Map Routingm

\*Team members :-

1. Wael Mohammed Abd El\_Moati.

ID :-2016170615.

Gmail🡪 Wael.mohamed91.22@gmail.com

1. Mahmoud Hamed Mohammed Hassan.

ID :-2016170605.

1. Gehad Khaled Sayed.

ID :-2016170142.

1. Ahmed Mohammed Saadallah

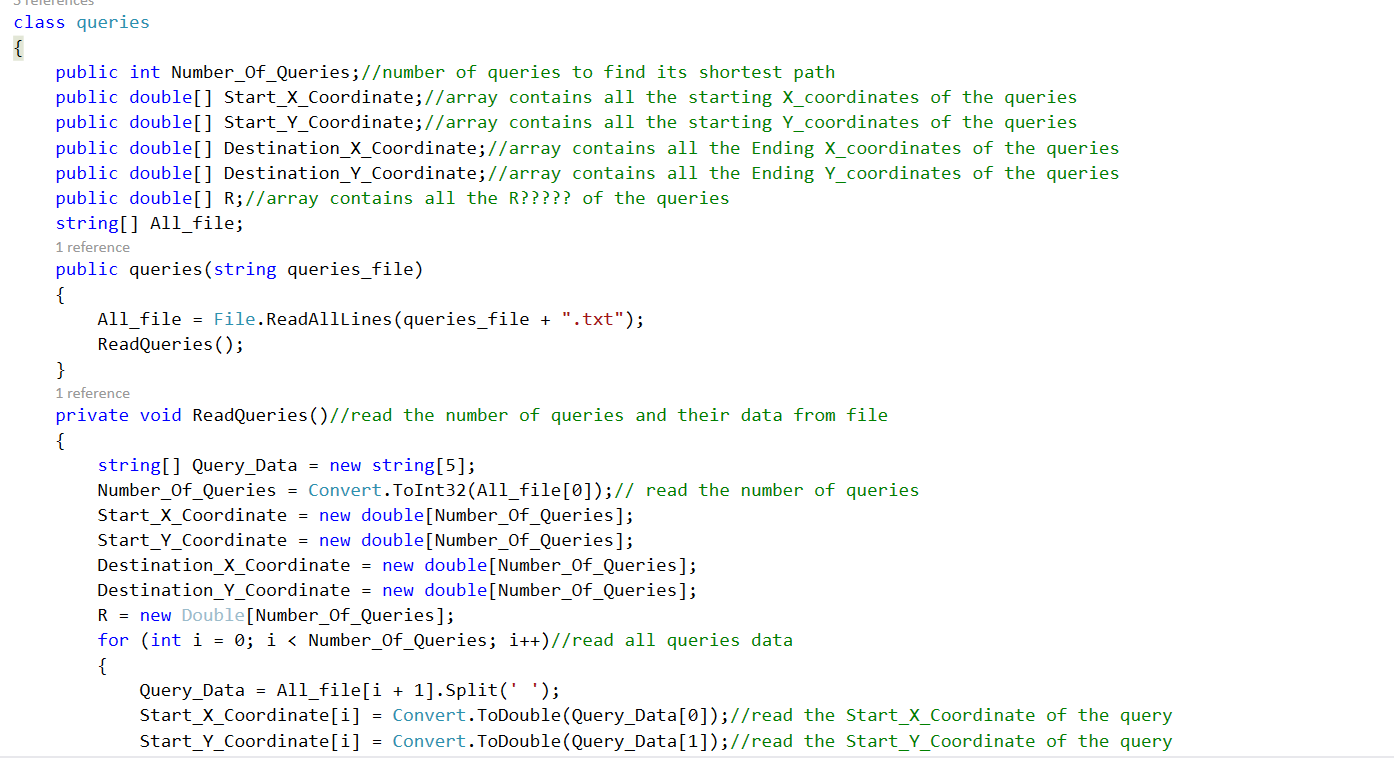
ID :-2016170048.

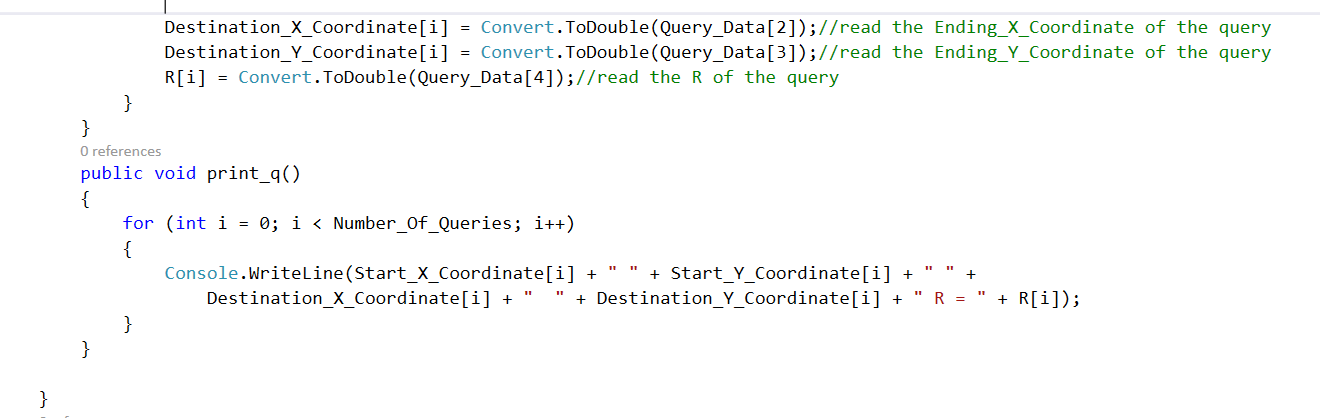
1. Asmaa Hamed Mohammed Abd El\_Rahman.

ID :-2016170089.

Now discuss the code (\*\_\*)

\*Let’s start with class queries :- this class read from **Queries** file (this file contain the coordinates of source and destination and R (R it’s maximum distance the person can walk for maximum R meters to get from an intersection to the source).





(\*\_\*) Hear We Discuss The Code Line by Line (\*\_\*)

/\* Notes :-

1)each vertices contain x,y coordinates .

2)each edge contain 2 vertices ,speed of route ,distance).

3)the meaning of start (source:The place of person ).

4)V::mean node or intersection .

5)E::mean edge or road .

6)n :: number of queries.

\*/

class queries

{

public int Number\_Of\_Queries;

🡺number of queries to find its shortest path

🡺Complexity:o(1).

public double[] Start\_X\_Coordinate;

🡺array contains all the starting X\_coordinates of the queries

🡺Complexity:o(1).

public double[] Start\_Y\_Coordinate;

🡺array contains all the starting Y\_coordinates of the queries

🡺Complexity:o(1).

public double[] Destination\_X\_Coordinate;

🡺array contains all the Ending X\_coordinates of the queries

🡺Complexity:o(1).

public double[] Destination\_Y\_Coordinate;

🡺array contains all the Ending Y\_coordinates of the queries

🡺Complexity:o(1).

public double[] R;

🡺array contains all the R????? of the queries

🡺Complexity:o(1).

string[] All\_file;

🡺Complexity:o(1).

public queries(string queries\_file)

{

All\_file = File.ReadAllLines(queries\_file + ".txt");

🡺Complexity:o(1).

ReadQueries();

🡺Complexity:o(1).

}

◆🡻🡻The complexity of function(ReadQueries)=o(body)=o(n).

private void ReadQueries()

🡺read the number of queries and their data from file

{

string[] Query\_Data = new string[5];

🡺Complexity:o(1).

Number\_Of\_Queries = Convert.ToInt32(All\_file[0]);

🡺read the number of queries

🡺Complexity:o(1).

Start\_X\_Coordinate = new double[Number\_Of\_Queries];

🡺Complexity:o(1).

Start\_Y\_Coordinate = new double[Number\_Of\_Queries];

🡺Complexity:o(1).

Destination\_X\_Coordinate = new double[Number\_Of\_Queries];

🡺Complexity:o(1).

Destination\_Y\_Coordinate = new double[Number\_Of\_Queries];

🡺Complexity:o(1).

R = new Double[Number\_Of\_Queries];

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=n\*o(1)=o(n).

for (int i = 0; i < Number\_Of\_Queries; i++)

🡺read all queries data

{

Query\_Data = All\_file[i + 1].Split(' ');

🡺Complexity:o(1).

Start\_X\_Coordinate[i] = Convert.ToDouble(Query\_Data[0]);

🡺read the Start\_X\_Coordinate of the query

🡺Complexity:o(1).

Start\_Y\_Coordinate[i] = Convert.ToDouble(Query\_Data[1]);

🡺read the Start\_Y\_Coordinate of the query

🡺Complexity:o(1).

Destination\_X\_Coordinate[i] = Convert.ToDouble(Query\_Data[2]);

🡺read the Ending\_X\_Coordinate of the query

🡺Complexity:o(1).

Destination\_Y\_Coordinate[i] = Convert.ToDouble(Query\_Data[3]);

🡺read the Ending\_Y\_Coordinate of the query

🡺Complexity:o(1).

R[i] = Convert.ToDouble(Query\_Data[4]);

🡺read the R of the query

🡺Complexity:o(1).

}

}

◆🡻🡻The complexity of function(print\_q)=o(body)=o(n).

public void print\_q()

{

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=n\*o(1)=o(n).

for (int i = 0; i < Number\_Of\_Queries; i++)

{

Console.WriteLine(Start\_X\_Coordinate[i] + " " + Start\_Y\_Coordinate[i] + " " +

Destination\_X\_Coordinate[i] + " " + Destination\_Y\_Coordinate[i] + " R = " + R[i]);

🡺Complexity:o(1).

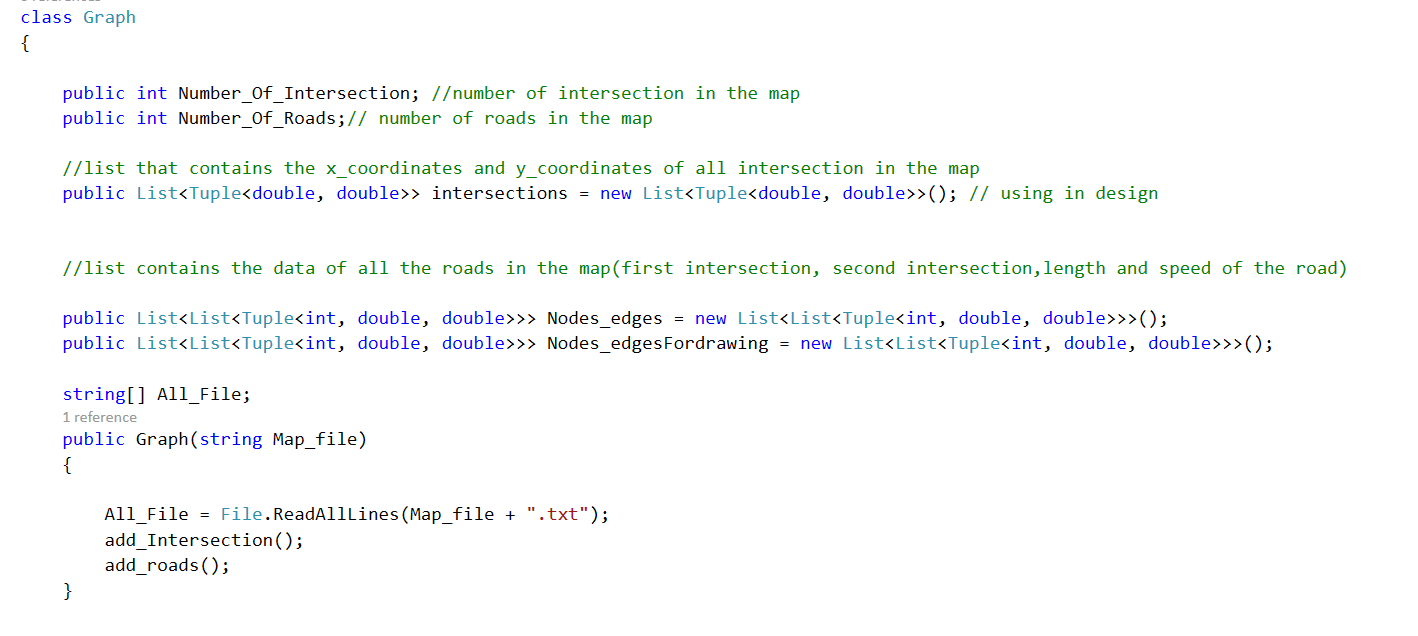
}

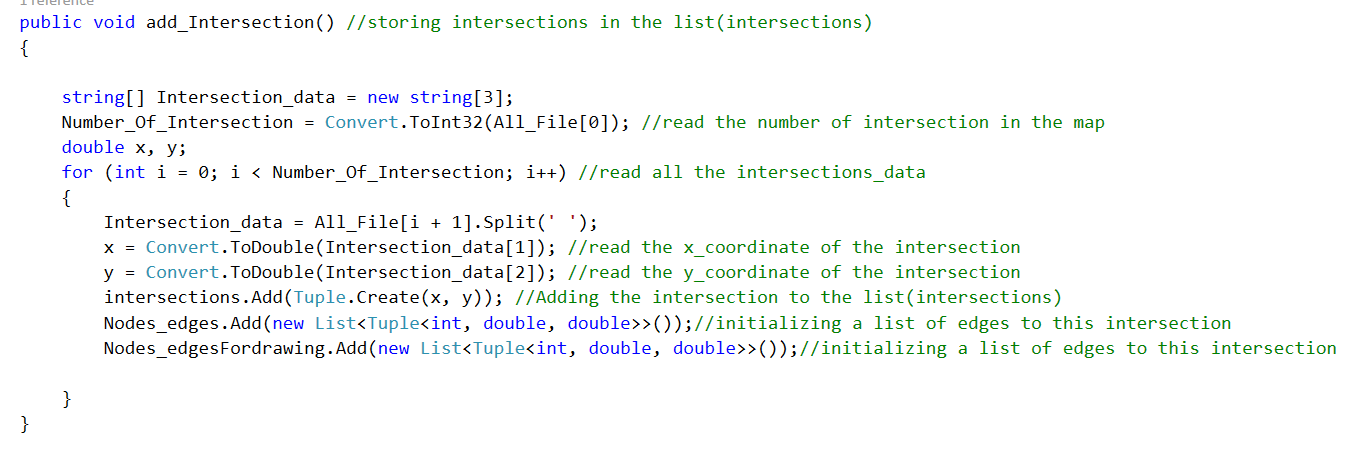
}

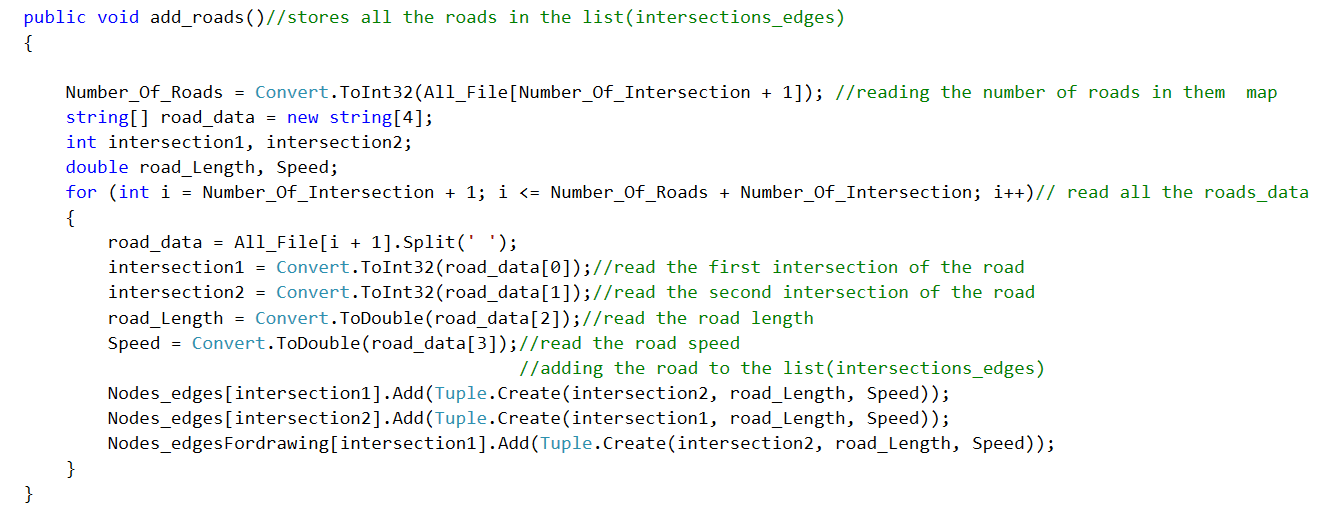
}

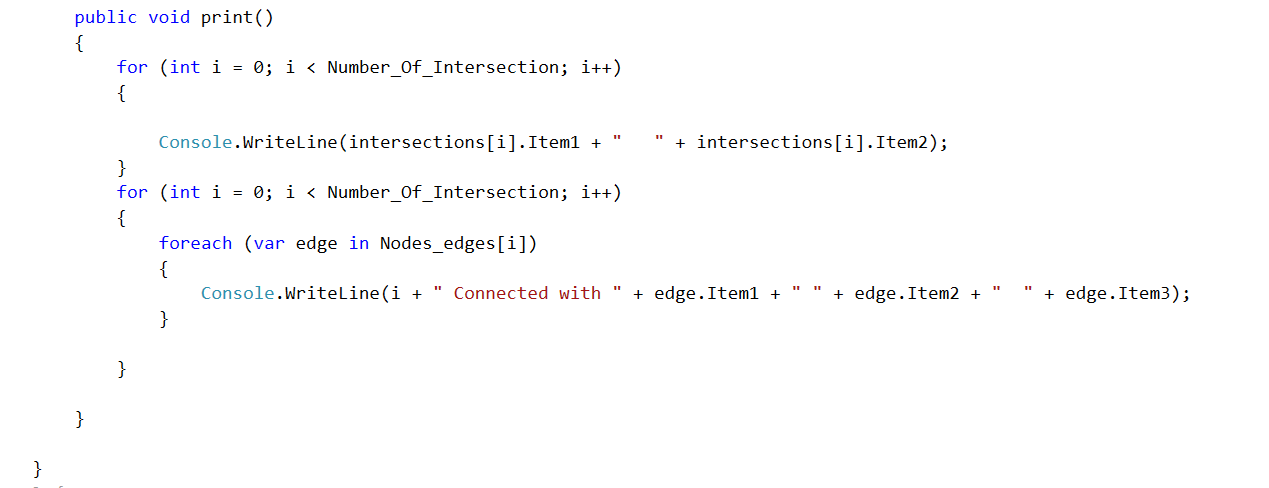
\*Now We Finish discussion of Class Queries(\*\_\*)

\*Let’s start with class graph :- this class read from map file (this file contain the number of vertices and for each vertices found its coordinates and R then number of edges , each edge contain 2 vertices with its speed and distance ).









(\*\_\*)Here WE Discuss Class Graph Line BY line (\*\_\*)

class Graph

{

public int Number\_Of\_Intersection;

🡺number of intersection in the map

🡺Complexity:o(1).

public int Number\_Of\_Roads;

🡺 number of roads in the map

🡺Complexity:o(1).

🡺list that contains the x\_coordinates and y\_coordinates of all intersection in the map

public List<Tuple<double, double>> intersections = new List<Tuple<double, double>>();

🡺Complexity:o(1).

🡺using in design

🡺 list contains the data of all the roads in the map(first intersection, second intersection,length and speed of the road)

public List<List<Tuple<int, double, double>>> Nodes\_edges = new List<List<Tuple<int, double, double>>>();

🡺Complexity:o(1).

public List<List<Tuple<int, double, double>>> Nodes\_edgesFordrawing = new List<List<Tuple<int, double, double>>>();

🡺Complexity:o(1).

string[] All\_File;

🡺Complexity:o(1).

public Graph(string Map\_file)

{

All\_File = File.ReadAllLines(Map\_file + ".txt");

🡺Complexity:o(1).

add\_Intersection();

🡺Complexity:o(1).

add\_roads();

🡺Complexity:o(1).

}

◆🡻🡻The complexity of function(add\_Intersection)=o(body)=o(V).

public void add\_Intersection()

🡺storing intersections in the list(intersections)

{

string[] Intersection\_data = new string[3];

🡺Complexity:o(1).

Number\_Of\_Intersection = Convert.ToInt32(All\_File[0]);

🡺read the number of intersection in the map

🡺Complexity:o(1).

double x, y;

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V\*o(1)=o(V).

for (int i = 0; i < Number\_Of\_Intersection; i++)

🡺read all the intersections\_data

{

Intersection\_data = All\_File[i + 1].Split(' ');

🡺Complexity:o(1).

x = Convert.ToDouble(Intersection\_data[1]);

🡺read the x\_coordinate of the intersection

🡺Complexity:o(1).

y = Convert.ToDouble(Intersection\_data[2]);

🡺read the y\_coordinate of the intersection

🡺Complexity:o(1).

intersections.Add(Tuple.Create(x, y));

🡺Adding the intersection to the list(intersections)

🡺Complexity:o(1).

Nodes\_edges.Add(new List<Tuple<int, double, double>>());

🡺initializing a list of edges to this intersection

🡺Complexity:o(1).

Nodes\_edgesFordrawing.Add(new List<Tuple<int, double, double>>());

🡺initializing a list of edges to this intersection

🡺Complexity:o(1).

}

}

◆🡻🡻The complexity of function(add\_roads)=o(body)=o(E).

public void add\_roads()

🡺stores all the roads in the list(intersections\_edges)

{

Number\_Of\_Roads = Convert.ToInt32(All\_File[Number\_Of\_Intersection + 1]); 🡺reading the number of roads in them map

🡺Complexity:o(1).

string[] road\_data = new string[4];

🡺Complexity:o(1).

int intersection1, intersection2;

🡺Complexity:o(1).

double road\_Length, Speed;

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V+E\*o(1)=o(V+E).

for (int i = Number\_Of\_Intersection + 1; i <= Number\_Of\_Roads + Number\_Of\_Intersection; i++)

🡺read all the roads\_data

{

road\_data = All\_File[i + 1].Split(' ');

🡺Complexity:o(1).

intersection1 = Convert.ToInt32(road\_data[0]);

🡺read the first intersection of the road

🡺Complexity:o(1).

intersection2 = Convert.ToInt32(road\_data[1]);

🡺read the second intersection of the road

🡺Complexity:o(1).

road\_Length = Convert.ToDouble(road\_data[2]);

🡺read the road length

🡺Complexity:o(1).

Speed = Convert.ToDouble(road\_data[3]);

🡺read the road speed

🡺Complexity:o(1).

🡺adding the road to the list(intersections\_edges)

Nodes\_edges[intersection1].Add(Tuple.Create(intersection2, road\_Length, Speed));

🡺Complexity:o(1).

Nodes\_edges[intersection2].Add(Tuple.Create(intersection1, road\_Length, Speed));

🡺Complexity:o(1).

Nodes\_edgesFordrawing[intersection1].Add(Tuple.Create(intersection2, road\_Length, Speed));

🡺Complexity:o(1).

}

}

◆🡻🡻The complexity of function(print)=o(body)=o(V).

public void print()

{

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V\*o(1)=o(V).

for (int i = 0; i < Number\_Of\_Intersection; i++)

{

Console.WriteLine(intersections[i].Item1 + " " + intersections[i].Item2);

🡺Complexity:o(1).

}

⍟◆🡻Complexity of nested loop =number of iterations \* o(inner loop)--🡪here=V\*o(v)=o(V^2).

for (int i = 0; i < Number\_Of\_Intersection; i++)

{

foreach (var edge in Nodes\_edges[i])

{

Console.WriteLine(i + " Connected with " + edge.Item1 + " " + edge.Item2 + " " + edge.Item3);

🡺Complexity:o(1).

}

}

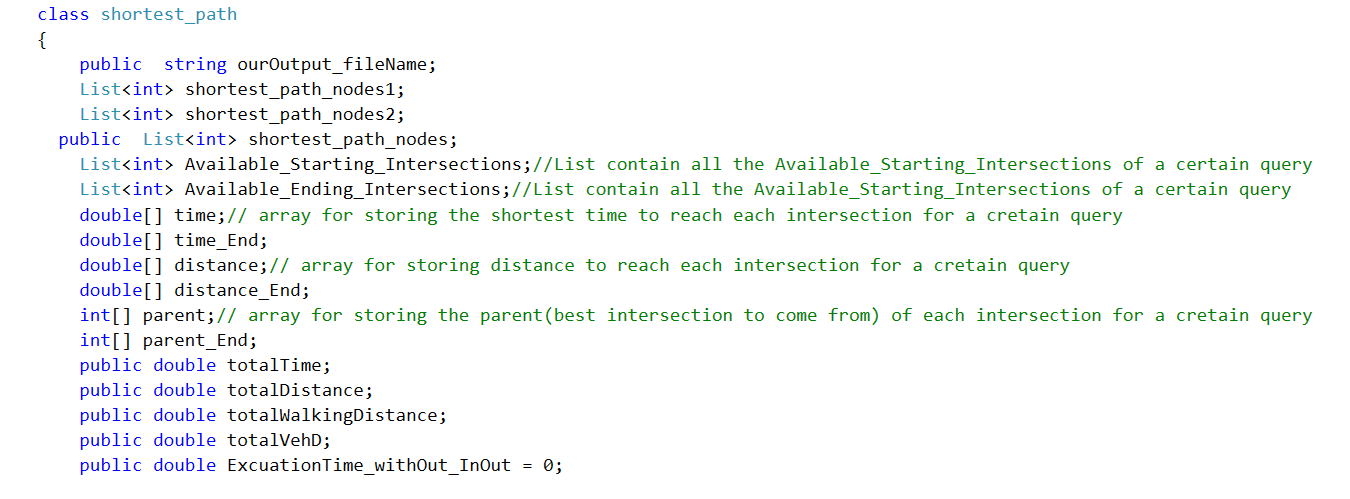
}

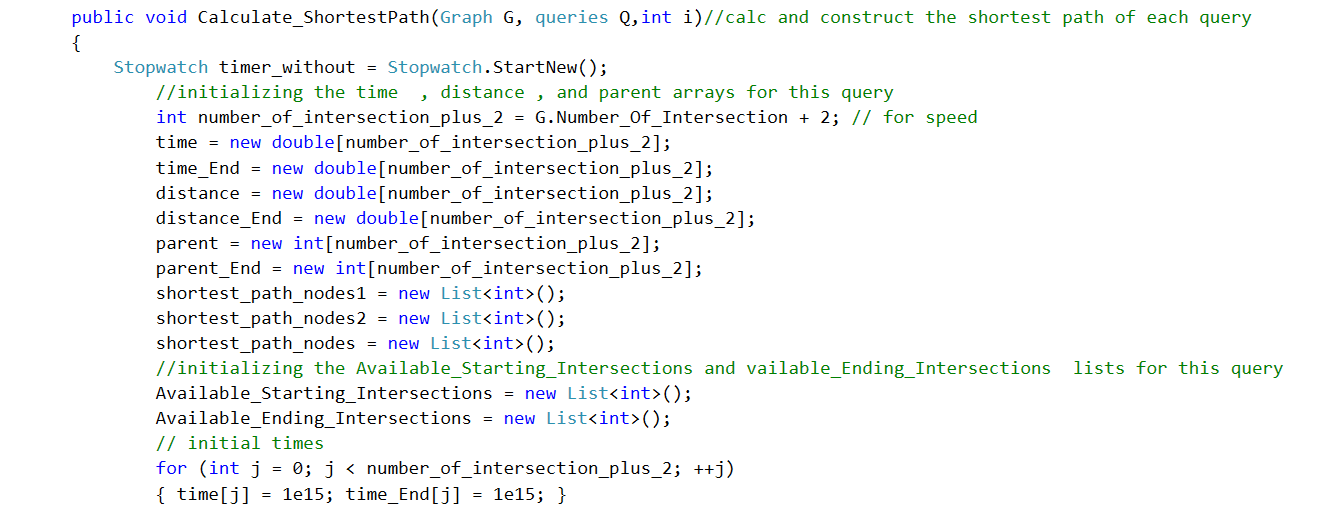
}

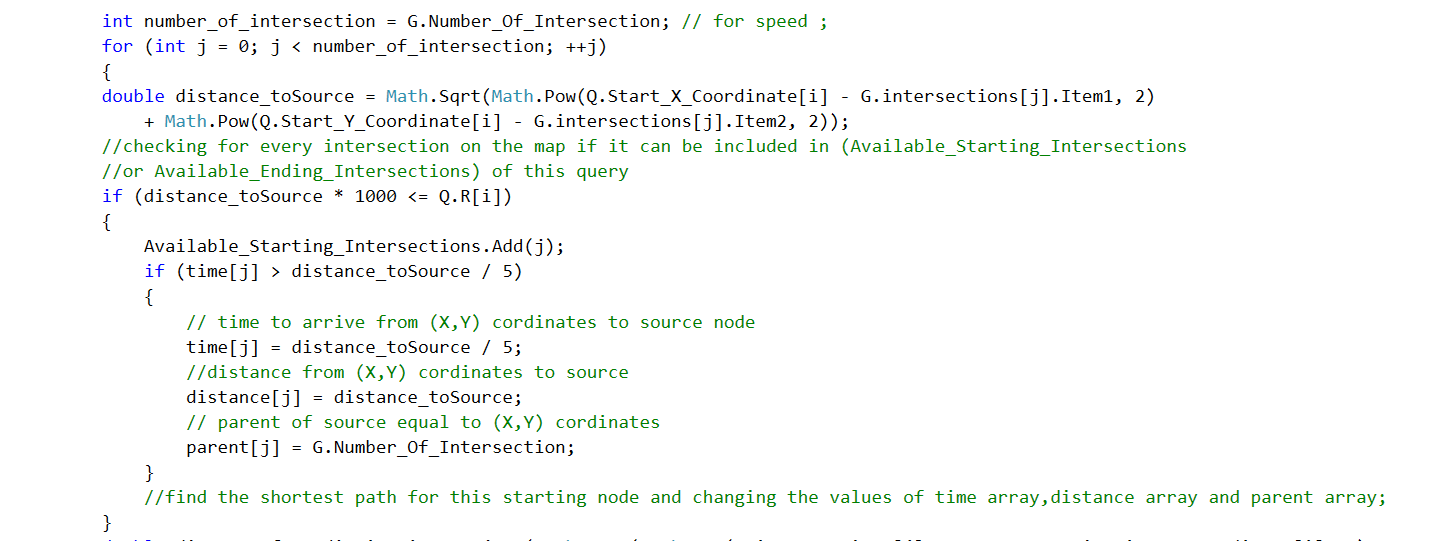
\*Now We Finish discussion of Class graph (\*\_\*)

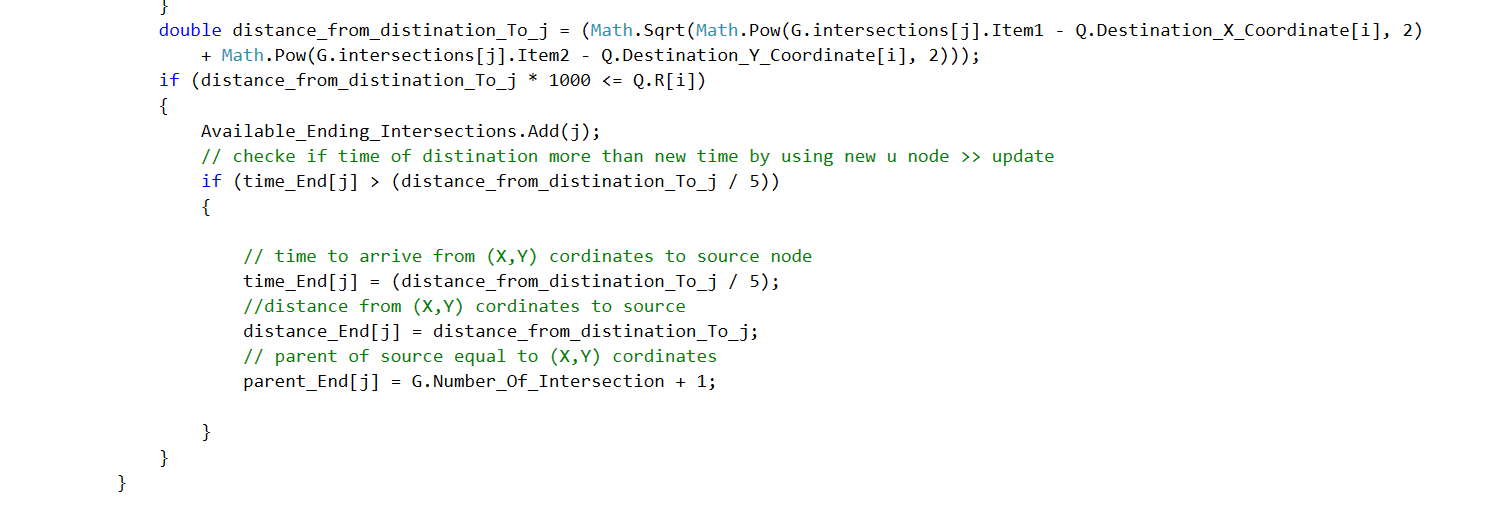
\*\*Let’s Start with class Shortest\_Path :-

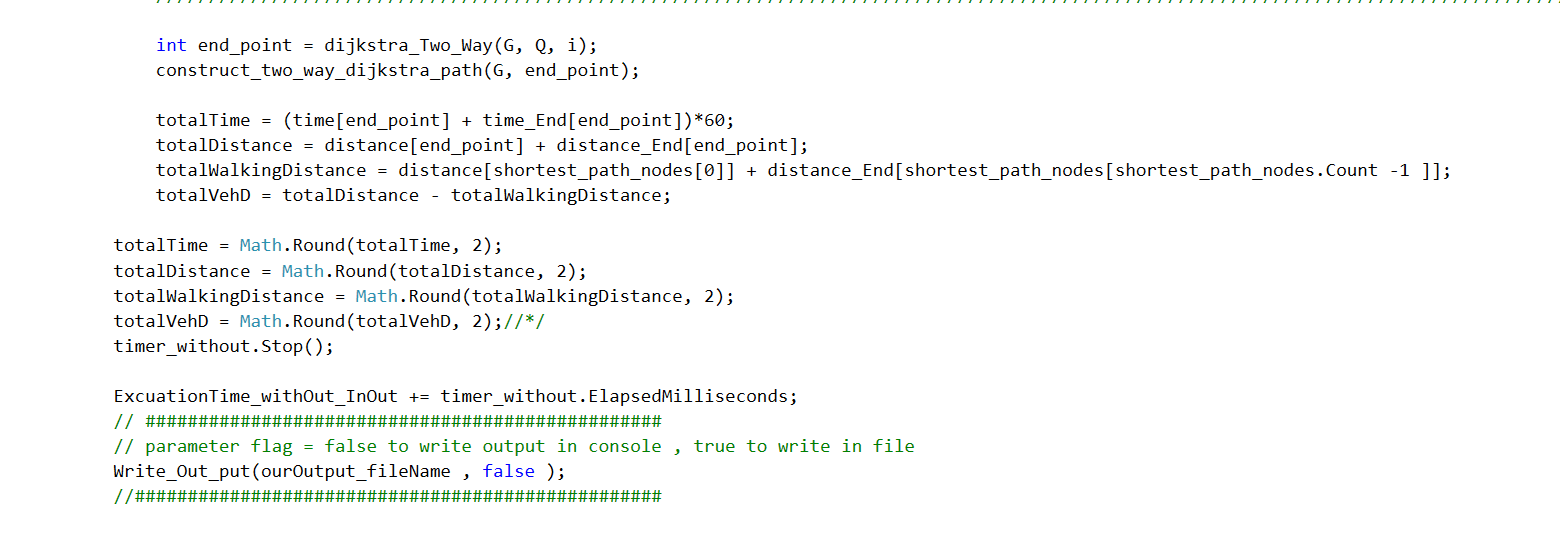
\*IN This Class We Calculate Shortest Path From Source to Destination depends on dijkstra algorithm .

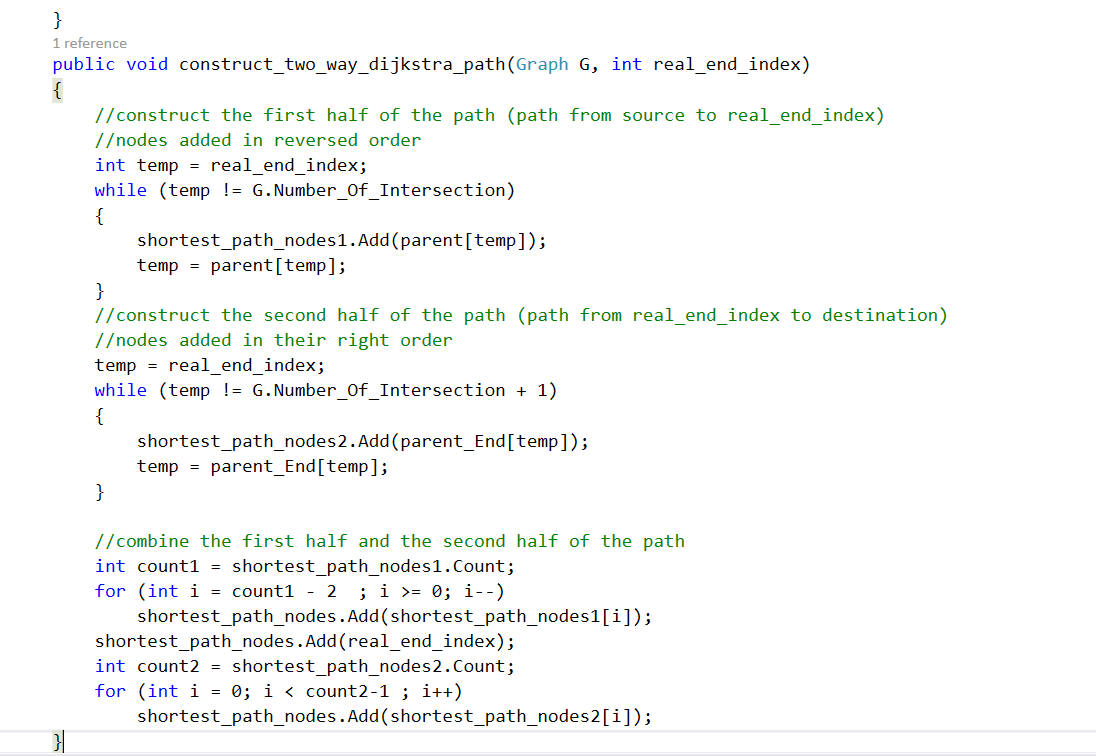


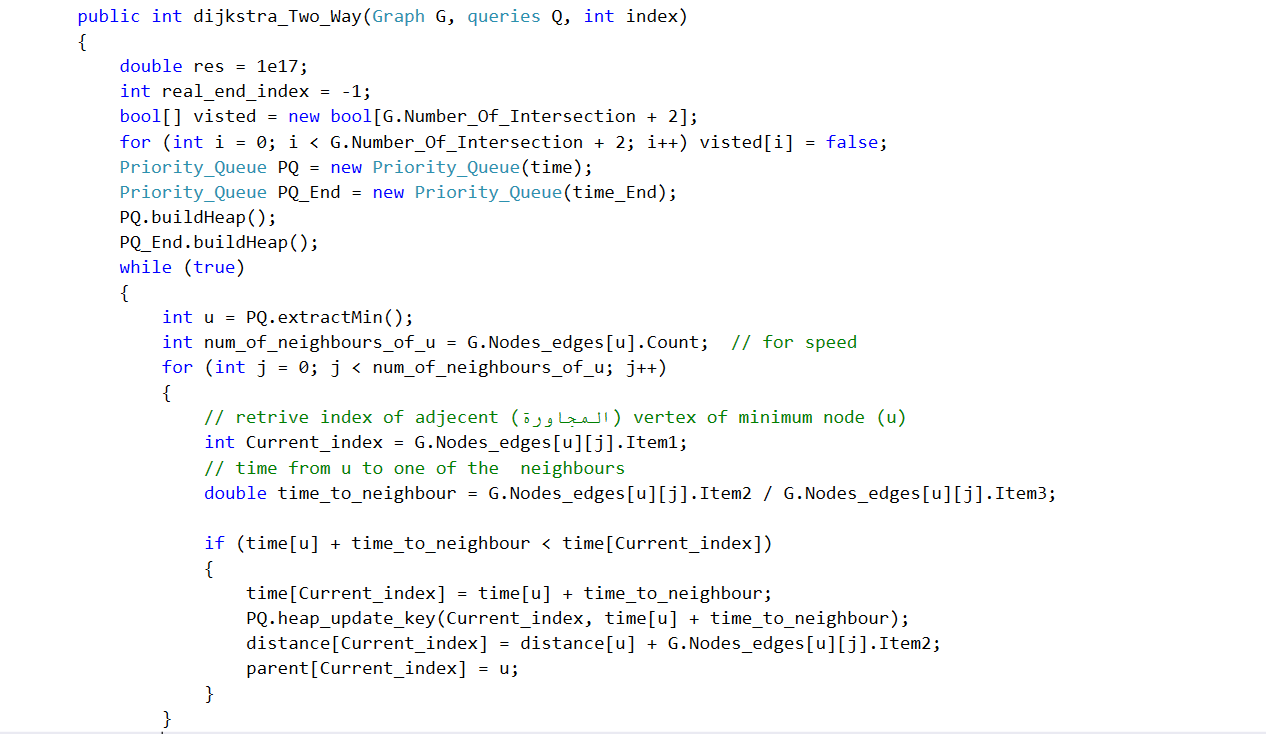


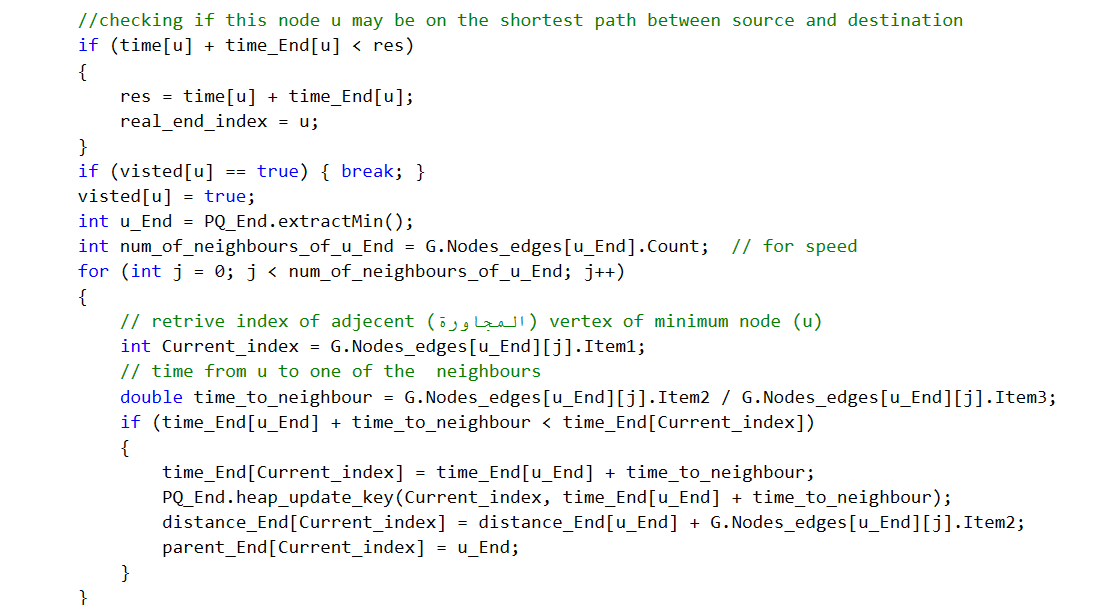


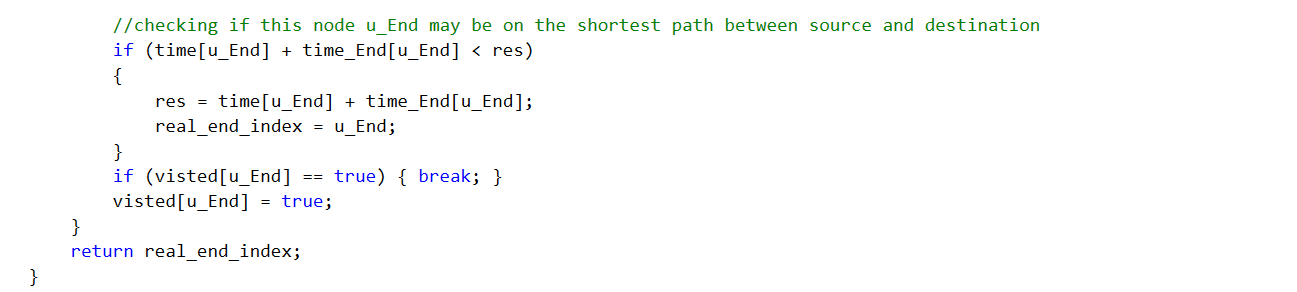


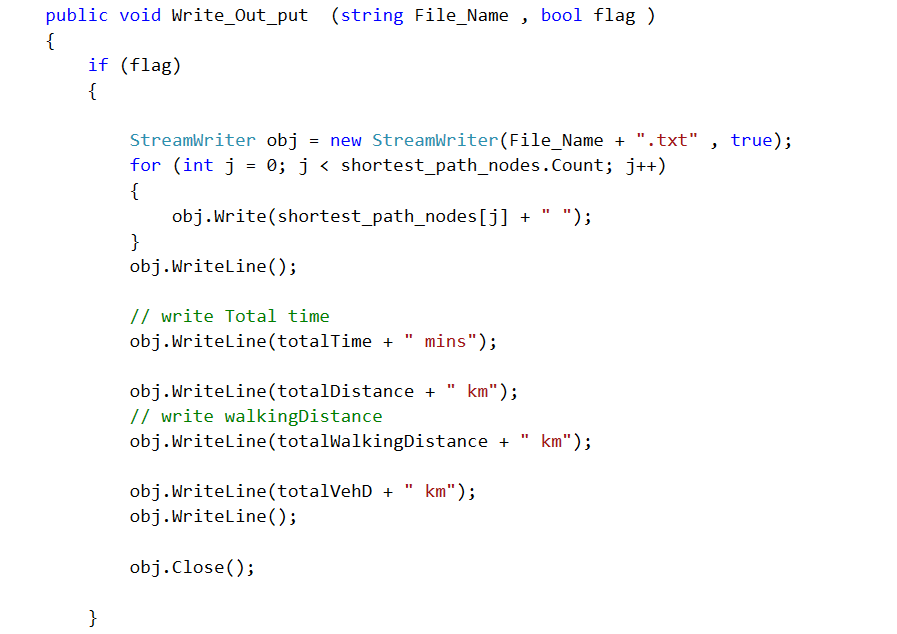


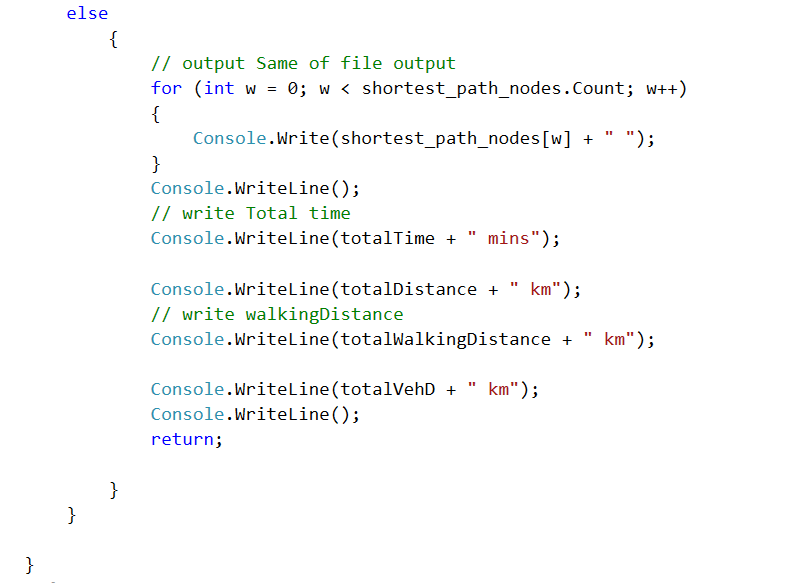












(\*\_\*)Here WE Discuss Class Shortest\_Path Line BY line (\*\_\*)

class shortest\_path

{

public string ourOutput\_fileName;

🡺Complexity:o(1).

List<int> shortest\_path\_nodes1;

🡺Complexity:o(1).

List<int> shortest\_path\_nodes2;

🡺Complexity:o(1).

public List<int> shortest\_path\_nodes;

🡺Complexity:o(1).

List<int> Available\_Starting\_Intersections;

🡺List contain all the Available\_Starting\_Intersections of a certain query

🡺Complexity:o(1).

List<int> Available\_Ending\_Intersections;

🡺List contain all the Available\_Starting\_Intersections of a certain query

🡺Complexity:o(1).

double[] time;

🡺array for storing the shortest time to reach each intersection for a cretain query

🡺Complexity:o(1).

double[] time\_End;

🡺Complexity:o(1).

double[] distance;

🡺 array for storing distance to reach each intersection for a cretain query

🡺Complexity:o(1).

double[] distance\_End;

🡺Complexity:o(1).

int[] parent;

🡺Complexity:o(1).

🡺 array for storing the parent(best intersection to come from) of each intersection for a cretain query

int[] parent\_End;

🡺Complexity:o(1).

public double totalTime;

🡺Complexity:o(1).

public double totalDistance;

🡺Complexity:o(1).

public double totalWalkingDistance;

🡺Complexity:o(1).

public double totalVehD;

🡺Complexity:o(1).

public double ExcuationTime\_withOut\_InOut = 0;

🡺Complexity:o(1).

◆🡻🡻The complexity of function(Calculate\_ShortestPath)=o(body)=o(E`log V`).

public void Calculate\_ShortestPath(Graph G, queries Q,int i)

🡺calc and construct the shortest path of each query

{

Stopwatch timer\_without = Stopwatch.StartNew();

🡺initializing the time , distance , and parent arrays for this query

🡺Complexity:o(1).

int number\_of\_intersection\_plus\_2 = G.Number\_Of\_Intersection + 2;

🡺for speed

🡺Complexity:o(1).

time = new double[number\_of\_intersection\_plus\_2];

🡺Complexity:o(1).

time\_End = new double[number\_of\_intersection\_plus\_2];

🡺Complexity:o(1).

distance = new double[number\_of\_intersection\_plus\_2];

🡺Complexity:o(1).

distance\_End = new double[number\_of\_intersection\_plus\_2];

🡺Complexity:o(1).

parent = new int[number\_of\_intersection\_plus\_2];

🡺Complexity:o(1).

parent\_End = new int[number\_of\_intersection\_plus\_2];

🡺Complexity:o(1).

shortest\_path\_nodes1 = new List<int>();

🡺Complexity:o(1).

shortest\_path\_nodes2 = new List<int>();

🡺Complexity:o(1).

shortest\_path\_nodes = new List<int>();

🡺Complexity:o(1).

🡺 initializing the Available\_Starting\_Intersections and vailable\_Ending\_Intersections lists for this query

Available\_Starting\_Intersections = new List<int>();

🡺Complexity:o(1).

Available\_Ending\_Intersections = new List<int>();

🡺 initial times

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V\*o(1)=o(V).

for (int j = 0; j < number\_of\_intersection\_plus\_2; ++j)

{

time[j] = 1e15; time\_End[j] = 1e15;

🡺Complexity:o(1).

}

int number\_of\_intersection = G.Number\_Of\_Intersection;

🡺for speed ;

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V\*o(1)=o(V).

for (int j = 0; j < number\_of\_intersection; ++j)

{

double distance\_toSource = Math.Sqrt(Math.Pow(Q.Start\_X\_Coordinate[i] - G.intersections[j].Item1, 2)

+ Math.Pow(Q.Start\_Y\_Coordinate[i] - G.intersections[j].Item2, 2));

🡺Complexity:o(1).

🡺checking for every intersection on the map if it can be included in (Available\_Starting\_Intersections or Available\_Ending\_Intersections) of this query

if (distance\_toSource \* 1000 <= Q.R[i])

{

Available\_Starting\_Intersections.Add(j);

🡺Complexity:o(1).

if (time[j] > distance\_toSource / 5)

{

🡺time to arrive from (X,Y) cordinates to source node

time[j] = distance\_toSource / 5;

🡺Complexity:o(1).

🡺distance from (X,Y) cordinates to source

distance[j] = distance\_toSource;

🡺Complexity:o(1).

🡺parent of source equal to (X,Y) cordinates

parent[j] = G.Number\_Of\_Intersection;

🡺Complexity:o(1).

}

🡺find the shortest path for this starting node and changing the values of time array,distance array and parent array;

}

double distance\_from\_distination\_To\_j = (Math.Sqrt(Math.Pow(G.intersections[j].Item1 - Q.Destination\_X\_Coordinate[i], 2)

+ Math.Pow(G.intersections[j].Item2 - Q.Destination\_Y\_Coordinate[i], 2)));

🡺Complexity:o(1).

if (distance\_from\_distination\_To\_j \* 1000 <= Q.R[i])

{

Available\_Ending\_Intersections.Add(j);

🡺Complexity:o(1).

🡺checke if time of distination more than new time by using new u node >> update

if (time\_End[j] > (distance\_from\_distination\_To\_j / 5))

{

🡺 time to arrive from (X,Y) cordinates to source node

time\_End[j] = (distance\_from\_distination\_To\_j / 5);

🡺Complexity:o(1).

🡺distance from (X,Y) cordinates to source

distance\_End[j] = distance\_from\_distination\_To\_j;

🡺Complexity:o(1).

🡺parent of source equal to (X,Y) cordinates

parent\_End[j] = G.Number\_Of\_Intersection + 1;

🡺Complexity:o(1).

}

}

}

int end\_point = dijkstra\_Two\_Way(G, Q, i);

🡺Complexity:o(E`log V`).

construct\_two\_way\_dijkstra\_path(G, end\_point);

🡺Complexity:o(V`).

totalTime = (time[end\_point] + time\_End[end\_point])\*60;

🡺Complexity:o(1).

totalDistance = distance[end\_point] + distance\_End[end\_point];

🡺Complexity:o(1).

totalWalkingDistance = distance[shortest\_path\_nodes[0]] + distance\_End[shortest\_path\_nodes[shortest\_path\_nodes.Count -1 ]];

🡺Complexity:o(1).

totalVehD = totalDistance - totalWalkingDistance;

🡺Complexity:o(1).

totalTime = Math.Round(totalTime, 2);

🡺Complexity:o(1).

totalDistance = Math.Round(totalDistance, 2);

🡺Complexity:o(1).

totalWalkingDistance = Math.Round(totalWalkingDistance, 2);

🡺Complexity:o(1).

totalVehD = Math.Round(totalVehD, 2);

🡺Complexity:o(1).

timer\_without.Stop();

🡺Complexity:o(1).

ExcuationTime\_withOut\_InOut += timer\_without.ElapsedMilliseconds;

🡺Complexity:o(1).

// #################################################

// parameter flag = false to write output in console , true to write in file

Write\_Out\_put(ourOutput\_fileName , false );

🡺Complexity:o(1).

//##################################################

◆🡻🡻The complexity of function(constract\_two\_way\_dijkstra\_path)=o(body)=o(v`).

public void construct\_two\_way\_dijkstra\_path(Graph G, int real\_end\_index)

{

🡺construct the first half of the path (path from source to real\_end\_index)

🡺 nodes added in reversed order

int temp = real\_end\_index;

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V`\*o(1)=o(V`).

while (temp != G.Number\_Of\_Intersection)

{

shortest\_path\_nodes1.Add(parent[temp]);

🡺Complexity:o(1).

temp = parent[temp];

🡺Complexity:o(1).

}

🡺 construct the second half of the path (path from real\_end\_index to destination)

🡺nodes added in their right order

temp = real\_end\_index;

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V`\*o(1)=o(V`).

while (temp != G.Number\_Of\_Intersection + 1)

{

shortest\_path\_nodes2.Add(parent\_End[temp]);

🡺Complexity:o(1).

temp = parent\_End[temp];

🡺Complexity:o(1).

}

🡺 combine the first half and the second half of the path

int count1 = shortest\_path\_nodes1.Count;

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V`\*o(1)=o(V`).

for (int i = count1 - 2 ; i >= 0; i--)

shortest\_path\_nodes.Add(shortest\_path\_nodes1[i]);

🡺Complexity:o(1).

shortest\_path\_nodes.Add(real\_end\_index);

🡺Complexity:o(1).

int count2 = shortest\_path\_nodes2.Count;

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V`\*o(1)=o(V`).

for (int i = 0; i < count2-1 ; i++)

shortest\_path\_nodes.Add(shortest\_path\_nodes2[i]);

🡺Complexity:o(1).

}

◆🡻🡻The complexity of function(dijkstra\_Two\_Way)=o(body)=o(E`LOG V`).

public int dijkstra\_Two\_Way(Graph G, queries Q, int index)

{

double res = 1e17;

🡺Complexity:o(1).

int real\_end\_index = -1;

🡺Complexity:o(1).

bool[] visted = new bool[G.Number\_Of\_Intersection + 2];

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V\*o(1)=o(V).

for (int i = 0; i < G.Number\_Of\_Intersection + 2; i++) visted[i] = false;

Priority\_Queue PQ = new Priority\_Queue(time);

🡺Complexity:o(1).

Priority\_Queue PQ\_End = new Priority\_Queue(time\_End);

🡺Complexity:o(1).

PQ.buildHeap();

🡺Complexity:o(V).

PQ\_End.buildHeap();

🡺Complexity:o(V).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=o(E`log V`).

while (true)

{

int u = PQ.extractMin();

🡺Complexity:o(log V).

int num\_of\_neighbours\_of\_u = G.Nodes\_edges[u].Count;

🡺Complexity:o(1).

🡺for speed

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=E`\*o(log V`)=o(E`log V`).

for (int j = 0; j < num\_of\_neighbours\_of\_u; j++)

{

🡺retrive index of adjecent (المجاورة) vertex of minimum node (u)

int Current\_index = G.Nodes\_edges[u][j].Item1;

🡺Complexity:o(1).

🡺 time from u to one of the neighbours

double time\_to\_neighbour = G.Nodes\_edges[u][j].Item2 / G.Nodes\_edges[u][j].Item3;

🡺Complexity:o(1).

if (time[u] + time\_to\_neighbour < time[Current\_index])

{

time[Current\_index] = time[u] + time\_to\_neighbour;

🡺Complexity:o(1).

PQ.heap\_update\_key(Current\_index, time[u] + time\_to\_neighbour);

🡺Complexity:o(log V`).

distance[Current\_index] = distance[u] + G.Nodes\_edges[u][j].Item2;

🡺Complexity:o(1).

parent[Current\_index] = u;

🡺Complexity:o(1).

}

}

🡺checking if this node u may be on the shortest path between source and destination

if (time[u] + time\_End[u] < res)

{

res = time[u] + time\_End[u];

🡺Complexity:o(1).

real\_end\_index = u;

🡺Complexity:o(1).

}

if (visted[u] == true) { break; }

🡺Complexity:o(1).

visted[u] = true;

🡺Complexity:o(1).

int u\_End = PQ\_End.extractMin();

🡺Complexity:o(log V`).

int num\_of\_neighbours\_of\_u\_End = G.Nodes\_edges[u\_End].Count;

🡺Complexity:o(1).

🡺for speed

for (int j = 0; j < num\_of\_neighbours\_of\_u\_End; j++)

{

🡺retrive index of adjecent (المجاورة) vertex of minimum node (u)

int Current\_index = G.Nodes\_edges[u\_End][j].Item1;

🡺Complexity:o(1).

🡺 time from u to one of the neighbours

double time\_to\_neighbour = G.Nodes\_edges[u\_End][j].Item2 / G.Nodes\_edges[u\_End][j].Item3;

🡺Complexity:o(1).

if (time\_End[u\_End] + time\_to\_neighbour < time\_End[Current\_index])

{

time\_End[Current\_index] = time\_End[u\_End] + time\_to\_neighbour;

🡺Complexity:o(1).

PQ\_End.heap\_update\_key(Current\_index, time\_End[u\_End] + time\_to\_neighbour);

🡺Complexity:o(log V`).

distance\_End[Current\_index] = distance\_End[u\_End] + G.Nodes\_edges[u\_End][j].Item2;

parent\_End[Current\_index] = u\_End;

🡺Complexity:o(1).

}

}

🡺checking if this node u\_End may be on the shortest path between source and destination

if (time[u\_End] + time\_End[u\_End] < res)

{

res = time[u\_End] + time\_End[u\_End];

🡺Complexity:o(1).

real\_end\_index = u\_End;

}

if (visted[u\_End] == true) { break; }

🡺Complexity:o(1).

visted[u\_End] = true;

🡺Complexity:o(1).

}

return real\_end\_index;

🡺Complexity:o(1).

}

◆🡻🡻The complexity of function(Write\_Out\_Put)=o(body)=o(V).

public void Write\_Out\_put (string File\_Name , bool flag )

{

if (flag)

{

StreamWriter obj = new StreamWriter(File\_Name + ".txt" , true);

🡺Complexity:o(1).

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V\*o(1)=o(V).

for (int j = 0; j < shortest\_path\_nodes.Count; j++)

{

obj.Write(shortest\_path\_nodes[j] + " ");

🡺Complexity:o(1).

}

obj.WriteLine();

🡺Complexity:o(1).

// write Total time

obj.WriteLine(totalTime + " mins");

🡺Complexity:o(1).

obj.WriteLine(totalDistance + " km");

🡺Complexity:o(1).

// write walkingDistance

obj.WriteLine(totalWalkingDistance + " km");

🡺Complexity:o(1).

obj.WriteLine(totalVehD + " km");

🡺Complexity:o(1).

obj.WriteLine();

🡺Complexity:o(1).

obj.Close();

🡺Complexity:o(1).

}

else

{

// output Same of file output

⍟◆🡻Complexity of loop =number of iterations \* o(body of loop)--🡪here=V\*o(1)=o(V).

for (int w = 0; w < shortest\_path\_nodes.Count; w++)

{

Console.Write(shortest\_path\_nodes[w] + " ");

🡺Complexity:o(1).

}

Console.WriteLine();

🡺Complexity:o(1).

// write Total time

Console.WriteLine(totalTime + " mins");

🡺Complexity:o(1).

Console.WriteLine(totalDistance + " km");

// write walkingDistance

🡺Complexity:o(1).

Console.WriteLine(totalWalkingDistance + " km");

🡺Complexity:o(1).

Console.WriteLine(totalVehD + " km");

🡺Complexity:o(1).

Console.WriteLine();

🡺Complexity:o(1).

return;

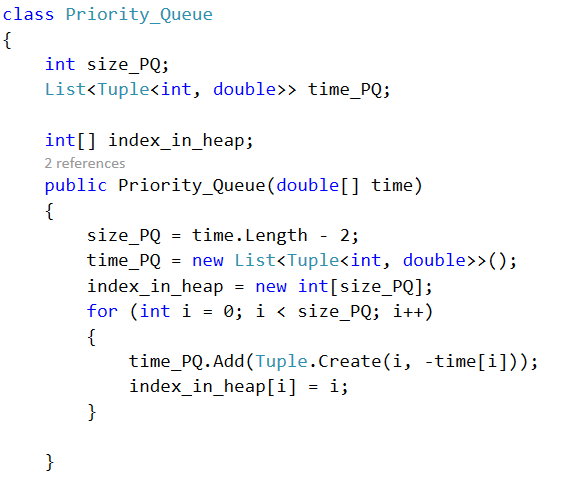
}

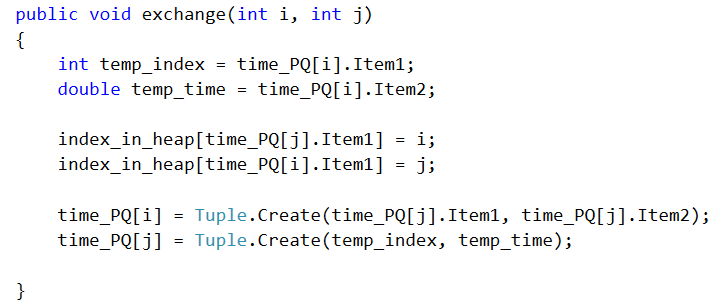
}

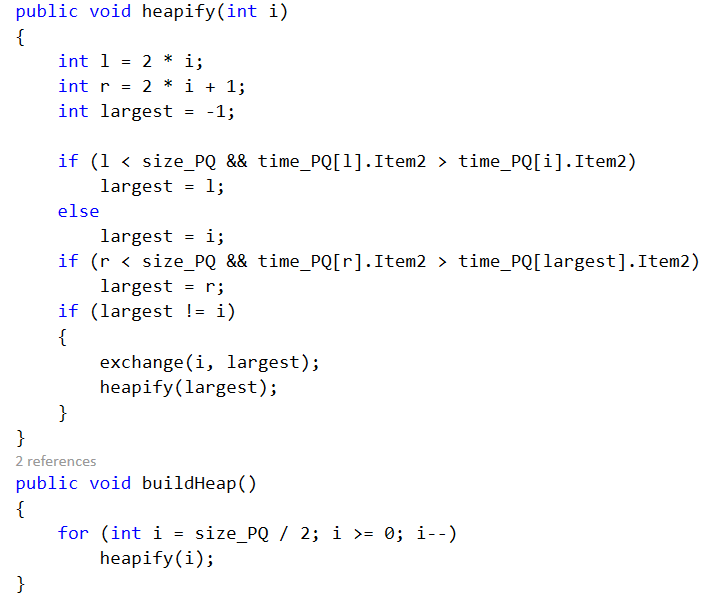
}

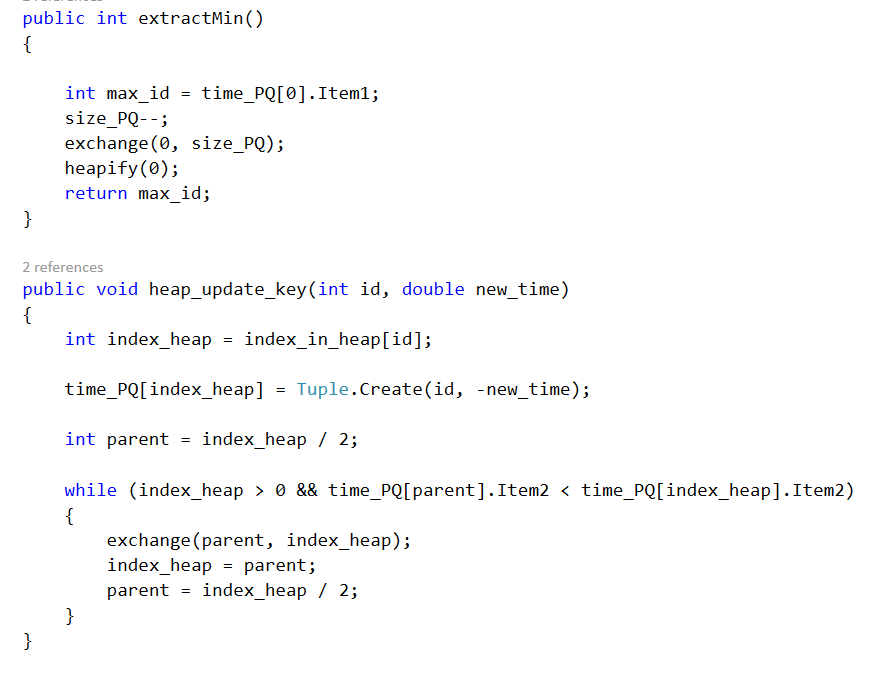
\*Now We Finish discussion of Class graph (\*\_\*)

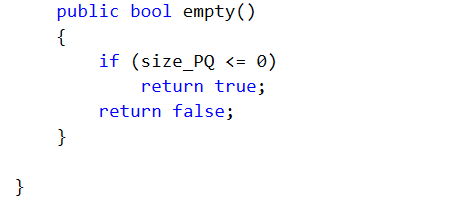
\*Let’s start with class Priority\_Queue 🡻











\*Here We Analysis Class Priority\_Queue Line by Line (\*\_\*).

🡻🡻🡻🡻

class Priority\_Queue

{

int size\_PQ;

List<Tuple<int, double>> time\_PQ;

🡺Complexity:o(1).

Dictionary<int, int> index\_in\_heap;

🡺Complexity:o(1).

public Priority\_Queue(double[] time)

{

size\_PQ = time.Length - 2;

🡺Complexity:o(1).

time\_PQ = new List<Tuple<int, double>>();

🡺Complexity:o(1).

index\_in\_heap = new Dictionary<int, int>();

🡺Complexity:o(1).

⍟◆🡾🡾Complexity of loop = number of iterations \* o(body)::here =size of queue\*o(1)=o(size of queue).

for (int i = 0; i < size\_PQ; i++)

{

time\_PQ.Add(Tuple.Create(i, -time[i]));

🡺Complexity:o(1).

index\_in\_heap[i] = i;

🡺Complexity:o(1).

}

}

◆🡺The complexity of function(exchange)=o(1).

public void exchange(int i, int j)

{

int temp\_index = time\_PQ[i].Item1;

🡺Complexity:o(1).

double temp\_time = time\_PQ[i].Item2;

🡺Complexity:o(1).

index\_in\_heap[time\_PQ[j].Item1] = i;

🡺Complexity:o(1).

index\_in\_heap[time\_PQ[i].Item1] = j;

🡺Complexity:o(1).

time\_PQ[i] = Tuple.Create(time\_PQ[j].Item1, time\_PQ[j].Item2);

🡺Complexity:o(1).

time\_PQ[j] = Tuple.Create(temp\_index, temp\_time);

🡺Complexity:o(1).

}

◆🡾The complexity of function(heapify)=o(log v).

public void heapify(int i)

{

int l = 2 \* i;

🡺Complexity:o(1).

int r = 2 \* i + 1;

🡺Complexity:o(1).

int largest = -1;

🡺Complexity:o(1).

🡾complexity of if &else condition=o(body)=o(1).

if (l < size\_PQ && time\_PQ[l].Item2 > time\_PQ[i].Item2)

largest = l;

🡺Complexity:o(1).

else

largest = i;

🡺Complexity:o(1).

🡾 complexity of if condition=o(body)=o(1).

if (r < size\_PQ && time\_PQ[r].Item2 > time\_PQ[largest].Item2)

largest = r;

🡺Complexity:o(1).

🡾 complexity of if condition=o(body)=o(1).

if (largest != i)

{

exchange(i, largest);

🡺Complexity:o(1).

heapify(largest);

🡺Complexity:o(log v).

}

}

◆🡾The complexity of function(buildHeap)=o(size of queue /2\*log (n)).

public void buildHeap()

{

⍟◆🡾🡾Complexity of loop = number of iterations \* o(body)::here =size of queue /2 \*o(log v).

for (int i = size\_PQ / 2; i >= 0; i--)

heapify(i);

🡺Complexity:o(log n).

}

◆🡾The complexity of function(extractMin)=o(log v).

public int extractMin()

{

int max\_id = time\_PQ[0].Item1;

🡺Complexity:o(1).

size\_PQ--;

🡺Complexity:o(1).

exchange(0, size\_PQ);

🡺Complexity:o(1).

heapify(0);

🡺Complexity:o(log v).

return max\_id;

🡺Complexity:o(1).

}

◆🡾The complexity of function(heap\_update\_key)=o(log v).

public void heap\_update\_key(int id, double new\_time)

{

int index\_heap = index\_in\_heap[id];

🡺Complexity:o(1).

time\_PQ[index\_heap] = Tuple.Create(id, -new\_time);

🡺Complexity:o(1).

int parent = index\_heap / 2;

🡺Complexity:o(1).

🡾complexity of while condition=#of iterations\*o(body)=o(log v).

while (index\_heap > 0 && time\_PQ[parent].Item2 < time\_PQ[index\_heap].Item2)

{

exchange(parent, index\_heap);

🡺Complexity:o(1).

index\_heap = parent;

🡺Complexity:o(1).

parent = index\_heap / 2;

🡺Complexity:o(1).

}

}

◆🡾The complexity of function(empty)=o(1).

public bool empty()

{

🡾complexity of if condition=o(body)=o(1).

if (size\_PQ <= 0)

return true;

🡺Complexity:o(1).

return false;

🡺Complexity:o(1).

}

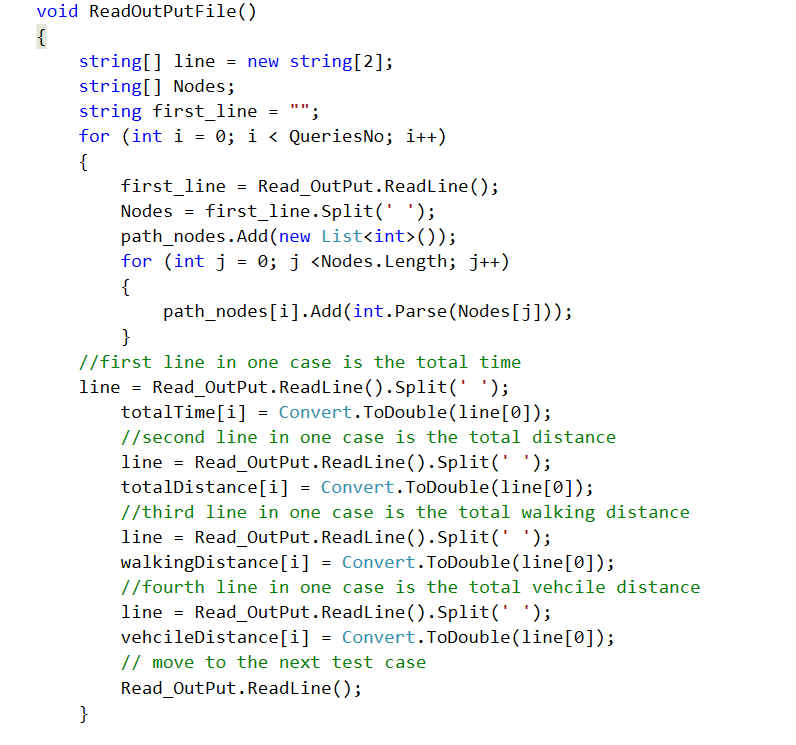
}

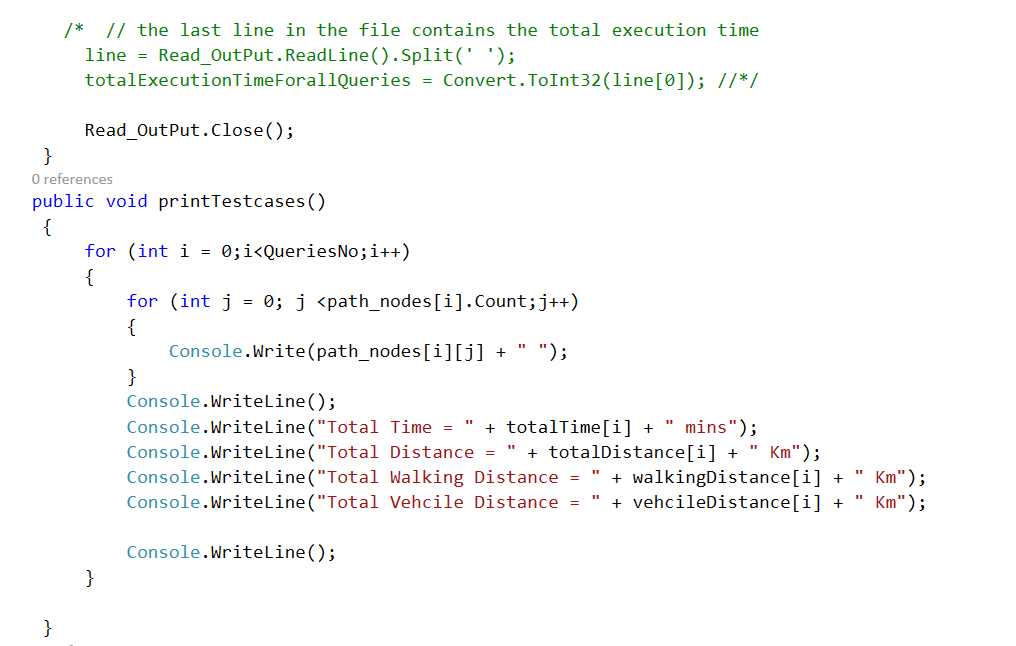
\*Now We Finish discussion of Class priority\_Queue(\*\_\*)

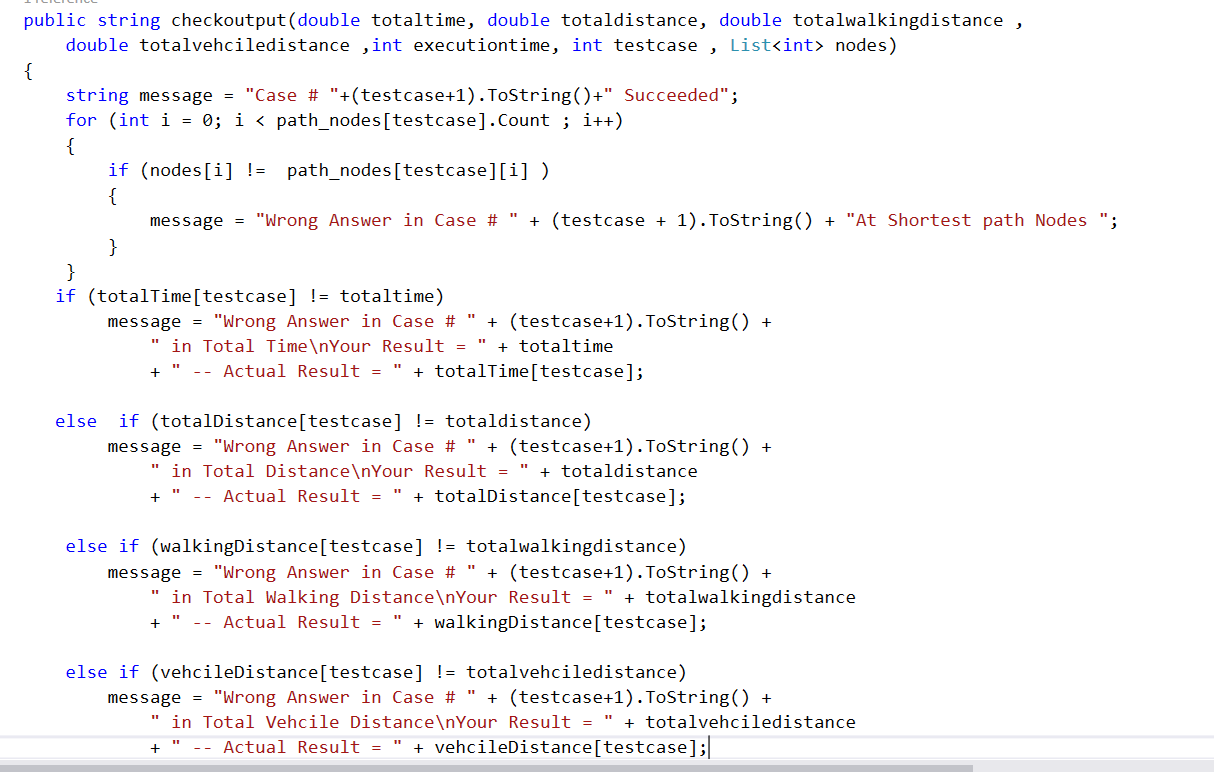
\*Let’s Start With Class CheckCorrectness(\*\_\*)

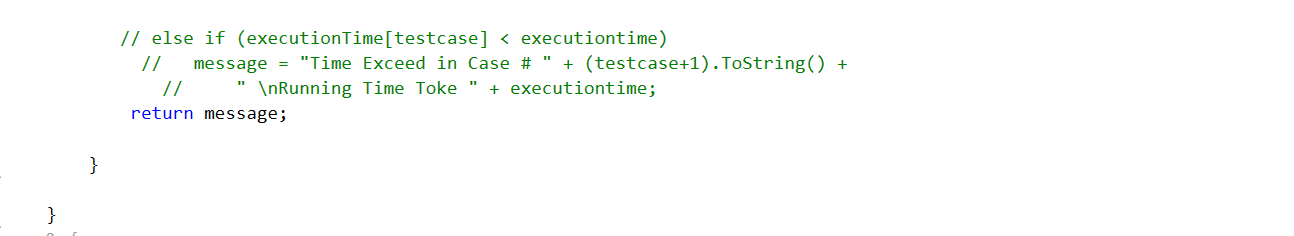


🡪complexity=O(V)



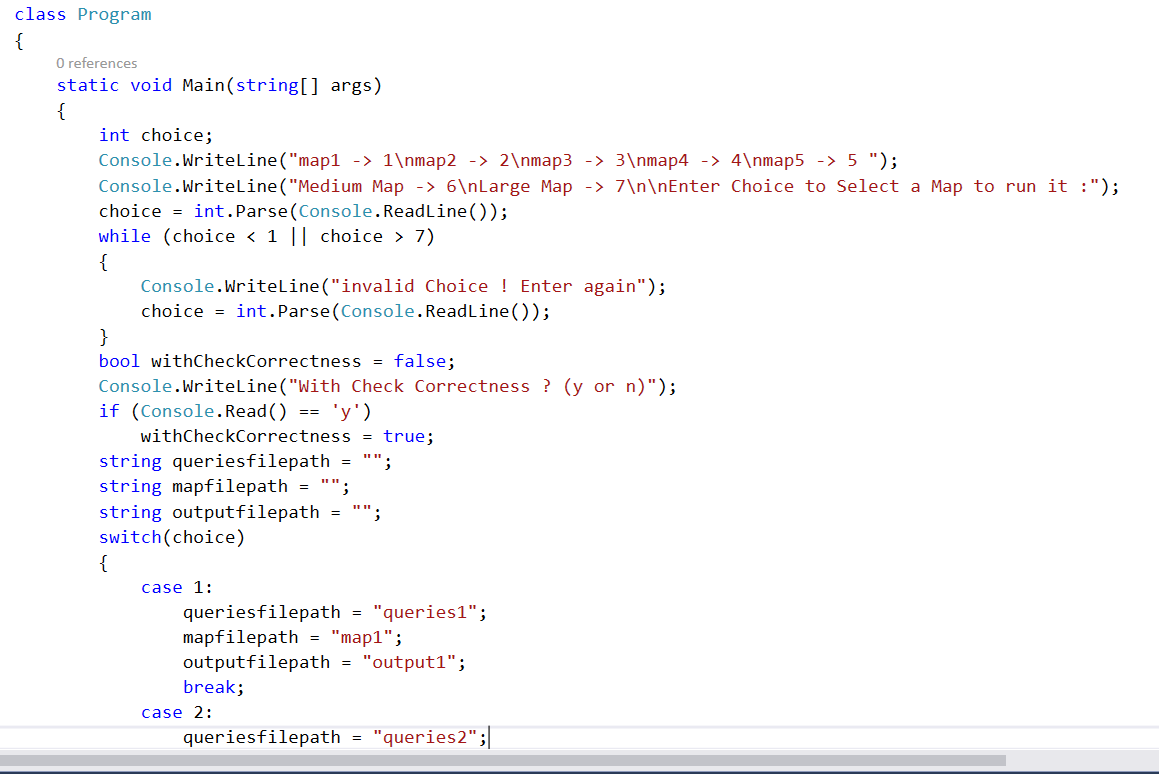




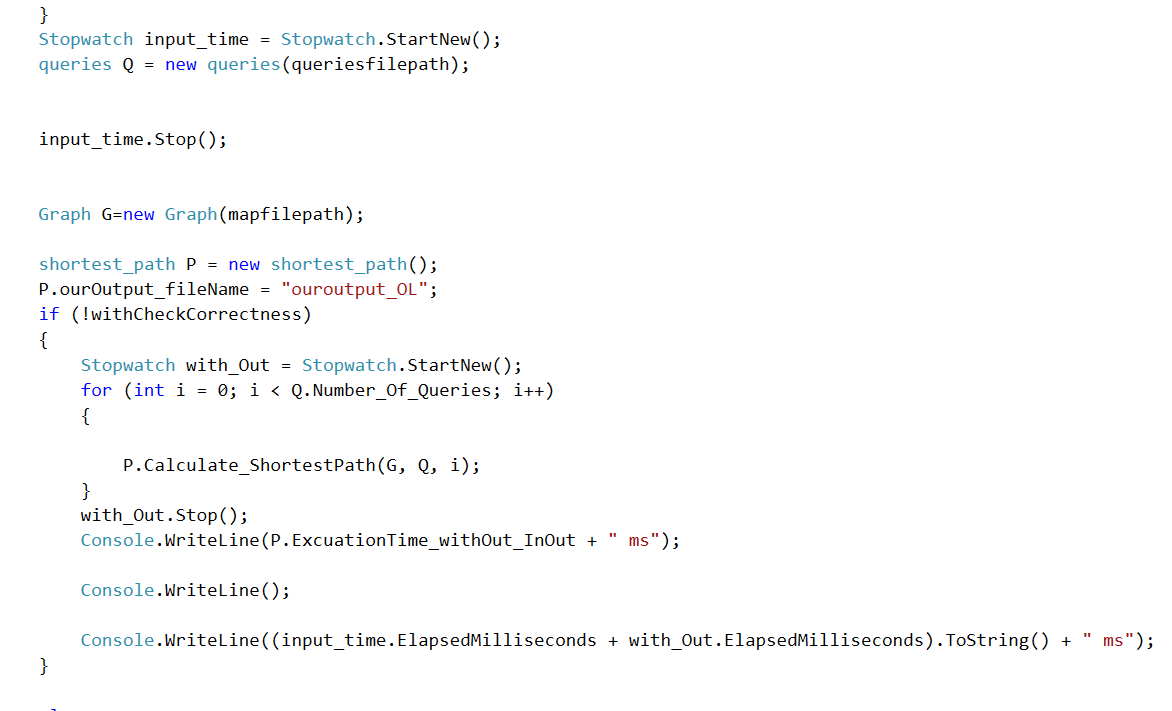


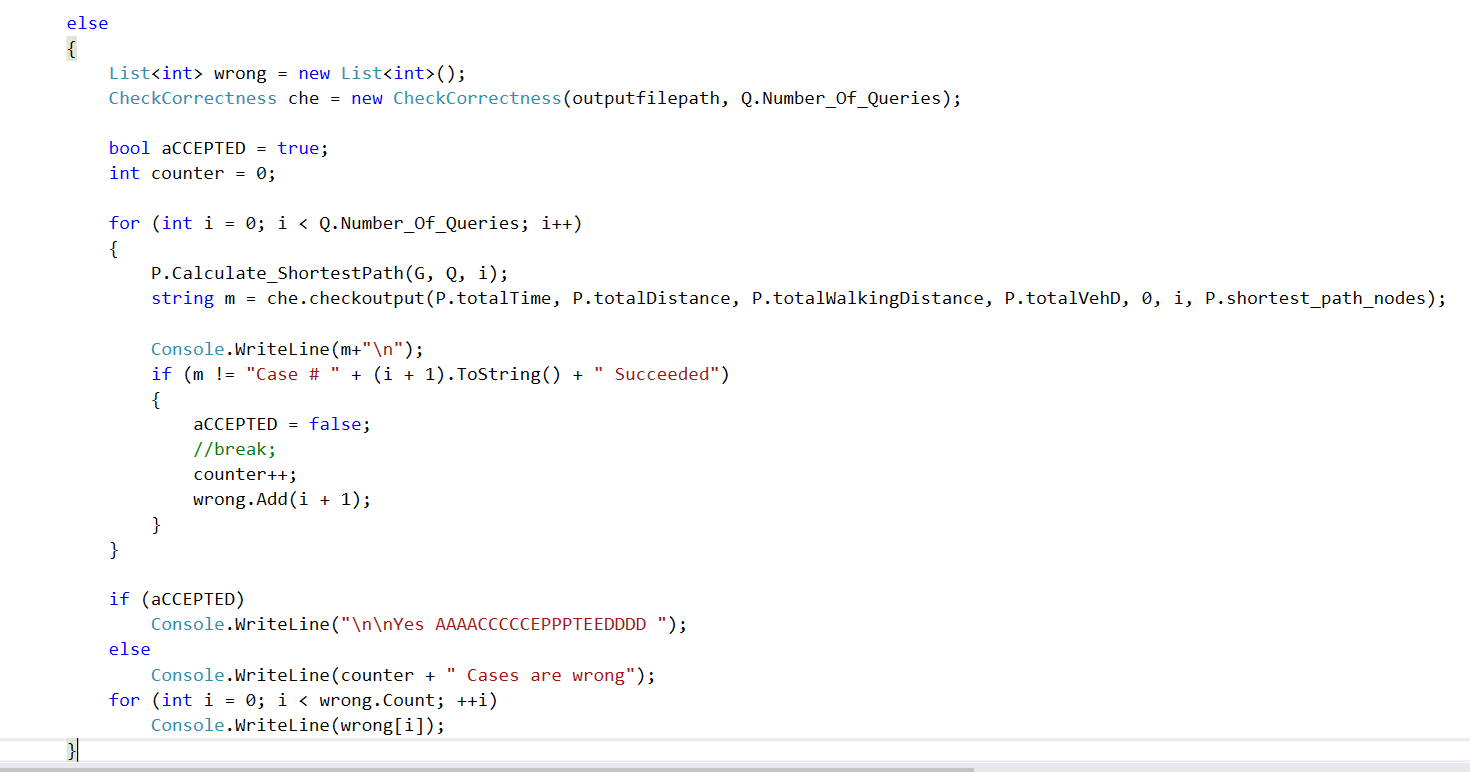
(\*\_\*)🡻🡻Analysis🡻🡻 (\*\_\*)

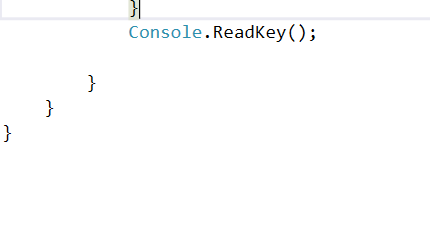
🡪complexity=O(Path\_ndes).











class Program

{

static void Main(string[] args)

{

int choice;

Console.WriteLine("map1 -> 1\nmap2 -> 2\nmap3 -> 3\nmap4 -> 4\nmap5 -> 5 ");

Console.WriteLine("Medium Map -> 6\nLarge Map -> 7\n\nEnter Choice to Select a Map to run it :");

choice = int.Parse(Console.ReadLine());

while (choice < 1 || choice > 7)

{

Console.WriteLine("invalid Choice ! Enter again");

choice = int.Parse(Console.ReadLine());

}

bool withCheckCorrectness = false;

Console.WriteLine("With Check Correctness ? (y or n)");

if (Console.Read() == 'y')

withCheckCorrectness = true;

string queriesfilepath = "";

string mapfilepath = "";

string outputfilepath = "";

switch(choice)

{

🡺Complexity:o(1).

case 1:

queriesfilepath = "queries1";

mapfilepath = "map1";

outputfilepath = "output1";

break;

case 2:

queriesfilepath = "queries2";

mapfilepath = "map2";

outputfilepath = "output2";

break;

case 3:

queriesfilepath = "queries3";

mapfilepath = "map3";

outputfilepath = "output3";

break;

case 4:

queriesfilepath = "queries4";

mapfilepath = "map4";

outputfilepath = "output4";

break;

case 5:

queriesfilepath = "queries5";

mapfilepath = "map5";

outputfilepath = "output5";

break;

case 6:

queriesfilepath = "OLQueries";

mapfilepath = "OLMap";

outputfilepath = "OLOutput";

break;

case 7:

queriesfilepath = "SFQueries";

mapfilepath = "SFMap";

outputfilepath = "SFOutput";

break;

}

Stopwatch input\_time = Stopwatch.StartNew();

queries Q = new queries(queriesfilepath);

input\_time.Stop();

Graph G=new Graph(mapfilepath);

shortest\_path P = new shortest\_path();

P.ourOutput\_fileName = "ouroutput\_OL";

if (!withCheckCorrectness)

{

Stopwatch with\_Out = Stopwatch.StartNew();

⍟◆🡾🡾Complexity of loop = number of iterations \* o(body)::here =o(E`log V`).

for (int i = 0; i < Q.Number\_Of\_Queries; i++)

{

P.Calculate\_ShortestPath(G, Q, i);

}

with\_Out.Stop();

Console.WriteLine(P.ExcuationTime\_withOut\_InOut + " ms");

Console.WriteLine();

Console.WriteLine((input\_time.ElapsedMilliseconds + with\_Out.ElapsedMilliseconds).ToString() + " ms");

}

🡺Complexity:o(1).

else

{

List<int> wrong = new List<int>();

CheckCorrectness che = new CheckCorrectness(outputfilepath, Q.Number\_Of\_Queries);

bool aCCEPTED = true;

int counter = 0;

for (int i = 0; i < Q.Number\_Of\_Queries; i++)

{

P.Calculate\_ShortestPath(G, Q, i);

🡺Complexity:o(E`log V`).

string m = che.checkoutput(P.totalTime, P.totalDistance, P.totalWalkingDistance, P.totalVehD, 0, i, P.shortest\_path\_nodes);

🡺Complexity:o(1).

Console.WriteLine(m+"\n");

if (m != "Case # " + (i + 1).ToString() + " Succeeded")

{

aCCEPTED = false;

//break;

counter++;

🡺Complexity:o(1).

wrong.Add(i + 1);

🡺Complexity:o(1).

}

}

if (aCCEPTED)

Console.WriteLine("\n\nYes AAAACCCCCEPPPTEEDDDD ");

🡺Complexity:o(1).

else

Console.WriteLine(counter + " Cases are wrong");

🡺Complexity:o(1).

for (int i = 0; i < wrong.Count; ++i)

Console.WriteLine(wrong[i]);

🡺Complexity:o(1).

}

Console.ReadKey();

🡺Complexity:o(1).

}

}

}

(\*\_\*)Thank\_You(\*\_\*)