# Problem 2: Graphs

Name: Muhammad Faraz Rashid  
Roll Num: I210659

Section: C

Introduction:

In this given problem this a modified version of the travelling salesman problem (TSP) which is a classic combinatorial optimization problem that has been studied extensively for many years. The modification is that we have a time constraint that we also have to consider for this problem. The problem asks for the shortest possible route that visits a set of vertices and returns to the vertices having visited each other vertex only once. TSP is a well-known NP-hard problem, meaning that there is no known algorithm that can solve the problem optimally in polynomial time.

Whilst researching for this problem it was discovered that many algorithms have been proposed to solve TSP, including dynamic programming methods such as the Held-Karp algorithm, heuristic algorithms such as simulated annealing, and others such as ant colony optimization.

In this report, a backtracking algorithm was chosen to solve TSP. While not always the most efficient algorithm for large-scale problems, backtracking offers a straightforward and easy-to-implement solution to the problem. In this report the pseudo code along with time complexities will be presented, followed by the resulting implementation, and finally how the test cases fared against.

## Pseudocode

## function getEdgeIndex(u, v):

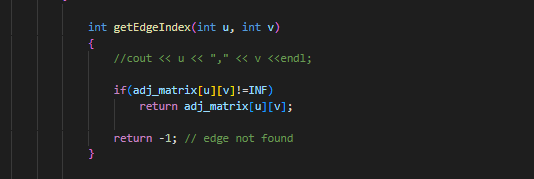
if adj\_matrix[u][v] is not INF:

return adj\_matrix[u][v]

else

return -1

Time Complexity: O(1)   
Basic Function Description:Returns the index from the adjacency matrix. If edge doesn’t exist i.e the adjacency matrix distance is infinity, it will return -1 to indicate the function doesn’t exist.



## function tspDFS():

best\_route = []

current\_route = [0]

visited = [False] \* vertices.size()

visited[0] = True

current\_time = 0

current\_distance = 0

best\_distance = INF

precalculation = 0

for i in range(vertices.size()):

precalculation += vertex\_times[i]

if precalculation > total\_time

print("Feasible Circuit Doesn't Exit ")

return

tspDFSUtil(best\_route, current\_route, visited, current\_time, current\_distance, best\_distance)

if best\_route:

print("Best Distance = ", best\_distance)

print("Best path: (", end="")

for i in range(len(best\_route)):

print(vertices[best\_route[i]], end=" ")

print(vertices[0], end="")

print(")")

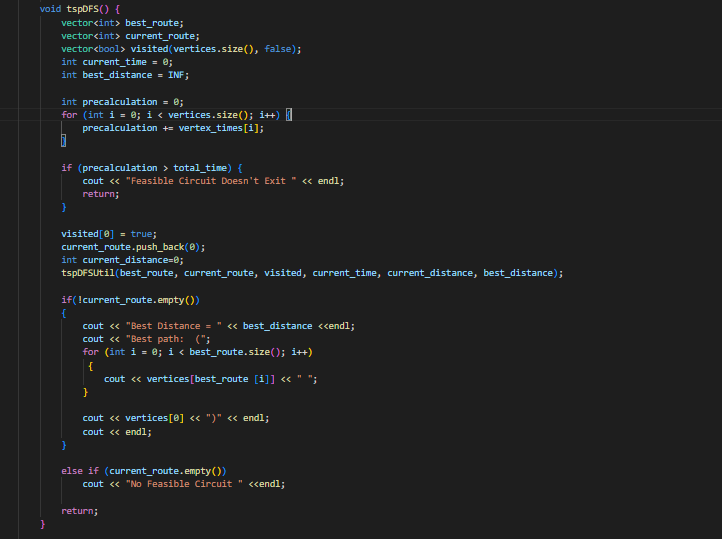
print()

else

print("No Feasible Circuit ")

### Time Complexity: O(n!)

### Basic Description: This function initializes the DFS traversal of the TSP solution space by setting up the initial parameters and calling the **tspDFSUtil()** function. It is a function to call the util function and display the output, hence the complexity is same as that of tspDFSUtil() which is also O(n!). It also firstly checks if the vertices time exceed total time, and if it does it returns from there directly. So in some cases like in test case 1 it returns the answer without even calling the util function thus saving us time and in which case the best case complexity is O(n)



### function tspDFSUtil(best\_route, current\_route, visited, current\_time, current\_distance, best\_distance):

if len(current\_route) == vertices.size():

total\_distance = calculateDistance(current\_route)

if total\_distance < best\_distance:

print("current best distance = ", best\_distance, " new best distance = ", total\_distance)

best\_distance = total\_distance

best\_route[:] = current\_route

return

for i in range(1, vertices.size()):

if not visited[i] and canVisit(vertices[i], current\_time, current\_route, current\_distance, best\_distance):

visited[i] = True

current\_route.append(i)

current\_time += vertex\_times[i]

tspDFSUtil(best\_route, current\_route, visited, current\_time, current\_distance, best\_distance)

current\_time -= vertex\_times[i]

current\_route.pop()

visited[i] = False

### Time Complexity: O(n!) Basic Description: This recursive function performs a depth-first search on the solution space of the TSP. It takes in the current best route, the current route being explored, the visited status of vertices, the current time, current distance traveled, and the best distance found so far. It explores all possible routes and updates the best distance and route if a better solution is found.

### function calculateDistance(route):

start\_index = find\_index(vertices[0])

last\_index = route[-1]

return\_index = adj\_matrix[last\_index][start\_index]

if return\_index == INF:

return INF

print("New route = ", end="")

total\_distance = 0

for i in range(1, len(route)):

u = route[i - 1]

v = route[i]

print(vertices[u], ",", vertices[v], "=", adj\_matrix[u][v], ",", end="")

if adj\_matrix[u][v] != INF:

total\_distance += adj\_matrix[u][v]

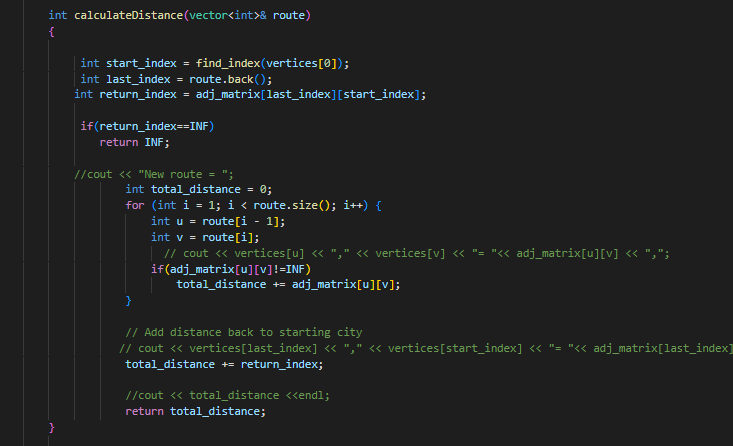
print(vertices[last\_index], ",", vertices[start\_index], "=", adj\_matrix[last\_index][start\_index], " total distance = ", end="")

total\_distance += return\_index

print(total\_distance)

return total\_distance

### Time Complexity: O(n) Basic Description: This function takes in a route and calculates the total distance traveled along that route by summing up the weights of the edges between vertices in the route.



### function canVisit(city, current\_time, current\_route, current\_distance, best\_time):

city\_index = find\_index(city)

last\_index = current\_route[-1]

if adj\_matrix[last\_index][city\_index] == INF:

return False

if adj\_matrix[last\_index][city\_index] + current\_distance > best\_time:

current\_distance -= adj\_matrix[last\_index][city\_index]

return False

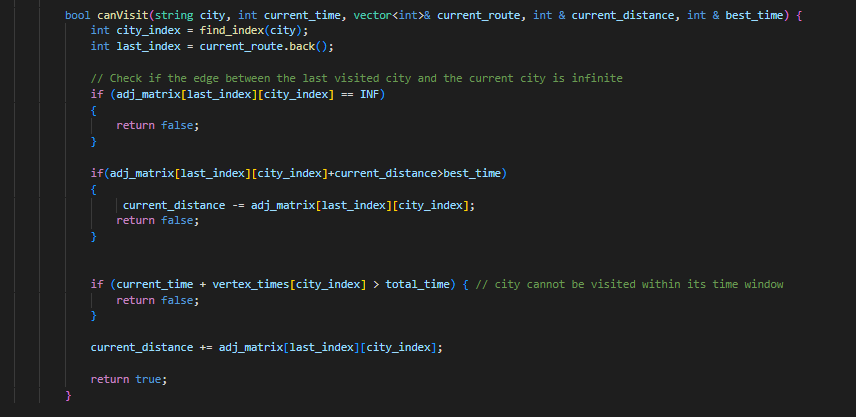
if current\_time + vertex\_times[city\_index] > total\_time:

return False

current\_distance += adj\_matrix[last\_index][city\_index]

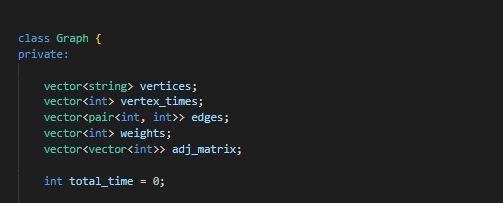
return True

### Time Complexity: O(1) Basic Description: It checks if the city can be visited at the current time and updates the current distance if it can be visited.



# Basic Data Structure and Reading File:

A graph structure was made using vectors and a set of basic functions and helper functions were created to read and construct the graph. A read file function was also made Overall, which has a time complexity of (m + n + k), where m is the number of edges, n is the number of vertices, and k is the number of vertices with specified times.

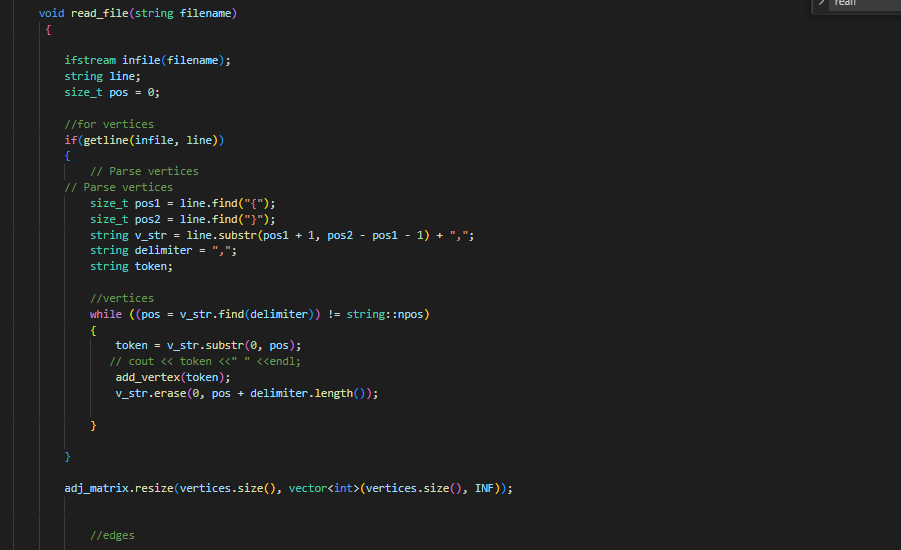


# 

# Overall Time Complexity:

The overall time complexity is O(n!) in the worst case where it visits all vertices. In the best case it is O(n) when it exists due to the time of each vertex being less than the total time.

Problems and Possible Improvements:  
  
The time complexity could have been improved if we implemented some dynamic programming like in the held-karp algorithm, but implementation might be difficult. Furthermore, the tspUtil() function could be improved by removing the calculate distance function and instead relying on a calculate distance variable, which I had tried to utilize but the implementation wasn’t working properly without. One of the problems I had initially faced was that I had come up with an adjacency list in the start which made returning edges increase in time complexity to O(n) and I was able to overcome that by convert it to an adjacency matrix.

Snippets of code were written similarly to this using getline and while loops with delimiters as well as the find function to extract the data from the test cases.  
  


# 

For the sake of not lengthening the report too long the full function hasn’t been included and the other functions like printing graph and dfs have been left out.

# Output:

As you can see the output is coming out as intended. The second test case has multiple optimal paths that can be the solution and it is also an answer.

# References and Sources:

https://www.geeksforgeeks.org/travelling-salesman-problem-implementation-using-backtracking/  
  
https://www.youtube.com/watch?v=9AIUTe4IkD0