	282.9	230.1	331.3	281.5	356.6	319.1	384.2	**404.3**

Table 2: Dijkstra's Algorithm table

According to the Dijkstra's Algorithm table above, the shortest distance is 404.3km. The shortest path that had determined using Dijkstra's Algorithm is shown as below:

KL → SM → SN → TP → PG → YP → SR → GP → TC

Figure 11: Shortest path result

To conclude, that the shortest path from the planned graph of the High-speed Rail should be from Kuala Lumpur, then to Semenyih, Senawang, Tampin, Pagoh, Yong Peng, Simpang Renggam, Gelang Patah, and finally to Tuas Checkpoint of Singapore.

Development of The Model into Programming Codes

According to the mathematical model constructed, the concept of discrete structure, Dijkstra's Algorithm is applied in this project. It is used to find the shortest path between Kuala Lumpur and Tuas Checkpoint with the vertices (stop point) and the edges (path linking between two stops) provided. To implement the algorithm, a Shortest Path Tree (SPT) was generated since some stops may connect the path with other stops. In this situation, the SPT can help in determining the minimum distance from the picked vertex to the adjacent vertex for every vertex. Generally, there are two sets that are held in SPT, one set contains the vertices that are already included in the SPT, and another set contains the vertices that haven't been included in the SPT. A vertex will be picked and it will be checked to see whether it is not included in the SPT. The vertex also should have the minimum distance to be included in the SPT. Then, this vertex will be marked as done and included in the SPT. All the steps are repeated until the destination point is reached. The pseudocode for this algorithm was prepared to develop the model into programming codes.

```
function dijkstra (graph, source)
   for each vertex v in graph:
      distance[v] <- INF
      previous[v] <- NULL
      if v != source, add v to Priority Queue Q

   distance[s] <- 0
   Q <- the set of all nodes in graph

   while Q is not empty:
      u <- node in Q with min distance[]
      remove u from Q

      for each adjacent v of u:
         temp <- distance[u] + distance_between (u, v)
         if temp < distance[v]
            distance[v] <- temp
            previous[v] <- u

   Return distance[], previous[]

End
```

Figure 12: Pseudocode of Dijkstra's Algorithm

Before constructing the programming codes, each stop point is represented by the numbers accordingly so that the graph is easy to be understood while implemented into programming codes. The related figure is shown below.

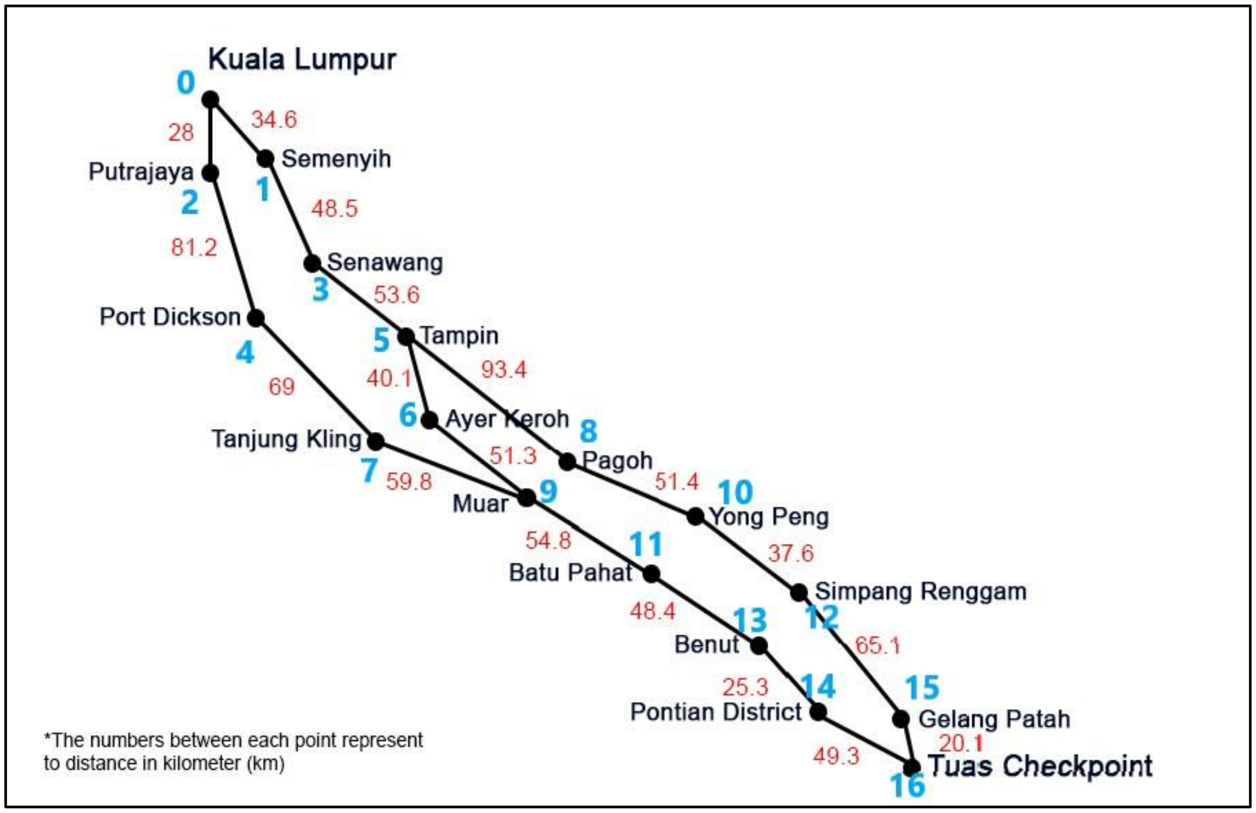
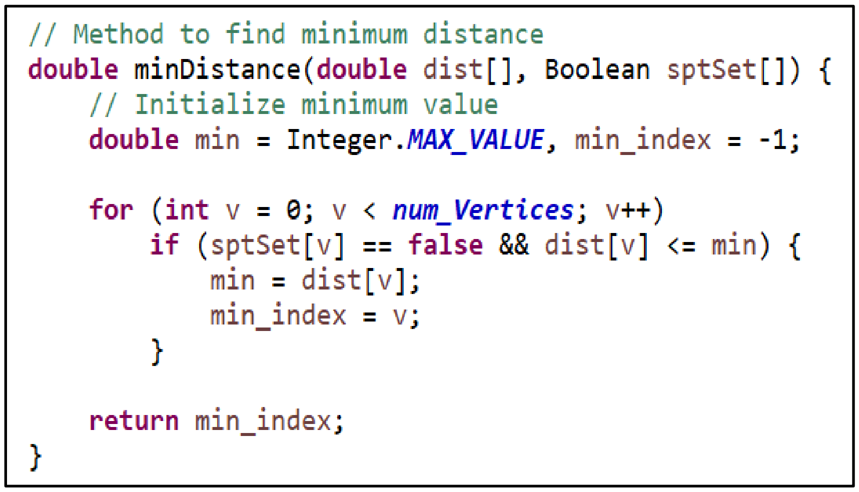


Figure 13: Vertices represented by numbers

The algorithm is implemented by using Java programming codes and the algorithm's steps were separated into several methods. The arrays and two-dimensional arrays are also applied in this program.

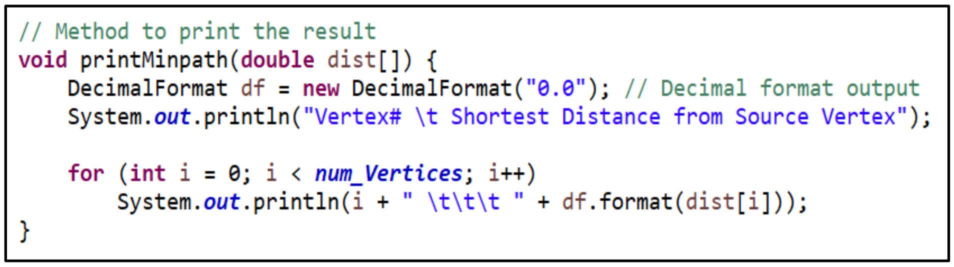
Based on Figure 14 shown below, it demonstrates a return value type method with a group of statements to perform the function of finding the minimum distance of the vertex **`v`**. The method **`minDistance()`** received two parameters which are **`dist[]`**, the distance of vertex and **`sptSet[]`**, the SPT set that contains the vertices. When going through all the vertices in the graph, the vertex index (or value) with minimum distance will be assigned in **`min_index`** and the minimum distance of the vertex will be assigned in **`dist[v]`** if and only the vertex **`v`** has not been included in the SPT. Then, it will return **`min_index`** for further operation in **`algo_dijkstra()`**.



```java
// Method to find minimum distance
double minDistance(double dist[], Boolean sptSet[]) {
    // Initialize minimum value
    double min = Integer.MAX_VALUE, min_index = -1;

    for (int v = 0; v < num_Vertices; v++)
        if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }

    return min_index;
}
```

Figure 14: Code of the method to find the minimum distance

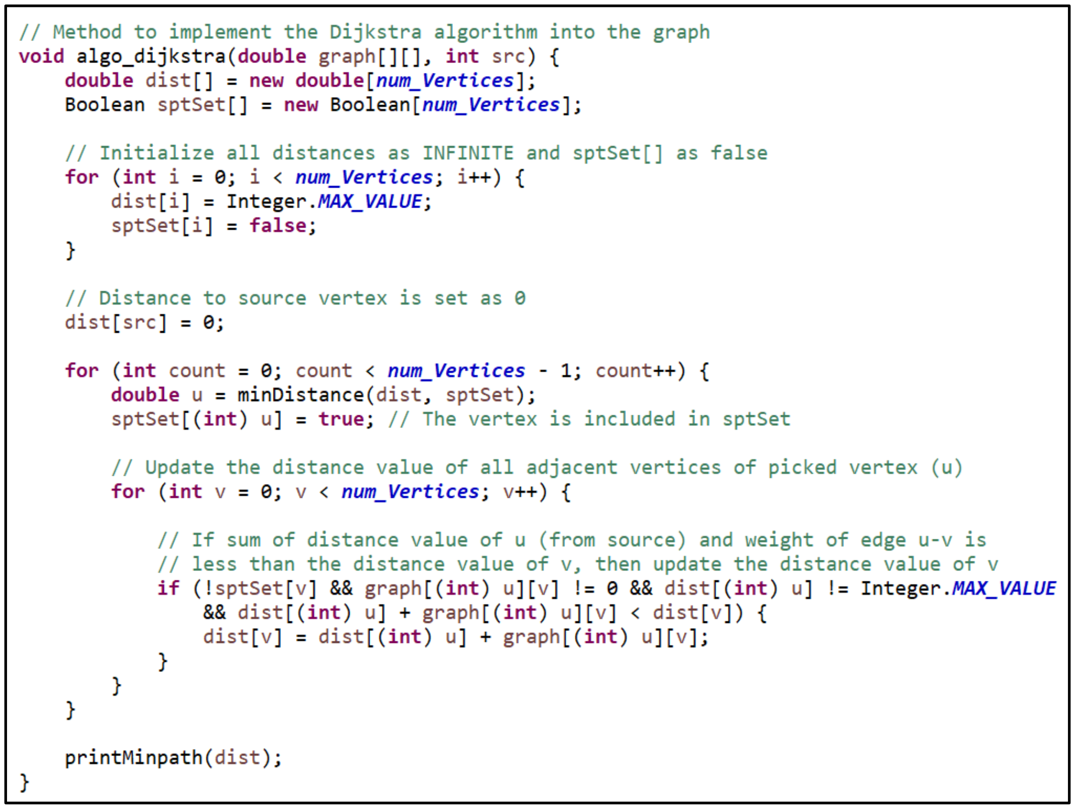
Figure 15 shows the **`printMinpath()`** method that prints out the result of the shortest path distance between the source node and destination node. The **`DecimalFormat`** constructor was created with format **`("0.0")`** so that the shortest path distance displayed is kept in one decimal place.



```java
// Method to print the result
void printMinpath(double dist[]) {
    DecimalFormat df = new DecimalFormat("0.0"); // Decimal format output
    System.out.println("Vertex# \t Shortest Distance from Source Vertex");

    for (int i = 0; i < num_Vertices; i++)
        System.out.println(i + " \t\t\t " + df.format(dist[i]));
}
```

Figure 15: Code of the method to print the result

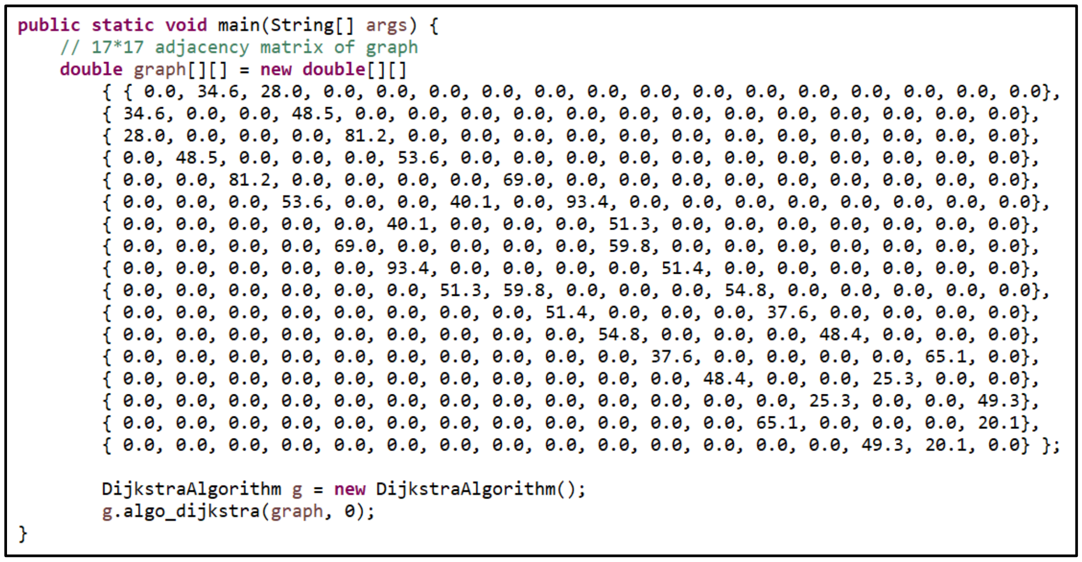
Figure 16 shows the **`algo_dijkstra()`** method that implements Dijkstra's Algorithm into the graph represented by the adjacency matrix. The method received two parameters which are **`graph[][]`**, the adjacency matrix based on the graph represented by numbers as shown in Figure 13 and **`src`**, the source node of the graph. First, the distance of the vertices, **`dist[i]`** is set to infinite and the SPT set, **`sptSet[i]`** is set to false, meaning that the vertices are not included in the SPT yet. The distance of the source vertex from itself, **`dist[src]`** is always set as 0. Then, the **`minDistance()`** method was called to find the minimum distance of vertex **`u`** (noted that **`u`** is always equal to **`src`** in the first iteration). After the method call, the vertex **`u`** is set as true, meaning that **`u`** is included in the SPT. Next, the distance value of all the adjacent vertices of picked vertex **`u`** will be updated. During the iteration, the distance value **`v`** will be updated if and only the sum of distance value **`u`** (from source) and edge's weight of **`u`** and **`v`** is smaller than the distance value of **`v`** (**`dist[(int) u] + graph[(int) u][v] < dist[v]`**). At the end of the method, the **`printMinpath()`** method was called.



```java
// Method to implement the Dijkstra algorithm into the graph
void algo_dijkstra(double graph[][], int src) {
    double dist[] = new double[num_Vertices];
    Boolean sptSet[] = new Boolean[num_Vertices];

    // Initialize all distances as INFINITE and sptSet[] as false
    for (int i = 0; i < num_Vertices; i++) {
        dist[i] = Integer.MAX_VALUE;
        sptSet[i] = false;
    }

    // Distance to source vertex is set as 0
    dist[src] = 0;

    for (int count = 0; count < num_Vertices - 1; count++) {
        double u = minDistance(dist, sptSet);
        sptSet[(int) u] = true; // The vertex is included in sptSet

        // Update the distance value of all adjacent vertices of picked vertex (u)
        for (int v = 0; v < num_Vertices; v++) {

            // If sum of distance value of u (from source) and weight of edge u-v is
            // less than the distance value of v, then update the distance value of v
            if (!sptSet[v] && graph[(int) u][v] != 0 && dist[(int) u] != Integer.MAX_VALUE
                && dist[(int) u] + graph[(int) u][v] < dist[v]) {
                dist[v] = dist[(int) u] + graph[(int) u][v];
            }
        }
    }

    printMinpath(dist);
}
```

Figure 16: Code of the method to find shortest path

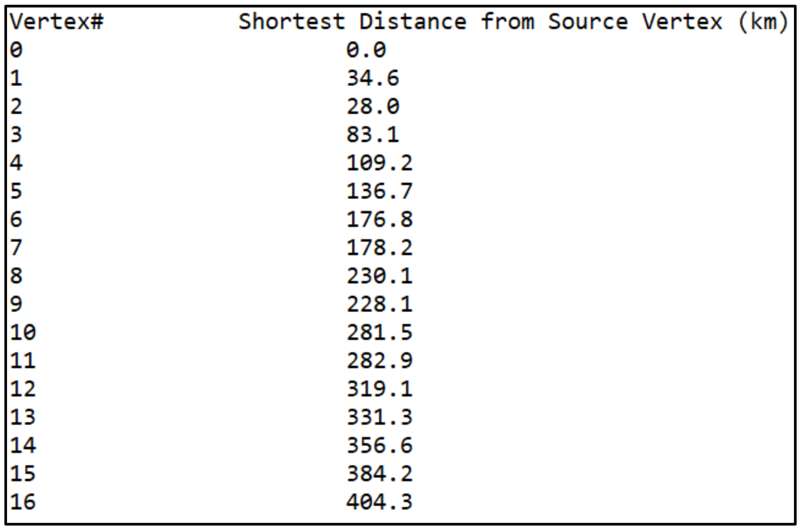
Figure 17 shows the main method to run this program. The two-dimensional array, **`graph[][]`** indicate the 17*17 adjacency matrix of the graph in Figure 13. The **`algo_dijkstra()`** method was called in this section.



```java
public static void main(String[] args) {
    // 17*17 adjacency matrix of graph
    double graph[][] = new double[][]
        { { 0.0, 34.6, 28.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        { 34.6, 0.0, 0.0, 48.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        { 28.0, 0.0, 0.0, 0.0, 81.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        { 0.0, 48.5, 0.0, 0.0, 0.0, 53.6, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        { 0.0, 0.0, 81.2, 0.0, 0.0, 0.0, 0.0, 69.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        { 0.0, 0.0, 0.0, 53.6, 0.0, 0.0, 40.1, 0.0, 93.4, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        { 0.0, 0.0, 0.0, 0.0, 0.0, 40.1, 0.0, 0.0, 0.0, 51.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        { 0.0, 0.0, 0.0, 0.0, 69.0, 0.0, 0.0, 0.0, 0.0, 59.8, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        { 0.0, 0.0, 0.0, 0.0, 0.0, 93.4, 0.0, 0.0, 0.0, 0.0, 51.4, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 51.3, 59.8, 0.0, 0.0, 0.0, 54.8, 0.0, 0.0, 0.0, 0.0, 0.0},
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 51.4, 0.0, 0.0, 0.0, 37.6, 0.0, 0.0, 0.0, 0.0},
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 54.8, 0.0, 0.0, 0.0, 48.4, 0.0, 0.0, 0.0},
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 37.6, 0.0, 0.0, 0.0, 0.0, 65.1, 0.0},
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 48.4, 0.0, 0.0, 25.3, 0.0, 0.0},
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 25.3, 0.0, 0.0, 49.3},
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 65.1, 0.0, 0.0, 0.0, 20.1},
        { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 49.3, 20.1, 0.0} };

        DijkstraAlgorithm g = new DijkstraAlgorithm();
        g.algo_dijkstra(graph, 0);
}
```

Figure 17: Code of the main method

Figure 18 shows the output displayed when the program was executed. The shortest path distance between the source node (Kuala Lumpur) and the destination node (Tuas Checkpoint) was 404.3km, which is the same as the result found in Dijkstra's Algorithm table.



```
Vertex#          Shortest Distance from Source Vertex (km)
0                       0.0
1                       34.6
2                       28.0
3                       83.1
4                       109.2
5                       136.7
6                       176.8
7                       178.2
8                       230.1
9                       228.1
10                      281.5
11                      282.9
12                      319.1
13                      331.3
14                      356.6
15                      384.2
16                      404.3
```

Figure 18: Output of the execution

Analysis of The Result

Based on the instructions of the project given, there is a path from Kuala Lumpur to Tuas Checkpoint, Singapore created for the High-speed Rail Project. In this project, a graph of the pathway from Kuala Lumpur to Tuas Checkpoint, Singapore is created.

In the case study before this, the minimum path linking Kuala Lumpur and Singapore for a High-speed Rail is required to be determined by a group of team members from Group 3 in BIC10103 Discrete Structure course. Graph theory is used in conjunction with Dijkstra's algorithm and programming codes for determining the shortest path between Kuala Lumpur and Singapore. There is a route which has been determined by team members linking from Kuala Lumpur and Singapore. The route which has been found out is Kuala Lumpur - Semenyih - Senawang - Tampin - Pagoh - Yong Peng - Simpang Renggam - Gelang Patah - Tuas Checkpoint (Singapore). In this project, Dijkstra's Algorithm and Java programming codes are among the methods used to find the shortest path between Kuala Lumpur and Singapore. In fact, the team members from Group 3 acquire a total of 17 vertices linking Kuala Lumpur and Singapore. The analysis revealed that the minimum path between Kuala Lumpur and Singapore for a High-speed Rail project is 404.3 kilometres (km) by utilising Dijkstra's Algorithm and Java programming codes. Both methods produce exactly the same results which is 404.3 km.

As a result, using Dijkstra's Algorithm and Java programming code, the route (Kuala Lumpur - Semenyih - Senawang - Tampin - Pagoh - Yong Peng - Simpang Renggam - Gelang Patah - Singapore’s Tuas Checkpoint) is the most ideal for building a high-speed rail to connect Kuala Lumpur and Singapore which has a total distance of 404.3 km. As compared with other types of transportation such as cars and buses, this high-speed rail system will significantly reduce travel time between Kuala Lumpur and Singapore.

References

Dijkstra’s Algorithm. (2021, December 16). In *Wikipedia*. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Estefania Cassingena Navone. (2020, September 28). *Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction.* https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/

International Mathematical Modeling Challenge. (n.d.). *What is mathematical modelling?* https://www.immchallenge.org.au/supporting-resources/what-is-mathematical-modelling

Isaac Computer Science. (n.d.). *Dijkstra’s algorithm*. https://isaaccomputerscience.org/concepts/dsa_search_dijkstra?examBoard=all&stage=all

James Clark. (2020, May 19). *Kuala Lumpur–Singapore high-speed rail*. https://futuresoutheastasia.com/kuala-lumpur-singapore-high-speed-rail/

Juha-Pekka Tolvanen. (2012, May 8). How to Integrate Models and Code. *InfoQ*. https://www.infoq.com/articles/combining-model-and-code/

Kenneth Leroy Busbee. (2009). *Programming Fundamentals*. Rebus Community. Retrieved January 10, 2022, from https://press.rebus.community/programmingfundamentals/

MathsIsFun. (n.d.). *Mathematical Models. https://www.mathsisfun.com/algebra/mathematical-models.html*

MyHSR Corporation Sdn. Bhd. (n.d.). *Project Overview - KL-SG HSR.* https://www.myhsr.com.my/kl-sg-hsr/project-overview

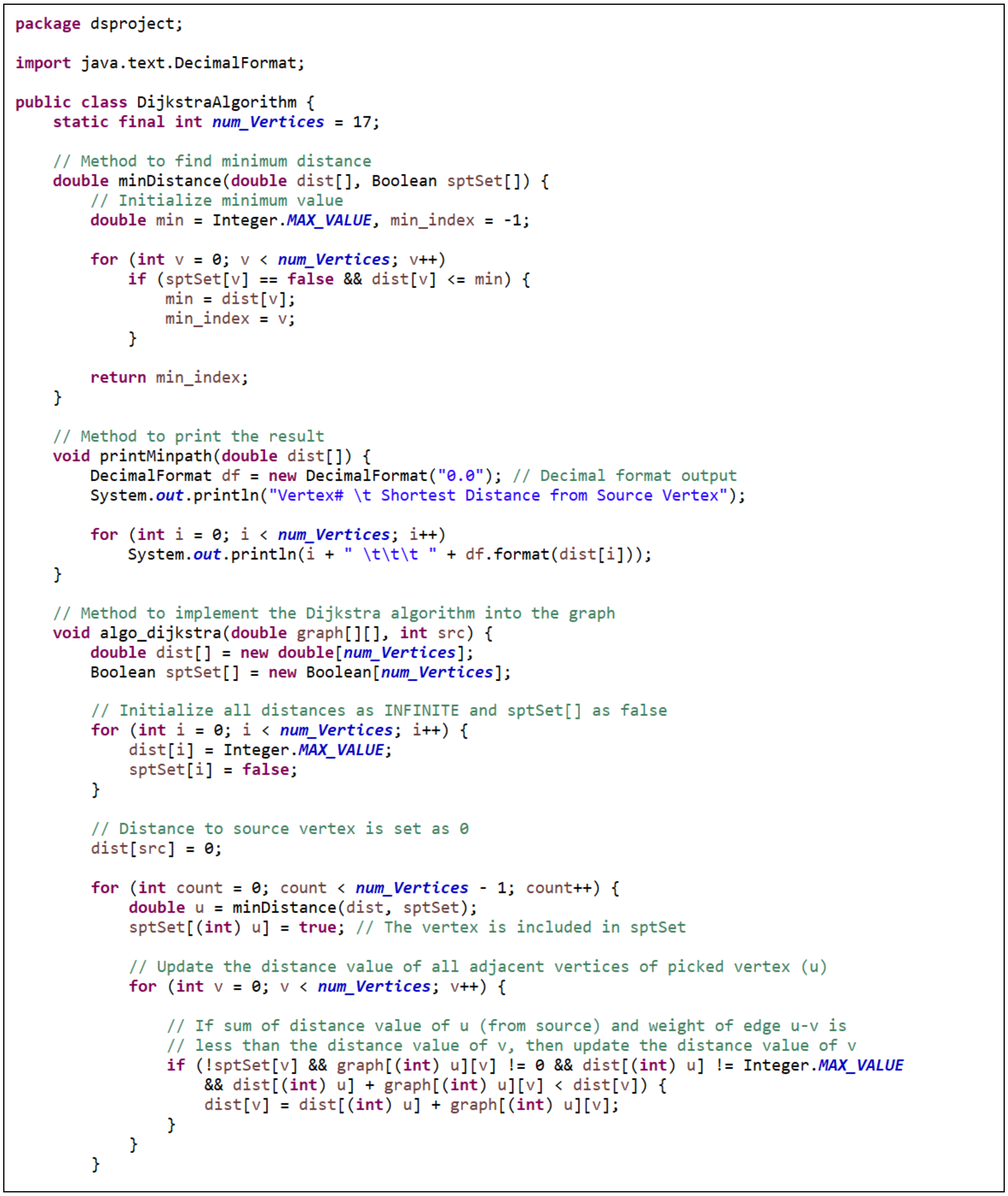
Neelam Tyagi. (2020, December 14). *Dijkstra’s Algorithm: The Shortest Path Algorithm*. https://www.analyticssteps.com/blogs/dijkstras-algorithm-shortest-path-algorithm

Appendix A

				V.Poor	Poor	Fair	Good	Excellent			
Project Evaluation Rubric											
CLO 1: Explain the concept of discrete structure clearly (C4:PLO7) 5%											
Assessment	Criteria	Sub-criteria	Level	1	2	3	4	5	Check	Weight	Score
Report	Ability to translate the real-world problem into discrete structure's concept	Identify a suitable mathematical model	C2							0.5	
	Ability to construct a model	Apply the mathematical model to the given case study	C3							1	
		Use data relevant to the problem	C3							0.5	
	Ability to interpret the solution	Interpret the solution	C4							2	
								TOTAL		4	$\frac{}{20} \times 5 =$
CLO 2: Manipulate knowledge in solving problems in the field of ICT (P4:PLO3) 5%											
Assessment	Criteria	Sub-criteria	Level	1	2	3	4	5	Check	Weight	Score
Presentation	Ability to perform an investigation on case study problem	Description on case study	P2							0.5	
		Correct use of mathematical notation	P2							0.5	
	Ability to use/adapt appropriate problem-solving technique/concepts	Establish proper mathematical model	P3							2	
	Ability to apply discrete structure concept into programming technique	Correct construction of problem model	P4							2	
		Correct algorithm applied	P4							2	
								TOTAL		7	$\frac{}{35} \times 5 =$
CLO 3: Solve programming problems using the concept of discrete structure (A4 : PLO6) 10%											
Assessment	Criteria	Sub-criteria	Level	1	2	3	4	5	Check	Weight	Score
Presentation	Ability to participate actively in group	Present the project using one/more suitable tool(s)	A2							1	
	Ability to solve problems in group	Comply to project instructions	A3							2	
	Ability to organize the project in group	Organization of group presentation	A4							2	
		Deliver the project clearly	A4							2	
									TOTAL	7	$\frac{}{35} \times 10 =$

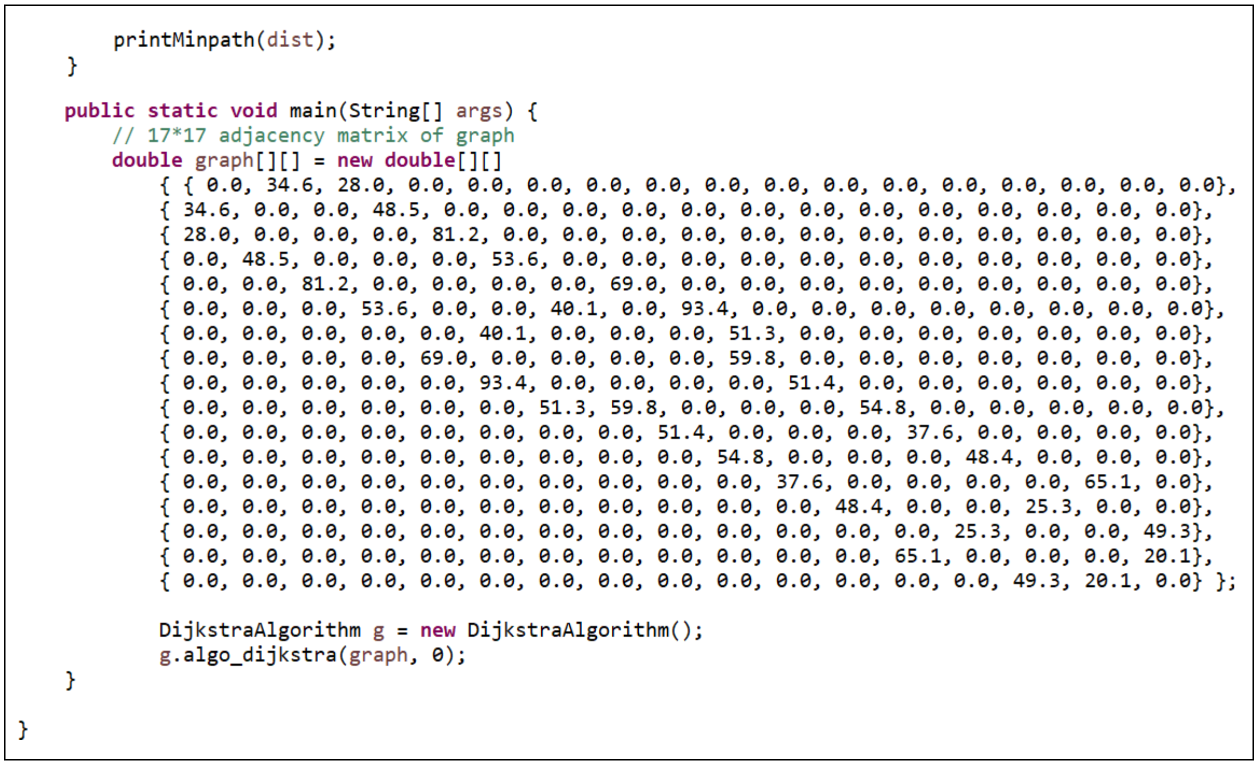
Appendix B

Java Source Code:



```java
package dsproject;

import java.text.DecimalFormat;

public class DijkstraAlgorithm {
    static final int num_Vertices = 17;

    // Method to find minimum distance
    double minDistance(double dist[], Boolean sptSet[]) {
        // Initialize minimum value
        double min = Integer.MAX_VALUE, min_index = -1;

        for (int v = 0; v < num_Vertices; v++)
            if (sptSet[v] == false && dist[v] <= min) {
                min = dist[v];
                min_index = v;
            }

        return min_index;
    }

    // Method to print the result
    void printMinpath(double dist[]) {
        DecimalFormat df = new DecimalFormat("0.0"); // Decimal format output
        System.out.println("Vertex# \t Shortest Distance from Source Vertex");

        for (int i = 0; i < num_Vertices; i++)
            System.out.println(i + " \t\t\t " + df.format(dist[i]));
    }

    // Method to implement the Dijkstra algorithm into the graph
    void algo_dijkstra(double graph[][], int src) {
        double dist[] = new double[num_Vertices];
        Boolean sptSet[] = new Boolean[num_Vertices];

        // Initialize all distances as INFINITE and sptSet[] as false
        for (int i = 0; i < num_Vertices; i++) {
            dist[i] = Integer.MAX_VALUE;
            sptSet[i] = false;
        }

        // Distance to source vertex is set as 0
        dist[src] = 0;

        for (int count = 0; count < num_Vertices - 1; count++) {
            double u = minDistance(dist, sptSet);
            sptSet[(int) u] = true; // The vertex is included in sptSet

            // Update the distance value of all adjacent vertices of picked vertex (u)
            for (int v = 0; v < num_Vertices; v++) {

                // If sum of distance value of u (from source) and weight of edge u-v is
                // less than the distance value of v, then update the distance value of v
                if (!sptSet[v] && graph[(int) u][v] != 0 && dist[(int) u] != Integer.MAX_VALUE
                    && dist[(int) u] + graph[(int) u][v] < dist[v]) {
                    dist[v] = dist[(int) u] + graph[(int) u][v];
                }
            }
        }
```

Figure 19: Java source code



```
        printMinpath(dist);
    }

    public static void main(String[] args) {
        // 17*17 adjacency matrix of graph
        double graph[][] = new double[][]
            { { 0.0, 34.6, 28.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            { 34.6, 0.0, 0.0, 48.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            { 28.0, 0.0, 0.0, 0.0, 81.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            { 0.0, 48.5, 0.0, 0.0, 0.0, 53.6, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            { 0.0, 0.0, 81.2, 0.0, 0.0, 0.0, 0.0, 69.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            { 0.0, 0.0, 0.0, 53.6, 0.0, 0.0, 40.1, 0.0, 93.4, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            { 0.0, 0.0, 0.0, 0.0, 0.0, 40.1, 0.0, 0.0, 0.0, 51.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            { 0.0, 0.0, 0.0, 0.0, 69.0, 0.0, 0.0, 0.0, 0.0, 59.8, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            { 0.0, 0.0, 0.0, 0.0, 0.0, 93.4, 0.0, 0.0, 0.0, 0.0, 51.4, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
            { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 51.3, 59.8, 0.0, 0.0, 0.0, 54.8, 0.0, 0.0, 0.0, 0.0, 0.0},
            { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 51.4, 0.0, 0.0, 0.0, 37.6, 0.0, 0.0, 0.0, 0.0},
            { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 54.8, 0.0, 0.0, 0.0, 48.4, 0.0, 0.0, 0.0},
            { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 37.6, 0.0, 0.0, 0.0, 0.0, 65.1, 0.0},
            { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 48.4, 0.0, 0.0, 25.3, 0.0, 0.0},
            { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 25.3, 0.0, 0.0, 49.3},
            { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 65.1, 0.0, 0.0, 0.0, 20.1},
            { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 49.3, 20.1, 0.0} };

            DijkstraAlgorithm g = new DijkstraAlgorithm();
            g.algo_dijkstra(graph, 0);
    }

}
```

Figure 20: Java source code (cont')