JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY

PROJECT SYNOPSIS

ALGORITHMS AND PROBLEM SOLVING PROJECT

TOPIC:TRIP TREKKER

BATCH:B2

SUBMITTED TO: MS. SHERRY GARG

SUBMITTED BY:

NAME	ENROLLMENT NO
DEEPAK	22103053
YASH RAUTELA	22103057
SIDHARTH BHARDWAJ	22103032

ABSTRACT

Map applications, as the name suggests, are implementations of maps in a virtual form with an ever increasing number of features that go beyond just showing directions. Some most commonly used navigation apps include Google Maps, Waze, Apple Maps etc. Map applications, now more than ever, have become nothing short of a necessity for most people. In order to plan trips or even daily commutes, it becomes important to know what route would suit one the most so they can use their time and money efficiently. There's also the modern problem of traffic which can cause undesirable delays if the route is not chosen properly.

The purpose of this project is to design a one-day itinerary for a traveler who wants to explore a specific location. The itinerary will include a variety of activities and destinations, based on the traveler's interests and preferences. The itinerary will also be designed to optimize the traveler's time and budget, while providing a memorable and enjoyable experience.

GOALS OF THE PROJECT :

- **<u>Planning a Full day tour:</u>** Planning a Full day visits of the city in a minimum time which can help in saving time and avoid confusion (Gives an efficient path) .User are requested to enter the source destination, Then Traveling Salesman problem / Hamiltonian cycle is applied to give a path (having a full route with same starting point and ending point) with minimum time travel and shows the time travel between the places.

- **<u>Full day Activities Planner:</u>** It provides you with the list of activities that can be performed at the particular location. The aim is that the user can participate in 2 maximum activities therefore we have used activity selection using Greedy approach.The users then choose their desired activities, and we have used quicksort to sort the activities on the basis of the end time, therefore avoiding the clash of time.

- **<u>Shortest path between two cities:</u>** It helps in finding the shortest path, users are requested to enter the district codes of the source and destination locations where they want to trace the path. Then Dijkstra's Algorithm is applied for the entered source and destination giving the shortest route which will help save your time and the hustle to find the ideal route.

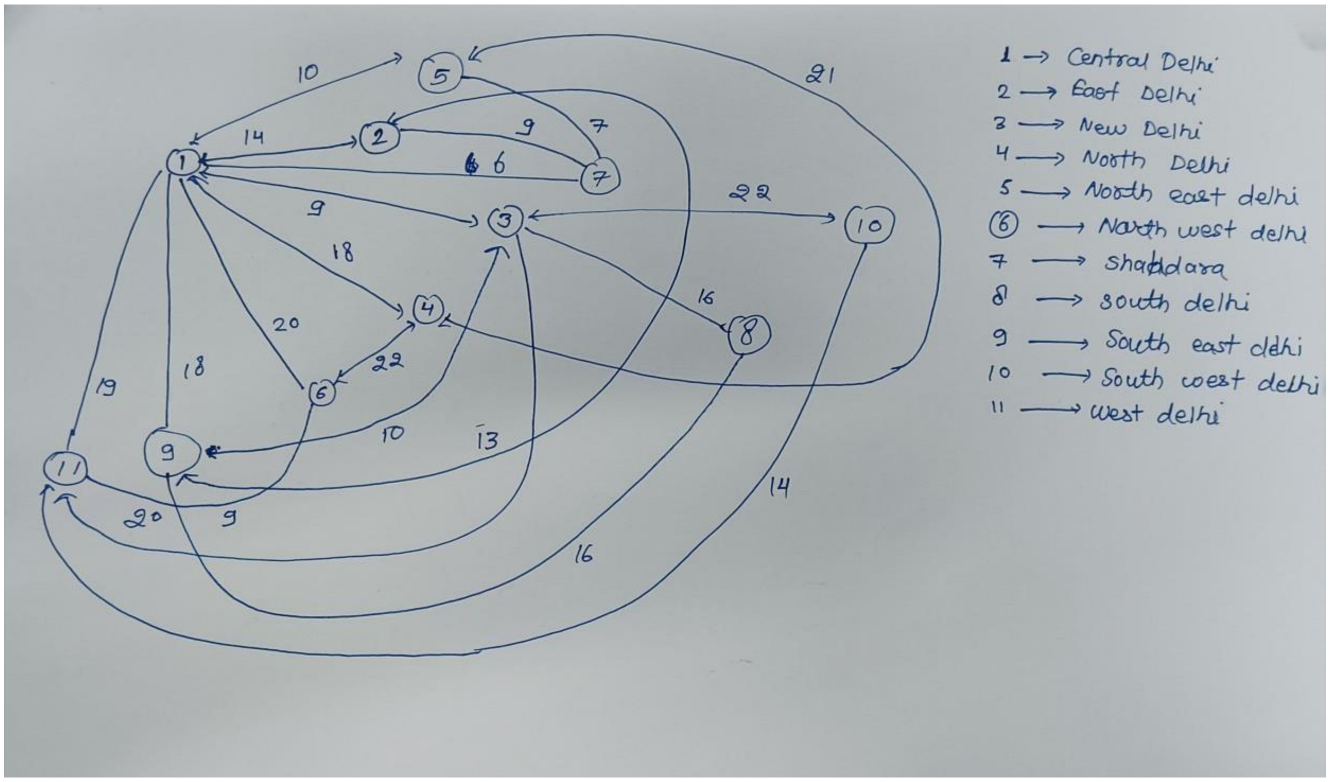
ALGORITHM USED :-

- **Activity Time Scheduling -** Using Greedy Algorithm - Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems.The greedy choice is to always pick the next activity whose finish time is the least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as the minimum finishing time activity
 Time Complexity-**O(n log n)**

- **Dijkstra algorithm -** Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph, which may represent, for example, road networks. 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. Dijkstra's original algorithm does not use a min-priority queue and runs in time Θ(|V|^{2})(where |V| is the number of nodes).

- **Travelling Salesman Problem Branch and Bound -** The Travelling salesman problem (TSP) is a classic optimization problem in computer science and operations research. It involves finding the shortest possible route that can be taken to visit a set of cities exactly once and return to the starting city with the minimum cost. The TSP is an NP-hard problem, which means that finding the exact optimal solution is computationally infeasible for large problem sizes. One popular approach for solving the TSP is the branch and bound algorithm. The basic idea behind this algorithm is to recursively divide the problem into smaller subproblems, and solve each subproblem separately. The algorithm keeps track of the current best solution found so far, and uses this to guide the search for better solutions.

- **Travelling Salesman Problem / Hamiltonian Cycle (finding minimum cost on basis of time) -** This states that, in a cycle, the same point or vertex should never be revisited. Given several points (cities) to be visited, the objective of the problem is to find the shortest possible route (called Hamiltonian Path) that visits each point exactly once and returns back to the starting point. This algorithm was then used at the core of the web-based tool (a practical use case) developed for release in public domain which helps users find an optimal round-trip route (i.e. Hamiltonian Path) among the points marked on the map.
Time complexity **-O(n^2∗2n)**



GRAPH OF THE PROJECT

```
#include<iostream>
#include<stdio.h>
#include<limits.h>
#include<cstdlib>
#include<ctime>
#include<conio.h>
#include<vector>
#include<string.h>
using namespace std;
const int N = 11;

// final_path[] stores the final solution ie, the
// path of the salesman.
int final_path[N+1];

// visited[] keeps track of the already visited nodes
// in a particular path
bool visited[N];

// Stores the final minimum weight of shortest tour.
int final_res = INT_MAX;

// Function to copy temporary solution to
// the final solution

void Swap (int* A, int a, int b)
{
   int temp=A[a];
   A[a]=A[b];
   A[b]=temp;
}

float Min(float a, float b)
{
   return (a<b) ? a : b;
```

```
}
int** fastTime=new int*[11];
//int fastTime[5][11];
int j=-1;

struct path
{
    int dist;
    int traffic;
    float time;
};

//structure containing info about limited time activities
struct act
{
    string Name;
    int start;
    int End;
};

//to Swap elements of vector of act type
void Swap (vector<act> &a, int n1, int n2)
{
  act temp=a[n1];
  a[n1]=a[n2];
  a[n2]=temp;
}

//partition function for quicksort function on the basis of end time of activities
int partition (vector<act> &a, int l, int r)
{
  int p=r;
  int j=l-1;
  for (int i=l; i<=r; i++)
  {
    if (a[i].End<a[p].End) {j++; Swap (a,i,j);}
  }
  Swap(a,p,j+1);
  return j+1;
}

//function to sort activities based on their end time
void quicksort (vector<act> &a, int l, int r)
{
  if (l>=r) return;
  int p = partition (a,l,r);
  quicksort(a,l,p-1);
  quicksort(a,p+1,r);
}

//Graph with all districts' info and various functions
class Graph
{
  //adjacency matrix of direct roads between districts
  //initialised with distances
  path adj[11][11]={{{10000},   {14},    {9},     {18},    {10},    {20},    {6},     {10000},        {18},
          {10000},        {19}},
          {{14},      {10000},        {10000},        {10000},        {10000},        {10000},        {9},
          {10000},        {13},    {10000},        {10000}},
          {{9},       {10000},        {10000},        {10000},        {10000},        {10000},
          {10000},        {16},    {10},    {22},    {20}},
          {{18},      {10000},        {10000},        {10000},        {21},    {22},    {10000},
          {10000},        {10000},        {10000},        {10000}},
          {{10},      {10000},        {10000},        {21},    {10000},        {10000},        {7},
          {10000},        {10000},        {10000},        {10000}},
          {{20},      {10000},        {10000},        {22},    {10000},        {10000},        {10000},
          {10000},        {10000},        {10000},        {9}},
          {{6},       {9},     {10000},        {10000},        {7},     {10000},        {10000},
          {10000},        {10000},        {10000},        {10000}},
          {{10000}, {10000},        {16},    {10000},        {10000},        {10000},        {10000},
          {10000},        {16},    {10000},        {10000}},
          {{18},      {13},    {10},    {10000},        {10000},        {10000},        {10000},        {16},
          {10000},        {10000},        {10000}},
        {{10000},  {10000},        {22},   {10000},       {10000},        {10000},        {10000},
    {10000},        {10000},        {10000},        {14}},

        {{19},      {10000},        {20},   {10000},       {10000},        {9},     {10000},
    {10000},        {10000},        {14},   {10000}}};


  string names[11]={"Central Delhi", "East Delhi", "New Delhi", "North Delhi", "North East Delhi", "North
West Delhi",

            "Shahdara", "South Delhi", "South East Delhi", "South West Delhi", "West Delhi"};

  int n;

  int speed;

  float minc;

  act activities[10];

  //array to store which activities are available to do in which district

  int availableacts[11][5]={{0,1,2,3,4},{0,5,7,8,9},{1,3,6,7,9},{2,4,5,6,8},{0,1,4,8,9},{0,4,5,6,7},

                  {2,3,4,7,8},{1,2,4,6,9},{0,1,3,5,7},{5,6,7,8,9},{3,4,6,8,9}};


public:


   void copyToFinal(int curr_path[])

{

    for (int i=0; i<N; i++)

        final_path[i] = curr_path[i];

    final_path[N] = curr_path[0];

}


// Function to find the minimum edge cost

// having an end at the vertex i

int firstMin(int i)

{

    int min = INT_MAX;

    for (int k=0; k<N; k++)

        if (adj[i][k].dist<min && i != k)

            min = adj[i][k].dist;

    return min;

}
```

```
// function to find the second minimum edge cost
// having an end at the vertex i
int secondMin(int i)
{
        int first = INT_MAX, second = INT_MAX;
        for (int j=0; j<N; j++)
        {
                if (i == j)
                        continue;

                if (adj[i][j].dist <= first)
                {
                        second = first;
                        first = adj[i][j].dist;
                }
                else if (adj[i][j].dist <= second &&
                                adj[i][j].dist != first)
                        second = adj[i][j].dist;
        }
        return second;
}

// function that takes as arguments:
// curr_bound -> lower bound of the root node
// curr_weight-> stores the weight of the path so far
// level-> current level while moving in the search
//               space tree
// curr_path[] -> where the solution is being stored which
//                       would later be copied to final_path[]
void TSPRec(int curr_bound, int curr_weight,
                        int level, int curr_path[])
{
        // base case is when we have reached level N which
        // means we have covered all the nodes once
        if (level==N)
```

```
{
        // check if there is an edge from last vertex in
        // path back to the first vertex
        if (adj[curr_path[level-1]][curr_path[0]].dist != 0)
        {
                // curr_res has the total weight of the
                // solution we got
                int curr_res = curr_weight +
                                adj[curr_path[level-1]][curr_path[0]].dist;

                // Update final result and final path if
                // current result is better.
                if (curr_res < final_res)
                {
                        copyToFinal(curr_path);
                        final_res = curr_res;
                }
        }
        return;
}

// for any other level iterate for all vertices to
// build the search space tree recursively
for (int i=0; i<N; i++)
{
        // Consider next vertex if it is not same (diagonal
        // entry in adjacency matrix and not visited
        // already)
        if (adj[curr_path[level-1]][i].dist != 0 &&
                visited[i] == false)
        {
                int temp = curr_bound;
                curr_weight += adj[curr_path[level-1]][i].dist;

                // different computation of curr_bound for
```

```
            // level 2 from the other levels
            if (level==1)
            curr_bound -= ((firstMin(curr_path[level-1]) +
                                    firstMin(i))/2);
            else
            curr_bound -= ((secondMin(curr_path[level-1]) +
                                    firstMin(i))/2);

            // curr_bound + curr_weight is the actual lower bound
            // for the node that we have arrived on
            // If current lower bound < final_res, we need to explore
            // the node further
            if (curr_bound + curr_weight < final_res)
            {
                curr_path[level] = i;
                visited[i] = true;

                // call TSPRec for the next level
                TSPRec(curr_bound, curr_weight, level+1,
                    curr_path);
            }

            // Else we have to prune the node by resetting
            // all changes to curr_weight and curr_bound
            curr_weight -= adj[curr_path[level-1]][i].dist;
            curr_bound = temp;

            // Also reset the visited array
            memset(visited, false, sizeof(visited));
            for (int j=0; j<=level-1; j++)
                visited[curr_path[j]] = true;
        }
    }
}
```

```
// This function sets up final_path[]
void TSP()
{
        int curr_path[N+1];

        // Calculate initial lower bound for the root node
        // using the formula 1/2 * (sum of first min +
        // second min) for all edges.
        // Also initialize the curr_path and visited array
        int curr_bound = 0;
        memset(curr_path, -1, sizeof(curr_path));
        memset(visited, 0, sizeof(curr_path));

        // Compute initial bound
        for (int i=0; i<N; i++)
                curr_bound += (firstMin(i) + secondMin(i));

        // Rounding off the lower bound to an integer
        curr_bound = (curr_bound&1)? curr_bound/2 + 1 :curr_bound/2;

        // We start at vertex 1 so the first vertex
        // in curr_path[] is 0
        visited[0] = true;
        curr_path[0] = 0;

        // Call to TSPRec for curr_weight equal to
        // 0 and level 1
        TSPRec(curr_bound, 0, 1, curr_path);
}




 void updatetime()
 {
    for (int i=0;i<n;i++)
```

```
        {
            for (int j=0;j<n;j++)
            {
                if (adj[i][j].dist!=10000)
                {
                    //random value of traffic is selected
                    adj[i][j].traffic=rand()%2;
                    if (adj[i][j].traffic==1) adj[i][j].time=((float)adj[i][j].dist)/((float)speed);
                    //speed is decreased in case of high traffic
                    else if (adj[i][j].traffic==2) adj[i][j].time=((float)adj[i][j].dist)/((float)(speed-15));
                    //speed is increased in case of low traffic
                    else adj[i][j].time=((float)adj[i][j].dist)/((float)(speed+15));
                }
                else adj[i][j].time=10000;
            }
        }
}

void updatetraffic()
{
        //to generate new random values for every unique time
        srand(time(0));
        updatetime();
}

Graph()
{
        n=11;
        speed=32;
        updatetraffic();

        //initialisation of activities
        activities[0]={"AB Exhibition ",10,12};
        activities[1]={"Heritage Walk ",6,9};
        activities[2]={"Fireworks show",21,22};
```

```
    activities[3]={"Light show    ",17,18};
    activities[4]={"CD Exhibition ",14,16};
    activities[5]={"Monuments Tour",11,14};
    activities[6]={"PQ Show       ",13,15};
    activities[7]={"Water show    ",10,11};
    activities[8]={"Birdwatching  ",5,7};
    activities[9]={"Art Gallery   ",16,17};
}

void updatespeed()
{
  system("CLS");
  int sp;
  while (1)
  {
    cout<<"\n\n          Enter new speed in km/h: ";
    cin>>sp;
    //validity of entered speed is checked
    if (sp>50) cout<<"          Speed cannot be greater than 50 km/h";
    else if (sp<=15) cout<<"          Enter valid speed in km/h";
    else
    {
      speed=sp;
      cout<<"\n\n          Speed has been updated.\n";
      updatetime();
      break;
    }
  }
  getch();
}

//to return type of traffic based on its value
string traffic (int i, int j)
{
  int temp=adj[i][j].traffic;
```

```
    if (temp==2) return "HIGH";
    else if (temp==1) return "NORMAL";
    else return "LOW";
}

//function to find min. cost path on basis of time using traveling salesman problem
float tsp (int Path[], int B[], int n, int l, float timesofar)
  {
     //check if last district is reached
     if (l==n-1) {
                  //check if current path is minimum costing
                 if (minc>timesofar+adj[B[n-1]][B[0]].time)
                 {
                     //copy currently best solution to B
                     minc=timesofar+adj[B[n-1]][B[0]].time;
                     for (int i=0;i<n;i++) Path[i]=B[i];
                 }
             }

     else
     {
        for (int i=l+1; i<n; i++)
        {
           Swap(B,i,l+1);
           //retract if current solution is not better than existing best solution
           if ((timesofar+adj[B[l]][B[l+1]].time)>minc);
           else minc=Min(minc,tsp(Path,B,n,l+1,timesofar+adj[B[l]][B[l+1]].time));
           Swap(B,i,l+1);
        }
     }

     return minc;
  }

void fulltour (int source)
```

```
 {
    system("CLS");
    minc=INT_MAX;
    int Path[n],B[n];
    for (int i=0;i<n;i++) Path[i]=B[i]=i;
    //set first element of path as entered source
    Path[0]=B[0]=source;
    Path[source]=B[source]=0;

    float t=tsp(Path,B,n,0,0.0);

    cout<<"\n\n        Here is the path we recommend you take to tour the entire city in minimum time.
Happy journey!\n\n";

    for (int i=0;i<n-1;i++)
    {
       cout<<"        "<<names[Path[i]]<<"\n          |\n          |  "<<adj[Path[i]][Path[i+1]].time<<" hrs";
       cout<<"\n          v\n";
    }

    cout<<"        "<<names[Path[n-1]]<<"\n          |\n          |  "<<adj[Path[n-1]][Path[0]].time<<" hrs";
    cout<<"\n          v\n"<<"        "<<names[Path[0]];

    cout<<"\n\n        Total time taken = "<<t<<" hours";

    getch();
 }

 void dayplan(int s)
 {
    system("CLS");

    cout<<"\n\n        Available activities in "<<names[s]<<": (Enter 1 if you want to participate and 0 if
not)\n\n";
```

```
vector<act> V;
int ch;

for (int i=0;i<5;i++)
{
   cout<<"        "<<activities[availableacts[s][i]].Name<<": "<<activities[availableacts[s][i]].start;
   cout<<":00 to "<<activities[availableacts[s][i]].End<<":00\n";
   cout<<"        ";
   cin>>ch;
   //add activities to vector V if user wants to do them
   if (ch==1) V.push_back(activities[availableacts[s][i]]);
}

if (V.empty()) cout<<"        No activities selected. Have a fun relaxing day!";
else
{
   //sort by end time
   quicksort(V,0,V.size()-1);

   int Count=0;
   Count++;

   int temp=0;
   cout<<"\n        For participating in max. no. of desired activities,";
   cout<<"\n        we recommend following the following schedule:\n\n";
   cout<<"        Name\t\t\tStart time\t\tEnd time\n        ";
   cout<<V[0].Name<<"\t\t"<<V[0].start<<":00\t\t"<<V[0].End<<":00"<<endl;
   for (int i=1;i<V.size();i++)
   {
      //check if new activity can be started based on end time of current activity
      if (V[i].start>=V[temp].End)
      {
         cout<<"        "<<V[i].Name<<"\t\t"<<V[i].start<<":00\t\t"<<V[i].End<<":00"<<endl;
         Count++;
         temp=i;
```

```
              }
          }

          cout<<"\n          Max. no. of activities: "<<Count;
      }

      getch();
  }



  int miniDist(int min_distance[], bool visited[])

{
    int minimum=INT_MAX,value;

    for(int k=0;k<11;k++)
    {
        if(visited[k]==false && min_distance[k]<=minimum)
        {
            minimum=min_distance[k];
            value=k;
        }
    }
    return value;
}

void DijkstraAlgo(int src,int dest)

{
    int min_distance[11];
    bool visited[11];



    for(int k = 0; k<11; k++)
    {
        min_distance[k] = INT_MAX;

        visited[k] = false;

    }


    min_distance[src] = 0;


    for(int k = 0; k<11; k++)

    {

        int m=miniDist(min_distance,visited);

        visited[m]=true;

        for(int k = 0; k<11; k++)

        {


            if(!visited[k] && adj[m][k].dist && min_distance[m]!=INT_MAX &&
min_distance[m]+adj[m][k].dist<min_distance[k])

                min_distance[k]=min_distance[m]+adj[m][k].dist;

        }

    }


    cout<<"Shortest distance from source vertex to destination vertex is :"<<endl;


    for(int k = 0; k<11; k++)


    {

        if (dest==k)

        {

            cout<<min_distance[k]<<endl;

        }

    }


    cout<<"distance of all vertices from source vertex  :"<<endl;


    for(int k = 0; k<11; k++)


    {
      cout<<k<<"\t\t\t"<<min_distance[k]<<endl;


   }
        getch();


}



  void menu()
  {
     int ch;
     while (1)
     {
        system("CLS");
        cout<<"          ********************************************************************************************************\n\n";
        cout<<"                              TripTrekker\n\n";
        cout<<"          ********************************************************************************************************\n\n";
        cout<<"       Choose an option\n\n";
        cout<<"       1. Plan a full city tour\n\n       2. Update speed (current speed = "<<speed<<"
km/h)";
        cout<<"\n\n       3. Shortest distance between 2 districts"  ;
                    cout<<"\n\n       4. Minimum distance to travel efficiently across the city";
                    cout<<"\n\n       5. Full day planner\n\n       6. Exit\n\n      ";
        cin>>ch;
        switch(ch)
        {
           case 1: {
                      system("CLS");
                      cout<<"\n\n";
                      for (int i=0;i<n;i++) cout<<"       "<<i<<". "<<names[i]<<endl;
                      int s;
                      while (1)
                      {
                         cout<<"\n\n       Enter your source district (no.): ";
                    cin>>s;
                    if (s<0||s>=11) cout<<"          Enter a valid no.";
                    else
                    {
                        fulltour(s);
                        break;
                    }
                }
                break;
            }
        case 2: updatespeed(); break;
                        case 3:{
                                system("CLS");
                cout<<"\n\n";
                for (int i=0;i<n;i++) cout<<"          "<<i<<". "<<names[i]<<endl;
                int s,d;
                while (1)
                {
                    cout<<"\n\n          Enter your source district (no.): ";
                    cin>>s;
                                                        cout<<"\n\n          Enter your destination
district (no.): ";
                    cin>>d;
                    if ((s<0||s>=11) && (d<0||d>=11)) cout<<"          Enter a valid no.";
                    else
                    {
                        DijkstraAlgo(s,d); break;
                    }
                }
                break;
                        }
                        case 4:{
         system("CLS");
                         cout<<"\n\n";
                for (int i=0;i<n;i++) cout<<"          "<<i<<". "<<names[i]<<endl;
```

```
};

                            TSP();
                            printf("Minimum distance required to travel : %d km\n", final_res);
                  printf("Path Taken : ");
                  for (int i=0; i<=11; i++)
                          printf("%d ", final_path[i]);
                getch();
                          break;

                          }

            case 5: system("CLS");
                    cout<<"\n\n      Choose where you're spending a whole day:\n\n";
                    for (int i=0;i<n;i++) cout<<"      "<<i<<". "<<names[i]<<endl;
                    int s;
                    while (1)
                    {
                      cout<<"\n\n      Enter your source district (no.): ";
                      cin>>s;
                      if (s<0||s>=11) cout<<"      Enter a valid no.";
                      else
                      {
                        dayplan(s);
                        break;
                      }
                    }
                    break;

            default: exit(0);
          }
        }
      }

};

int main()
```

```
{
        for(int i=0;i<11;i++)
   {
        fastTime[i]=new int[11];
        for(int j=0;j<11;j++)
        {
                 fastTime[i][j]=-1;
                 }
        }
   Graph G;
   G.menu();
        return 0;
}
```