**DATA STRUCTURES 4A PROJESİ**

AHMET GÜL – BİRGÜL ÖZLEM

05110000002 - 05110000004

**KULLANILAN DİL VE PLATFORM:** C# - Visual Studio 2010

**PROBLEMİN KISA TANIMI:** Bir graph yapısının oluşturulması ve oluşturulan bu yapı üzerinde Dijkstra, MST, BFS gibi algoritmaların uygulanarak; shortest path, en küçük kapsayan ağacın bulunması ve genişlik öncelikli dolaşılmasıdır.

**YAZILIM GELİŞTİRME İÇİN HARCANAN SÜRE:** Ahmet Gül – 40 saat

Birgül Özlem – 40 saat

**KULLANILAN SINIFLAR ve İÇLERİNDEKİ METOTLAR**

class PriorityQ : Minimum kapsayan ağaç bulunurken kullanılacak olan öncelik kuyruğunun yapıldığı classtır içindeki metotlar aşağıda açıklanmıştır.

public PriorityQ() : Bu metot yapıcı metot olup , kenar cinsinden oluşturulacak olan dizinin max. boyut kadar büyüklükte oluşturulduğu yerdir.Ayrıca size değişkene ilk değer olarak sıfır atanır.

public void insert(Edge item) : Bu metot kenar cinsinden oluşturulan diziye parametre olarak gönderilen kenar tipindeki değişkenin eklendiği yerdir.Ekleme işlemi uygulandıktan sonra size değişkeni +1 arttırılarak dizinin boyutu da tutulmuş olur.

public Edge removeMin() : Bu metot Edge tipindeki dizinin en dışta duran elemanını yani en küçük elemanı döndürür.

public void removeN(int n): Bu metot parametre olarak aldığı n sayısına tekamül eden sıradaki elemanı siler. Silme işlemi yaparken eleman sırasını da düzeltir.

public Edge peekMin() : Bu metot min. olan elemanı döndürür.

public int getSize() : Bu metot size değişkenini yani şuana kadar dizide olan toplam eleman sayısını döndürür.

public bool isEmpty() : Bu metot bool tipinde dizinin boş olup olmadığını kontrol eder.Eğer boşsa true değerini gönderir.Dizinin boş olması size ‘ın 0 olması demektir.

public Edge peekN(int n) : Bu metot n sayısını parametre olarak alır ve bu n sayısının olduğu sıradaki elemanı döndürür.

public int find(int findDex) : Belirtilen verinin kuyrukta olup olmadığını bulan metottur.

class Edge : Bu class köşeleri tutmamızı sağlar. İçinde bu köşenin uzunluğu , bu köşenin başlangıç tepesinin indeksini ve bu köşenin bitiş tepesinin indeksini tutar.

public Edge(int sv, int dv, int d) : Yapıcı metottur ve parametre olarak aldığı değerleri(köşenin başlangıç tepesinin indeksi, köşenin bitiş tepesinin indeksi ve köşenin uzunluğunu ) atar.

class Edge\_other : String şeklinde değişkenleri olan ve köşeleri tutmakla görevli classtır.

public Edge\_other(string vertex1, string vertex2,int distance) : Yapıcı metottur.Kenarın uzunluğunu , başlanıgıç ve bitiş tepelerini parametre olarak alarak , ilkleme işlemi yapar.

class Vertex : Bu class tepeleri tutmamıza ve bu tepeleri görüntülememizi sağlar.

public Vertex(string lab) : Yapıcı metot olup , parametre olarak aldığı stringi harf adına atar. Böylelikle bu köşenin bir adı olmuş olur.

public void display() : Bu metot tepenin adını ekrana yazdırır.

class Dijkstra : Bu class Dijkstra algoritmasını çözmek için oluşturulmuştur ve içinde graph matrisi , dijkstranın çözümü ve genişlik öncelikli dolaşmak için metotlar bulunur.

public Dijkstra(int boyut, int[,] pArray, int a, int b) : Yapıcı metottur. Parametre olarak aldığı sayı boyutunda uzunlukları tutan bir matris oluşturur ve yine bu boyut büyüklüğünde Nodes dizisini oluşturur.Ayrıca parametre olarak alınan kullanıcı tarafından main’de girilmiş olan hangi tepeden hangi tepe arasındaki sonuçların görüntülenmek istediğini belirten ilk ve son değişkenlerine başlangıç ve hedef indeksini atar.Graph matrisi üzerinde işlem yapılacağı için bu matrisi distanceMat ‘ına kopyalar.Daha sonra dolaşılacak olan tepeler için Nodes dizisinde her birine index değerine göre dolaşılmadığı için -1 atadık.Bu işlemden sonra kullanıcının istediği düğüm gezileceği için ona -1 değerini atanır ve böylelikle oradan geçilmiş olduğu anlaşılır. Son olarak ise D dizisine “ilk” tepesinin kullanıcının istediği tepenin diğer tüm tepelere doğrudan olan uzaklıkları atanır.

public void DijkstraSolving(Queue que, int[] min\_bulan\_dizi, int current\_tour) : Bu metot Dijkstra algoritmasının işletildiği metottur.Bu metot parametre olarak Kuyruk tipinde que ‘yu alır ve öncelikli olarak D dizisindeki en küçük elemanı bulunur ve bu elemanın indeksi minNode’a atanır.Daha sonra minNode değişkeni Nodes düğümünde indeks olarak kullanılarak o tepeden geçildiği işaretlenir.Bu işlemden sonra D ‘nin minNode’daki değeri ve bu değerin komşu tepelere olan uzaklıkları toplamı D dizisindeki değerlerden küçükse daha küçük yol bulunacağı için D’nin o indeksine en küçük uzunluk atanıyor.en\_kucuk dizisine minNode değerinin bir fazlası atanır.Bu işlemlerden sonra en\_kucuk dizisindeki son indeksinden 1 önceki indeksteki değer ki bu minNode değeri oluyor que kuyruğa ekleniyor.

public void run(Queue que, int[] min\_bulan\_dizi) : Bu metot Dijkstra’nın çözümünü ekrana adım adım yazdırmak için kullanılıyor.Matrisimizdeki boyut kadar for döngüsü ile dönerek DijkstraSolving metodu boyut kadar çağırılarak bütün uzunluk olasılıklarının hesaplanmasını sağlıyor.

public void bfs(Vertex[] vertex\_list, int start) : Queue tipinde kuyruk tanımlayarak başlıyoruz. Kontrol[] dizisinde de geçtiğimiz düğümleri belirtmek amacıyla ilgili indeksi true yapıyoruz. Başlayacak olduğumuz düğümü görüntületip while döngüsüne giriyoruz ve kuyruk tamamen boşalana kadar çıkmıyoruz. Sıradaki düğümü kuyruktan çıkartıp sonrasında gelmesi gereken düğüm için tekrar while ile işaretlenmemiş ve sırası gelmiş düğümü bulana kadar dönüyoruz. Bu döngüde doğru düğümü kuyruğa ekliyoruz. Her iki while döngüsü de bittiğinde kontrol dizimizi default haline geri döndürmeyi de unutmuyoruz.

public int getAdjUnvisitedVertex(int v): Bu metot daha önce işaretlenmemiş olan düğümleri tespit etmemi: Bu metot daha önce işaretlenmemiş olan düğümleri tespit etmemize yarıyor.

class Graph : Bu class en küçük kapsayan ağacı bulmak içindir ve bu classın içinde graph oluşturulur , en küçük kapsayan ağaç bu graph için bulunur.

# public Graph() : Yapıcı metottur.İçinde tepelerden oluşan bir dizi , ve aynı boyutta bir [adjacency](https://eksisozluk.com/adjacency-matrix--1687366) matris oluşturulur.Bu matrisin elemanlarına sonsuz değeri atanır.Daha sonra PriorityQ classı üzerinden thePQ adlı Queue oluşturulur.

public void addVertex(String lab): Bu metot tepe eklemek içindir ve Vertex tipinde vertexList dizisi oluşturulur. Parametre olarak aldığı string ise Vertex classında kullanılır.

public void addEdge(int start, int end, int weight):Bu metot kenar eklemek içindir.Parametre olarak aldığı start ve end değerlerini indeks olarak kullanılır ve matriste bu yerlere gene parametre olarak aldığı weight yani kenarın ağırlığını atar. public void displayVertex(int v) :Bu metot integer tipinde aldığı v parametresini indeks olarak kullaranak vertexList’in o indeksindeki harf\_adı’nı yazdırıyor.

public void mstw() : En küçük kapsayan ağacı bulan metottur. Bu metot öncelikli olarak etkin olan tepeye 0 değerini atar ve ağaçtaki tepe sayısı graphtaki tepe sayısından 1 (while işlemi)küçük olduğu sürece etkin olan tepeyi vertexList’te indeksleyerek sırasıyla geçer.Geçtiği tepeye ağaçta olduğu için true değerini atar ve ağaçtaki tepe sayısını +1 arttırır.Daha sonra vertexList’teki her tepe için for döngüsüyle adjMat’teki currentVert’in tüm tepelere olan uzaklığını putInPQ fonk.u ‘na gönderir.Eğer kuyrukta tepe yoksa GRAPH BAĞLANTILI değildir yazısını ekrana yazdırır.Daha sonra PQ kuyruğundaki en küçük kenarı alır ve bu kenarın bağlı olduğu tepeleri ekrana yazdırır.(While işlemi biter.)En son olarak ise vertexLİst’teki bütün tepelerin ağaçta olup olmadığı durumu false olarak işaretlenir.

public void putInPQ(int newVert, int newDist): Aynı tepeye giden yol üzerinde başka kenar olup olmadığına bakan metottur. Mstw metodunda kullanılır.

**KULLANILAN VERİ YAPILARI**

private Edge[] queArray : Edge tipinde değişken tutan dizidir.PriorityQ classında kullanılır.

public int[,] distanceMat : Bu matris Dijkstra Class’ında kullanılır ve her bir tepenin birbirlerine doğrudan olan uzaklığını yani kenarların ağırlıklarını tutar.

public int[] D : Bu dizi Dijkstra Class’ında kullanılır ve bir tepenin diğer tepelere doğrudan olan uzaklıklarını tutar.

public int[] Nodes : Bu dizi de Dijkstra Class’ında kullanılır ve indeksleme görevi görür.Bu dizideki bir elemanın değeri -1 ise o eleman yani o köşeden geçildiğini gösteriyor.

int [] en\_kucuk : DijkstraSolving metodunda kullanılan bu dizi kuyruğa eklemeden önce görev alır ve bu dizi içinde en kısa yolu bulmak için minNode +1 tutulur ve bu minNode+1 daha sonra kuyruğa eklenerek güzergah oluşturulur.

private Vertex[] vertexList : Bu dizi Graph Class’ında kullanılmış olup Vertex tipinde tepeleri tutar.

private int[,] adjMat :Bu matris graphta hangi tepe hangi tepeye doğrudan olarak ne kadar uzunlukta ya da farklı bir deyişle hangi tepeler arasındaki kenarın ağırlığının ne olduğunu tutar.

private PriorityQ thePQ : PriorityQ tipinde bir öncelik kuyruğudur ve en küçük kapsayan ağaç bulunurken kullanılır.Graph’ın bağlantılı olup olmadığını ve en kısa kenarın bulunması sırasında bu kuyruk kullanılır.

int[,] dizi: Random değerler üreterek içini doldurduğumuz ve bize iki köşe arasındaki ağırlığı tutan matris.

int[] min\_bulan\_dizi: Dijkstra probleminin çözümünde kullanılması için main de oluşturulup dijk.run( ) metoduna parametre olarak gönderilen dizi.

Random r : Her köşe çifti için çalışma zamanında rastgele değerler atanması için oluşturduğumuz değişken.

Queue que : Dijkstra probleminin çözümünde en kısa yolu bulması için kulandığımız kuyruğumuz.

Vertex[] vertex\_list : Her indeksinde Vertex tipinde bir köşe tutan ve en kısa yol algoritmamızda kullandığımız dizi.

Graph theGraph : Tüm köşe ve kenar yapılarını içinde barındıran veri yapımız.

Dijkstra dijk : Dijkstra problemini çözmek için oluşturduğumuz veri yapımız. Çözüm aşamasında bunun metodlarını kullanıyoruz.

Graph gerceklesecek\_graph : Telefon bağlantı durumunu tutan Graph tipindeki nesnemiz.

Edge\_other [] list\_of\_edge : Gerceklesecek\_graph nesnesindeki her kenarın verilerini tutan nesne.

**SP, BFS ve MST ilişkisi:**

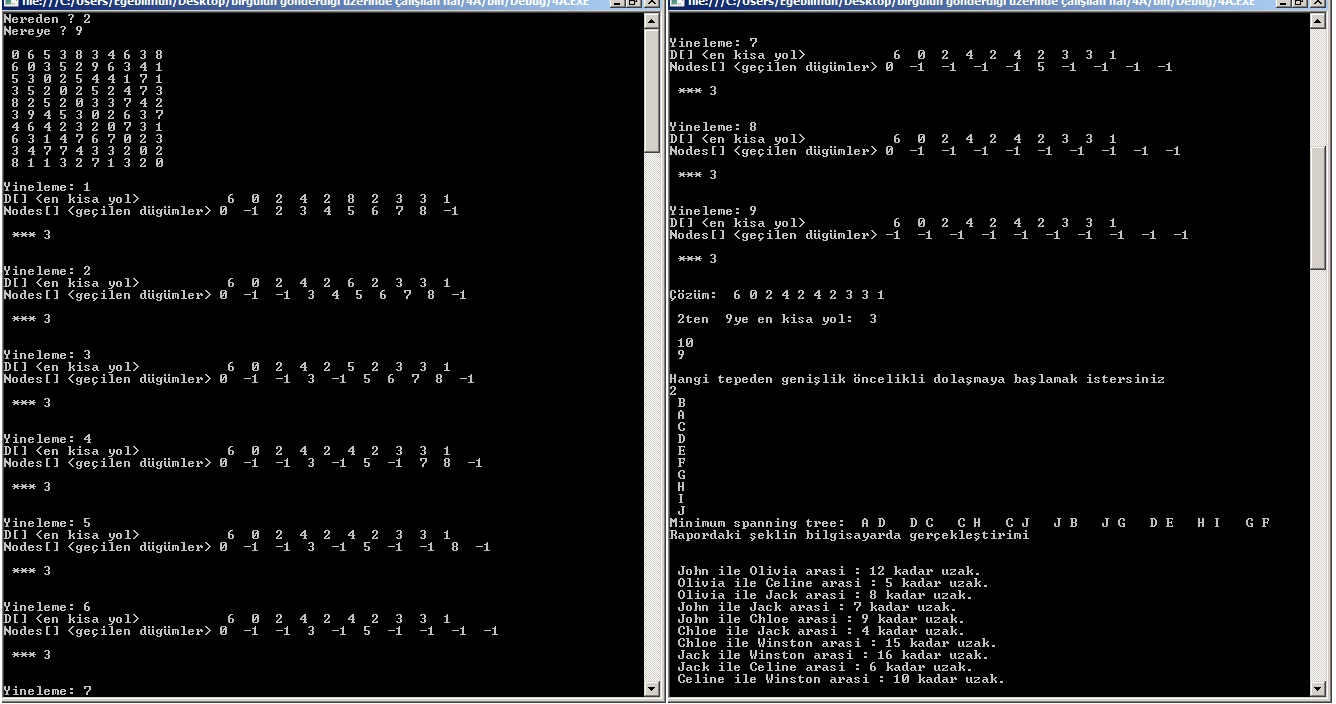
Shortest path : Shortest path bu örnekte birbiri ile bağlantılı olan kişilerden her birinin bir diğerine en kısa sürede nasıl ulaşacağını ve ne kadar sürede ulaşacağını bulmamızı sağlar.

Mininmum spanning tree: Bu örnekte birbiri ile bağlantı içinde olan insanları olası bir acil durumda en az ücret ile en çok kişiye nasıl ulaşılabileceği hakkında bizlere bilgi verir. Ayrıca tek bir kişiden hangi aktarmaları kullanarak hangi kişiye ne kadar sürede ulaşabileceğini gösterir.

Breadth first search : Ağaç veri yapısında her bir derinlikteki tüm elemanlara ulaşılıp sonra bir alt derinliğe inilerek tüm düğümlerin incelendiği algoritma. Bu örnekte ise kullanıcının istediği düğümden başlayarak tüm düğümleri level order gibi gezmemizi sağlıyor. Ve birbiriyle konuşma süresi daha az olan kişileri sırası ile bulur.

**KULLANICI KILAVUZU**

Program açıldığında öncelikle nereden kısmına 1-10 arasında bir değer, sonra nereye kısmına 1-10 arasında bir değer kullanıcıdan girilecektir. Console’da matris ve her yinelemedeki geçilen düğüm ve yenilenen en kısa yollar görüntülenir. Sonra kullanıcının istediği iki yer arasındaki en kısa yol ve oraya nereden gidileceği görüntülenir. Sonrasında genişlik öncelikli dolaşma algoritmasını çalıştırmanız için başlamak istediğiniz tepenin numarası 1-10 aralığında girmeniz gerekmektedir.Genişlik öncelikli dolaşma sırası listelenir. Daha sonra konsolda minimum spanning tree algoritması gereği tüm tepeler arasındaki minumum maliyetli yollar listelenir. En son olarak bir grup insanın telefon bağlantı durumu ekranda görüntülenir.



**KISITLAMALAR:**

* Aralık belirtilmemiştir, fakat 1-10 aralığında tamsayı girmeniz gerekmektedir.
* Sizden genişlik öncelikli arama yapmak için sayı istenirken sayı girmeden “enter”a basmayınız. Lütfen 1-10 aralığında tamsayı giriniz.

**KAYNAK KOD**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Collections;

namespace dijk\_1

{

class PriorityQ

{

// array in sorted order, from max at 0 to min at size-1

private readonly int SIZE = 20;

private Edge[] queArray;

private int size;

public PriorityQ() // constructor

{

queArray = new Edge[SIZE];

size = 0;

}

public void insert(Edge item) // insert item in sorted order

{

int j;

for (j = 0; j < size; j++) // find place to insert

if (item.distance >= queArray[j].distance)

break;

for (int k = size - 1; k >= j; k--) // move items up

queArray[k + 1] = queArray[k];

queArray[j] = item; // insert item

size++;

}

public Edge removeMin() // remove minimum item

{

return queArray[--size];

}

public void removeN(int n) // remove item at n

{

for (int j = n; j < size - 1; j++) // move items down

queArray[j] = queArray[j + 1];

size--;

}

public Edge peekMin() // peek at minimum item

{

return queArray[size - 1];

}

public int getSize() // return number of items

{

return size;

}

public bool isEmpty() // true if queue is empty

{

return (size == 0);

}

public Edge peekN(int n) // peek at item n

{

return queArray[n];

}

public int find(int findDex) // find item with specified

{ // endVertex value

for (int j = 0; j < size; j++)

if (queArray[j].endVertex == findDex)

return j;

return -1;

}

} // end class PriorityQ

class Edge

{

public int startVertex; // index of a vertex starting edge

public int endVertex; // index of a vertex ending edge

public int distance; // distance from src to dest

public Edge(int sv, int dv, int d) // constructor

{

startVertex = sv;

endVertex = dv;

distance = d;

}

} // end class Edge

class Edge\_other

{

public string vertex1;

public string vertex2;

public int distance;

public Edge\_other(string vertex1, string vertex2,int distance)

{

this.vertex1 = vertex1;

this.vertex2 = vertex2;

this.distance = distance;

}

}

class Vertex

{

public string harf\_adı; // label (e.g. ‘A’)

public bool isInTree;

public Vertex(string lab) // constructor

{

harf\_adı = lab;

isInTree = false;

}

public void display()

{

Console.WriteLine(" " + harf\_adı);

}

// -------------------------------------------------------------

} // end class Vertex

class Dijkstra

{

public int[,] distanceMat; // her köşenin diğer köşeler ile arasındaki uzaklık veya ağırlık matrisi

public int[] D; // 0. köşeden diğer köşelere ulan en kısa yolları tutan matris

public int[] Nodes; // -1 olan ilgili indis o köşeden geçildiği gösteriyor

public int tur1 = 0;

public int tur2 = 0;

public int ilk; // kullanıcıdan alınan "nereden" düğümü

public int son; // kullanıcıdan alınan "nereye" düğümü

public bool[] kontrol = new bool[10];

public Dijkstra(int boyut, int[,] pArray, int a, int b)

{

distanceMat = new int[boyut, boyut];

Nodes = new int[boyut];

D = new int[boyut];

tur1 = boyut;

ilk = a;

son = b;

for (int i = 0; i < tur1; i++)

{

for (int j = 0; j < tur1; j++)

{

distanceMat[i, j] = pArray[i, j]; // iki for ile main de random sayılar ile oluşturulup yollanmış olan matrisi

} // distanceMat matrisine kopyaladık.

}

for (int i = 0; i < tur1; i++)

{

Nodes[i] = i; // dolaşılan her düğümü -1 yapmadan önce hepsine -1 den farklı olarak indeksini atadık

}

Nodes[ilk - 1] = -1; // kullanıcıdan alınan "nereden" düğümü ile başladık ve -1 atadık

for (int i = 0; i < tur1; i++)

{

D[i] = distanceMat[ilk - 1, i]; // D[] matrisine "nereden" düğümünün diğer tüm düğümler ile olan ilk uzaklıklarını atadık

}

}

public void DijkstraSolving(Queue que, int[] min\_bulan\_dizi, int current\_tour)

{

int a = 0;

int minValue = Int32.MaxValue; // sonsuz olmasa da en yüksek integer değerden başlayarak bulabildiğimiz en kısa yolu buluyoruz

int minNode = 0;

for (int i = 0; i < tur1; i++)

{

if (Nodes[i] == -1) // eğer ilgili düğümün Nodes[] değeri -1 ise daha önce o düğümden geçilmiş demektir

continue; // o sebeple continue diyerek for döngüsüne devam ediyoruz

if (D[i] > 0 && D[i] < minValue)

{ // eğer o düğümden ilk kez geçilecekse ve D[] değeri şimdiye kadarki ağırlığından küçük ise

minValue = D[i]; // yeni ve daha kısa bir yol bulmuşuz demektir

minNode = i; // o düğümün ağırlığını değiştiriyoruz

}

}

Nodes[minNode] = -1; // for döngüsünden çıktıktan Nodes[] daki ilgili düğümü -1 yaparak, o düğümden geçtiğimizi belirtiyoruz

int [] en\_kucuk = new int[tur1];

for (int i = 0; i < tur1; i++)

{

if ((D[minNode] + distanceMat[minNode, i]) < D[i] && distanceMat[minNode, i] != 0)

{ // i'nin bu yeni düğümden önceki en kısa yolunun ve distanceMat[minNode, i] deki uzunluğunun toplamı

D[i] = minValue + distanceMat[minNode, i];// D[i] den küçükse daha kısa yol bulunmuş demektir, atıyoruz.

en\_kucuk[i] = minNode+1;

}

}

que.Enqueue(en\_kucuk[son-1]);

}

public void run(Queue que, int[] min\_bulan\_dizi)

{

for (tur2 = 1; tur2 < tur1; tur2++) // boyut kadar dönüyoruz

{

DijkstraSolving(que, min\_bulan\_dizi,tur2);

Console.WriteLine("\n\nYineleme: " + tur2); // her for döngüsünde Nodes[] ile şuanki düğümü ayrıca kullanıcıdan alınan

Console.Write("D[] <en kisa yol> "); // "nereden" düğümünden tüm düğümlere giden en kısa yolları D[] ile takip edebiliyoruz.

for (int i = 0; i < tur1; i++)

Console.Write(D[i] + " ");

Console.WriteLine("");

Console.Write("Nodes[] <geçilen dügümler> ");

for (int i = 0; i < tur1; i++)

Console.Write(Nodes[i] + " ");

Console.WriteLine("");

Console.WriteLine("\n \*\*\* " + D[son - 1]);

}

}

public void bfs(Vertex[] vertex\_list, int start) // breadth-first search --> derinlik öncelikli dolaşma

{

Queue kuyruk = new Queue();

int v2;

kontrol[start] = true; // geçtiğimiz düğümü işaretliyoruz

vertex\_list[start].display(); // ilk olarak başladığımız düğümü ekrana görüntületiyoruz

kuyruk.Enqueue(start); // başlamak istediğimiz düğümü kuyruğumuza ekliyoruz

while (kuyruk.Count != 0) // kuyruk boşalıncaya kadar döngüden çıkmıyoruz

{

int v1 = (int)kuyruk.Dequeue(); // kuyruktan sıradaki düğümü çıkarıyoruz

while ((v2 = getAdjUnvisitedVertex(v1)) != -1) // işaretlenmemiş komşumuz kalana kadar dönüyoruz

{

kontrol[v2] = true; // geçtiğimiz düğümü işaretliyoruz

vertex\_list[v2].display(); //işaretlediğimiz düğümü ekrana görüntületiyoruz

kuyruk.Enqueue(v2); // ve kuyruğa ekliyoruz.

}

}

for (int j = 0; j < kontrol.Count(); j++) // kontrol dizimizi default haline geri döndürüyoruz

kontrol[j] = false;

}

public int getAdjUnvisitedVertex(int v)

{

for (int j = 0; j < tur1; j++)

if (distanceMat[v, j] >= 1 && kontrol[j] == false)

return j;

return -1;

} // end getAdjUnvisitedVertex()

}

class Graph

{

private readonly int MAX\_VERTS = 20;

private readonly int INFINITY = 1000000;

private Vertex[] vertexList; // list of vertices

private int[,] adjMat; // adjacency matrix

private int nVerts; // current number of vertices

private int currentVert;

private PriorityQ thePQ;

private int nTree; // number of verts in tree

public Graph() // constructor

{

vertexList = new Vertex[MAX\_VERTS];

adjMat = new int[MAX\_VERTS, MAX\_VERTS]; // adjacency matrix

nVerts = 0;

for (int j = 0; j < MAX\_VERTS; j++) // set adjacency

for (int k = 0; k < MAX\_VERTS; k++) // matrix to 0

adjMat[j, k] = INFINITY;

thePQ = new PriorityQ();

} // end constructor

public void addVertex(String lab)

{

vertexList[nVerts++] = new Vertex(lab);

}

public void addEdge(int start, int end, int weight)

{

adjMat[start, end] = weight;

adjMat[end, start] = weight;

}

public void displayVertex(int v)

{

Console.Write(vertexList[v].harf\_adı);

}

public void mstw() // minimum spanning tree

{

currentVert = 0; // start at 0

while (nTree < nVerts - 1) // while not all verts in tree

{ // put currentVert in tree

vertexList[currentVert].isInTree = true;

nTree++;

// insert edges adjacent to currentVert into PQ

for (int j = 0; j < nVerts; j++) // for each vertex,

{

if (j == currentVert) // skip if it’s us

continue;

if (vertexList[j].isInTree) // skip if in the tree

continue;

int distance = adjMat[currentVert, j];

if (distance == INFINITY) // skip if no edge

continue;

putInPQ(j, distance); // put it in PQ (maybe)

}

if (thePQ.getSize() == 0) // no vertices in PQ?

{

Console.WriteLine(" GRAPH NOT CONNECTED");

return;

}

// remove edge with minimum distance, from PQ

Edge theEdge = thePQ.removeMin();

int sourceVert = theEdge.startVertex;

currentVert = theEdge.endVertex;

// display edge from source to current

Console.Write(" " + vertexList[sourceVert].harf\_adı);

Console.Write(" " + vertexList[currentVert].harf\_adı);

Console.Write(" ");

} // end while(not all verts in tree)

// mst is complete

for (int j = 0; j < nVerts; j++) // unmark vertices

vertexList[j].isInTree = false;

} // end mstw

public void putInPQ(int newVert, int newDist)

{

// is there another edge with the same destination vertex?

int queueIndex = thePQ.find(newVert);

if (queueIndex != -1) // got edge’s index

{

Edge tempEdge = thePQ.peekN(queueIndex); // get edge

int oldDist = tempEdge.distance;

if (oldDist > newDist) // if new edge shorter,

{

thePQ.removeN(queueIndex); // remove old edge

Edge theEdge =

new Edge(currentVert, newVert, newDist);

thePQ.insert(theEdge); // insert new edge

}

// else no action; just leave the old vertex there

} // end if

else // no edge with same destination vertex

{ // so insert new one

Edge theEdge = new Edge(currentVert, newVert, newDist);

thePQ.insert(theEdge);

}

} // end putInPQ()

// -------------------------------------------------------------

} // end class Graph

class Program

{

static void Main(string[] args)

{

int boyut = 10;

int[,] dizi = new int[boyut, boyut];

int rastgele, x, y;

Random r = new Random();

Queue que = new Queue();

int[] min\_bulan\_dizi = new int[boyut];

Vertex[] vertex\_list = new Vertex[boyut];

Graph theGraph = new Graph();

for (int i = 0; i < boyut; i++) // random sayılar ile matrisime uzunluk değerleri atıyorum

{

for (int j = i + 1; j < boyut; j++)

{

rastgele = r.Next(1, boyut);

if (i != j) // tabi eğer matrisin köşesinde değilsek

{

dizi[i, j] = rastgele;

dizi[j, i] = rastgele;

theGraph.addEdge(i, j, rastgele);

}

if (dizi[i, j] == 0 || dizi[j, i] == 0)

{

dizi[i, j] = Int32.MaxValue; //infinity

dizi[j, i] = Int32.MaxValue; //infinity

}

}

string going;

switch (i)

{

case 0:

going = "A";

vertex\_list[0] = new Vertex(going);

theGraph.addVertex(going);

break;

case 1:

going = "B";

vertex\_list[1] = new Vertex(going);

theGraph.addVertex(going);

break;

case 2:

going = "C";

vertex\_list[2] = new Vertex(going);

theGraph.addVertex(going);

break;

case 3:

going = "D";

vertex\_list[3] = new Vertex(going);

theGraph.addVertex(going);

break;

case 4:

going = "E";

vertex\_list[4] = new Vertex(going);

theGraph.addVertex(going);

break;

case 5:

going = "F";

vertex\_list[5] = new Vertex(going);

theGraph.addVertex(going);

break;

case 6:

going = "G";

vertex\_list[6] = new Vertex(going);

theGraph.addVertex(going);

break;

case 7:

going = "H";

vertex\_list[7] = new Vertex(going);

theGraph.addVertex(going);

break;

case 8:

going = "I";

vertex\_list[8] = new Vertex(going);

theGraph.addVertex(going);

break;

case 9:

going = "J";

vertex\_list[9] = new Vertex(going);

theGraph.addVertex(going);

break;

}

}

Console.Write("Nereden ? "); // kullanıcıdan iki integer değer alıyoruz

x = (int)Convert.ToInt32(Console.ReadLine()); // nereden en kısa yol ile nereye gitmek istedikleri

Console.Write("Nereye ? ");

y = (int)Convert.ToInt32(Console.ReadLine());

Dijkstra dijk = new Dijkstra(boyut, dizi, x, y); // Dijkstra tipinde dijk değişkeni tanımlıyoruz

for (int m = 0; m < boyut; m++)

{

Console.WriteLine("");

for (int n = 0; n < boyut; n++)

{

Console.Write(" " + dijk.distanceMat[m, n]); // ve takip edilmesi kolay olsun diye tüm matrisi ekrana görüntületiyoruz

}

}

dijk.run(que, min\_bulan\_dizi); // Dijkstra's shortest path algoritmasını kullanarak en kısa yolu buluyoruz

Console.Write("\n\nÇözüm: ");

foreach (int i in dijk.D)

{

Console.Write(" "+i); // nereden den tüm köşelere giden en kısa yollar

}

Console.WriteLine("\n\n " + x + "ten " + y + "ye en kisa yol: " + dijk.D[y - 1]); // mutlu son =D

que.Enqueue(y);

Console.WriteLine("");

foreach (object o in que) // dijkstra'nın bulduğu en kısa yolda hangi yollardan geçtiğini ekrana yazdırıyoruz

{

if ((int)o != 0)

Console.WriteLine(" " + o + " ");

}

Console.WriteLine(" ");

Console.WriteLine("Hangi tepeden genişlik öncelikli dolaşmaya başlamak istersiniz");

int start = (int)Convert.ToInt16(Console.ReadLine());

dijk.bfs(vertex\_list, (start - 1));

Console.Write("Minimum spanning tree: ");

theGraph.mstw(); // minimum spanning tree

Console.WriteLine();

Console.WriteLine("Rapordaki şeklin bilgisayarda gerçekleştirimi");

Graph gerceklesecek\_graph = new Graph();

string eklenecek\_john = "John";

gerceklesecek\_graph.addVertex(eklenecek\_john);

string eklenecek\_olivia = "Olivia";

gerceklesecek\_graph.addVertex(eklenecek\_olivia);

string eklenecek\_celine = "Celine";

gerceklesecek\_graph.addVertex(eklenecek\_celine);

string eklenecek\_jack = "Jack";

gerceklesecek\_graph.addVertex(eklenecek\_jack);

string eklenecek\_chloe = "Chloe";

gerceklesecek\_graph.addVertex(eklenecek\_chloe);

string eklenecek\_winston = "Winston";

gerceklesecek\_graph.addVertex(eklenecek\_winston);

Edge\_other [] list\_of\_edge = new Edge\_other[10];

list\_of\_edge[0] = new Edge\_other(eklenecek\_john, eklenecek\_olivia, 12);

list\_of\_edge[1] = new Edge\_other(eklenecek\_olivia, eklenecek\_celine, 5);

list\_of\_edge[2] = new Edge\_other(eklenecek\_olivia, eklenecek\_jack, 8);

list\_of\_edge[3] = new Edge\_other(eklenecek\_john, eklenecek\_jack, 7);

list\_of\_edge[4] = new Edge\_other(eklenecek\_john, eklenecek\_chloe, 9);

list\_of\_edge[5] = new Edge\_other(eklenecek\_chloe, eklenecek\_jack, 4);

list\_of\_edge[6] = new Edge\_other(eklenecek\_chloe, eklenecek\_winston, 15);

list\_of\_edge[7] = new Edge\_other(eklenecek\_jack, eklenecek\_winston, 16);

list\_of\_edge[8] = new Edge\_other(eklenecek\_jack, eklenecek\_celine, 6);

list\_of\_edge[9] = new Edge\_other(eklenecek\_celine, eklenecek\_winston, 10);

Console.WriteLine("\n");

String a, b;

int d;

for (int i = 0; i < 10; i++)

{

a = list\_of\_edge[i].vertex1;

b = list\_of\_edge[i].vertex2;

d = list\_of\_edge[i].distance;

Console.WriteLine(" " + a + " ile " + b + " arasi : " + d + " kadar uzak.");

}

Console.ReadKey();

}

}

}