



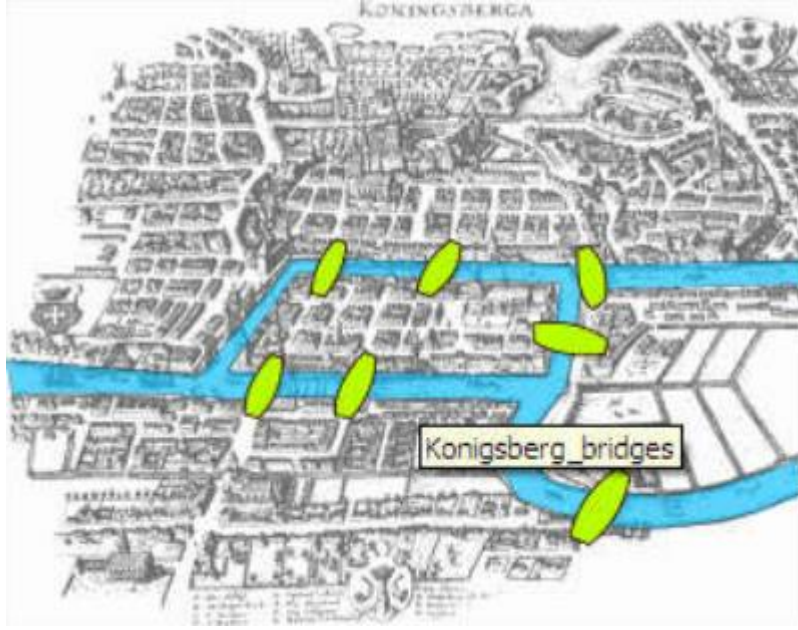
**YAZILIM KALİTE GÜVENCE VE TESTİ FİNAL PROJE
DOKÜMANTASYONU**

16542005 – Alperen Şişman

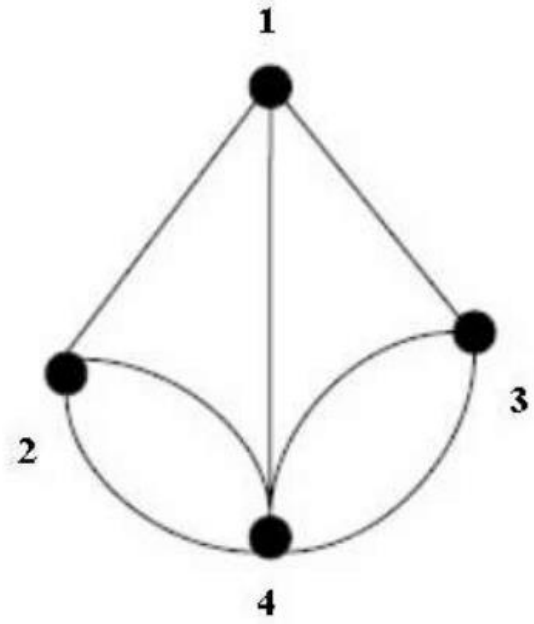
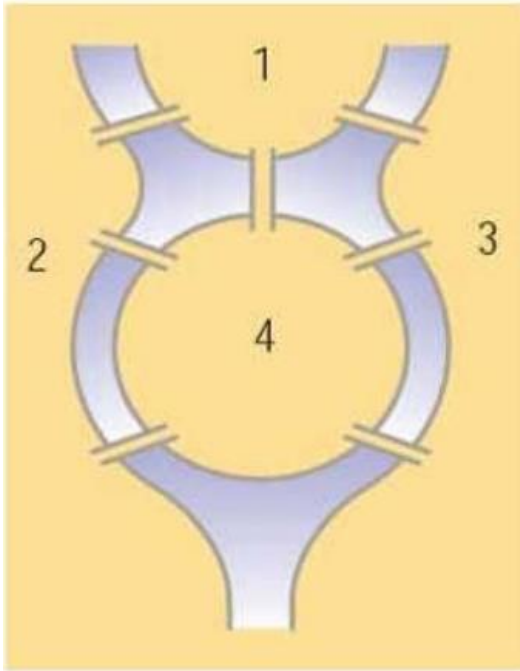
1. Giriş

Matematikte graf ya da çizge, nesne çiftlerinin bir anlamda "ilişkili" olduğu bir dizi nesne kümesini belirleyen bir yapıdır[1]. Nesneler, köşeler (ayrıca düğümler veya noktalar olarak da adlandırılır) adı verilen matematiksel soyutlamalara karşılık gelir ve ilgili düğüm çiftlerinin her birine bir kenar, ayrıntı (bağlantı veya çizgi olarak da adlandırılır) adı verilir. Tipik olarak bir graf, kenarları için çizgiler veya eğriler ile birleştirilen, düğümler için bir nokta veya daire kümesi olarak diyagram şeklinde gösterilir. Graflar ayrık matematikte çalışmanın amaçlarından biridir. 1736'da Euler'in yazdığı "Königsberg'in yedi köprüsü" isimli (Şekil 1) meşhur makalesi ile graflar ortaya çıkar[2]. O yıllarda Königsberg'deki Pregel nehri Kneiphof adasının iki yanından akmakta ve üzerinde yedi farklı köprü bulunmaktaydı. Şehir halkı, her bir köprüden sadece bir kez geçerek tüm kıyıları dolaşmanın ve tekrar başlangıç noktasına dönmenin mümkün olup olmadığını merak ediyordu. İsviçreli matematikçi Leonhard Euler Şekil 2 'de görüldüğü gibi, önce problemi hat ve düğümlerden oluşan bir yapıya dönüştürdü. Sonradan graf adı verilen bu yapıyı kullanarak problemin çözümüne ulaştı. Königsberg halkı aradıkları yolu bulamamakta haklıydılar, zira böyle bir yol yoktu. Graflar bu süreçten sonra zamanla geliştirildi ve 12 adet graf ortaya konuldu[1]. Bu Graflar;

- Yönlendirilmiş Graf
- Düzenli Graf
- Tam Graf
- Sonlu Graf
- Bağlı Graf
- İki parçalı Graf
- Yol Graf
- Düzlemsel Graf
- Çember Graf
- Ağırlıklı Graf
- Ağaç
- Çoklu Ağ



Şekil 1 Königsberg'in yedi köprüsü



Şekil 2 Leonhard Euler Hat ve Düğümler Yapısı

Dijkstra Algoritması[4], en kısa yol problemi ağırlıklı bir grafta herhangi bir tepeden diğer tüm tepelere en kısa yolun bulunması problemidir. Dijkstra algoritması bu problemin çözümü için kullanılan en popüler algoritmalarından biridir. Adını Hollandalı Matematikçi ve Bilgisayar Bilimci Edsger Dijkstra'dan alır. Dijkstra algoritması sezgisel açgözlü bir algoritmadır. Yani en iyi sonucu vermeyi garanti etmez.

Bellman-Ford algoritması[3], bir graf üzerindeki, bir kaynaktan bir hedefe giden en kısa yolu bulmaktır. Bir $s \in V$ kaynağından tüm $v \in V$ 'lere bütün kısa yol uzunluklarını bulur. Bir negatif ağırlık çevrimi olduğunu saptar. Algoritma ağırlıklı graflar üzerinde çalışır ve bir anlamda Dijkstra algoritmasının iyileştirilmiş olarak düşünülebilir. Bu algoritma aslında Dijkstra algoritmasından daha kötü bir performansa sahiptir ancak graftaki ağırlıklarıneksi(-) olması durumunda Dijkstra'nın tersine başarılı çalışır.

Bu projede graf dolaşım algoritmaları arasında testler gerçekleştirilecektir. Gerçekleştireceğimiz testlerde Dijkstra ve Bellman-Ford algoritmalarının kullanacağız. Kullanılacak test teknikleri;

- Notasyon Analizi,
- Performans Testi,
- Döngüsel Karmaşıklık Testi,

Teknikleridir.

2. Kullanılacak Test Teknikleri

2.1 Algoritma Analizi

Algoritma, matematikte ve bilgisayar biliminde bir işi yapmak için tanımlanan, bir başlangıç durumundan başladığında, açıkça belirlenmiş bir son durumunda sonlanan, sonlu işlemler kümesidir. İyi algoritmayı belirlemek için uygulanan testler veya yapılan işlemler Algoritma Analizi'nin konusudur[5]. Aynı problemi çözen birden fazla algoritma arasından en iyisini seçme tekniğidir.

2.1.1 Algoritmaların Yönleri

Algoritma yönleri 6 başlıkta ifade edilir, bunlar[5];

1. Algoritmaları Tasarlama

- Bulmacaların (puzzle) parçalarını birleştirme,
- Veri yapılarını seçme,
- Problemin çözümü için temel yaklaşımlar seçme,
- En popüler tasarım stratejileri böl ve fethet (divide&conquer), açgözlü(greedy), dinamik programlama, özyineleme (recursive).

2. Algoritma ifadesi ve uygulanması

- Algoritmayı tasarladıktan sonra sözde kod (pseudocode),
- İfadesinin belirlenmesi ve problem için uygulanması,
- Bu konudaki endişeler, netlik, özlülük, etkinlik vb.

3. Algoritma Analizi (Çözümlemesi)

- Algoritma analizi, algoritmayı gerçekte uygulamadan, bir algoritmayı çalıştırabilmek için gereken kaynakların (zaman, yer gibi) araştırılması demektir.

4. Çözümünüzün yeterince iyi olup olmadığını görmek için alt ve üst sınırları karşılaştırma

- Algoritma analizi problemi çözmek için bize alt ve üst sınırları verir

5. Algoritma veya programı doğrulama

- Algoritmanın verilen tüm olası girişler için hesaplama yaptığını ve doğru çıkış ürettiğini göstermektir.

6. Algoritmaların test edilmesi

Test için iki aşama vardır;

- Hata ayıklama (Debugging): Programın örnek veriler üzerinde çalıştırılması sırasında oluşan hataları tespit etme ve onları düzeltme işlemi.
- Profil oluşturma (Profiling): Çeşitli veriler üzerinde programın çalıştırılması ve sonuçların hesaplaması için gerekli zamanın (ve alan) ölçülmesi işlemi.

2.2 Performans Testi

Algoritmanın performansı[6], bir algoritma için hızlı ve sorunsuz çalışması olarak tanımlanabilir. Çeşitli kodlama yöntemleri ile performansı düşük bir algoritma performansı arttırılabilir veya yüksek performanslı algoritma geliştirilebilir. Algoritma performansını, Performans Testi / Raporu yaparak anlamak mümkündür. Performans testinin, algoritmanın nasıl çalıştığını gösteren ve algoritma hatalarını bulmakla ilgili bir test olmadığı unutulmamalıdır. Yapılacak analiz sonrası ihtiyaç duyuluyorsa algoritma performans arttırmaya yönelik çalışmalar yapılmalıdır. Bu çalışmalar ise çıkarılacak doğru bir yol haritası ile profesyonelce yapılmalıdır. Bilgisayar bilimlerinde algoritmanın performansını ölçmek için “zaman” kullanmak popüler bir yaklaşımdır. Bu teknik ile algoritmanın başlangıç zamanı ve bitiş zamanı ölçülür. Başlangıç ve bitiş zamanları arasındaki fark test edilen algoritmanın performansı olarak kabul edilebilir. Karşılaştırma amaçlı aldığımız bu zaman farkı, aynı problemi çözen diğer algoritmalarda da sırayla test edilir. En kısa süreyi veren algoritma bizim seçeceğimiz en iyi algoritma olabilir.

2.3 Döngüsel Karmaşıklık Testi

Döngüsel karmaşıklık basit bir ifade ile kaynak kodda bulunan karar sayılarını belirlemek için kullanılmaktadır[9]. Bu sayı ne kadar yüksekse sistem o kadar karmaşık olmaktadır. Kodun karmaşıklığını azaltmak ve test durumlarını belirlemek için kullanılır. Döngüsel karmaşıklığı hesaplamak için; yollar, düğümler ve bağlı olmayan yollar hesaplanır. Bu hesaplamalar formülde(1) yerine koyulur ve döngüsel karmaşıklık değeri elde edilir.

$$DK = Y - N + 2P \quad (1)$$

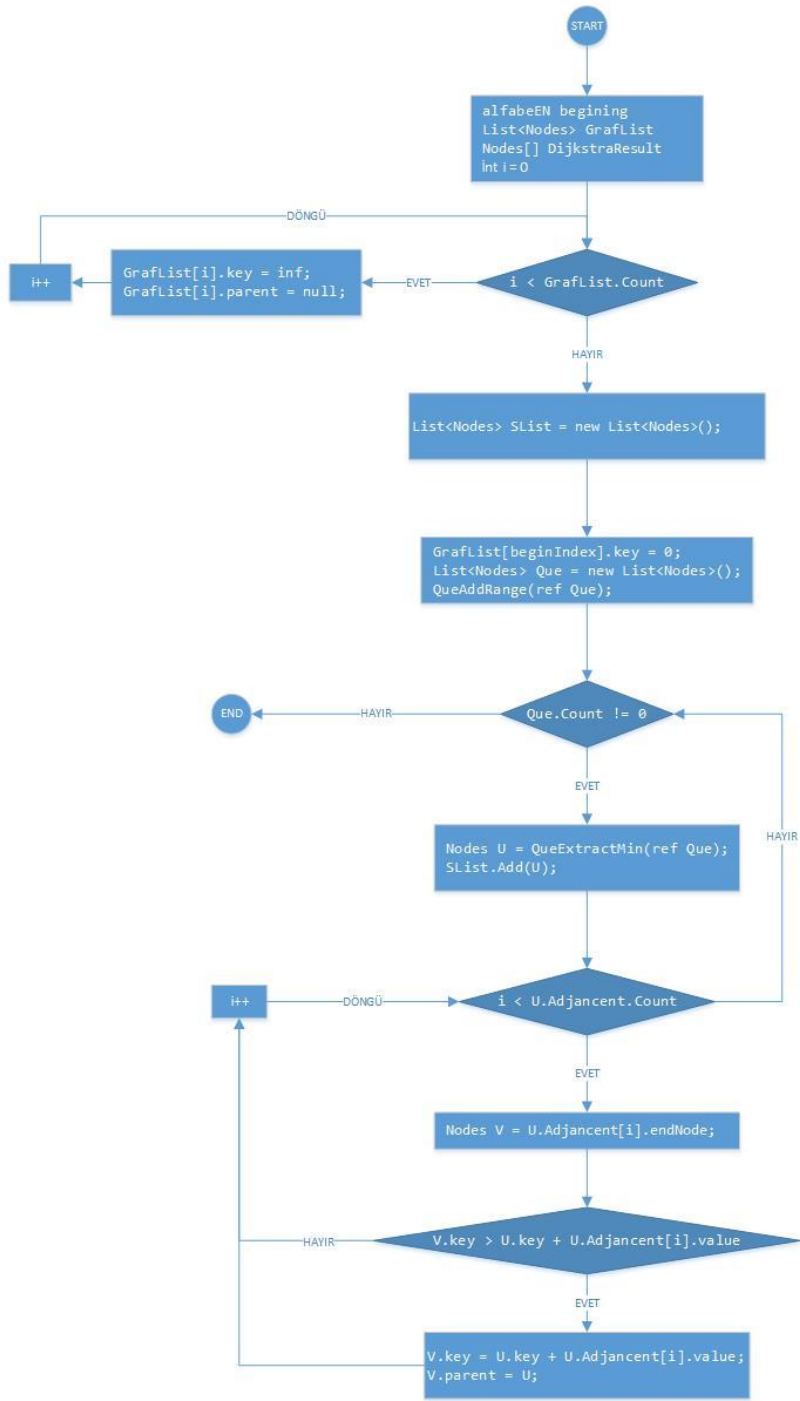
Yukarıdaki Formüle(1) Göre;

- Y = Toplam Yol Sayısı
- N = Toplam Düğüm Sayısı
- P = Toplam Bağlı Olmayan Yol Sayısı

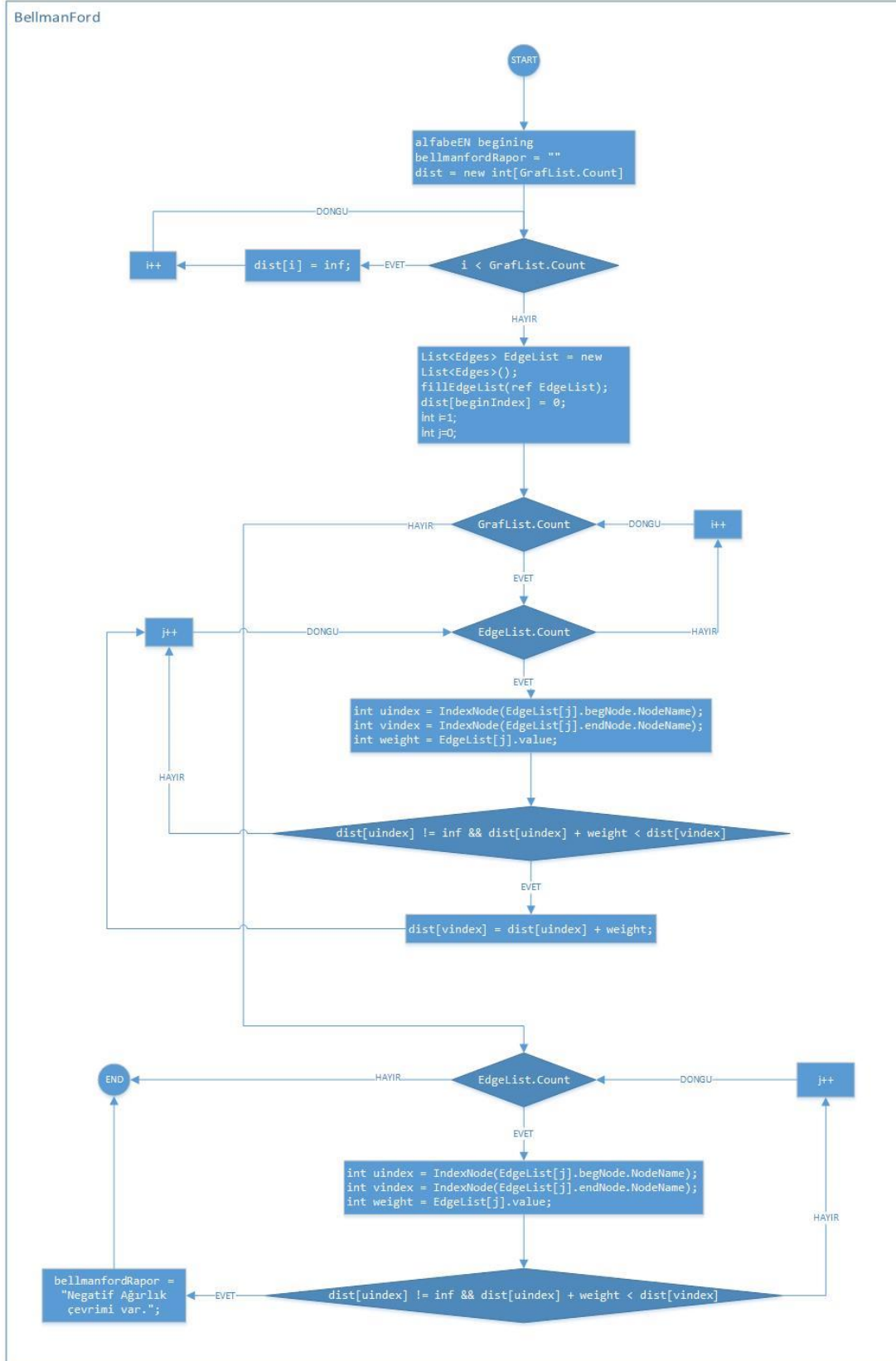
3. UML Etkinlik ve Sınıf Diyagramları



Şekil 3 Sınıf Diyagramı



Şekil 4 Dijkstra Algoritması Diyagramı



Şekil 5 Bellman-Ford Algoritması Diyagramı

4. Uygulama Süreci

3.1 Algoritma Analizi

3.1.1 Dijkstra Algoritması

Dijkstra algoritmasının analizini yaparken “Tokalaşma Kuramı(Handshaking Lemma)” kavramıyla karşılaşıyoruz. Herhangi bir grafta düğümlerin dereceleri toplamı, kenarların sayısının 2 katıdır [10]. Yani bir kenar için 2 düğüm(başlangıç ve son) gerekir. Bu kuramdan yola çıkarak tokalaşmanın bir tarafı şekil 3 ‘deki while döngüsünden geliyor yani V, tokalaşan diğer taraf ise kenarın bağlı olduğu V düğümünden geliyor. Bu durumda çarpım halinin dışında ifade edebiliriz.

```
// V -> Düğüm Sayısı, E -> Tokalaşma Kuramı
for (int i = 0; i < GrafList.Count; i++) // (A Bölümü) V kere
{
    GrafList[i].key = inf; // 1
    GrafList[i].parent = null; // 1
}
List<Nodes> SList = new List<Nodes>(); // (B Bölümü) 1
GrafList[beginIndex].key = 0; // (B Bölümü) 1
List<Nodes> Que = new List<Nodes>(); // (B Bölümü) 1
Que.AddRange(ref Que); // (B Bölümü) V kere
while (Que.Count != 0) // (C Bölümü) V kere
{
    Nodes U = Que.ExtractMin(ref Que); // LogV
    SList.Add(U); // 1
    for (int i = 0; i < U.Adjacent.Count; i++) // (D Bölümü) Tokalaşma Kuramı O(E)
    {
        Nodes V = U.Adjacent[i].endNode; // 1
        if (V.key > U.key + U.Adjacent[i].value) // 1
        {
            V.key = U.key + U.Adjacent[i].value; // 1
            V.parent = U; // 1
        }
    }
}
```

Şekil 6 Dijkstra Algoritması - C#

Şekil 3 ‘de C# ile kodlanmış Dijkstra Algoritmasını incelersek;

- A bölümünde, düğüm sayısı kadar dönen bir döngümüz var içerisinde 1 adımlık 2 adet kod satırı vardır. A bölümünden “2V” terimi geliyor,
- B bölümünde, 1 adımlık 3 adet kod satırı vardır ve 1 adet V adımlık döngümüz var. B bölümünde “V + 3” terimleri geliyor,
- C bölümünde, bütün düğümleri while döngüsü ile alıyoruz ve buradan “V” çarpanı elde ediyoruz. İçerisinde bulunan “LogV ve 1” terimleri ile dışardan hesaplayabileceğimiz “VLogV + V” terimleri geliyor,
- D bölümünde, Tokalaşma Kuramı ve içerisinde bulunan 4 satır ile “4E” elde etmiş oluruz.

Bütün bu verileri toplayıp formüle eklersek;

Adım 1)	$4E + V \cdot \log V + 4V + 3$	(2)
Adım 2)	$O(4E) + O(V \log V + 4V + 3)$	(3)
Adım 3)	$O(E) + O(V \log V)$	(4)
Adım 4)	$O(E + V \log V)$	(5)

Yukarıda formül haline getirilmiş adımları incelersek;

- Adım 1’de (2) , algoritmadan elde etmiş olduğumuz veriler bir bütün halinde ifade edildi,
- Adım 2’de (3), Tokalaşma Kuramı ve geri kalan bölüm ayrı notasyonlarda ifadeye alındı,
- Adım 3’de (4), katsayılardan ve düşük dereceli terimlerden kurtulduk,
- Adım 4’de (5), Büyük O notasyonlarının birleşim kümeleri alındı.

3.1.2 Bellman-Ford Algoritması

Şekil 4 ‘de C# ile kodlanmış Bellman-Ford Algoritmasını incelersek;

- A bölümünde, 1 adet “V” adımlık döngümüz, 1 adet “E” adımlık döngümüz ve 1 adımlık 3 adet kod satırımız bulunmakta. A bölümünden “E + V + 3” terimleri geliyor.
- B bölümünde, bütün düğümleri for döngüsü ile alıyoruz ve buradan “V” çarpanı elde ediyoruz.
- C bölümünde, bütün kenarları for döngüsü ile alıyoruz ve buradan “E” çarpanı elde ediyoruz. İçerisinde bulunan 5 adet kod satırı ile “5E” geliyor.
- B ve C Bölümlerini bir bütün halinde ifade edersek bu bölümlerden , “V*(5E)” geliyor.
- D bölümünde, negatif ağırlık çevrimi kontrol ediliyor. Eğer ki bu bölümde negatif ağırlık çevrimi yok ise, D bölümünden “4E” geliyor. Negatif ağırlık çevrimi en son adımda yani E. Adımda gelir ise, D bölümünden “4E + 2” geliyor.

Bütün bu verileri toplayıp formüle eklersek;

$$\text{Adım 1)} \quad V \cdot (5E) + 5E + V + 5 \quad (6)$$

$$\text{Adım 2)} \quad O(VE) \quad (7)$$

Yukarıda formül haline getirilmiş adımları incelersek;

- Adım 1’de (6) , algoritmadan elde etmiş olduğumuz veriler bir bütün halinde ifade edildi,
- Adım 2’de (7), katsayılardan ve düşük dereceli terimlerden kurtulduk.

```
// V -> Düğüm Sayısı, E -> Kenar Sayısı
dist = new int[GrafList.Count]; //(A Bölümü) 1
for (int i = 0; i < GrafList.Count; i++) //(A Bölümü) V kere
{
    dist[i] = inf; // 1
}
List<Edges> EdgeList = new List<Edges>(); //(A Bölümü) 1
fillEdgeList(ref EdgeList); //(A Bölümü) E
dist[beginIndex] = 0; //(A Bölümü) 1

for (int i = 1; i < GrafList.Count; i++) //(B Bölümü) V kere
{
    for (int j = 0; j < EdgeList.Count; j++) //(C Bölümü) E kere
    {
        int uindex = IndexNode(EdgeList[j].begNode.NodeName); // 1
        int vindex = IndexNode(EdgeList[j].endNode.NodeName); // 1
        int weight = EdgeList[j].value; // 1
        if (dist[uindex] != inf && dist[uindex] + weight < dist[vindex]) // 1
        {
            dist[vindex] = dist[uindex] + weight; // 1
        }
    }
}

for (int j = 0; j < EdgeList.Count; j++) //(D Bölümü) E kere
{
    int uindex = IndexNode(EdgeList[j].begNode.NodeName); // 1
    int vindex = IndexNode(EdgeList[j].endNode.NodeName); // 1
    int weight = EdgeList[j].value; // 1
    if (dist[uindex] != inf && dist[uindex] + weight < dist[vindex]) // 1
    {
        bellmanfordRapor = "Negatif Ağırlık çevrimi var."; // 1
        return; // 1
    }
}
```

Şekil 7 Bellman-Ford Algoritması - C#

3.2 Performans Testi

Performans testini uygularken dijkstra ve bellman-ford algoritmaları için aynı graf(Şekil 8) kullanılacaktır. Negatif ağırlık çevrimi olmayacaktır. Her bir Algoritma 10 defa çalıştırılıp, çıkan sonuçların ortalaması alınacaktır.

```
public void grafBuilder()
{
    GrafList = new List<Nodes>();
    AddNode(alfabeEN.A);
    AddNode(alfabeEN.B);
    AddNode(alfabeEN.C);
    AddNode(alfabeEN.D);
    AddNode(alfabeEN.E);
    AddNode(alfabeEN.F);
    AddNode(alfabeEN.G);
    AddNode(alfabeEN.H);
    AddNode(alfabeEN.I);
    AddNode(alfabeEN.J);
    AddNode(alfabeEN.K);
    AddNode(alfabeEN.L);

    AddEdge(alfabeEN.A, alfabeEN.E, 6);
    AddEdge(alfabeEN.A, alfabeEN.B, 2);
    AddEdge(alfabeEN.B, alfabeEN.D, 1);
    AddEdge(alfabeEN.B, alfabeEN.F, 2);
    AddEdge(alfabeEN.B, alfabeEN.I, 5);
    AddEdge(alfabeEN.C, alfabeEN.A, 1);
    AddEdge(alfabeEN.D, alfabeEN.C, 3);
    AddEdge(alfabeEN.E, alfabeEN.D, 4);
    AddEdge(alfabeEN.F, alfabeEN.E, 2);
    AddEdge(alfabeEN.F, alfabeEN.C, 1);
    AddEdge(alfabeEN.F, alfabeEN.J, 1);
    AddEdge(alfabeEN.G, alfabeEN.L, 5);
    AddEdge(alfabeEN.H, alfabeEN.G, 2);
    AddEdge(alfabeEN.I, alfabeEN.L, 2);
    AddEdge(alfabeEN.I, alfabeEN.H, 2);
    AddEdge(alfabeEN.J, alfabeEN.F, 1);
    AddEdge(alfabeEN.J, alfabeEN.G, 4);
    AddEdge(alfabeEN.K, alfabeEN.E, 1);
    AddEdge(alfabeEN.K, alfabeEN.J, 1);
    AddEdge(alfabeEN.K, alfabeEN.H, 1);
    AddEdge(alfabeEN.L, alfabeEN.K, 1);
}
```

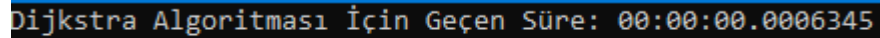
Şekil 8 grafBuilder Metodu ve Ağırlıklar

3.2.1 Dijkstra Algoritması

```
class Program
{
    0 başvuru
    static void Main(string[] args)
    {
        Graflar grf = new Graflar();
        grf.grafBuilder();
        Stopwatch stw = new Stopwatch();
        stw.Start();
        grf.Dijkstra(Graflar.alfabeEN.A);
        stw.Stop();
        Console.WriteLine("Dijkstra Algoritması İçin Geçen Süre: " + stw.Elapsed);
        stw.Reset();
        Console.ReadKey();
    }
}
```

Şekil 9 Dijkstra Algoritması İçin Performans Testi Algoritması

1. Performans Testi Sonucu

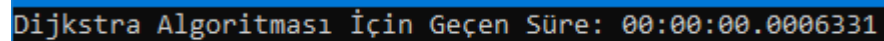


Dijkstra Algoritması İçin Geçen Süre: 00:00:00.0006345

Şekil 10 1. Performans Testi Sonucu

Geçen Süre: $6,345 \times 10^{-4}$ Milisaniye(ms)

2. Performans Testi Sonucu



Dijkstra Algoritması İçin Geçen Süre: 00:00:00.0006331

Şekil 11 2. Performans Testi Sonucu

Geçen Süre: $6,331 \times 10^{-4}$ Milisaniye(ms)

3. Performans Testi Sonucu

```
Dijkstra Algoritması İçin Geçen Süre: 00:00:00.0006239
```

Şekil 12 3. Performans Testi Sonucu

Geçen Süre: $6,239 \times 10^{-4}$ Milisaniye(ms)

4. Performans Testi Sonucu

```
Dijkstra Algoritması İçin Geçen Süre: 00:00:00.0006236
```

Şekil 13 4. Performans Testi Sonucu

Geçen Süre: $6,236 \times 10^{-4}$ Milisaniye(ms)

5. Performans Testi Sonucu

```
Dijkstra Algoritması İçin Geçen Süre: 00:00:00.0006377
```

Şekil 14 5. Performans Testi Sonucu

Geçen Süre: $6,377 \times 10^{-4}$ Milisaniye(ms)

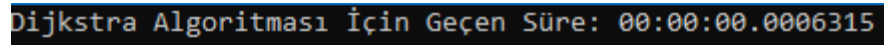
6. Performans Testi Sonucu

```
Dijkstra Algoritması İçin Geçen Süre: 00:00:00.0006327
```

Şekil 15 6. Performans Testi Sonucu

Geçen Süre: $6,327 \times 10^{-4}$ Milisaniye(ms)

7. Performans Testi Sonucu

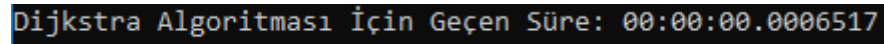


```
Dijkstra Algoritması İçin Geçen Süre: 00:00:00.0006315
```

Şekil 16 7. Performans Testi Sonucu

Geçen Süre: $6,315 \times 10^{-4}$ Milisaniye(ms)

8. Performans Testi Sonucu

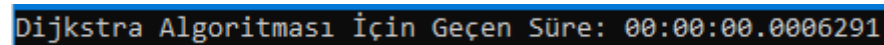


```
Dijkstra Algoritması İçin Geçen Süre: 00:00:00.0006517
```

Şekil 17 8. Performans Testi Sonucu

Geçen Süre: $6,517 \times 10^{-4}$ Milisaniye(ms)

9. Performans Testi Sonucu

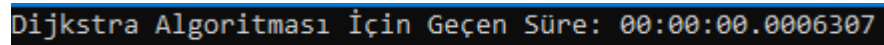


```
Dijkstra Algoritması İçin Geçen Süre: 00:00:00.0006291
```

Şekil 18 9. Performans Testi Sonucu

Geçen Süre: $6,291 \times 10^{-4}$ Milisaniye(ms)

10. Performans Testi Sonucu



```
Dijkstra Algoritması İçin Geçen Süre: 00:00:00.0006307
```

Şekil 19 10. Performans Testi Sonucu

Geçen Süre: $6,307 \times 10^{-4}$ Milisaniye(ms)

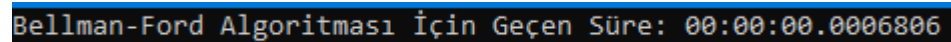
3.2.2 Bellman-Ford Algoritması

```
class Program
{
    0 başvuru
    static void Main(string[] args)
    {
        Graflar grf = new Graflar();
        grf.grafBuilder();
        Stopwatch stw = new Stopwatch();
        stw.Start();
        grf.BellmanFord(Graflar.alfabeEN.A);
        stw.Stop();
        Console.WriteLine("Bellman-Ford Algoritması İçin Geçen Süre: " + stw.Elapsed);
        stw.Reset();

        Console.ReadKey();
    }
}
```

Şekil 20 Dijkstra Algoritması İçin Performans Testi Algoritması

1. Performans Testi Sonucu

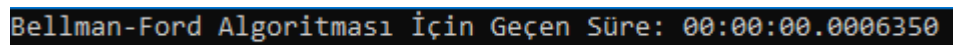


Bellman-Ford Algoritması İçin Geçen Süre: 00:00:00.0006806

Şekil 21 1. Performans Testi Sonucu

Geçen Süre: $6,806 \times 10^{-4}$ Milisaniye(ms)

2. Performans Testi Sonucu



Bellman-Ford Algoritması İçin Geçen Süre: 00:00:00.0006350

Şekil 22 2. Performans Testi Sonucu

Geçen Süre: $6,350 \times 10^{-4}$ Milisaniye(ms)

3. Performans Testi Sonucu

```
Bellman-Ford Algoritması İçin Geçen Süre: 00:00:00.0006254
```

Şekil 23 3. Performans Testi Sonucu

Geçen Süre: $6,254 \times 10^{-4}$ Milisaniye(ms)

4. Performans Testi Sonucu

```
Bellman-Ford Algoritması İçin Geçen Süre: 00:00:00.0006891
```

Şekil 24 4. Performans Testi Sonucu

Geçen Süre: $6,891 \times 10^{-4}$ Milisaniye(ms)

5. Performans Testi Sonucu

```
Bellman-Ford Algoritması İçin Geçen Süre: 00:00:00.0006219
```

Şekil 25 5. Performans Testi Sonucu

Geçen Süre: $6,219 \times 10^{-4}$ Milisaniye(ms)

6. Performans Testi Sonucu

```
Bellman-Ford Algoritması İçin Geçen Süre: 00:00:00.0006213
```

Şekil 26 6. Performans Testi Sonucu

Geçen Süre: $6,213 \times 10^{-4}$ Milisaniye(ms)

7. Performans Testi Sonucu

```
Bellman-Ford Algoritması İçin Geçen Süre: 00:00:00.0006243
```

Şekil 27 7. Performans Testi Sonucu

Geçen Süre: $6,243 \times 10^{-4}$ Milisaniye(ms)

8. Performans Testi Sonucu

```
Bellman-Ford Algoritması İçin Geçen Süre: 00:00:00.0006655
```

Şekil 28 8. Performans Testi Sonucu

Geçen Süre: $6,655 \times 10^{-4}$ Milisaniye(ms)

9. Performans Testi Sonucu

```
Bellman-Ford Algoritması İçin Geçen Süre: 00:00:00.0006723
```

Şekil 29 9. Performans Testi Sonucu

Geçen Süre: $6,723 \times 10^{-4}$ Milisaniye(ms)

10. Performans Testi Sonucu

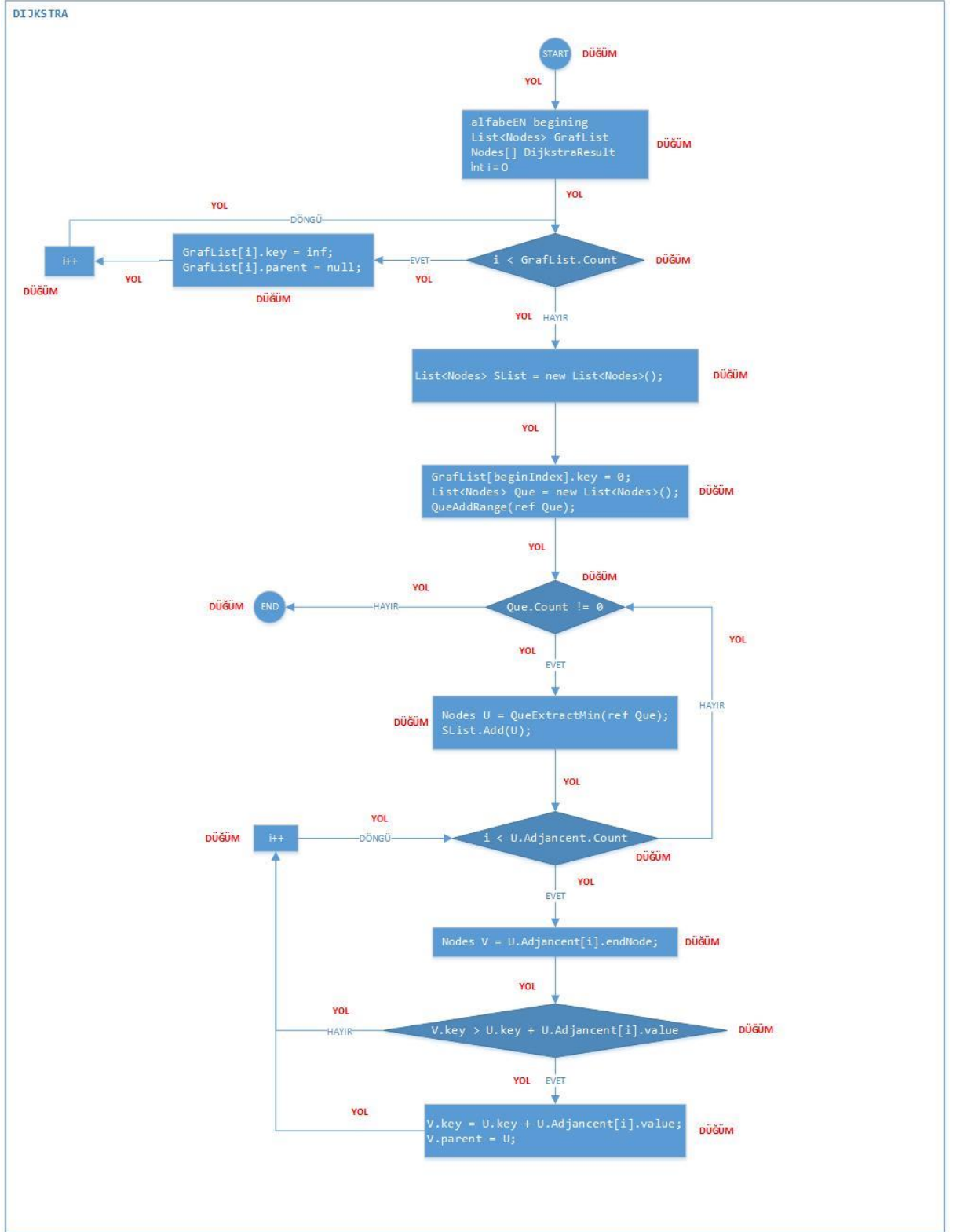
```
Bellman-Ford Algoritması İçin Geçen Süre: 00:00:00.0006488
```

Şekil 30 10. Performans Testi Sonucu

Geçen Süre: $6,488 \times 10^{-4}$ Milisaniye(ms)

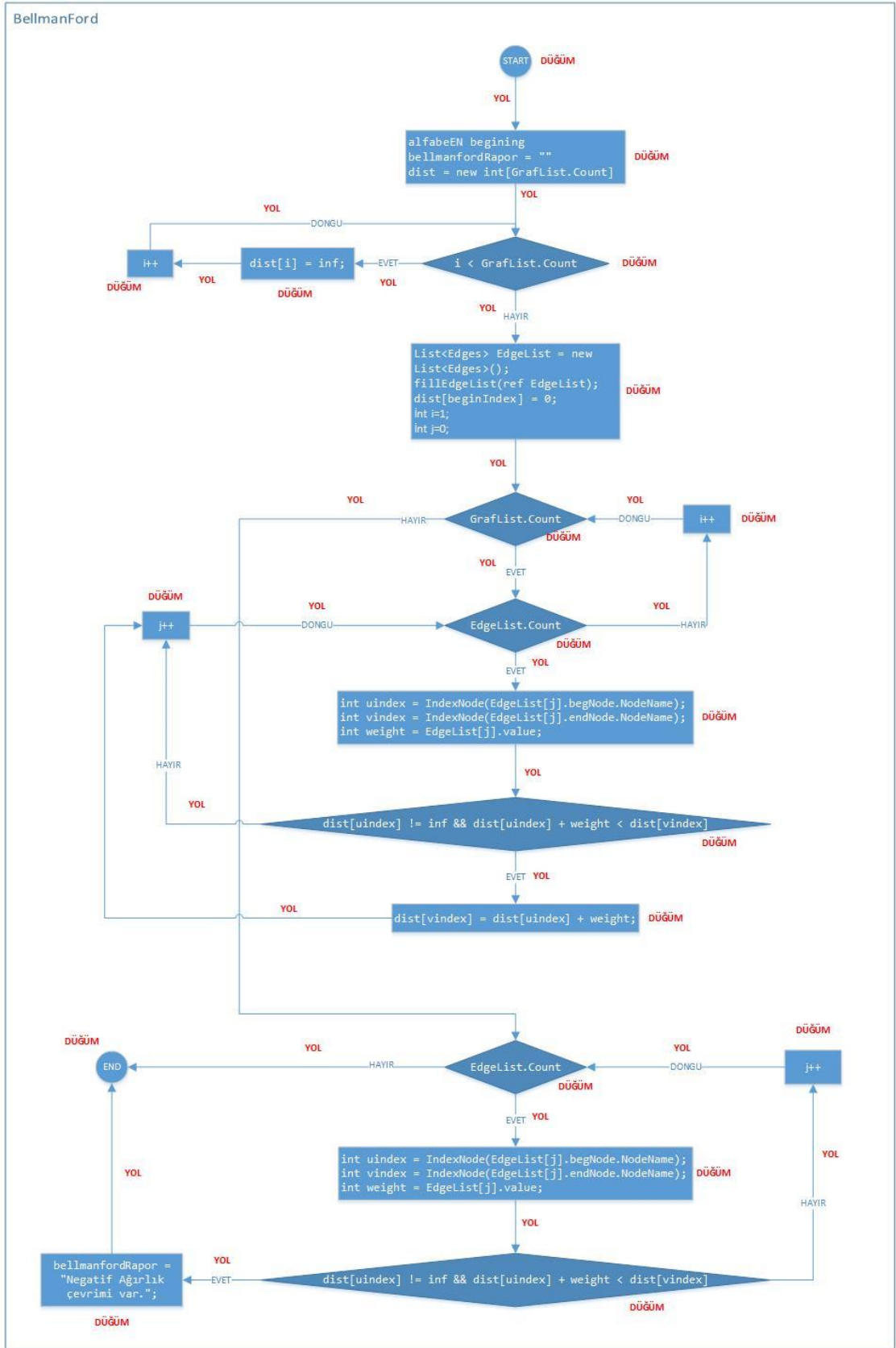
3.4 Döngüsel Karmaşıklık Testi

3.4.1 Dijkstra Algoritması



Şekil 31 Dijkstra Algoritması Döngüsel Karmaşıklık Hesaplaması

3.4.2 Bellman-Ford Algoritması



Şekil 32 Bellman-Ford Algoritması Döngüsel Karmaşıklık Hesaplaması

5. Uygulama Sonucu

5.1 Algoritma Analizi

5.1.1 Dijkstra Algoritması

Dijkstra Algoritmasının Büyük O notasyonu “ $E + V\log V$ ” olarak bulundu. $O(E + V\log V)$

5.1.2 Bellman-Ford Algoritması

Bellman-Ford Algoritmasının Büyük O notasyonu “ $V * E$ ” olarak bulundu. $O(V * E)$

5.1.3 Karşılaştırma Sonucu

Dijkstra ve Bellman-Ford algoritmalarının karşılaştırılmasında 12 adet düğüm(V) ve 21 adet kenar kullanılacaktır(Şekil 8).

Dijkstra algoritması için Tokalaşma Kuramı geçerli olacağı için;

- $E = 42$ (Tokalaşma Kuramı)
- $V = 12$
- $O(E + V\log V) = 42 + 12 * \log 12 = 42 + 12 * 3,6 = 42 + 43,2$
- Sonuç: 85,2 birim geliyor

Bellman-Ford algoritması için;

- $E = 21$
- $V = 12$
- $O(V * E) = 12 * 21$
- Sonuç: 252 birim geliyor.

Algoritma analizi karşılaştırması sonucunda Bellman-Ford algoritmasının notasyonu daha büyük olduğu için; Bellman-Ford algoritması, Dijkstra algoritmasından daha yavaş çalışmaktadır.

5.2 Performans Testi

5.2.1 Dijkstra Algoritması

Dijkstra algoritmasının performans sürelerini toplarsak;

$$\text{Adım 1)} \quad Y = 6,345 \times 10^{-4} + 6,331 \times 10^{-4} + \quad (8)$$

$$6,239 \times 10^{-4} + 6,236 \times 10^{-4} +$$

$$6,377 \times 10^{-4} + 6,327 \times 10^{-4} +$$

$$6,315 \times 10^{-4} + 6,517 \times 10^{-4} +$$

$$6,291 \times 10^{-4} + 6,307 \times 10^{-4}$$

$$\text{Adım 2)} \quad Y = 10^{-4} \times (6,345 + 6,331 + \quad (9)$$

$$6,239 + 6,236 + 6,377 + 6,327$$

$$+ 6,315 + 6,517 + 6,291 +$$

$$6,307)$$

$$\text{Adım 3)} \quad Y = 10^{-4} \times (63,285) \quad (10)$$

$$\text{Adım 4)} \quad Y = 63,285 \times 10^{-4} \quad (11)$$

Y teriminin ortalamasını alırsak;

Adım 1)	$Y = (63,285)/10 \times 10^{-4}$	(12)
Adım 2)	$Y = (6,3285) \times 10^{-4}$	(13)

Sonuç: 6,3285 Milisaniye(ms) geliyor.

5.2.2 Bellman-Ford Algoritması

Dijkstra algoritmasının performans sürelerini toplarsak;

$$\begin{aligned} \text{Adım 1)} \quad Y &= 6,806 \times 10^{-4} + 6,350 \times 10^{-4} + & (14) \\ & 6,254 \times 10^{-4} + 6,891 \times 10^{-4} + \\ & 6,219 \times 10^{-4} + 6,213 \times 10^{-4} + \\ & 6,243 \times 10^{-4} + 6,655 \times 10^{-4} + \\ & 6,723 \times 10^{-4} + 6,488 \times 10^{-4} \\ \text{Adım 2)} \quad Y &= 10^{-4} \times (6,806 + 6,350 + & (15) \\ & 6,254 + 6,891 + 6,219 + 6,213 \\ & + 6,243 + 6,655 + 6,723 + \\ & 6,488) \\ \text{Adım 3)} \quad Y &= 10^{-4} \times (64,842) & (16) \\ \text{Adım 4)} \quad Y &= 64,842 \times 10^{-4} & (17) \end{aligned}$$

Y teriminin ortalamasını alırsak;

Adım 1)	$Y = (64,842)/10 \times 10^{-4}$	(18)
Adım 2)	$Y = (6,4842) \times 10^{-4}$	(19)

Sonuç: 6,4842 Milisaniye(ms) geliyor.

5.2.3 Karşılaştırma Sonucu

Dijkstra ve Bellman-Ford algoritmalarının performans karşılaştırılmasında 12 adet düğüm(V) ve 21 adet kenar kullanıldığı için geçen süreler çok küçük kaydedilmiştir. Bilgisayar bilimlerinde 5binden düşük döngüsel adımlar göz ardı edilebilir bir performans farkı oluşturur. Fakat 2 algoritmanın performans karşılaştırmasında küçük de olsa farklılık gözüküyor.

Dijkstra algoritmasından ortalama 6,3285 ms ve Bellman-Ford algoritmasından ortalama 6,4842 ms elde ediyoruz.

	$6,4842 - 6,3285 = 0,1557$	(20)
--	----------------------------	------

Aradaki Farkı hesaplırsak(20), 0.1557 ms geliyor. Bu fark çok büyük bir süre olarak gözükme de düğüm ve kenar sayısı arttıkça aradaki farkta orantılı olarak artacaktır. Sonuç olarak Bellman-Ford algoritması, Dijkstra algoritmasından daha yavaştır.

5.3 Döngüsel Karmaşıklık Testi

5.3.1 Dijkstra Algoritması

Dijkstra Algoritmasının döngüsel karmaşıklık testini yaparken Şekil 31 'den yararlanacağız. Şekil 31'e göre;

- Toplam Yol Sayısı(Y) = 18 Adet
- Toplam Düğüm Sayısı(N) = 15 Adet
- Bağlı Olmayan Yol Sayısı = 1 Adet(End Düğümü)

Bu terimleri formülde yerlerine yazarsak;

$$\text{Döngüsel Karmaşıklık(DK)} = 18 - 15 + 2*(1)$$

Dijkstra Algoritmasının döngüsel karmaşıklığını "5" buluyoruz.

5.3.2 Bellman-Ford Algoritması

Bellman-Ford Algoritmasının döngüsel karmaşıklık testini yaparken Şekil 32 'den yararlanacağız. Şekil 32'e göre;

- Toplam Yol Sayısı(Y) = 24 Adet
- Toplam Düğüm Sayısı(N) = 19 Adet
- Bağlı Olmayan Yol Sayısı = 1 Adet(End Düğümü)

Bu terimleri formülde yerlerine yazarsak;

$$\text{Döngüsel Karmaşıklık(DK)} = 24 - 19 + 2*(1)$$

Bellman-Ford Algoritmasının döngüsel karmaşıklığını "7" buluyoruz.

5.3.3 Karşılaştırma Sonucu

Dijkstra ve Bellman-Ford algoritmalarının döngüsel karmaşıklık testi sonucunda, Dijkstra algoritmasının 5 ve Bellman-Ford algoritmasının 7 olduğunu görüyoruz. Sonuç olarak Bellman-Ford algoritması, dijkstra algoritmasından daha fazla puana sahip olduğundan; Bellman-Ford algoritması, dijkstra algoritmasından daha karmaşıktır diyebiliriz.

6. Sonuç

Bu çalışmanın sonucunda, Bellman-Ford algoritması Dijkstra algoritmasına göre; algoritma analizindeki, performans testindeki ve döngüsel karmaşıklık testindeki verilere göre daha yavaş ve daha karmaşık çıkmıştır. Fakat pratikte Dijkstra algoritması negatif ağırlıklı kenarlarda hata verirken, Bellman-Ford Algoritması negatif ağırlık çevrimlerini rapor edebiliyor.

7. Not

2021 Yılı'nın kasım ayında Yazılım Kalite Güvencesi ve Testi dersinde gündüz ve gece grubu öğrencilerine Jenkins kurulumu ve Jenkins ile test teknikleri gösterilmiştir. Jenkins kurulumu internette bulunabilse de, Jenkins test teknikleri internette doğru sonuç üretmemektedir. Bu nedenden dolayı derste çalışan ve düzgünce işleyen test tekniği tarafımdan mühendislik esaslarına özel geliştirilmiştir. Derste gösterilen test tekniğinin içerisinde performans testi'de bulunmaktadır.

8. Kaynakça

- [1] https://web.karabuk.edu.tr/ismail.karas/759/Sunu1_esas.pdf ,(Erişim Tarihi 18.12.2021).
- [2] [https://tr.wikipedia.org/wiki/Graf_\(matematik\)](https://tr.wikipedia.org/wiki/Graf_(matematik)) ,(Erişim Tarihi 18.12.2021).
- [3] <https://www.halildurmus.com/2020/10/10/bellman-ford-algoritmasi> ,(Erişim Tarihi 18.12.2021).
- [4] <https://m-caliskanyurek.medium.com/en-k%C4%B1sa-yol-problemi-shortest-path-problem-dijkstra-algoritmas%C4%B1-ve-c-kodu-792bc51b318d> ,(Erişim Tarihi 18.12.2021).
- [5] E. Tanyıldızı, Algoritma Analizi Dersi, Algoritma Analizi Giriş.pdf, (Erişim Tarihi 18.12.2021).
- [6] <https://binbiriz.com/blog/yazilim-performansi-nedir> ,(Erişim Tarihi 18.12.2021).
- [8] R. Daş, Yazılım Kalite Güvencesi ve Testi, 4.Bölüm TestStratejileri.pdf, (Erişim Tarihi 19.12.2021).
- [9] R. Daş ve T.B. Alakuş, Yazılım Kalite Güvence ve Testi, 9.Bölüm Yazılım Metrikleri 2.pdf, (Erişim Tarihi 19.12.2021).
- [10] https://www.youtube.com/watch?v=_ZSE0wVpBBo ,(Erişim Tarihi 20.12.2021).