

**DISCRETE MATHEMATICS**

**TEB1053 / TDB1013**

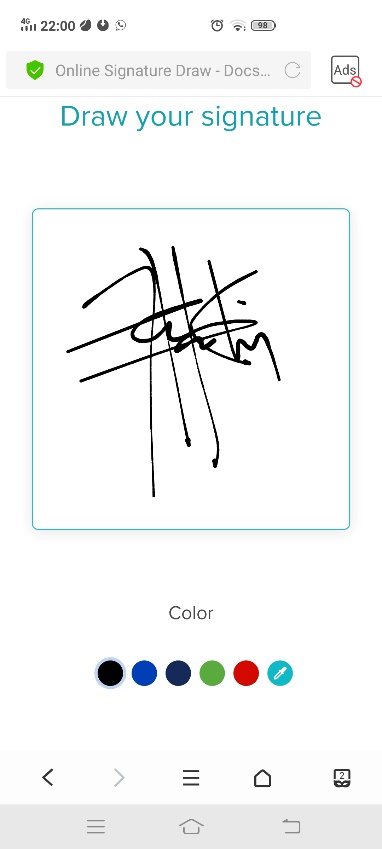
**January 2021 Semester**

|  |  |  |
| --- | --- | --- |
| Name | Student ID | Photo |
| Tan Hon Chi | 19000788 |  |
| Tan Yu Ting | 19000505 |  |
| Rayan Roy A/L Edward | 19000652 |  |
| Muhammad Aiman Bin Mohd Suhaimi | 19000840 |  |
| Amir Afiq Bin Farhan Muaddi | 20000431 |  |

**Computer Project Title – Question 5**

**Certification of Originality**

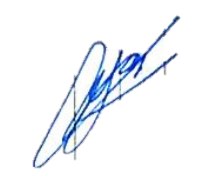
I, Tan Hon Chi (19000788) confirm that I have actively involved in this assignment and I have attended all the discussion meetings. Besides, I acknowledge that this work is the result of our own work.

Signature: 

I, Tan Yu Ting (19000505) confirm that I have actively involved in this assignment and I have attended all the discussion meetings. Besides, I acknowledge that this work is the result of our own work.

Signature: 

I, Rayan Roy A/L Edward (19000652) confirm that I have actively involved in this assignment and I have attended all the discussion meetings. Besides, I acknowledge that this work is the result of our own work.

Signature: 

I, Muhammad Aiman Bin Mohd Suhaimi (19000840) confirm that I have actively involved in this assignment and I have attended all the discussion meetings. Besides, I acknowledge that this work is the result of our own work.

Signature: 

I, Amir Afiq Bin Farhan Muaddi (20000431) confirm that I have actively involved in this assignment and I have attended all the discussion meetings. Besides, I acknowledge that this work is the result of our own work.

Signature: 

**Table of Content**

|  |  |  |
| --- | --- | --- |
| **No.** | **Description** | **Page No.** |
| **1.0** | **Problem Description** | **5** |
| **2.0** | **Objectives** | **6** |
| **3.0** | **Matrix Graph** | **7 – 10** |
| **4.0** | **Pseudocode Used** | **11 – 16** |
| **5.0** | **Source codes in Mathematica** | **17 – 27** |
| **6.0** | **Screenshots of Input / Output** | **28** |
| **7.0** | **Analysis of Results** | **29 – 45** |
| **8.0** | **Conclusion** | **46** |
| **9.0** | **References** | **47** |

1. **Problem Description**

Travelling salesman problem is about a salesman who complete his works by travelling a number of cities and does not mainly care what is the factors that have to be thought in this journey. The salesman will just keep travelling until all the cities have been reached by him. However, this journey will not bring any benefits to his work because the way he completes his work will spend a lot of time. Wasting time on unplanned travel will not only affect the work efficiency, but also will face the problem of delayed work. Besides, the long journey will bring higher cost travel which will cause the salesman may not be able to effort it. This is because the more distance he travels, the more fuel to be added, and this will cause higher cost on refilling the fuel. Therefore, to make sure the long journey will not bring any bad effects to the salesman, Dijkstra’s Shortest Path Algorithm is suitable to be used to find and count the shortest path between the source node and all other nodes in the journey. By having Dijkstra’s Shortest Path Algorithm, the cost travel is able to be controlled and will not exceed the budget. Apart from that, it is also able to shorten the travel distance and minimize the time travel.

\

1. **Objective**

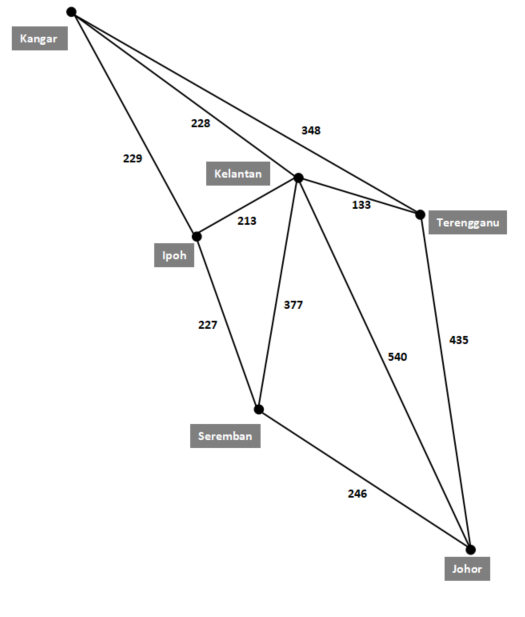
Travelling while working to several cities in Malaysia is an interesting and enjoyable experience which is also need a complete planning on several factors so that the journey will not be too rushing or troublesome. Besides, to avoid on having high cost and problem on wasting time, Dijkstra’s Shortest Path Algorithm is suitable to be used so that the salesman will have a better journey. Dijkstra’s Shortest Path Algorithm is said to be important for this work because it is able to count and provide the shortest path between other nodes and the departure node which represents as source node. Other that than, the weights of every edge will be used to count and so that the shortest path of the journey can be counted and provided. By having a planned journey, work quality will be improved and better experience on travelling will be experienced by the salesman.

1. **Matrix graph**

Named the cities and connect all the cities creating possible routes.



Create a graph and assigned actual distances.



* 1. **Adjacency Matrix Graph**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| NODE | KANGAR | TERENGGANU | KELANTAN | IPOH | SEREMBAN | JOHOR |
| KANGAR | 0 | 348 | 228 | 229 | 0 | 0 |
| TERENGGANU | 348 | 0 | 133 | 0 | 0 | 435 |
| KELANTAN | 228 | 133 | 0 | 213 | 377 | 435 |
| IPOH | 229 | 0 | 213 | 0 | 227 | 0 |
| SEREMBAN | 0 | 0 | 377 | 227 | 0 | 246 |
| JOHOR | 0 | 435 | 540 | 0 | 246 | 0 |

* 1. **Number of Vertices**

Definition of vertices: a vertex (node) is the fundamental unit of which graphs are formed.

1. Kangar
2. Tereangganu
3. Kelantan
4. Ipoh
5. Seremban
6. Johor

There are a total of **6 vertices**.

* 1. **Total Number of Edges**

Definition of edge: an edge is a connecting point between two vertices.

1. Kangar – Ipoh
2. Kagnar – Kelantan
3. Kangar – Terangganu
4. Ipoh – Kelantan
5. Ipoh – Seremban
6. Kelantan – Terangganu
7. Kelantan – Johor
8. Kelantan – Seremban
9. Terengganu – Johor
10. Seremban – Johor

There are a total of **10 edges**.

* 1. **Degree of Each Node / Vertex**

Definition of degree: the degree of a vertex is the number of edges connecting to a vertex.

1. Kangar = 3
2. Ipoh = 3
3. Kelantan = 5
4. Teregganu = 3
5. Seremban =3
6. Johor =3
7. **Pseudocode Used**

1.0 Start.

2.0 Import the defaultdict and deque functions from library named collections.

3.0 Create a class named Graph.

4.0 Define all the functions needed in the class Graph.

5.0 Define function \_\_init\_\_ as the constructor with the parameters of variables self and vertices.

5.1 Assign the variables self.V, self.graph, self.all\_destination\_name, self.edges and self.distances.

6.0 Define addEdge function that has parameters of variables self, u and v where u and v are the integer value representing the name of cities.

6.1 To append the edges for first graph (all paths graph)

6.2 Append the nodes into the list named graph

7.0 Define add\_node function that has parameters of variables self and value where the value is name of node.

7.1 To assign the nodes for second graph (shortest path graph)

7.2 Assign the name of cities into the set named all\_destination\_name

8.0 Define add\_edge function that has parameters of variables self, from\_node, to\_node and weight\_destination. From\_node and to\_node are the name of nodes while weight\_destination is the distance between the two particular nodes.

8.1 Append the edges between two nodes which are from\_node and to\_node as undirected graph into self.edges.

8.2 Assign the variable weight\_destination which is the value of distance between two nodes (from\_node and to\_node) into self.destination.

9.0 Define printAllPaths function that has the parameters of variables self, s and d where s is the integer for starting node and d is the integer for destination(ending node).

9.1 Define list named visited and mark all the vertices as FALSE (not visited).

9.2 Create an empty list named path.

9.3 Call a function named printAllPathsUtil.

10.0 Define printAllPathsUtil function that has the parameters of variables self, u, d, visited, path. Variable u is the integer for starting point and d is the integer for destination(ending point). Variable visited and path are the lists that will be used in the function.

10.1 Assign the element with the index of u in the list named visited to be TRUE.

10.2 Append the integer representing the starting point into the list named path.

10.3 If the number for starting point equals to number for destination

10.3.1 Convert the elements in path into type of string and store into list named outputs.

10.3.2 Define variable x that stores the value of length of outputs list

10.3.3 For each elements in the outputs list

10.3.3.1 Convert the string of number into its represented city name.

10.3.4 Append an empty string into the outputs list.

10.3.5 Assign variable sum and initialize to zero.

10.3.6 For the variable i with values starting from 0 to (length of path list-1)

10.3.6.1 Compare the elements of outputs with the index of i and i+1

10.3.6.1.1 Increase the value of sum according to the string values available in the index of i and i+1 for outputs

10.3.6.1.2 Break the loop is the element at index of i+1 is an empty string.

10.3.6.2 Delete the last element in the outputs list

10.3.6.3 Display the value of variable sum and all the elements of outputs list.

10.4 else

10.4.1 Assign variable i for each elements in self.graph list with the index of starting point (u)

10.4.1.1 If the element in visited list with index of i is FALSE

10.4.1.1.1 Call back the printAllPathsUtil function (own function )

10.5 Delete the last element of list named path

10.6 Assign the element in visited list with the index of starting point(u) to become FALSE

11.0 Define dijkstra function that has the parameters of variables graph and starting. Variable is the object in class Graph (graph2) and starting is the name of starting point.

11.1 Assign dictionary named visited with the name of starting point as key and 0 as the value. Key of the dictionary representing city names while the value represent distance.

11.2 Declare an empty dictionary named path.

11.3 Declare a set named all\_destination\_name with all the city names

11.4 While there is value in the all\_destination\_name

11.4.1 Declare variable min\_distance\_destination and initialize to None.

11.4.2 Declare variable node for all the elements in the set all\_destination\_name

11.4.2.1 If the variable node is in the visited dictionary

14.2.1.1 If the min\_distance\_destination has no value

14.2.1.1.1 Assign the value of node into min\_distance\_destination.

14.2.1.2 Else, if the elements of the visited dictionary with the key of node is less than element with key of min\_distance\_destination.

11.4.3 Remove the value min\_distance\_destination from the set all\_destination\_name

11.4.4 Declare variable current\_weight and assigned with the value of element in visited dictionary with the key of min\_distance\_destination.

11.4.5 Declare variable edge and assigned with the elements in graph.edges

11.4.5.1 Call the try function.

11.4.5.1.1 Declare variable weight and assigned the value which is the sum of current\_weight and graph.distances.

11.4.5.2 Call the except function.

11.4.5.2.1 Call the continue function.

11.4.5.3 If the variable edge is not available in visited dictonary or the variable weight is smaller that the element of visited dictionary with the key of edge.

11.4.5.3.1 Assigned the value weight into the visited dictionary with key of edge.

11.4.5.3.2 Assigned min\_distance\_destination into the path dictionary with key of edge.

11.5 Return the visited and path dictionaries.

12.0 Define shortest\_path function that has the parameter of variables graph, origin and final\_destination. Variable origin is the name of starting point, final\_destination is the name of ending point and graph is the object.

12.1 Call the dijkstra function and the values which are dictionaries are returned from the function to the variables named visited and paths.

12.2 Declare a variable full\_path as deque.

12.3 Declare a variable \_destination and assigned with the element in paths dictionary with key of final\_destination.

12.4 While \_destination is not equal to origin(starting point)

12.4.1 Append the \_destination into full\_path by calling the function appendleft to append the value to the left end of the deque

12.4.2. Assign the elements in paths dictionary with key of \_destination to the variable \_destionation.

12.5 Append the origin into full\_path by calling the function appendleft to append the value to the left end of the deque.

12.6 Append the final\_destination into full\_path by calling the function append the append the value to the right end of the deque.

12.7 Display the value of visited dictionary with the key of final\_destination and display the variable full\_path.

13.0 Finish defining the class and all the methods.

14.0 Design and create first graph to determine all possible paths.

14.1 Declare an object g.

14.2 Call the addEdge function by using the object g.

15.0 Create a list named CITYnamelist and stored the cities name in it.

16.0 Prompt and get the variable username and username2 where username is the starting point and the username2 is the destination (ending point).

17.0 While the username/username2 is not in the CITYnamelist.

17.1 Prompt and get for username/username2 again.

18.0 Compare the username and username2 with the cities name using if statement.

18.1 Declare variable s for username and variable d for username2.

18.2 Assign respective integer for the variables s and d according to the values of username and username2.

19.0 Call the function printAllPaths using the object g.

20.0 Declare an object graph.

21.0 Declare a variable node for all the elements in CITYnamelist.

21.1 Call add\_node function using object graph to add the nodes.

22.0 Call add\_edge function using object graph

23.0 Call the function shortest\_path.

24.0 End.

1. **Source codes in Mathematica**

#combine all path and shortest path

from collections import defaultdict, deque

class Graph:

def \_\_init\_\_(self,vertices):

self.V= vertices

self.graph = defaultdict(list)

self.all\_destination\_name = set()

self.edges = defaultdict(list)

self.distances = {}

#for all path

def addEdge(self,u,v):

self.graph[u].append(v)

#self.graph[v].append(u) << can add this to convert directed graph to undirected

#for shortest

def add\_node(self, value):

self.all\_destination\_name.add(value)

#for shortest

def add\_edge(self, from\_node, to\_node, weight\_destination):

self.edges[from\_node].append(to\_node) # from Kangar in list to append Ipoh

self.edges[to\_node].append(from\_node) # from Ipoh in list to append Kangar

self.distances[(from\_node, to\_node)] = weight\_destination

self.distances[(to\_node, from\_node)] = weight\_destination

def printAllPaths(self,s, d):

#Marking all the vertices as not visited

visited =[False]\*(self.V)

#Create an array to store paths

path = []

#Calling a recursive function for printing all paths

self.printAllPathsUtil(s, d,visited, path)

def printAllPathsUtil(self, u, d, visited, path):

visited[u]= True

path.append(u)

if u == d:

outputs = [str(x) for x in path ]

x=len(outputs)

for i in range(x):

if(outputs[i]=="0"):

outputs[i]="Kangar"

if(outputs[i]=="1"):

outputs[i]="Ipoh"

if(outputs[i]=="2"):

outputs[i]="Kelantan"

if(outputs[i]=="3"):

outputs[i]="Terengganu"

if(outputs[i]=="4"):

outputs[i]="Seremban"

if(outputs[i]=="5"):

outputs[i]="Johor"

outputs.append("")

#print (outputs)

sum = 0

for i in range(x):

if(outputs[i]=="Kangar" and outputs[i+1]=="Ipoh"):

sum = sum + 229

if(outputs[i]=="Kangar" and outputs[i+1]=="Kelantan"):

sum = sum + 228

if(outputs[i]=="Kangar" and outputs[i+1]=="Terengganu"):

sum = sum + 348

if(outputs[i]=="Ipoh" and outputs[i+1]=="Kangar"):

sum = sum + 229

if(outputs[i]=="Ipoh" and outputs[i+1]=="Kelantan"):

sum = sum + 213

if(outputs[i]=="Ipoh" and outputs[i+1]=="Seremban"):

sum = sum + 227

if(outputs[i]=="Kelantan" and outputs[i+1]=="Kangar"):

sum = sum + 228

if(outputs[i]=="Kelantan" and outputs[i+1]=="Ipoh"):

sum = sum + 213

if(outputs[i]=="Kelantan" and outputs[i+1]=="Terengganu"):

sum = sum + 133

if(outputs[i]=="Kelantan" and outputs[i+1]=="Seremban"):

sum = sum + 377

if(outputs[i]=="Kelantan" and outputs[i+1]=="Johor"):

sum = sum + 540

if(outputs[i]=="Terengganu" and outputs[i+1]=="Kangar"):

sum = sum + 348

if(outputs[i]=="Terengganu" and outputs[i+1]=="Kelantan"):

sum = sum + 133

if(outputs[i]=="Terengganu" and outputs[i+1]=="Johor"):

sum = sum + 435

if(outputs[i]=="Seremban" and outputs[i+1]=="Ipoh"):

sum = sum + 227

if(outputs[i]=="Seremban" and outputs[i+1]=="Kelantan"):

sum = sum + 377

if(outputs[i]=="Seremban" and outputs[i+1]=="Johor"):

sum = sum + 246

if(outputs[i]=="Johor" and outputs[i+1]=="Seremban"):

sum = sum + 246

if(outputs[i]=="Johor" and outputs[i+1]=="Kelantan"):

sum = sum + 540

if(outputs[i]=="Johor" and outputs[i+1]=="Terengganu"):

sum = sum + 435

if(outputs[i]=="Kangar" and outputs[i+1]==""):

break

if(outputs[i]=="Ipoh" and outputs[i+1]==""):

break

if(outputs[i]=="Kelantan" and outputs[i+1]==""):

break

if(outputs[i]=="Terangganu" and outputs[i+1]==""):

break

if(outputs[i]=="Seremban" and outputs[i+1]==""):

break

if(outputs[i]=="Johor" and outputs[i+1]==""):

break

outputs.pop()

print(sum, end = " km\t: ")

for i in range(x):

outputs[i] = outputs[i].upper()

print (outputs)

else:

for i in self.graph[u]:

if visited[i]==False:

self.printAllPathsUtil(i, d, visited, path)

path.pop()

visited[u]= False

def dijkstra(graph, starting):

visited = {starting: 0}

print("%s"%visited)

path = {}

all\_destination\_name = set(graph.all\_destination\_name)

while all\_destination\_name :

min\_distance\_destination= None

for node in all\_destination\_name :

if node in visited:

if min\_distance\_destination is None:

min\_distance\_destination = node

elif visited[node] < visited[min\_distance\_destination]:

min\_distance\_destination = node

if min\_distance\_destination is None:

break

# remove the min\_distance\_destination from the set of nodes

all\_destination\_name .remove(min\_distance\_destination)

current\_weight = visited[min\_distance\_destination]

for edge in graph.edges[min\_distance\_destination]:

try:

weight = current\_weight + graph.distances[(min\_distance\_destination, edge)]

except:

continue

if edge not in visited or weight < visited[edge]:

visited[edge] = weight

path[edge] = min\_distance\_destination

return visited, path

def shortest\_path(graph, origin, final\_destination):

visited, paths = dijkstra(graph, origin)

full\_path = deque()

\_destination = paths[final\_destination]

while \_destination != origin:

full\_path.appendleft(\_destination)

\_destination = paths[\_destination]

full\_path.appendleft(origin)

full\_path.append(final\_destination)

print("%s km , %s" %(visited[final\_destination], list(full\_path)))

#main function

# kangar 0 ipoh 1 kelantan 2 terenggganu 3 SEREMBAN 4JOHOR 5

# Graph 1

g = Graph(11)

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(0, 3)

g.addEdge(1, 2)

g.addEdge(1, 4)

g.addEdge(2, 3)

g.addEdge(2, 4)

g.addEdge(4, 5)

g.addEdge(3, 5)

g.addEdge(2, 5)

#Add the bottom part for undirected

g.addEdge(1, 0)

g.addEdge(2, 0)

g.addEdge(3, 0)

g.addEdge(2, 1)

g.addEdge(4, 1)

g.addEdge(3, 2)

g.addEdge(4, 2)

g.addEdge(5, 4)

g.addEdge(5, 3)

g.addEdge(5, 2)

CITYnamelist = ["KANGAR", "IPOH", "KELANTAN", "TERENGGANU", "SEREMBAN", "JOHOR"]

print("Available city :", end="")

print(CITYnamelist)

print("City A : Departure")

print("City B : Destination")

print("\n")

username = input("CITY A : ").upper()

while(username not in CITYnamelist):

username = input("CITY A :").upper()

username2 = input("CITY B : ").upper()

while(username2 not in CITYnamelist):

username2 = input("CITY B :").upper()

print("\nCity A to City B : %s -------> %s "%(username,username2))

print("-------------------------------------------------------")

print("\nTo display all possible paths : ")

if(username == "KANGAR"): s=0

if(username == "IPOH"): s=1

if(username == "KELANTAN"): s=2

if(username == "TERENGGANU"): s=3

if(username == "SEREMBAN"): s=4

if(username == "JOHOR"): s=5

if(username2 == "KANGAR"): d=0

if(username2 == "IPOH"): d=1

if(username2 == "KELANTAN"): d=2

if(username2 == "TERENGGANU"): d=3

if(username2 == "SEREMBAN"): d=4

if(username2 == "JOHOR"): d=5

print ("\nThese are the all paths from node %s to %s : " %(username, username2))

g.printAllPaths(s, d)

#second graph for shortest path

graph = Graph(11)

for node in CITYnamelist:

graph.add\_node(node)

#Graph 2

graph.add\_edge("KANGAR", "IPOH", 229)

graph.add\_edge("KANGAR", "KELANTAN", 228)

graph.add\_edge("KANGAR", "TERENGGANU" , 348)

graph.add\_edge("IPOH", "KELANTAN", 213)

graph.add\_edge("IPOH", "SEREMBAN", 227)

graph.add\_edge("KELANTAN", "TERENGGANU", 133)

graph.add\_edge("KELANTAN", "SEREMBAN", 377)

graph.add\_edge("SEREMBAN", "JOHOR", 246)

graph.add\_edge("TERENGGANU", "JOHOR", 435)

graph.add\_edge("KELANTAN", "JOHOR", 540)

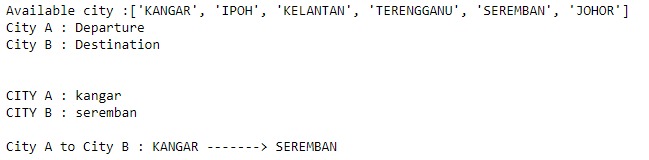
print("-------------------------------------------------------")

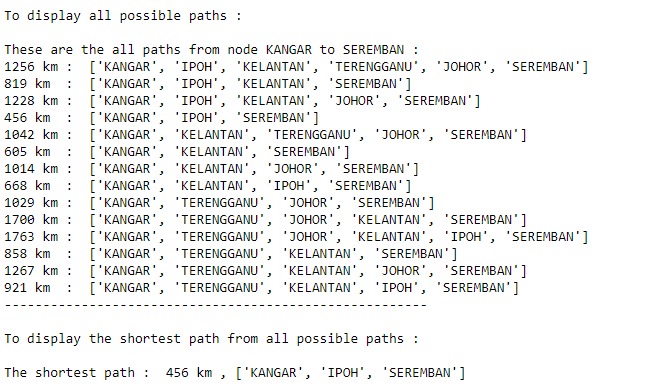
print("\nTo display the shortest path from all possible paths : ")

print("\nThe shortest path : ", end=" ")

shortest\_path(graph, username, username2)

1. **Screenshots of Input / Output**





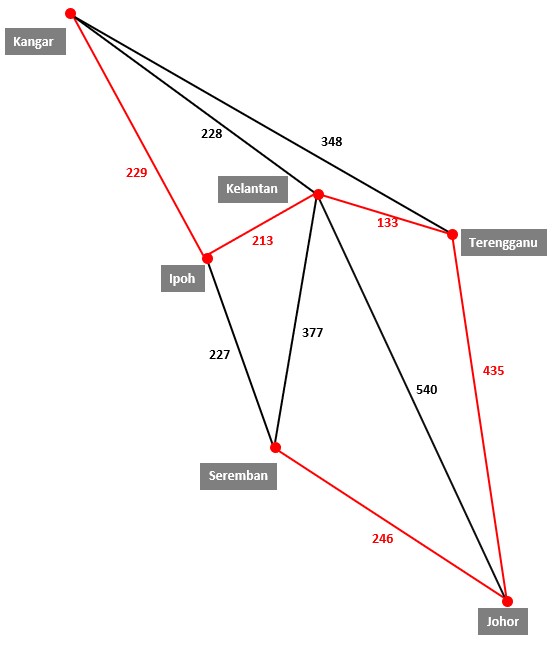
1. **ANALYSIS OF RESULTS**

OUTPUTS OF THE ABOVE CODING

Input City A : Kangar

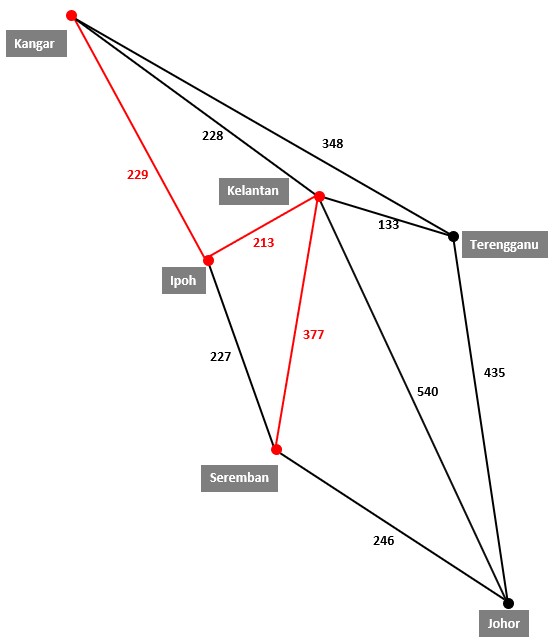
Input City B : Seremban

Total outputs: 14 ways



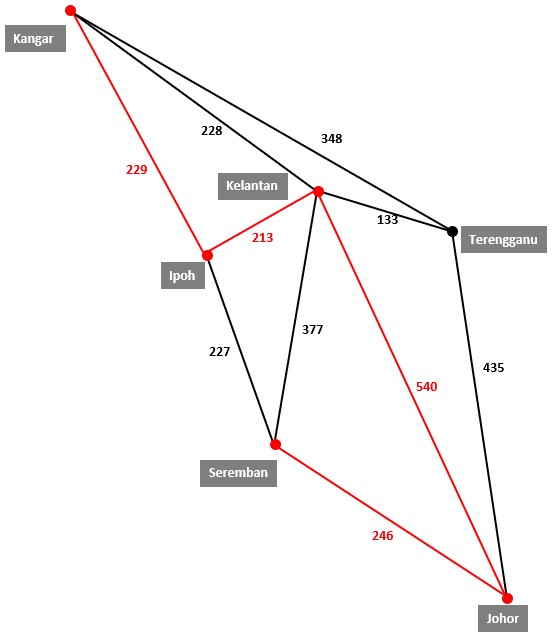
Kangar-Ipoh-Kelantan-Terengganu-Johor-Seremban

229+213+133+435+246 = 1256 KM



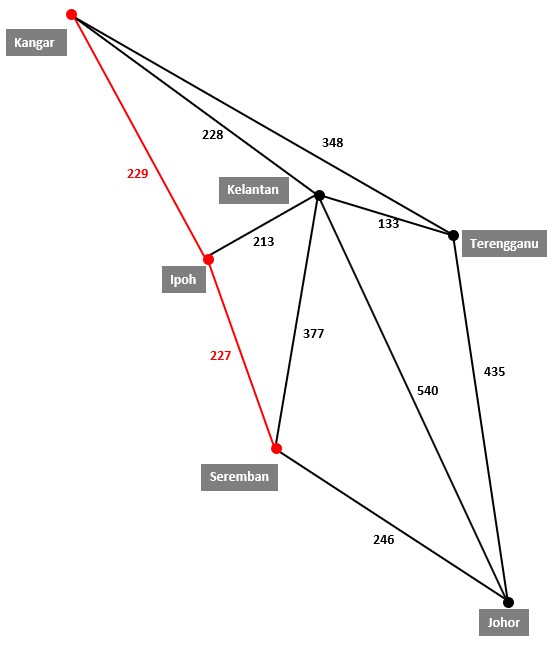
Kangar-Ipoh-Kelantan-Seremban

229+213+377 = 819 Km



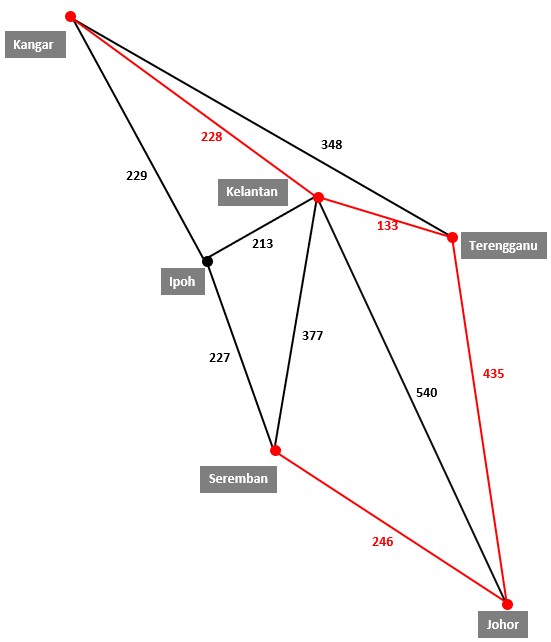
Kangar-Ipoh-Kelantan-Johor-Seremban

229+213+540+246 = 1228 Km



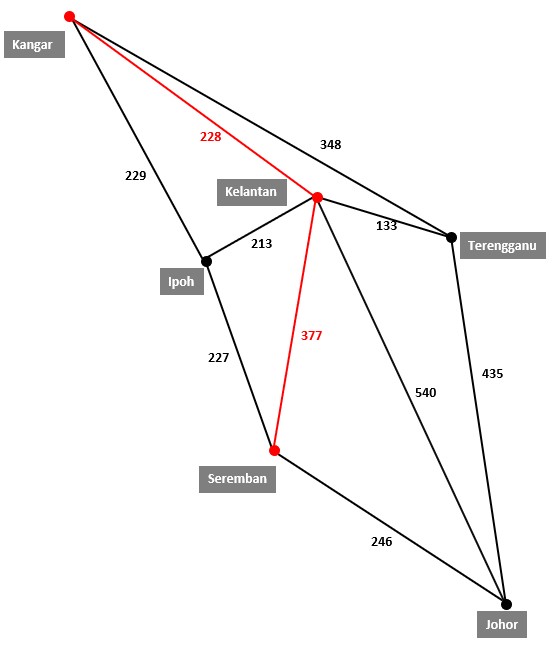
Kangar-Ipoh-Seremban

229+227 = 456 Km



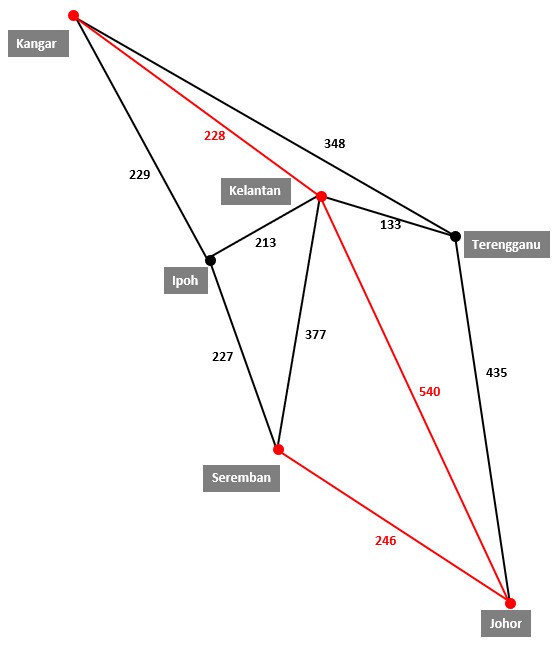
Kangar-Kelantan-Terengganu-Johor-Seremban

228+133+435+246 = 1042 Km



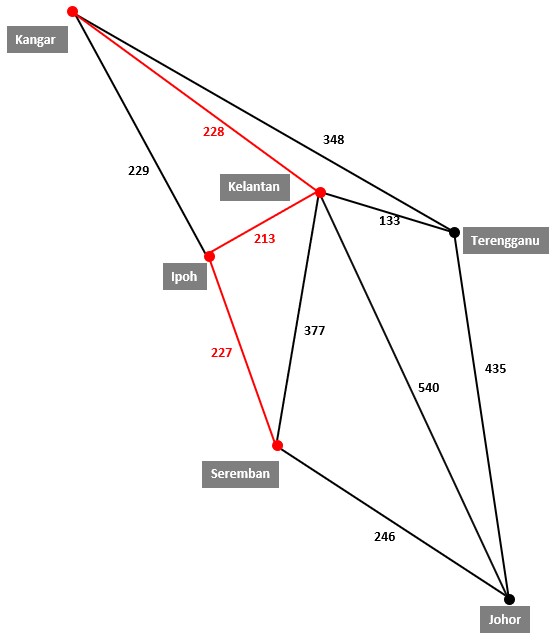
Kangar-Kelantan-Seremban

228+377 = 605 Km



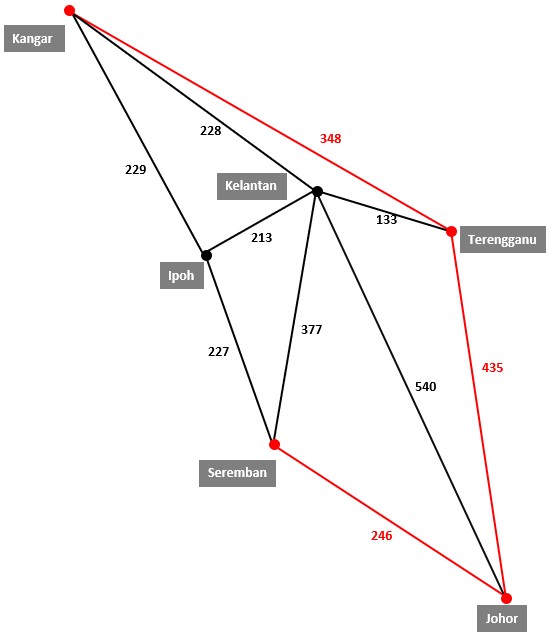
Kangar-Kelantan-Johor-Seremban

228+540+246 = 1014 Km



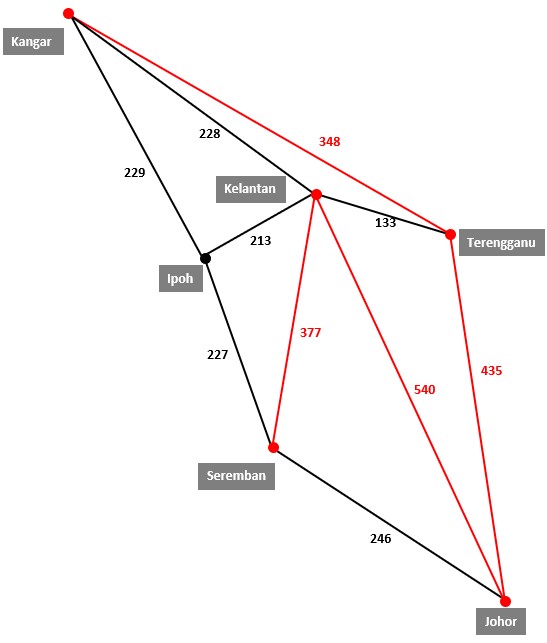
Kangar-Kelantan-Ipoh-Seremban

228+213+227 = 668 Km



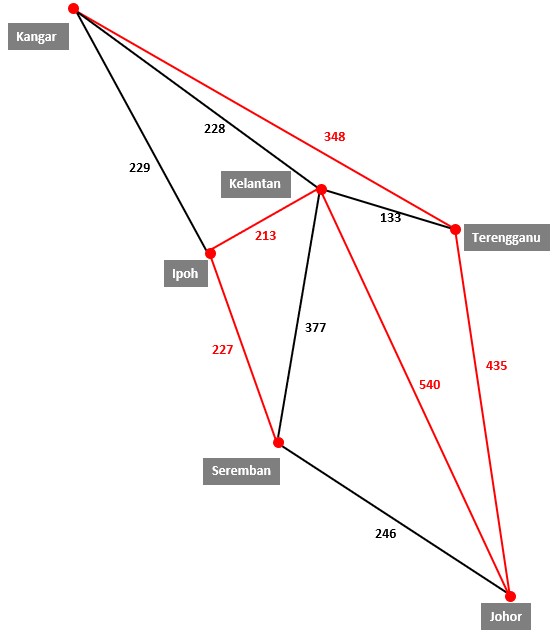
Kangar-Terengganu-Johor-Seremban

348+435+246 = 1029 Km



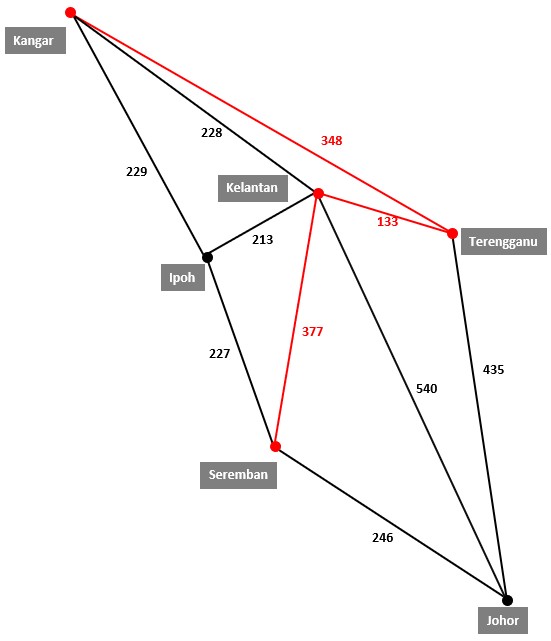
Kangar-Terengganu-Johor-Kelantan-Seremban

348-435-540-377 = 1700 Km



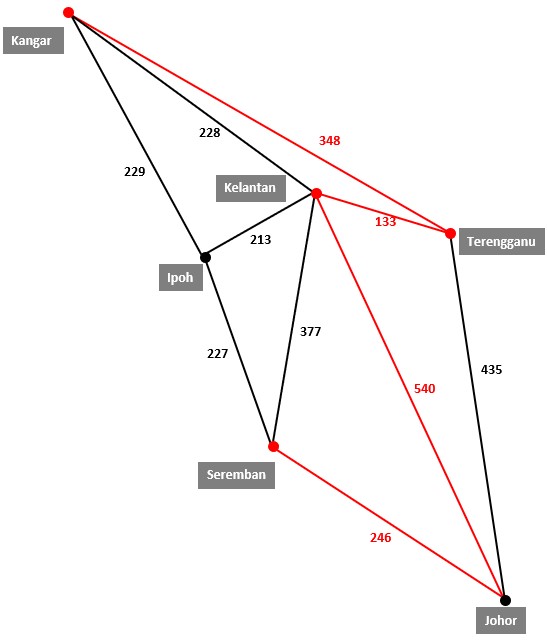
Kangar-Terengganu-Johor-Kelantan-Ipoh-Seremban

348+435+540+213+227 = 1763 Km



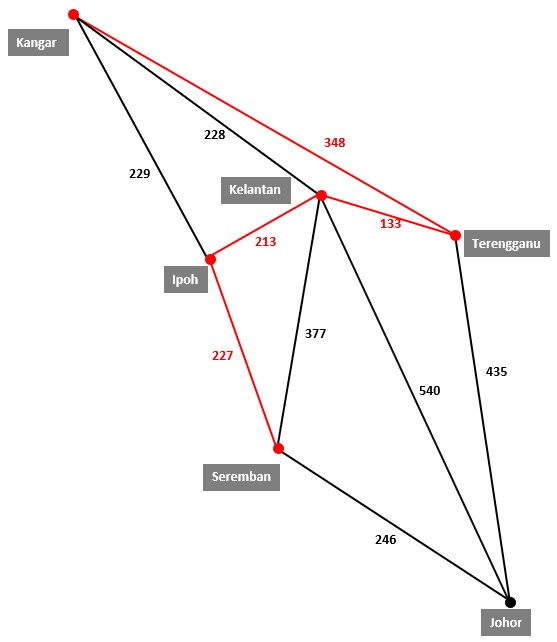
Kangar-Terengganu-Kelantan-Seremban

348+133+377 = 858 Km



Kangar-Terengganu-Kelantan-Johor-Seremban

348+133+540+246 = 1267 Km

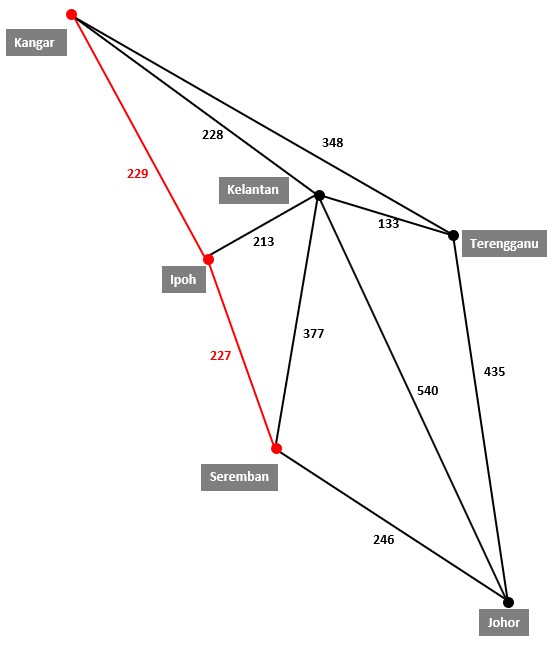


Kangar-Terengganu-Kelantan-Ipoh-Seremban

348+133+213+227 = 921 Km

* 1. **Analysis of Results**

Based on the results shown, the graphs were created by connecting the vertices through edges via adjacency matrix. The **shortest vertex** from the coding given would give a diagram of:



These 3 vertices are connected through vertices via a sequence of matrices. The first vertex is labeled as Kangar, and this vertex is connected to only one other vertex, Ipoh.

From the input of the coding, all six vertices represent 6 different states. Every vertex is interconnected by edges with different values. The value of the edges, in the diagram above, represents the distance, in km, of the states between one another. Kelantan is in the middle of the map, which is connected to all the other states, Kangar, Terengganu, Johor, Seremban, and Ipoh. The rest of the states on the other hand, are mainly connected to a maximum of three other states. Kangar is only connected to three states, Ipoh, Kelantan and Terengganu. Ipoh is connected to Kangar, Kelantan and Seremban. Seremban is connected to Ipoh, Kelantan and Johor. Johor is connected to Seremban, Kelantan and Terengganu. Terengganu is connected to Johor, Kelantan and Kangar.

Based on the graph, it is clearly shown the distance from each city varies. Kelantan to Terengganu is134km, Kelantan to Ipoh is 135.35km, Kelantan to Johor is 479.64km, Kelantan to Seremban is 297.70km, Kelantan to Kangar is 231.06 km, Ipoh to Kangar is 226.91km, Ipoh to Seremban is 229.93km, Johor to Seremban is 246.26km, Johor to Terengganu is 434.95km, Terengganu to Kangar is 348.15 km. Thus, the purpose of the coding is to present the shortest path travelled from one city to the other as well as the total distance travelled for those specific cities.

As shown in the input and the output of the coding, the shortest path can be easily observed. As reference to the input, Kangar -> Seremban, the shortest path from Kangar to Seremban can easily be determined. The shortest path from Kangar to Seremban is by passing by Ipoh, which give a total distance of 456km whereas the furthest distance from Kangar to Seremban is by going through Terengganu, Kelantan, Johor, which gives a total distance of 1267km.

By taking the shortest path, we will be able to save on many resources. These resources include fuel, money, time and energy. Unnecessary usage of fuel can be avoided here by taking the shortest path, which also reduces the amount of money used to purchase petrol, thus, reducing carbon footprint. A shorter time taken to travel from different cities mean lesser amount of energy used. Energy in the form of fuel intake by the vehicles as well as energy needed during the drive. In conclusion, taking the shortest path provides many benefits.

1. **Conclusion**

To have the fastest and shortest journey which is from departure city, Kangar to destination city, Seremban, the journey from Kangar to Ipoh then to Seremban is the most suitable path that can be travelled. This result is achieved by having the concept Dijkstra’s Shortest Path Algorithm which is used to calculate the shortest path of the road. Taking the shortest path has many benefits such as the time taken for a journey will be shorter, cost of the journey will not exceed the budget and the uses on the fuel is able to be reduced. Besides, travelling salesman problem will also be solved so that the salesman will not face any problems on his sales work.

1. **References**

**G**

GeeksforGeeks. (2020, January 23). *Python Program to convert List of Integer to List of String*. <https://www.geeksforgeeks.org/python-program-to-convert-list-of-integer-to-list-of-string/>

*Python Operators*. (n.d.-b). W3schools.Com. <https://www.w3schools.com/python/python_operators.asp>

*Python List/Array Methods*. (n.d.). W3schools.Com. <https://www.w3schools.com/python/python_ref_list.asp>

**N**

Navone, E. C. (2020, November 19). *Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction*. FreeCodeCamp.Org. <https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/#:%7E:text=Dijkstra’s%20Algorithm%20finds%20the%20shortest,node%20and%20all%20other%20nodes>

**R**

Raj, V. (2020, August 8). *Print all paths from a given source to a destination in Python*. CodeSpeedy. <https://www.codespeedy.com/print-all-paths-from-a-given-source-to-a-destination-in-python/>

Rosa, M. (2016). *Modified Python implementation of Dijkstra’s Algorithm (https://gist.github.com/econchick/4666413)*. Gist. <https://gist.github.com/mdsrosa/c71339cb23bc51e711d8>