Trip Planner

By
- Mayuri Balajee (205002047)
- Gunanicaa Arun (205002035)

Problem Statement

- One of the main difficulties faced by many people is planning a proper trip.
- To solve this issue, the proposed project is a trip planner which gives people an optimized travel plan by finding places & trips that match the most with their preferences and by providing as narrowed-down suggestions as possible.
- This project will help users not have to compromise with their preferences/factors and skip the tedious task of researching on the internet.

Nature of Data and Processing

- List of places stored in Hash tables
- The hash keys arranged in priority order using priority queues based on different factors
- Edge weights have been given and the optimum path to visit all the places once and return to the starting point has been implemented

Input from the user:

- If they wish to specify factors that might affect their trip
- If yes, which factor out of the given options

Data Structures Used

- Graphs
- Hash Tables
- Priority Queues

Graphs

1. We use graphs to compute the shortest path between two locations
2. The optimum plan with what places to visit will be suggested
3. The most efficient path will be suggested to the user.
4. Time complexity is **O(V+E)** where V is the number of vertices in the graph and E is number of edges in the graph
5. Because of its non-linear structure, helps in understanding complex problems and their visualization.

```
def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)

    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)

    for i in next_permutation:
        #print(next_permutation)

        current_pathweight = 0

        k = s
        for j in i:
            current_pathweight += graph[k][j]
            #print(i)
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        min_path = min(min_path, current_pathweight)
        perm_arr.append(i)
        min_vals.append(min_path)
```

Graphs Comparison

Adjacency matrix, adjacency list, edges list

- **The adjacency matrix is most helpful in cases where the graph doesn't contain a large number of nodes.**
- **Adjacency lists, on the other hand, are a great option when we need to continuously access all the neighbors of some node *u*.**
- **we use edges lists when we have an enormous amount of nodes that can't be stored inside the memory, with only a few edges**

Usage of backtracking algorithm for TSP (NP-hard Problem)

- DP Approach Time complexity - O(n!)
  - Space complexity: Exponential
  - Not suitable for large no. of vertices
- Same time complexity for branch and bound
- Backtracking approach: O(n!)
  - Space complexity: O(N)

Hash Tables

1. The list of places to visit will be stored in a hash table
2. The hash table keys will hence be stored in a priority queue.
3. This is done as each place will have its corresponding key value, and hence easier to objectively choose the more important place to visit.
4. The main advantage of hash tables over other data structures is speed
5. The access time of an element is on average O(1)
6. Hash tables are particularly efficient when the maximum number of entries can be predicted in advance

```
# quadratic probing
def hashing(table, tsize, arr, N):

    for i in range(N):

        hv = len(arr[i]) % tsize

        if (table[hv] == -1):
            table[hv] = arr[i]

        else:

            for j in range(tsize):

                t = (hv + j * j) % tsize

                if (table[t] == -1):

                    table[t] = arr[i]
                    break

    printArray(table, L)
```

Hash Table - Collision Solution

Chaining, Open Addressing: Linear Probing, Quadratic Probing, Double Hashing

- Separate chaining can lead to empty spaces in the table, the list in the positions can be very long
- Linear Probing causes clustering
- Double hashing can cause thrashing (system spend major time on solving the faults)

- QB is easier to implement
- It solves the problem of clustering

Hash Table vs other data structures

- Binary Trees
  - medium complexity to implement (assuming you can't get them from a library)
  - inserts are O(logN)
  - lookups are O(logN)
- Linked lists (unsorted)
  - low complexity to implement
  - inserts are O(1)
  - lookups are O(N)
- Hash tables
  - high complexity to implement yet since this application is meant to have a large dataset
  - inserts are O(1) on average
  - lookups are O(1) on average

Priority Queues

1. Priority queue is used to store all the places in the hash table dataset and along with an auto generated priority.
2. This suggests the most important places to visit.
3. Additionally we have also incorporated additional priority values depending on the user requirements
4. The priority queue keeps changing and the optimized path will be displayed.
5. The major advantage of using a priority queue is that you will be able to quickly access the highest priority item with a time complexity of just O(1).
6. Time complexity: O(log n) - for inserting data

```
class PriorityQueueNode:

def __init__(self, value, pr):

    self.data = value
    self.priority = pr
    self.next = None
```

```
# Implementation of Priority Queue
class PriorityQueue:

    def __init__(self):

        self.front = None

    def isEmpty(self):

        return True if self.front == None else False

    def push(self, value, priority):

        if self.isEmpty() == True:

            self.front = PriorityQueueNode(value,
                                           priority)

            return 1

        else:
            if self.front.priority > priority:

                newNode = PriorityQueueNode(value,
                                            priority)

                newNode.next = self.front
                self.front = newNode
                return 1
```

```
            else:
                temp = self.front

                while temp.next:

                    if priority <= temp.next.priority:
                        break

                    temp = temp.next

                newNode = PriorityQueueNode(value,
                                            priority)
                newNode.next = temp.next
                temp.next = newNode

                return 1

    def pop(self):


        if self.isEmpty() == True:
            return

        else:


            self.front = self.front.next
            return 1
```

Inferences

- Usage of graphs to find the optimised path by using travelling salesman problem.
- Implementation of priority queue to find the places having highest preferences to visit, according to the factors given by the user.
- Hash tables are used to store the data set(list of places) where the time required to access the data is very less.

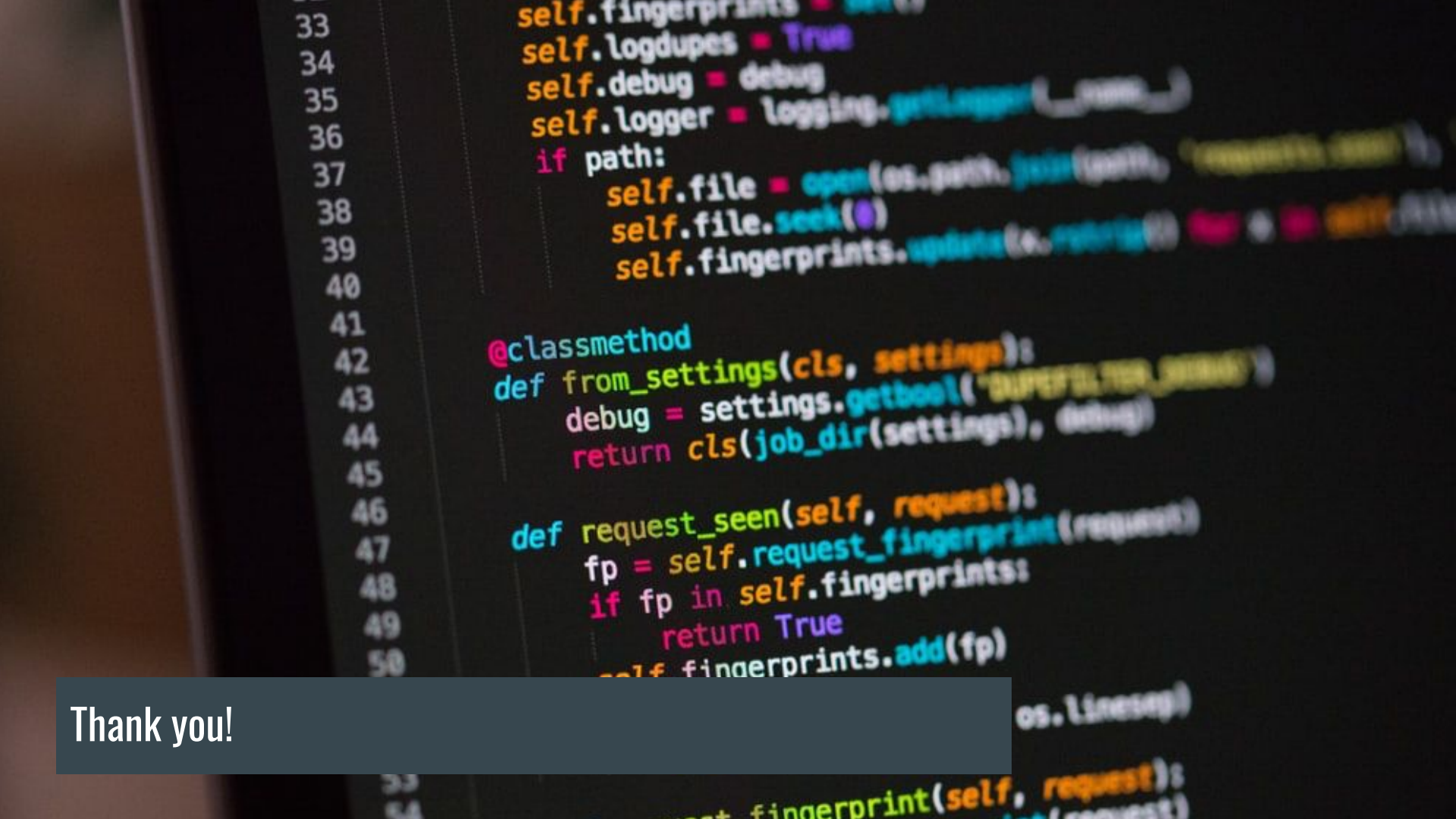
Gap Analysis

Our project currently is limited to a small sample space, and to only list of places in one city.

The data taken from the user is very minimal in the current program, which makes it less interactive and less personalized.

Possible Improvements

- Assigning weights based on a more detailed manner (graphs)
- Increasing the dataset to having more cities and places (hashtable)
- Incorporating more factors to decide the final plan and implementing an appropriate way to assign weights based on the data collected


Thank you!

Advanced Data Structures Project

Documentation

Team Members:

Gunanicaa Arun 205002035

Mayuri Balajee 205002047

TRIP PLANNER

1. Problem Statement:

One of the main difficulties faced by many people is planning a proper trip.
To solve this issue, the proposed project is a trip planner which gives people an optimized travel plan by finding places & trips that match the most with their preferences and by providing as narrowed-down suggestions as possible.
This project will help users not have to compromise with their preferences/factors and skip the tedious task of researching on the internet.

2. Nature of Data and Processing:

- The list of places are stored in a hash table.
- The hash keys arranged in priority order using priority queues based on different factors
- Edge weights have been given and the optimum path to visit all the places once and return to the starting point has been implemented

Input from the user:

Firstly the users have the option to choose if they want to select any factor or just let the project suggest a default solution.
If they want to choose a fator they can choose from the following:

a. Family friendly
b. Budget
c. Distance

3. Data Structures used:

Graphs:

Graph is a non-linear data structure consisting of nodes and edges. Here in this case, we use graphs to compute the shortest path between two locations. The optimum plan with what places to visit will be suggested (taking into account shortest distance between

places) , however if the user wants to change their preferences, the most efficient path will be suggested to the user.
Time complexity: Time complexity is **O(V+E)** where V is the number of vertices in the graph and E is number of edges in the graph. So here the time complexity is O(110)
By using graphs we can easily find the shortest path, neighbors of the nodes, and many more.Graphs are used to implement algorithms like DFS and BFS.
It helps in organizing data.Because of its non-linear structure, helps in understanding complex problems and their visualization.

Hash Table:

The list of places to visit will be stored in a hash table. The hash table keys will hence be stored in a priority queue. This is done as each place will have its corresponding key value, and hence easier to objectively choose the more important place to visit.
The main advantage of hash tables over other data structures is speed . The access time of an element is on average O(1), therefore lookup could be performed very fast. Hash tables are particularly efficient when the maximum number of entries can be predicted in advance.

Priority Queue:

Priority queue is used to store all the places in the hash table dataset and along with an autogenerated priority. This suggests the most important places to visit. Additionally we have also incorporated additional priority values depending on the user requirements , in this way the priority queue keeps changing and the optimized path will be displayed.
The major advantage of using a priority queue is that we will be able to quickly access the highest priority item with a time complexity of just O(1).
Time complexity: O(log n) - for inserting data

Comparison of data structures used:

Graphs Comparison:

- The adjacency matrix is most helpful in cases where the graph doesn't contain a large number of nodes.
- Adjacency lists, on the other hand, are a great option when we need to continuously access all the neighbors of some node *u*.
- we use edges lists when we have an enormous amount of nodes that can't be stored inside the memory, with only a few edges

Usage of backtracking algorithm for TSP (NP-hard Problem)

- DP Approach Time complexity - O(n!)
    - Space complexity: Exponential
    - Not suitable for large no. of vertices
- Same time complexity for branch and bound
- Backtracking approach: O(n!)
    - Space complexity: O(N)

Hash Table :
The different types of hashing techniques are as follows:

- Chaining, Open Addressing: Linear Probing, Quadratic Probing, Double Hashing
- Separate chaining can lead to empty spaces in the table, the list in the positions can be very long
- Linear Probing causes clustering
- Double hashing can cause thrashing (system spend major time on solving the faults)
- QB is easier to implement
- It solves the problem of clustering

Hash Table vs other data structures

- Binary Trees
    - medium complexity to implement (assuming you can't get them from a library)
    - inserts are O(logN)
    - lookups are O(logN)
- Linked lists (unsorted)
    - low complexity to implement
    - inserts are O(1)
    - lookups are O(N)
- Hash tables
    - high complexity to implement yet since this application is meant to have a large dataset
    - inserts are O(1) on average
    - lookups are O(1) on average

4. **Project source code:**

```
# Function to print an array
def printArray(arr, n):

    #printing the hashtable(Array)
    for i in range(n):
        print(arr[i], end = " ")

# quadratic probing
def hashing(table, tsize, arr, N):

    for i in range(N):

        hv = len(arr[i]) % tsize

        if (table[hv] == -1):
            table[hv] = arr[i]

        else:

            for j in range(tsize):

                t = (hv + j * j) % tsize

                if (table[t] == -1):

                    table[t] = arr[i]
                    break

    printArray(table, L)


# Driver code

print("\n\n--------------------TRIP PLANNER--------------------")
```

```
print("\nTrip Planner based in Chennai. Based on your preference we will tell you the top 10 places to visit in Chennai.\n")
print("Happy Planning! :) \n")

print("Hashtable Contents: \n")

arr = [
"Marina-Beach","VGP-Snow-Kingdom","Mahabalipuram","EA-Mall","Santhome-Church",
    "Pondy_Bazar","Kabali_Temple","Senmozhi_Poonga",
"Vandaloor_Zoo","Muttukadu-Boat-House"]

N = 10

# Size of the hash table
L = 2*N

hash_table = [0] * L

# Initializing the hash table
for i in range(L):
    hash_table[i] = -1

hashing(hash_table, L, arr, N)

index_arr=[None]*L
j=0
for i in range(0,L):
    if (hash_table[i]!=-1):
        index_arr[j]=i
        j+=1



print("\n\nHash-Keys of the Hashtable: ")
for i in range(0,len(index_arr)):
    if (index_arr[i]!=None):
        print(index_arr[i], end = " ")
#print(index_arr)"'
```

```
class PriorityQueueNode:

 def __init__(self, value, pr):

    self.data = value
    self.priority = pr
    self.next = None

# Implementation of Priority Queue
class PriorityQueue:

    def __init__(self):

        self.front = None

    def isEmpty(self):

        return True if self.front == None else False

    def push(self, value, priority):

        if self.isEmpty() == True:

            self.front = PriorityQueueNode(value,
                                           priority)

            return 1

        else:
            if self.front.priority > priority:

                newNode = PriorityQueueNode(value,
                                            priority)

                newNode.next = self.front
                self.front = newNode
                return 1

            else:
                temp = self.front
```

```
            while temp.next:

                if priority <= temp.next.priority:
                    break

                temp = temp.next

            newNode = PriorityQueueNode(value,
                                        priority)
            newNode.next = temp.next
            temp.next = newNode

            return 1

    def pop(self):


        if self.isEmpty() == True:
            return

        else:


            self.front = self.front.next
            return 1


    def peek(self):

        if self.isEmpty() == True:
            return
        else:
            return self.front.data

    def traverse(self):


        if self.isEmpty() == True:
            return "Queue is Empty!"
```

```
        else:
            temp = self.front
            while temp:
                print(temp.data, end = "\n")
                temp = temp.next

# Driver code
if __name__ == "__main__":

    print("\n\nList of factors which will affect your trip:\n1.Distance\n2.Budget\n3.Familiy
friendly\n\n")
    print("Do you want to choose any factor? \n1.Yes\n2.No \n")
    pref=int(input("Enter option: "))
    if (pref==1):
        choice=int(input("Enter based on which factor you want to prioritize your trip: "))
        print("\nTop 10 places to visit in Chennai based on chosen priority! ")

        if (choice ==1):

            pq = PriorityQueue()

            pq.push(arr[2], 1,)
            pq.push(arr[8], 2)
            pq.push(arr[4], 3)
            pq.push(arr[0], 4)
            pq.push(arr[9], 5)
            pq.push(arr[1], 6)
            pq.push(arr[7], 7)
            pq.push(arr[6], 8)
            pq.push(arr[3], 9)
            pq.push(arr[5], 10)

            pq.traverse()

            pq.pop()
        elif (choice ==2):

            pq = PriorityQueue()

            pq.push(arr[1], 1,)
```

```
            pq.push(arr[5], 2)
            pq.push(arr[2], 3)
            pq.push(arr[4], 4)
            pq.push(arr[6], 5)
            pq.push(arr[1], 6)
            pq.push(arr[3], 7)
            pq.push(arr[7], 8)
            pq.push(arr[9], 9)
            pq.push(arr[8], 10)

            pq.traverse()

            pq.pop()

        elif (choice==3):

            pq = PriorityQueue()

            pq.push(arr[9], 1,)
            pq.push(arr[0], 2)
            pq.push(arr[2], 3)
            pq.push(arr[3], 4)
            pq.push(arr[8], 5)
            pq.push(arr[1], 6)
            pq.push(arr[5], 7)
            pq.push(arr[7], 8)
            pq.push(arr[4], 9)
            pq.push(arr[6], 10)

            pq.traverse()

            pq.pop()
        else:
            print("\nEnter valid option!\n")
    elif (pref==2):
        print("\nThis is the suggested plan taking all the factors into consideration!\n")

        pq = PriorityQueue()

        pq.push(arr[9], 1,)
```

```
            pq.push(arr[0], 2)
            pq.push(arr[2], 3)
            pq.push(arr[3], 4)
            pq.push(arr[8], 5)
            pq.push(arr[1], 6)
            pq.push(arr[5], 7)
            pq.push(arr[7], 8)
            pq.push(arr[4], 9)
            pq.push(arr[6], 10)

            pq.traverse()

            pq.pop()
        else:
            print("\nEnter valid option!\n")

#TSP
from sys import maxsize
from itertools import permutations
V = N

def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)

    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)

    for i in next_permutation:
        #print(next_permutation)

        # store current Path weight(cost)
        current_pathweight = 0

        # compute current path weight
```

```
    k = s
    for j in i:
        current_pathweight += graph[k][j]
        #print(i)
        k = j
    current_pathweight += graph[k][s]

    # update minimum
    min_path = min(min_path, current_pathweight)
    perm_arr.append(i)
    min_vals.append(min_path)

#print(min_vals)
#print(perm_arr)
x=0
for a in range(0,len(min_vals)):
    if min_vals[a]==min_path:
        x=a
        break
#printing the path
print("\n\n",s,end=" ")
for y in range(0,V-1):
    print(" -> ",perm_arr[x][y],end=" ")
print(" ->",s,end=" ")

'"#storing the path in an array
path_arr=[0]*(V+1)
path_arr[0]=0
for i in range(0,V-1):
    path_arr[i+1]=perm_arr[x][i]
    #print("\n\n",perm_arr[x][i])
print(path_arr)

#accessing the hash table contents
final_arr=[]
for p in range(0,len(path_arr)):
    q=path_arr[p]
    index_arr[q]!=None
    r=index_arr[p]
    final_arr.append(hash_table[r])"'
```

```
    return min_path


# matrix representation of graph
if (pref==1):
    if (choice==1):
        graph = [[6, 3, 34, 4, 3, 66, 4, 34, 6, 10], [20, 34, 34, 34, 2, 9, 7, 6, 2, 5],
            [38, 11, 3, 4, 89, 34, 34, 3, 34, 10], [7, 34, 3, 66, 7, 2, 89, 7, 55, 34],
            [9, 2, 38, 7, 24, 9, 3, 34, 11, 2], [11, 4, 20, 9, 4, 95, 34, 34, 10, 6],
            [9, 38, 66, 3, 4, 20, 3, 11, 5, 55], [3, 7, 55, 2, 9, 6, 2, 3, 9, 95],
            [10, 20, 4, 66, 2, 6, 34, 7, 89, 55], [20, 34, 2, 66, 89, 9, 38, 55, 3, 95]]


    elif(choice==2):
        graph = [[3, 7, 55, 2, 9, 6, 2, 3, 9, 95],[20, 34, 34, 34, 2, 9, 7, 6, 2, 5],
            [38, 11, 3, 4, 89, 34, 34, 3, 34, 10], [20, 34, 2, 66, 89, 9, 38, 55, 3, 95],
            [9, 38, 66, 3, 4, 20, 3, 11, 5, 55],[9, 2, 38, 7, 24, 9, 3, 34, 11, 2],
            [11, 4, 20, 9, 4, 95, 34, 34, 10, 6], [6, 3, 34, 4, 3, 66, 4, 34, 6, 10],
            [10, 20, 4, 66, 2, 6, 34, 7, 89, 55],[7, 34, 3, 66, 7, 2, 89, 7, 55, 34]]

    elif(choice==3):
        graph = [[11, 4, 20, 9, 4, 95, 34, 34, 10, 6],[3, 7, 55, 2, 9, 6, 2, 3, 9, 95],
                [20, 34, 34, 34, 2, 9, 7, 6, 2, 5], [38, 11, 3, 4, 89, 34, 34, 3, 34, 10],
                [20, 34, 2, 66, 89, 9, 38, 55, 3, 95],[9, 38, 66, 3, 4, 20, 3, 11, 5, 55],
                [9, 2, 38, 7, 24, 9, 3, 34, 11, 2], [6, 3, 34, 4, 3, 66, 4, 34, 6, 10],
                [7, 34, 3, 66, 7, 2, 89, 7, 55, 34],[10, 20, 4, 66, 2, 6, 34, 7, 89, 55]]

    else:
        print("Enter valid option!")

else:
    graph = [[9, 2, 38, 7, 24, 9, 3, 34, 11, 2],[6, 3, 34, 4, 3, 66, 4, 34, 6, 10],
            [7, 34, 3, 66, 7, 2, 89, 7, 55, 34], [38, 11, 3, 4, 89, 34, 34, 3, 34, 10],
             [11, 4, 20, 9, 4, 95, 34, 34, 10, 6], [20, 34, 34, 34, 2, 9, 7, 6, 2, 5],
             [3, 7, 55, 2, 9, 6, 2, 3, 9, 95], [9, 38, 66, 3, 4, 20, 3, 11, 5, 55],
            [20, 34, 2, 66, 89, 9, 38, 55, 3, 95],[10, 20, 4, 66, 2, 6, 34, 7, 89, 55] ]

z=index_arr[0]
s = 0
```

```
min_vals=[]
perm_arr=[]

print("\n\nThe final order of places to visit such that it has the minimum cost is: ")
travellingSalesmanProblem(graph, s)

print("\n\nThe numbers here represent the following places: \n")

for x in range(0,len(index_arr)):
        if (index_arr[x]!=None):
            y=index_arr[x]
            print(x,hash_table[y])

print("\n\nHave a safe journey!\n\n")
```

5. Output:

```
gunanicaaarun@Gunanicaas-MacBook-Air final % python3 final.py


--------------------TRIP PLANNER-------------------

Trip Planner based in Chennai. Based on your preference we will tell you the top 10 places to visit in Chennai.

Happy Planning! :)

Hashtable Contents:

Muttukadu-Boat-House -1 -1 -1 -1 -1 -1 EA-Mall -1 -1 -1 Pondy_Bazar Marina-Beach Mahabalipuram Kabali_Temple Santhome-Church VGP-Snow-Kingdom Vandaloor_Zoo -1 Senmozhi_Poon
ga

Hash-Keys of the Hashtable:
0 7 11 12 13 14 15 16 17 19
```

```
List of factors which will affect your trip:
1.Distance
2.Budget
3.Familiy friendly


Do you want to choose any factor?
1.Yes
2.No

Enter option: 2

This is the suggested plan taking all the factors into consideration!

Muttukadu-Boat-House
Marina-Beach
Mahabalipuram
EA-Mall
Vandaloor_Zoo
VGP-Snow-Kingdom
Pondy_Bazar
Senmozhi_Poonga
Santhome-Church
Kabali_Temple
```

```
The final order of places to visit such that it has the minimum cost is:


 0  ->  6  ->  3  ->  7  ->  8  ->  2  ->  5  ->  9  ->  4  ->  1  -> 0

The numbers here represent the following places:

0 Muttukadu-Boat-House
1 EA-Mall
2 Pondy_Bazar
3 Marina-Beach
4 Mahabalipuram
5 Kabali_Temple
6 Santhome-Church
7 VGP-Snow-Kingdom
8 Vandaloor_Zoo
9 Senmozhi_Poonga


Have a safe journey!
```

```
List of factors which will affect your trip:
1.Distance
2.Budget
3.Familiy friendly


Do you want to choose any factor?
1.Yes
2.No

Enter option: 1
Enter based on which factor you want to prioritize your trip: 1

Top 10 places to visit in Chennai based on chosen priority!
Mahabalipuram
Vandaloor_Zoo
Santhome-Church
Marina-Beach
Muttukadu-Boat-House
VGP-Snow-Kingdom
Senmozhi_Poonga
Kabali_Temple
EA-Mall
Pondy_Bazar


The final order of places to visit such that it has the minimum cost is:


 0  ->  6  ->  3  ->  5  ->  1  ->  8  ->  4  ->  9  ->  2  ->  7  -> 0

The numbers here represent the following places:

0 Muttukadu-Boat-House
1 EA-Mall
2 Pondy_Bazar
3 Marina-Beach
4 Mahabalipuram
5 Kabali_Temple
6 Santhome-Church
7 VGP-Snow-Kingdom
8 Vandaloor_Zoo
9 Senmozhi_Poonga
```

```
Do you want to choose any factor?
1.Yes
2.No

Enter option: 1
Enter based on which factor you want to prioritize your trip: 2

Top 10 places to visit in Chennai based on chosen priority!
VGP-Snow-Kingdom
Pondy_Bazar
Mahabalipuram
Santhome-Church
Kabali_Temple
VGP-Snow-Kingdom
EA-Mall
Senmozhi_Poonga
Muttukadu-Boat-House
Vandaloor_Zoo


The final order of places to visit such that it has the minimum cost is:


 0  ->  6  ->  9  ->  5  ->  1  ->  8  ->  4  ->  3  ->  2  ->  7  -> 0

The numbers here represent the following places:

0 Muttukadu-Boat-House
1 EA-Mall
2 Pondy_Bazar
3 Marina-Beach
4 Mahabalipuram
5 Kabali_Temple
6 Santhome-Church
7 VGP-Snow-Kingdom
8 Vandaloor_Zoo
9 Senmozhi_Poonga


Have a safe journey!
```

```
Do you want to choose any factor?
1.Yes
2.No

Enter option: 1
Enter based on which factor you want to prioritize your trip: 3

Top 10 places to visit in Chennai based on chosen priority!
Muttukadu-Boat-House
Marina-Beach
Mahabalipuram
EA-Mall
Vandaloor_Zoo
VGP-Snow-Kingdom
Pondy_Bazar
Senmozhi_Poonga
Santhome-Church
Kabali_Temple


The final order of places to visit such that it has the minimum cost is:


 0  ->  1  ->  6  ->  9  ->  4  ->  2  ->  8  ->  5  ->  3  ->  7  -> 0

The numbers here represent the following places:

0 Muttukadu-Boat-House
1 EA-Mall
2 Pondy_Bazar
3 Marina-Beach
4 Mahabalipuram
5 Kabali_Temple
6 Santhome-Church
7 VGP-Snow-Kingdom
8 Vandaloor_Zoo
9 Senmozhi_Poonga


Have a safe journey!
```

```
Do you want to choose any factor?
1.Yes
2.No

Enter option: 4

Enter valid option!
```

6. **Inferences:**

Usage of graphs to find the optimized path by using traveling salesman problem.
Implementation of priority queue to find the places having highest preferences to visit, according to the factors given by the user.
Hash tables are used to store the data set(list of places) where the time required to access the data is very less.

7. **Gap Analysis :**

Our project currently is limited to a small sample space, and to only a list of places in one city.

The data taken from the user is very minimal in the current program, which makes it less interactive and less personalized.

Possible Improvements:

- Assigning weights based on a more detailed manner (graphs)
- Increasing the dataset to having more cities and places (hashtable)
- Incorporating more factors to decide the final plan and implementing an appropriate way to assign weights based on the data collected