**FCIS Map**

* Ahmed Hassan Ezz El Din.
* Haneen Tamer .
* Omar Hossam.
* Reem Osama Mohamed Marie.
* Tasneem Hani Shafiek.



class GraphOperations

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using UnityEngine;

public class GraphOperations

{

private static void printVertex(Vertex v)

{

Debug.Log("Vertex " + v.getData()+ " ");//Ꝋ(1)

}

private static void printEdge(Edge e)

{

Debug.Log(" edge " + e.getAdj().getData() + " of weight "+ e.getWeight());//Ꝋ(1)

}

public static void Dijsktra(Graph g, Dictionary<Vertex, VertexColour> vertices, Vertex src, Vertex dest)// O(E LogV)

{

PriorityQueue<Vertex, int> Q = new PriorityQueue<Vertex, int>();

foreach (Vertex v in g.getAllVertices())//O(V LogN)

{

vertices[v] = new VertexColour();//Ꝋ(1)

if (v.Equals(src))//Ꝋ(1)

{

vertices[v].color = VertexColour.black;//Ꝋ(1)

vertices[v].dist = 0;//Ꝋ(1)

}

Q.Enqueue(v, vertices[v].dist);//O(LogN)

}

while (Q.Count() != 0)//O(E LogV)

{

//Touch every vertex only once : O(LogV)

Vertex u = Q.Dequeue();//Ꝋ(1)

vertices[u].color = VertexColour.black;//Ꝋ(1)

if (u.Equals(dest)) return;//Ꝋ(1)

foreach (Edge e in g.getNeighbours(u)) //Ꝋ(E) " Maximum Iterations "

{

Vertex v = e.getAdj();//Ꝋ(1)

int distToTry = vertices[u].dist + e.getWeight();//Ꝋ(1)

if (vertices[v].color.Equals(VertexColour.white)//Ꝋ(1)

&& vertices[v].dist > distToTry)

{

Q.update\_value(v, distToTry);//Ꝋ(1)

vertices[v].dist = distToTry;//Ꝋ(1)

vertices[v].parent = u;//Ꝋ(1)

}

}

}

}

public static void printDijsktra(Dictionary<Vertex, VertexColour> vertices, Vertex v,List<Vertex> x)//O(V)

{

if (vertices[v].parent != null) printDijsktra(vertices, vertices[v].parent,x);//O(V)

x.Add(v); //Ꝋ(1)

}

public static List<Vertex> printShortestPath(Graph g, Vertex src, Vertex dest) //O(E LogV)

{

List<Vertex> temp= new List<Vertex>(); //Ꝋ(1)

Dictionary<Vertex, VertexColour> vertices = new Dictionary<Vertex, VertexColour>(); //Ꝋ(1)

Dijsktra(g, vertices, src, dest); // O( E LogV)

Debug.Log("Shortest Path from " + src.getData() + " to " + dest.getData() + " costs " + vertices[dest].dist); //Ꝋ(1)

printDijsktra(vertices, dest,temp);//O(V)

return temp; //Ꝋ(1)

}

public static void BellmanFord(Graph graph, Vertex source, Vertex Destination)//O(V\*E)

{

int verticesCount = graph.numVertices(); //theta(1)

int edgesCount = graph.numEdges(); //theta(1)

Dictionary<Vertex, int> distance = new Dictionary<Vertex, int>(); //theta(1)

List<Vertex> AllVertices = graph.getAllVertices(); //theta(1)

Dictionary<Vertex, Vertex> Parent = new Dictionary<Vertex, Vertex>(); //theta(1)

List<Edge> temp = new List<Edge>(); //theta(1)

foreach (Vertex u in graph.getAllVertices()) // #iteration \*body = v \* neighbors = O(V \* E)

{

foreach (Edge e in graph.getNeighbours(u)) //#iteration \*body= E "neighbors" \* theta(1) = O(neighbors)

{

e.Src = u; // theta(1)

temp.Add(e);// theta(1)

}

}

foreach (Vertex x in graph.getAllVertices()) // O(V)

{

distance[x] = int.MaxValue; // theta(1)

}

distance[source] = 0; //theta(1)

Parent[source] = null; //theta(1)

for (int i = 0; i < verticesCount; ++i) // #iteration \* body = V \* E = O(V\*E)

{

foreach (Edge e in temp) //#iteration \* body =E \* theta(1)=O(E)

{

Vertex u = e.Src; //theta(1)

Vertex v = e.getAdj(); //theta(1)

int weight = e.getWeight(); //theta(1)

if (distance[u] != int.MaxValue && distance[u] + weight < distance[v]) //theta(1)

{

distance[v] = distance[u] + weight; //theta(1)

Parent[v] = u; //theta(1)

}

}

}

printPath(Parent, Destination); //O(V)

GraphManager.TotalCost = distance[Destination]; //theta(1)

}

private static void printPath(Dictionary<Vertex,Vertex> parent, Vertex v)//O(V)

{

if (parent[v] != null)

printPath(parent, parent[v]);//O(V)

GraphManager.ShortestPath.Add(v);//theta(1)

}

public static List<Vertex> BfsShotestPath(Graph g,Vertex source, Vertex destination)//O(V+E)

{

Queue<Vertex> Q = new Queue<Vertex>();//theta(1)

Q.Enqueue(source);//theta(1)

Vertex currentOne;//theta(1)

while (Q.Count() != 0)//theta(V)

{

currentOne = Q.Dequeue(); //theta(N)

foreach (Edge e in g.getNeighbours(currentOne))//theta(Neighbours)

{

Vertex v = e.getAdj();//theta(1)

int weight = e.getWeight();//theta(1)

if (v.state == "unVisited")//theta(1)

{

v.state = "Visited";//theta(1)

v.distance = currentOne.distance + weight;//theta(1)

v.parent = currentOne;//theta(1)

Q.Enqueue(v);//theta(1)

}

else if (v.state == "Visited")//theta(1)

{

if (v.distance > currentOne.distance + weight)//theta(1)

{

v.distance = currentOne.distance + weight;//theta(1)

v.parent = currentOne;//theta(1)

}

}

}

currentOne.state = "Finished";//theta(1)

}

GraphManager.TotalCost = destination.distance;//theta(1)

return backTrack(destination, source);//O(V)

}

private static List<Vertex> backTrack(Vertex destination, Vertex source)//O(V)

{

List<Vertex> path = new List<Vertex>();//theta(1)

Vertex currenOne = destination;//theta(1)

while (currenOne.parent != null)//O(V)

{

GraphManager.ShortestPath.Add(currenOne);//theta(1)

currenOne = currenOne.parent;//theta(1)

}

GraphManager.ShortestPath.Add(currenOne);//theta(1)

return path;//theta(1)

}

public static void FloydWarshall(Graph graph, Vertex Source, Vertex Destination) // O(v^3)

{

List<Vertex> temp = graph.getAllVertices();// O(1)

List<Edge> AllEdges = new List<Edge>();// O(1)

List<List<int>> initDistances = new List<List<int>>();// O(1)

foreach (Vertex u in graph.getAllVertices())// O(V\*E)

{

foreach (Edge e in graph.getNeighbours(u))// O(E)

{

e.Src = u;// O(1)

AllEdges.Add(e);// O(1)

}

}

for (int i = 0; i < graph.getAllVertices().Count; i++)// O(V^2)

{

initDistances.Add(new List<int>());// O(1)

for (int j = 0; j < graph.getAllVertices().Count; j++)// O(V)

{

initDistances[i].Add(999);// O(1)

}

}

for (int i = 0; i < graph.getAllVertices().Count; i++)// O(V\*E)

{

foreach (Edge e in AllEdges)// O(E)

{

if (e.getWeight() < initDistances[temp.IndexOf(e.Src)][temp.IndexOf(e.getAdj())])// O(1)

initDistances[temp.IndexOf(e.Src)][temp.IndexOf(e.getAdj())] = e.getWeight();// O(1)

}

}

int[,] distance = new int[graph.numVertices(), graph.numVertices()];// O(1)

int[,] path = new int[graph.numVertices(), graph.numVertices()];// O(1)

for (int i = 0; i < graph.numVertices(); i++)// O(V^2)

{

for (int j = 0; j < graph.numVertices(); j++) // O(V)

{

distance[i, j] = initDistances[i][j];// O(1)

path[i, j] = j;// O(1)

}

}

for (int v = 0; v < graph.numVertices(); v++)// O(V)

{

distance[v, v] = 0;// O(1)

path[v, v] = v;// O(1)

}

for (int k = 0; k < graph.numVertices(); k++)// theta(V^3)

{

for (int i = 0; i < graph.numVertices(); i++)// O(V^2)

{

for (int j = 0; j < graph.numVertices(); j++)// O(V)

{

if (distance[i, k] + distance[k, j] < distance[i, j])// O(1)

{

distance[i, j] = distance[i, k] + distance[k, j];// O(1)

path[i, j] = path[i, k];// O(1)

}

}

}

}

List<int> SP = new List<int>();// O(1)

SP = GetPath(temp.IndexOf(Source), temp.IndexOf(Destination), path);// O(V)

for (int i = 0; i < SP.Count; i++)// O(Shortest Path Length)

{

GraphManager.ShortestPath.Add(graph.getAllVertices()[SP[i]]);// O(1)

}

GraphManager.TotalCost = distance[temp.IndexOf(Source), temp.IndexOf(Destination)];

//Printfloyd(distance, graph.numVertices(),graph);

}

public static List<int> GetPath(int u, int v, int[,] path)// O(V)

{

List<int> temp = new List<int>();// O(1)

temp.Add(u);// O(1)

while (u != v) // O(V)

{

u = path[u, v];// O(1)

temp.Add(u);// O(1)

}

return temp;// O(1)

}

}

class Edge

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using UnityEngine;

public class Edge

{

private int weight;

private Vertex dest;

public Edge( Vertex dest, int w)

{

this.dest = dest;//Ꝋ(1)

this.weight = w;//Ꝋ(1)

}

public override bool Equals(System.Object obj)

{

if (!(obj is Edge)) return false;//Ꝋ(1)

Edge e = (Edge)obj;//Ꝋ(1)

return dest.Equals(e.dest);//Ꝋ(1)

}

public override int GetHashCode()

{

return dest.GetHashCode();//Ꝋ(1)

}

public Vertex getAdj() { return this.dest; }//Ꝋ(1)

public int getWeight() { return this.weight; }//Ꝋ(1)

}

class Vertex

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using UnityEngine;

public class Vertex : IComparable

{

private String data;

public GameObject NodeObject;

public int SceneNumber;

public int NodeIndex;

public Vertex(String data)

{

this.data = data;//Ꝋ(1)

}

public override bool Equals(System.Object obj)

{

if (!(obj is Vertex)) return false;//Ꝋ(1)

Vertex v = (Vertex)obj;//Ꝋ(1)

return data.Equals(v.data);//Ꝋ(1)

}

public override int GetHashCode()

{

return data.GetHashCode();//Ꝋ(1)

}

public String getData()

{

return this.data;//Ꝋ(1)

}

public int CompareTo(object obj)

{

return 0;//Ꝋ(1)

}

}

class Graph

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using UnityEngine;

public class Graph

{

int v;

int e;

Dictionary<Vertex, HashSet<Edge>> dict;

public Graph()

{

e = 0;//Ꝋ(1)

v = 0;//Ꝋ(1)

dict = new Dictionary<Vertex, HashSet<Edge>>();//Ꝋ(1)

}

public void insertVertex(Vertex v)

{

dict[v] = new HashSet<Edge>();//Ꝋ(1)

this.v += 1;//Ꝋ(1)

}

public void insertEdge(Vertex v1, Vertex v2, int w)

{

dict[v1].Add(new Edge(v2, w));//Ꝋ(1)

dict[v2].Add(new Edge(v1, w));//Ꝋ(1)

this.e += 1;//Ꝋ(1)

}

public bool areAdjacent(Vertex v1, Vertex v2)

{

return dict[v1].Contains(new Edge(v2, 0));//O(E)

}

public void removeEdge(Vertex v1, Vertex v2)

{

dict[v1].Remove(new Edge(v2, 0));//Ꝋ(1)

dict[v2].Remove(new Edge(v1, 0));//Ꝋ(1)

this.e = this.e - 1;//Ꝋ(1)

}

public void removeVertex(Vertex v)//Total = #Iterations \* Ꝋ(1) = Ꝋ(Neighbours)

{

foreach (Edge e in dict[v])

{

Vertex n = e.getAdj();//Ꝋ(1)

removeEdge(n, v);//Ꝋ(1)

}

dict.Remove(v);//Ꝋ(1)

this.v = this.v - 1;//Ꝋ(1)

}

public HashSet<Edge> getNeighbours(Vertex v1)

{

return new HashSet<Edge>(dict[v1]);//Ꝋ(1)

}

public List<Vertex> getAllVertices()

{

return new List<Vertex>(dict.Keys);//O(Keys)

}

public int numVertices()

{

return v;//Ꝋ(1)

}

public int numEdges()

{

return e;//Ꝋ(1)

}

}

class VertexColour

public class VertexColour

{

public static string white = "White";

public static string grey = "Grey";

public static string black = "Black";

public string color;

public int dist;

public Vertex parent;

public VertexColour()

{

color = VertexColour.white;//Ꝋ(1)

dist = int.MaxValue;//Ꝋ(1)

parent = null;//Ꝋ(1)

}

}

class PriorityQueue

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using UnityEngine;

public class PriorityQueue<T, Int>

{

private List<KeyValuePair<T, int>> data;

private Dictionary<T, int> positions;

private int getParentIndex(int i) { return (i / 2); } //Ꝋ(1)

private int getLeftChildIndex(int i) { return 2 \* i; }//Ꝋ(1)

private int getRightChildIndex(int i) { return (2 \* i) + 1; }//Ꝋ(1)

private void swap(int indx1, int indx2)//Total = Ꝋ(1)

{

KeyValuePair<T, int> tmp = data[indx1];//Ꝋ(1)

data[indx1] = data[indx2];//Ꝋ(1)

positions[data[indx1].Key] = indx1;//Ꝋ(1)

data[indx2] = tmp;//Ꝋ(1)

positions[data[indx2].Key] = indx2;//Ꝋ(1)

}

private void min\_heapify(int i)// Total = O(LogN)

{

int left = getLeftChildIndex(i);//Ꝋ(1)

int right = getRightChildIndex(i);//Ꝋ(1)

int min = i;//Ꝋ(1)

if (left < data.Count//Ꝋ(1)

&& data[left].Value < data[i].Value)

{

min = left;//Ꝋ(1)

}

if (right < data.Count//Ꝋ(1)

&& data[right].Value < data[min].Value)

{

min = right;//Ꝋ(1)

}

if (min != i)//Ꝋ(1)

{

swap(i, min);//Ꝋ(1)

min\_heapify(min);//O(LogN)

}

}

public PriorityQueue()

{

this.data = new List<KeyValuePair<T, int>>();//Ꝋ(1)

this.positions = new Dictionary<T, int>();//Ꝋ(1) }

public PriorityQueue(int n)

{

this.data = new List<KeyValuePair<T, int>>(n);//Ꝋ(N)

this.positions = new Dictionary<T, int>(n);//Ꝋ(N)

}

public int Count()

{

return data.Count - 1;//Ꝋ(1)

}

public void Enqueue(T item, int weight)//Total = O(LogN)

{

if (data.Count == 0) data.Add(new KeyValuePair<T, int>());//Ꝋ(1)

KeyValuePair<T, int> pair = new KeyValuePair<T, int>(item, int.MaxValue);//Ꝋ(1)

data.Add(pair);//Ꝋ(1)

positions[item] = data.Count - 1;//Ꝋ(1)

decrease\_value(data.Count - 1, weight);//O(LogN)

}

public T Dequeue()//Total = O(LogN)

{

T min = data[1].Key;//Ꝋ(1)

data[1] = data[data.Count - 1];//Ꝋ(1)

positions[data[1].Key] = 1;//Ꝋ(1)

data.RemoveAt(data.Count - 1);//Ꝋ(1)

positions.Remove(min);//Ꝋ(1)

min\_heapify(1);//O(LogN)

return min;//Ꝋ(1)

}

public void update\_value(T item, int weight)//O(LogN)

{

int pos = positions[item]; //Ꝋ(1)

decrease\_value(pos, weight);//O(LogN)

}

private void decrease\_value(int indx, int weight)

{

data[indx] = new KeyValuePair<T, int>(data[indx].Key, weight);//Ꝋ(1)

int parent = getParentIndex(indx);//Ꝋ(1)

while (indx > 1 && data[parent].Value > data[indx].Value)// Total = #Iterations\*Ꝋ(1) = O(LogN)

{

swap(indx, parent);//Ꝋ(1)

indx = getParentIndex(indx);//Ꝋ(1)

parent = getParentIndex(indx);//Ꝋ(1)

}

}

}

**Comparison between Algorithms**

|  |  |  |
| --- | --- | --- |
|  | Dijkstra | BFS |
| complexity | O(E\*Log(v)) | O(V+E) |
| Functionality | gets shortest path from one node to all nodes , considered greedy | traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbor nodes (nodes which are directly connected to source node). |

|  |  |  |
| --- | --- | --- |
|  | BellMan Ford | Floyd Warshall |
| complexity | O(V\*E) | O(V^3) |
| Functionality | gets shortest path from one node to all other nodes, contains negative edge weights but doesn’t contain negative cycles. Not considered greedy | gets shortest path between all pairs of vertices, negative edges allowed |