Hayrullah URCAN
23537676020

A. DİJKSTRA ALGORİTMASI

- **Çalışma Mantığı:**

Dijkstra algoritması, bir grafın bir başlangıç düğümünden tüm diğer düğümlere olan en kısa yolları bulmak için kullanılır. Pozitif ağırlıklı kenarlarla çalışır.

- **Karmaşıklık:**

Zaman Karmaşıklığı: $O\ (\ V^{\ 2}\ )$(Priority Queue kullanılırsa $O\ (\ E\log V)$

Uzay Karmaşıklığı: $O\ (\ V\ )$

- **Algoritma:**

1. Başlangıç düğümünü seç ve bu düğümün maliyetini 0 olarak ayarla.
2. Tüm diğer düğümlerin maliyetini sonsuz yap.
3. Başlangıç düğümünü bir öncelik kuyruğuna ekle.
4. Kuyruk boş olana kadar şu adımları tekrarla:
    - Kuyruktan minimum maliyetli düğümü al.
    - Bu düğümün komşularını güncelle ve kuyrukta maliyetleri değiştir.

- **C# Kodu**

```
using System;
using System.Collections.Generic;

public class Graph
{
    // Grafın düğüm sayısını tutar
    private int Vertices;

    // Komşuluk listesini, her düğüm için bir liste tutar
    private List<Tuple<int, int>>[] adjacencyList;

    // Constructor, düğüm sayısını alır ve komşuluk listesini başlatır
    public Graph(int vertices)
    {
        // Parametre olarak alınan düğüm sayısını Vertices değişkenine atar
        Vertices = vertices;
        // Her düğüm için bir komşuluk listesi oluşturmak üzere 'adjacencyList' dizisini başlatır
        adjacencyList = new List<Tuple<int, int>>[vertices];
        // Tüm düğümler için komşuluk listesini oluşturur
        for (int i = 0; i < vertices; i++)
        {
            // Her düğüm için yeni bir liste başlatır
            adjacencyList[i] = new List<Tuple<int, int>>();
        }
    }
```

```
// Graf'a kenar ekler
public void AddEdge(int u, int v, int weight)
{
   // Kenarı (u, v) ve ağırlığını (weight) u'nun komşuluk listesine ekler
   adjacencyList[u].Add(new Tuple<int, int>(v, weight));
   // Eğer yönsüz grafsa, aynı kenarı (v, u) ve ağırlığını v'nin komşuluk listesine de ekler
   adjacencyList[v].Add(new Tuple<int, int>(u, weight));
}

// Dijkstra algoritması, kaynak düğümden diğer düğümlere en kısa yolu bulur
public void Dijkstra(int source)
{
   // Mesafe dizisi, başlangıçta tüm mesafeler sonsuz olarak ayarlanır
   var distance = new int[Vertices];
   // Öncelik kuyruğu, düğümleri ve mesafeleri tutar
   var priorityQueue = new SortedSet<Tuple<int, int>>();
   // Başlangıçta tüm mesafeleri sonsuz olarak ayarla
   for (int i = 0; i < Vertices; i++)
   {
      distance[i] = int.MaxValue;
   }
   // Kaynak düğümün mesafesini 0 olarak ayarla ve kuyruğa ekle
   distance[source] = 0;
   priorityQueue.Add(new Tuple<int, int>(0, source));

   // Öncelik kuyruğu boşalana kadar devam et
   while (priorityQueue.Count > 0)
   {
      // Kuyruğun en küçük elemanını al ve kuyruktan çıkar
      var current = priorityQueue.Min;
      priorityQueue.Remove(current);

      // Şu anki düğümü al
      int u = current.Item2;

      // Şu anki düğümün tüm komşularını kontrol et
      foreach (var neighbor in adjacencyList[u])
      {
         // Komşu düğümü ve kenar ağırlığını al
         int v = neighbor.Item1;
         int weight = neighbor.Item2;

         // Yeni bir kısa yol bulunduysa, mesafeyi güncelle ve kuyruğa ekle
         if (distance[u] + weight < distance[v])
         {
            // Mevcut komşu mesafesini kuyruktan çıkar
            priorityQueue.Remove(new Tuple<int, int>(distance[v], v));
            // Yeni kısa yolu güncelle
            distance[v] = distance[u] + weight;
            // Güncellenmiş mesafeyle kuyruğa tekrar ekle
            priorityQueue.Add(new Tuple<int, int>(distance[v], v));
```

```
                }
            }
        }

        // Sonuçları yazdır
        Console.WriteLine("Vertex Distance from Source");
        for (int i = 0; i < Vertices; i++)
        {
            // Her düğümün mesafesini yazdır
            Console.WriteLine($"{i}\t {distance[i]}");
        }
    }
}
```

B. KRUSKAL ALGORİTMASI

- **Çalışma Mantığı:**

Kruskal algoritması, bir grafın minimum maliyetli kapsama ağacını (MST) bulmak için kullanılır. Kenarları artan ağırlık sırasına göre ekleyerek çalışır ve döngü oluşturmaktan kaçınır.

- **Karmaşıklık:**

Zaman Karmaşıklığı: O (E log E) veya O (E log V)

Uzay Karmaşıklığı: O ($E + V$)

- **Algoritma:**

1. Tüm kenarları artan ağırlık sırasına göre sırala.
2. Boş bir MST seti başlat.
3. Her kenarı sırayla ekle, eğer döngü oluşturmuyorsa MST'ye ekle.
4. Tüm düğümler kapsanana kadar bu işlemi devam ettir.

- **C# Kodu:**

```
using System;
using System.Collections.Generic;

public class KruskalGraph
{
   // Grafın düğüm sayısını tutar
   private int Vertices;

   // Kenarların listesini tutar
   private List<Edge> edges = new List<Edge>();

   // Constructor, düğüm sayısını alır
   public KruskalGraph(int vertices)
   {
      // Parametre olarak alınan düğüm sayısını Vertices değişkenine atar
      Vertices = vertices;
   }

   // Graf'a kenar ekler
   public void AddEdge(int u, int v, int weight)
   {
      // Kenarları (u, v) ve ağırlığını (weight) Edge nesnesi olarak edges listesine ekler
      edges.Add(new Edge(u, v, weight));
   }

   // Kruskal algoritması, minimum yayılı ağacını (MST) bulur
   public void KruskalMST()
```

```
{
    // Kenarları ağırlıklarına göre sıralar
    edges.Sort((a, b) => a.Weight.CompareTo(b.Weight));

    // Ebeveyn dizisi, her düğümü kendi ebeveyni olarak başlatır
    var parent = new int[Vertices];
    for (int i = 0; i < Vertices; i++)
        // Her düğümü başlangıçta kendi ebeveyni olarak atar
        parent[i] = i;

    // 'Find' fonksiyonu, bir düğümün kökünü bulur ve sıkıştırma (path compression) yapar
    int Find(int i)
    {
        // Eğer düğüm kendi ebeveyni ise kendisini döndürür
        if (parent[i] == i)
            return i;
        // Değilse, ebeveynini bulur ve path compression yapar
        return parent[i] = Find(parent[i]);
    }

    // 'Union' fonksiyonu, iki kümenin köklerini birleştirir
    void Union(int i, int j)
    {
        // İki düğümün köklerini bulur
        int a = Find(i);
        int b = Find(j);
        // Birinin kökünü diğerine bağlar
        parent[a] = b;
    }

    // MST'deki kenarları tutacak liste
    var result = new List<Edge>();

    // Sıralı kenarlar üzerinden geçer
    foreach (var edge in edges)
    {
        // Kenarın başlangıç düğümünü alır
        int u = edge.U;
        // Kenarın bitiş düğümünü alır
        int v = edge.V;

        // Bu düğümlerin kök kümelerini bulur
        int setU = Find(u);
        int setV = Find(v);

        // Eğer u ve v farklı kümelere aitse, bu kenarı MST'ye ekle ve kümeleri birleştir
        if (setU != setV)
        {
            // Kenarı sonuç listesine ekler
            result.Add(edge);
            // İki kümeyi birleştirir
```

```
            Union(setU, setV);
          }
        }

        // MST'deki kenarları yazdırır
        Console.WriteLine("Edges in MST");
        foreach (var edge in result)
        {
          // Her kenarın başlangıç ve bitiş düğümlerini ve ağırlığını yazdırır
          Console.WriteLine($"{edge.U} - {edge.V}: {edge.Weight}");
        }
    }

    // Kenar sınıfı, bir kenarın iki düğümünü ve ağırlığını tutar
    private class Edge
    {
        // Kenarın başlangıç düğümü
        public int U, V, Weight;

        // Kenar yapıcı fonksiyonu, başlangıç düğümü, bitiş düğümü ve ağırlığı alır
        public Edge(int u, int v, int weight)
        {
            // Parametre olarak alınan değerleri sınıf değişkenlerine atar
            U = u;
            V = v;
            Weight = weight;
        }
    }
}
```

C. PRİM ALGORİTMASI

- **Çalışma Mantığı:**
- Prim algoritması, bir grafın minimum maliyetli kapsama ağacını (MST) bulmak için kullanılır. Başlangıç düğümünden başlayarak, mevcut ağaca en düşük ağırlıklı kenarı ekleyerek çalışır.

- **Karmaşıklık:**

Zaman Karmaşıklığı: O (V^2) (Priority Queue kullanılırsa O (E log V))

Uzay Karmaşıklığı: O (V)

- **Algoritma:**

1. Başlangıç düğümünü seç ve bu düğümün maliyetini 0 olarak ayarla.
2. Tüm diğer düğümlerin maliyetini sonsuz yap.
3. Bir MST seti başlat ve başlangıç düğümünü bu sete ekle.
4. MST'ye en düşük maliyetli kenarı ekle ve bu kenarın diğer ucunu MST'ye ekle.
5. Tüm düğümler MST'ye eklenene kadar bu işlemi tekrar et.

- **C# Kodu:**

```
using System;
using System.Collections.Generic;

public class PrimGraph
{
   // Grafın düğüm sayısını tutar
   private int Vertices;

   // Komşuluk listesini, her düğüm için bir liste tutar
   private List<Tuple<int, int>>[] adjacencyList;

   // Constructor, düğüm sayısını alır ve komşuluk listesini başlatır
   public PrimGraph(int vertices)
   {
      // Parametre olarak alınan düğüm sayısını Vertices değişkenine atar
      Vertices = vertices;
      // Her düğüm için bir komşuluk listesi oluşturmak üzere 'adjacencyList' dizisini başlatır
      adjacencyList = new List<Tuple<int, int>>[vertices];
      // Tüm düğümler için komşuluk listesini oluşturur
      for (int i = 0; i < vertices; i++)
      {
         // Her düğüm için yeni bir liste başlatır
         adjacencyList[i] = new List<Tuple<int, int>>();
```

```
    }
}

// Graf'a kenar ekler
public void AddEdge(int u, int v, int weight)
{
    // Kenarı (u, v) ve ağırlığını (weight) u'nun komşuluk listesine ekler
    adjacencyList[u].Add(new Tuple<int, int>(v, weight));
    // Aynı kenarı (v, u) ve ağırlığını v'nin komşuluk listesine de ekler
    adjacencyList[v].Add(new Tuple<int, int>(u, weight));
}

// Prim algoritması, minimum yayılı ağacını (MST) bulur
public void PrimMST()
{
    // Anahtar değerlerini tutar ve başlangıçta tüm anahtarları sonsuz yapar
    var key = new int[Vertices];
    // MST'yi temsil eden ebeveyn dizisi
    var parent = new int[Vertices];
    // MST'ye dahil olup olmadığını takip eder
    var mstSet = new bool[Vertices];

    // Başlangıçta tüm anahtarları sonsuz ve mstSet dizisini false yapar
    for (int i = 0; i < Vertices; i++)
    {
        key[i] = int.MaxValue;
        mstSet[i] = false;
    }

    // İlk düğümün anahtarını 0 yaparak başlatır
    key[0] = 0;
    // İlk düğümün ebeveynini -1 yaparak başlatır (MST'nin kökü)
    parent[0] = -1;

    // Tüm düğümler için MST setine eklenene kadar döngü
    for (int count = 0; count < Vertices - 1; count++)
    {
        // Anahtar değeri en düşük olan düğümü seçer ve MST setine ekler
        int u = MinKey(key, mstSet);
        mstSet[u] = true;

        // Seçilen düğümün komşularını günceller
        foreach (var neighbor in adjacencyList[u])
        {
            int v = neighbor.Item1;
            int weight = neighbor.Item2;

            // Eğer komşu düğüm MST setinde değilse ve anahtar değeri güncellenebiliyorsa
            if (!mstSet[v] && weight < key[v])
            {
                // Ebeveynini ve anahtar değerini günceller
```

```
                parent[v] = u;
                key[v] = weight;
            }
        }
    }

    // MST'yi yazdırır
    PrintMST(parent);
  }

  // Anahtar değeri en düşük olan düğümü bulan yardımcı fonksiyon
  private int MinKey(int[] key, bool[] mstSet)
  {
    // Başlangıçta minimum değeri ve indeksini tanımlar
    int min = int.MaxValue, minIndex = -1;

    // Tüm düğümler üzerinde döner
    for (int v = 0; v < Vertices; v++)
    {
      // Eğer düğüm MST setinde değilse ve anahtar değeri minimumdan küçükse
      if (mstSet[v] == false && key[v] < min)
      {
        // Yeni minimum değeri ve indeksini günceller
        min = key[v];
        minIndex = v;
      }
    }

    // En düşük anahtar değerine sahip düğümün indeksini döner
    return minIndex;
  }

  // MST'yi yazdıran yardımcı fonksiyon
  private void PrintMST(int[] parent)
  {
    Console.WriteLine("Edge \tWeight");
    // Tüm düğümler üzerinde döner
    for (int i = 1; i < Vertices; i++)
    {
      // Ebeveyn düğümünden bu düğüme olan kenarı ve ağırlığını yazdırır
      Console.WriteLine($"{parent[i]} - {i}\t{key[i]}");
    }
  }
}
```