PROJECT REPORT

Data Structure and Algorithms
(UCS 406)

TOPIC

“Finding Shortest Path and Nearest Food Point in THAPAR INSTITUTE”

Submitted To: Dr. RAJIV KUMAR

Submitted By:

CHANDNI MITTAL	101611015
DAMANPREET SINGH	101611017
TANISH CHARAYA	101611056
ANMOL BAJWA	101783049

Submitted to the

Computer Science & Engineering Department

Thapar Institute of Engineering and Technology, Patiala

May 2018

TABLE OF CONTENT

1. <u>Problem Formulation</u>

Making a Map for Thapar University.

We have made a map for Thapar university itself in which we have taken into the consideration the problems faced by the students as sometimes they are unable to find the particular area which they are interested in visiting. This map will help them to find the block where they want to go. We have used Dijkstra’s algorithm to find the shortest path so that they can reach their place of interest in shortest time possible and we have used the concept of searching the food points.

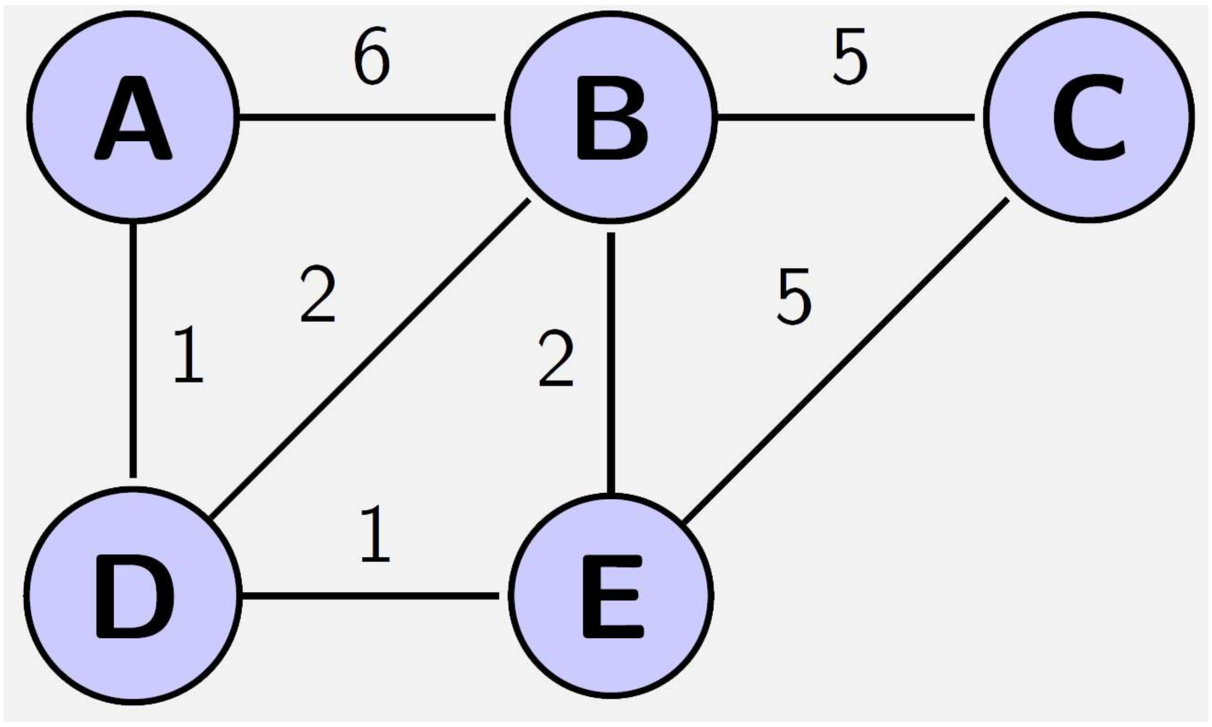


Figure 1.1

2. <u>Analysis of problem w.r.t. Data structure and Algorithmic technique used.</u>

2.1 Create a graph: With the help of adjacency matrix we will create a graph joining various places in Thapar. The graph will be undirected so if mat[i][j] equals to (a) then mat[j][i] will also be equal to (a). Where (a) is the distance between place (i) and (j).

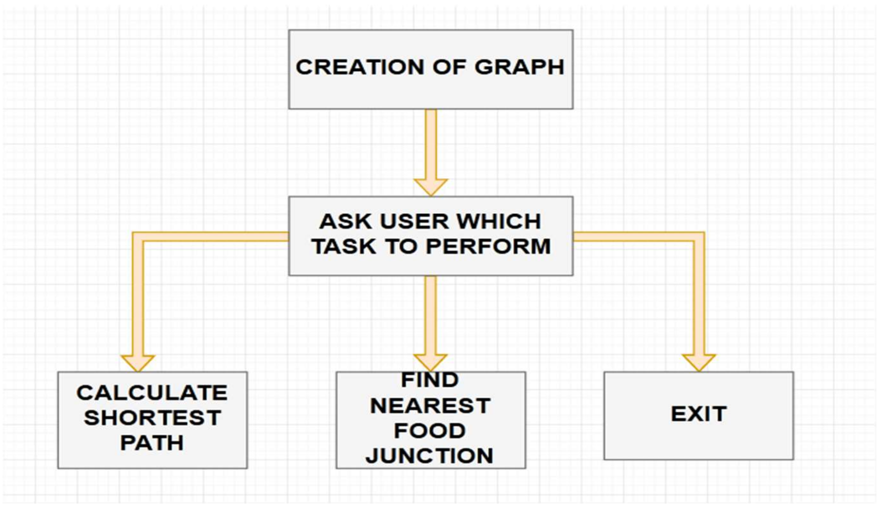
2.2 User input: Here user is provided with three options of either calculating the shortest path between two places that is source and destination, or to find the nearest food point from a certain location, or exit. This is implemented using switch having three cases.

2.3 On the basis of selection made by user three different outputs are possible:

2.3.1 Option 1: If user want to know the shortest path between two places then ask the index number corresponding to source and destination. Then this will return the shortest path between source and destination. And more over it will also give information about all the places which are along the way. This will provide a more clearer idea about the path followed.

2.3.2 Option 2: If user want to know about the nearest food point from a certain location then ask the index number corresponding to the location from which a user want to find the distance. Then here source will be the current location and the destination will be the nearest food point which is calculated using spanning tree.

2.3.3 Option 3: If user want to exit the program the this can be done by selecting third option and program will be successfully terminated.



3. <u>Concepts used:</u>

3.1 Spanning tree: A **spanning tree** T of an undirected graph *G* is a subgraph that is a tree which includes all of the vertices of *G*, with minimum possible number of edges. In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree. If all of the edges of *G* are also edges of a spanning tree *T* of *G*, then *G* is a tree and is identical to *T* (that is, a tree has a unique spanning tree and it is itself).

A tree is a connected undirected graph with no cycles. It is a spanning tree of a graph *G* if it spans *G* (that is, it includes every vertex of *G*) and is a subgraph of *G* (every edge in the tree belongs to *G*). A spanning tree of a connected graph *G* can also be defined as a maximal set of edges of *G* that contains no cycle, or as a minimal set of edges that connect all vertices.

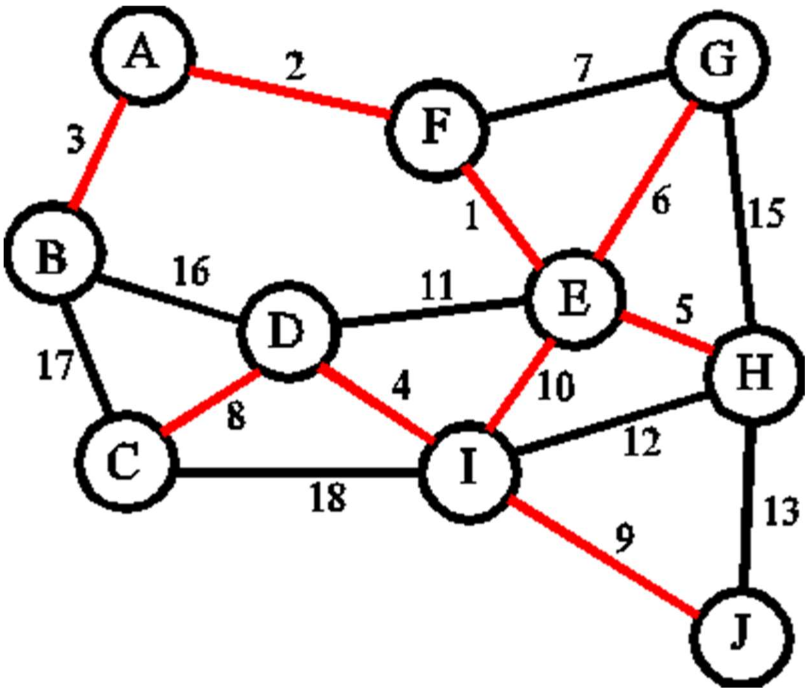


Figure 3.1 A sample spanning tree

3.2 Dijkstra's algorithm: It is an algorithm for finding the shortest path between nodes in a graph. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Suppose you would like to find the *shortest path* between two intersections on a city map: a *starting point* and a *destination*. Dijkstra's algorithm initially marks the distance (from the starting point) to every other intersection on the map with *infinity*. This is done not to imply there is an infinite distance, but to note that those intersections have not yet been visited; some variants of this method simply leave the intersections' distances *unlabeled*. Now, at each iteration, select the *current intersection*. For the first iteration, the current intersection will be the starting point, and the distance to it (the intersection's label) will be *zero*. For subsequent iterations (after the first), the current intersection will be a *closest unvisited intersection* to the starting point (this will be easy to find).

From the current intersection, *update* the distance to every unvisited intersection that is directly connected to it. This is done by determining the *sum* of the distance between an unvisited intersection and the value of the current intersection, and re-labeling the unvisited intersection with this value (the sum), if it is less than its current value. In effect, the intersection is relabeled if the path to it through the current intersectio is shorter than the previously known paths. To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabeled intersection if you label/relabel it, and erase all others pointing to it. After you have updated the distances to each neighboring intersection, mark the current intersection as *visited*, and select an unvisited intersection with minimal distance (from the starting point) – or the lowest label—as the current intersection. Nodes marked as visited are labeled with the shortest path from the starting point to it and will not be revisited or returned to.

Continue this process of updating the neighboring intersections with the shortest distances, then marking the current intersection as visited and moving onto a closest unvisited intersection until you have marked the destination as visited. Once you have marked the destination as visited (as is the case with any visited intersection) you have determined the shortest path to it, from the starting point, and can *trace your way back, following the arrows in reverse*; in the algorithm's implementations, this is usually done (after the algorithm has reached the destination node) by following the nodes' parents from the destination node up to the starting node; that's why we also keep track of each node's parent.

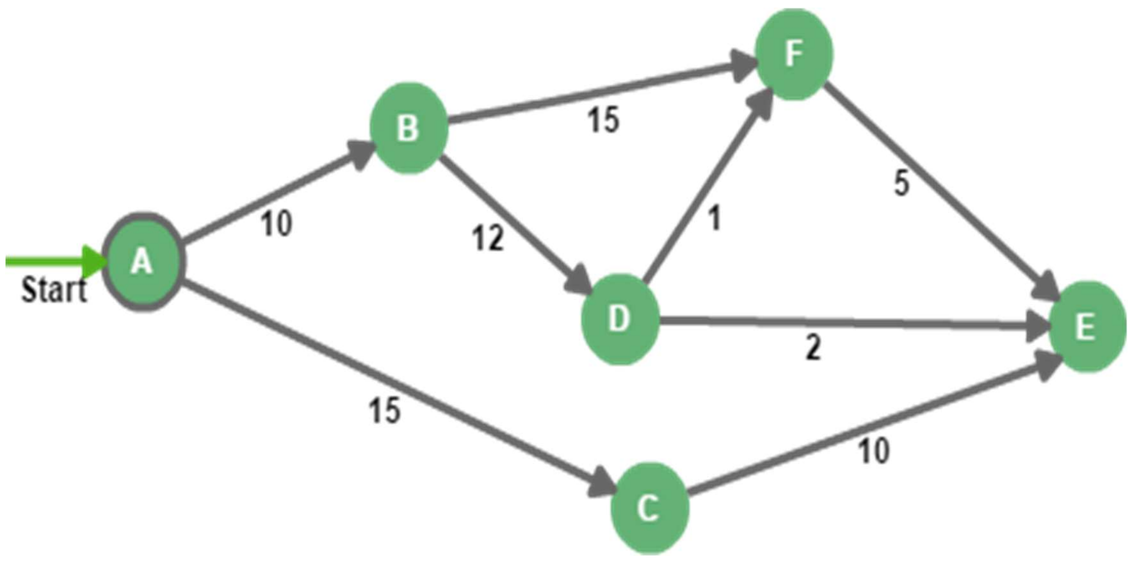


Figure 3.2 A directional weighted graph

3.3 Backtracking: Backtracking helps in solving an overall issue by finding a solution to the first sub-problem and then recursively attempting to resolve other sub-problems based on the solution of the first issue. If the current issue cannot be resolved, the step is backtracked and the next possible solution is applied to previous steps, and then proceeds further. In fact, one of the key things in backtracking is recursion. It is also considered as a method of exhaustive search using divide and conquer. A backtracking algorithm ends when there are no more solutions to the first sub-problem.

Backtracking is an algorithm which can help achieve implementation of nondeterminism. It takes a depth-first search of a given issue space. It is used mostly in logic programming languages like Prolog. Wherever backtracking can be applied, it is faster than the brute force technique, as it eliminates a large number of candidates with a single test.

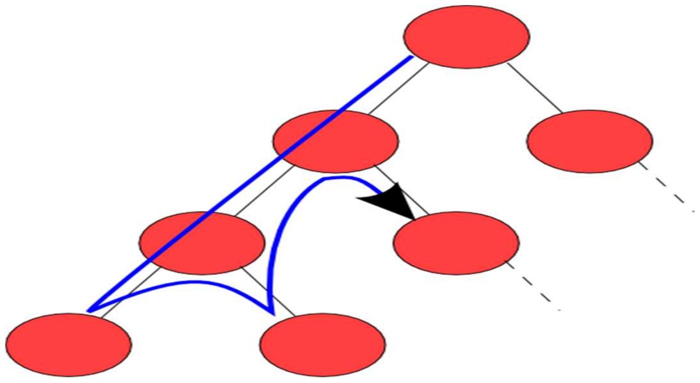


Fig 3.3 A figure depicting the concept of backtracking

4. Methodology used form of pseudocode/flowchart/algorithm

4.1 DJIKSTRA(int s,int sptset())

```
Function to calculate the shortest distance between any two nodes
1. Initialize spanning tree array sptset equal to -1 for all values of N.
2. Initialize distance matrix array with max possible value of integer datatype and set dist[src]=0
3.repeat step 4 to 9 till number of visited nodes(n) are less than N
4. repeat this from k ranging from 0 to N
Call check() function to see if the particular node is visited or not
if check() returns true i.e. the node is node visited
then
        find an 'i' such that the dist[i] is minimum

5.set sptset[n] equal to i

6.increment n

7.Repeat this from j varying form 0 to N
        update the distance matrix using
        min_distance = dist[i] + mat[i][j]

8.if check() is true (i.e. node is unvisited) and distance is minimum
        then set dist[i]=min_distance

 9.now check for the next node which is unvisited
```

4.2 BACKTRACK(int r,int z)

Function to calculate the path followed in order to achieve minimum weight

```
1.set flag equal to 1
2. repeat step 3 to-- while j varies from 0 to N
3.if path exists between r and j
    then
        check if dist[r]-mat[r][j] equals to dist[j]
        if true
            then add j to the path and
             call backtrack again with r now being equal to j
```

4.3 CHECK (int i , int sptset())

```
1. Initialise f = 0 i.e the node is visited
2. repeat while j varies from 0 to N
3.check if i belongs to spanningtree array
if yes then
 set f=false and break free from the loop
4. return f
```

4.4 SHORTPATH(int src , int dest)

To calculate the shortest path we need to call dijkastra function which will tell us the minimum weight between source and destination

To find the intermediate nodes travelled to reach destination we will need to call backtrack function

```
1.Call dijkstra()
```

2.Call Backtrack()

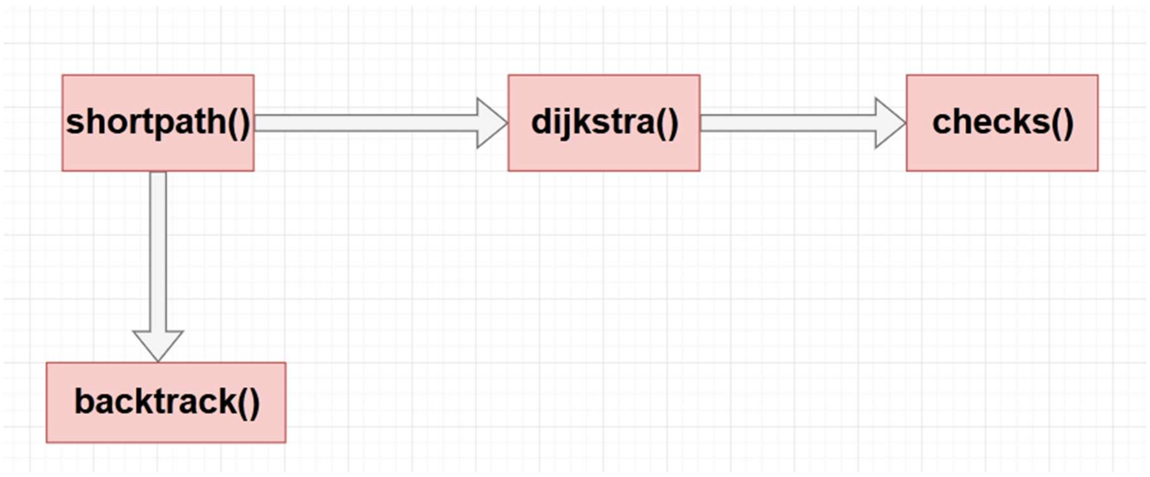


Fig4.1 Flowchart depicting the function calls required in order to find the shortest path between source and destination

4.5 FOOD(int loc)

Finding the nearest food point is same as finding the shortest path between current location and available food points.

1.Call Dijkstra() until the new visited node obtained is a food point

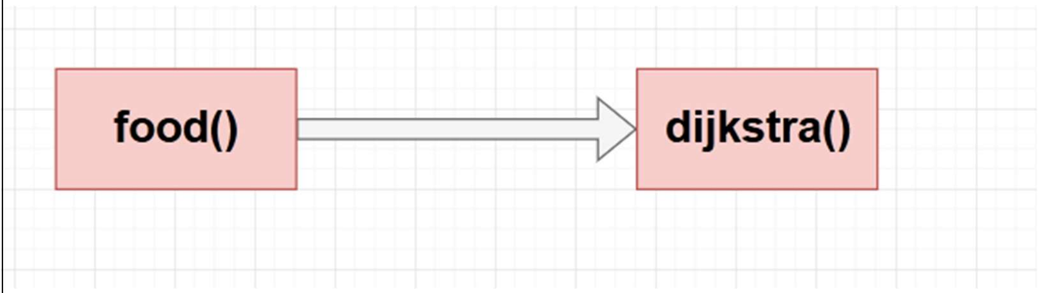
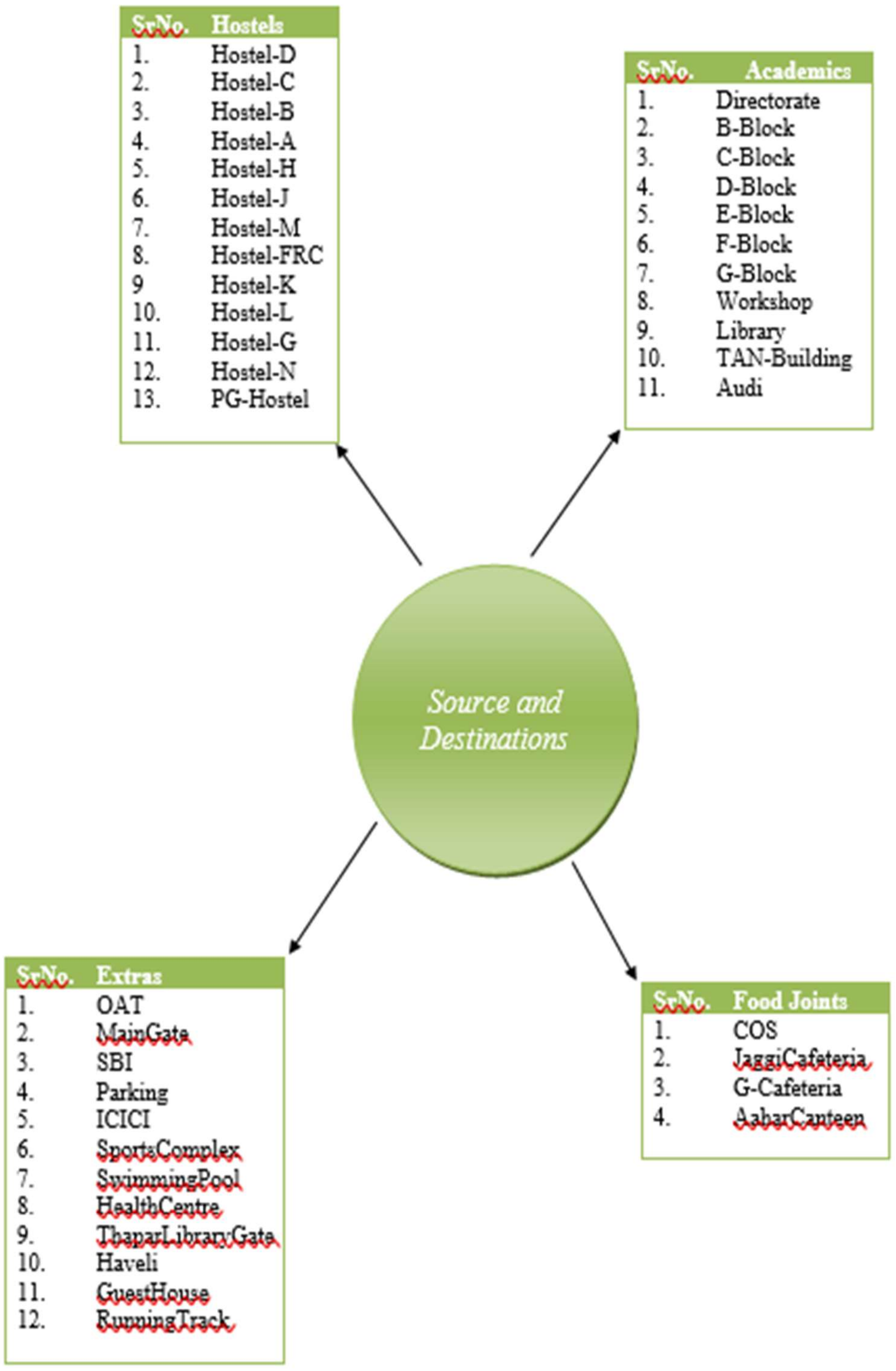


Fig4.2 Flowchart depicting the function calls required in order to find the nearest food joint from desired location



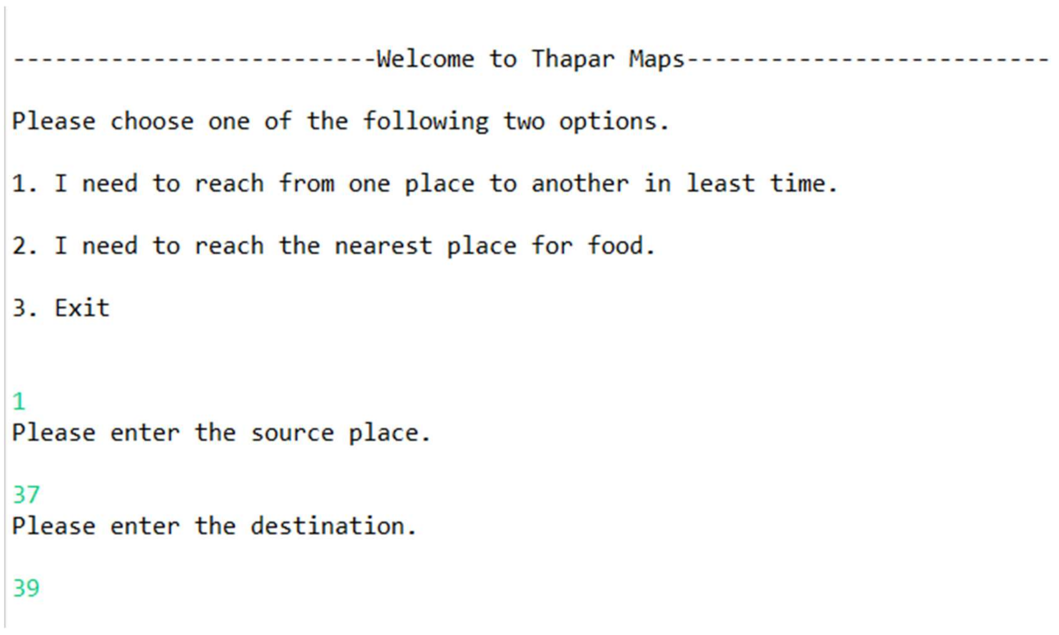
SrNo.	Hostels
1.	Hostel-D
2.	Hostel-C
3.	Hostel-B
4.	Hostel-A
5.	Hostel-H
6.	Hostel-J
7.	Hostel-M
8.	Hostel-FRC
9	Hostel-K
10.	Hostel-L
11.	Hostel-G
12.	Hostel-N
13.	PG-Hostel

SrNo.	Academics
1.	Directorate
2.	B-Block
3.	C-Block
4.	D-Block
5.	E-Block
6.	F-Block
7.	G-Block
8.	Workshop
9.	Library
10.	TAN-Building
11.	Audi

SrNo.	Extras
1.	OAT
2.	MainGate
3.	SBI
4.	Parking
5.	ICICI
6.	SportsComplex
7.	SwimmingPool
8.	HealthCentre
9.	ThaparLibraryGate
10.	Haveli
11.	GuestHouse
12.	RunningTrack

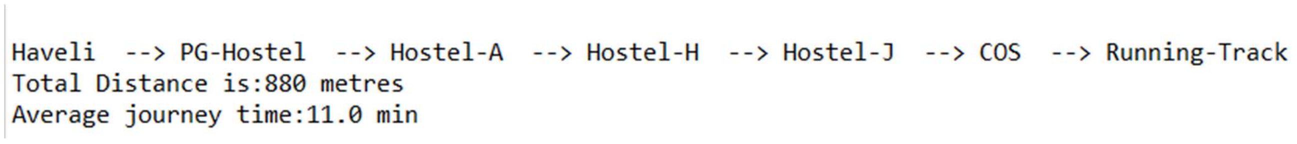
SrNo.	Food Joints
1.	COS
2.	JaggiCafeteria
3.	G-Cafeteria
4.	AaharCanteen

5. CONSOLE OUTPUT:



```
--------------------------Welcome to Thapar Maps--------------------------

Please choose one of the following two options.

1. I need to reach from one place to another in least time.

2. I need to reach the nearest place for food.

3. Exit


1
Please enter the source place.

37
Please enter the destination.

39
```

Fig 5.1 Take input from user



```
Haveli  --> PG-Hostel  --> Hostel-A  --> Hostel-H  --> Hostel-J  --> COS  --> Running-Track
Total Distance is:880 metres
Average journey time:11.0 min
```

Fig5.2 Output will look like this

RESULT/CONCLUSION:

For a large institution like Thapar university it becomes very tough to find a areas or blocks where we want to go. The project has successfully solved the purpose by making Thapar maps. The proposed analysis solves the problem by telling the user about information of all the blocks in their institution sharing similar interest. The complexity can be altered by using linked lists, matrix and arrays. Time complexity of the above code is O(V^2) as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed O(V+E) times. The inner loop has decreaseKey() operation which takes O(LogV) time. So overall time complexity is O(E+V)*O(LogV) which is O((E+V)*LogV) = O(ELogV) This project can be further linked to database management to store group information and corresponding information of all people. Also a portal can be made using Graphical User Interface.

7. Bibliography:

1. https://www.geeksforgeeks.org/graph-and-its-representations/

2. http://www.tutorialdost.com/Cpp-Programming-Tutorial/25-Cpp-Structure.aspx

3. https://www.jdoodle.com/online-compiler-c++

4. https://www.sanfoundry.com/cpp-program-   implement-adjacency-matrix/

5. https://www.techopedia.com/definition/17837/backtracking

6.https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm