GEZGİN KARGO PROBLEMİ

Taha Batuhan TÜRK – Müberra ÇELİK
180202007 - 180202102

Bilgisayar Mühendisliği Bölümü
Kocaeli Üniversitesi
180202007@kocaeli.edu.tr - 180202102@kocaeli.edu.tr

***Özet*–Proje , başlangıç şehri Kocaeli olan bir kargo firmasının siparişlerini en kısa yoldan yerlerine ulaştırmayı hedefleyen , nesneye yönelik programlama mantığına dayanarak, Java dilinde yazılmış bir masaüstü uygulamasıdır.**

Anahtar Kelimeler : nesneye yönelik programlama , Java , veri yapıları , grafik arayüz, graph, dijkstra, NP-Hard problem.

I. GİRİŞ

Program Türkiye'deki 81 ilin ve bu illerin komşularının birbirlerine olan uzaklıklarını tutan "txt" dosyasından okuma yaparak verileri bir listeye aktarır. Bu verileri kullanarak her ilin birbirleriyle olan en kısa mesafelerini ve rotalarını alternatifleri ile birlikte dijkstra algoritması yardımı ile kendi geliştirdiğimiz rota algoritmasını kullanarak **"35 ± 10"** saniyenin altında hesaplar. Hesaplanan bu rotaları en az maliyetliden başlanarak sıralanacak şekilde ara yüz kullanılarak ekranda yazdırır ve bir harita üzerinde çizdirir.

II. YÖNTEM

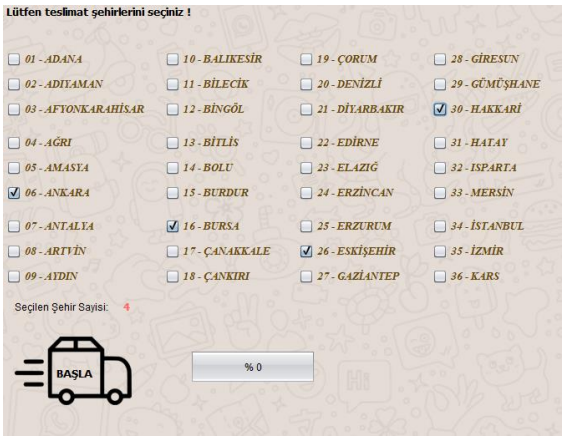
Program NetBeans ortamında Java programlama dili ile yazılmıştır. Projede Inheritance , Polymorphism, Veri Yapıları, Graph Teorisi ve Swing kavramları kullanılmıştır. Kodda Constructor (Yapıcı) metotlar tanımlanmıştır. Kodun içeriği metotlar , classlar , döngüler ve if – else koşul durumlarından oluşmaktadır.

III. DENEYSEL SONUÇLAR

Kodu ilk çalıştırdığımızda kullanıcıyı adından Şekil.1 deki gibi teslimat yapılacak şehirleri seçebileceği 81 il ve seçtiği bu şehirlerin rotalarının hesaplanmasını bir buton yardımı ile başlatarak , güzergahlarını harita üzerinde çizdirebilecek bir ara yüz karşılar.

Şekil.1 Kullanıcının karşısına çıkan arayüz ekranı

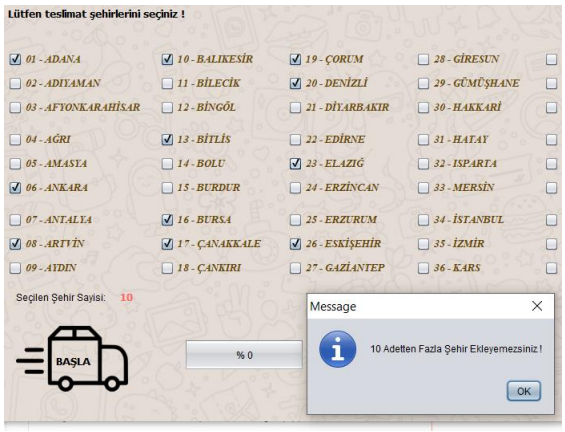
Kullanıcı checkBox yardımı ile şekil 2. deki gibi en az 3 ve en çok 10 şehir seçebilmek koşulu ile teslimat yapmak istediği şehirleri seçer. Seçilen şehirlerin sayısı kullanıcı şehir seçtikçe artar ve ara yüzde gösterilir.



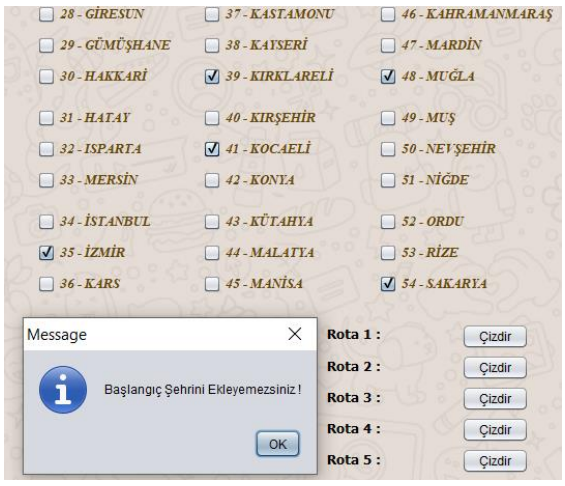
Şekil.2 Kullanıcının seçtiği şehirler

3 taneden daha az şehir seçilmiş ise "Başla" butonuna tıklandıktan sonra (Şekil.2.a),eğer 10 taneden daha fazla ve başlangıç noktası olan şehir seçilmiş ise daha butona tıklanmadan (Şekil.2.b) ve (Şekil.2.c) deki gibi program kullanıcıya bir hata mesajı verir.

Şekil2.a 3 adetten az şehir seçildiğinde ekrana gelen uyarı mesajı

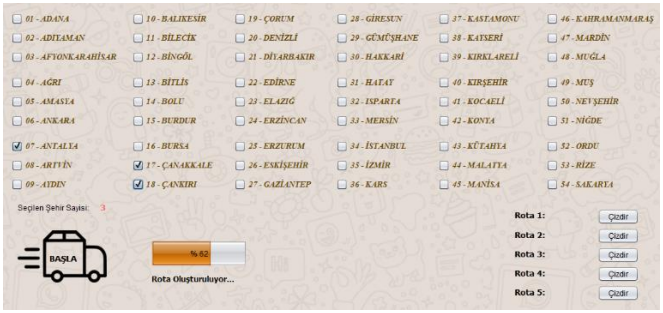


Şekil.2.b 10 adetten fazla şehir seçildiğinde ekrana gelen uyarı mesajı



Şekil.2.c Başlangıç noktası olan Kocaeli şehri seçildiğinde ekrana gelen uyarı mesajı

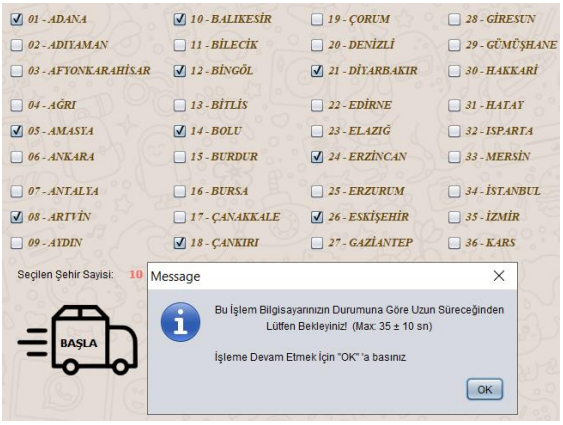
Kullanıcı koşullara uygun teslimat şehirlerini seçtikten sonra "başla" butonuna tıklar ve program alternatif en kısa yolların mesafelerini ve rotalarını hesaplamaya başlar. Hesaplama yapılırken "Progress bar" dolmaya başlar (Şekil.3), %100 olduğunda ise hesaplama işlemi sona ererek alternatif en kısa 5 yolun mesafelerini ekrana yazdırır. (Şekil.4)



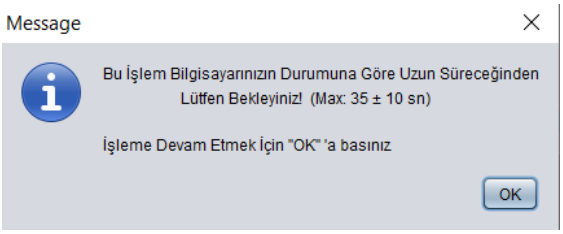
Şekil.3 Progress Bar'ın dolması.

Şekil.4 Hesaplamalar tamamlandıktan sonra rotaların ekranda listelenmesi.

Teslimat yapılacak şehirler 10 adet seçilmiş ise; (Şekil.5) ve (Şekil.5.a) daki gibi olmak üzere kullanıcının bilgisayarına bağlı olarak rotaların hesaplanma süresi değişeceğinden maksimum 35 ± 10 saniye beklenmesi için kullanıcıya bir uyarı mesajı verilir.



Şekil.5 10 tane şehir seçilmesi ve ekrana uyarı mesajının gelmesi

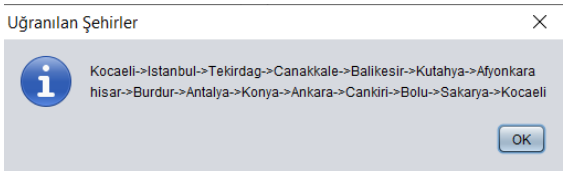


Şekil.5.a Gelen uyarı mesajının içeriği

Kullanıcı çizdirmek istediği rotayı "Çizdir" butonuna tıklayarak harita üzerinde görüntüler. Seçilen teslimat şehirleri kırmızı daire ile belirtilmiştir. (Şekil.6)

Şekil.6 Çizdirilen rotanın harita üzerindeki görüntüsü.

İstenilen rota çizdirildikten sonra "Rotayı Göster" butonuna tıklanarak oluşan rota görüntülenir. (Şekil.6.a)



Şekil.6.a Çizdirilen güzergahın gösterilmesi

Son olarak kullanıcı uygulamayı kapatmadan yeni bir simülasyon başlatabilir. Ara yüzde istenilen şehirler eklenilip çıkarıldıktan sonra tekrar "Başla" butonuna tıklanarak istenilen yeni rotalar hesaplanır.

IV. YALANCI KOD

Parametre olarak aldığı şehir ile graphtaki diğer tüm şehirler arasındaki en kısa mesafeyi hesaplayan Dijkstra fonksiyonunun kod parçacığı :

```
public void sehirDijkstra(Vertex sehir) {
    Vertex aktifSehir=sehir;
    for (int i = 0; i < sehirler.size(); i++) {
        for (int i = 0; i < sehirler.size(); i++) {
            if(aktif sehirden diğer tüm şehirlere olan mesafeleri, yeni mesafe eski mesafeden küçükse ){
                sehrin mesafesini güncelle
            }
        }
        aktifSehiri kullanıldı yap. Kullanılmayan,
        en kısa mesafeye sahip sehri aktifsehir yap.
    }
}
```

Dijkstra fonksiyonundan hemen sonra çağrılan ve aktif değerleri bellekte tutmak için minSehirler2'ye ekleyen guzergahEkle fonksiyonunun kod parçacığı:

```
public void guzergahEkle(Vertex sehir, int index)
{
for (int i = 0; i < sehirler.size(); i++) {
    ArrayList<Vertex> cumle2;
    Vertex guzergah = sehirler.get(i)
    cumle2.add(new Vertex(guzergah.minMesafe);
```

```
    while (guzergah != sehirler.get(sehir.plaka - 1))
{
        cumle2.add(guzergah);
        guzergah = guzergah.parent;
      }
      cumle2.add(sehir);
      minSehirler2.add(new
ArrayList<ArrayList<Vertex>>());
      minSehirler2.get(index).add(cumle2);
   }
}
```

Dosyadan okuma yapıp "sehirler' graphını oluşturan fonksiyonun kod parçacığı.

```
public void dosyaOkuma() throws
FileNotFoundException, IOException {
      while (scanner.hasNextLine()) {
          komsuluk.txt'den oku.
          sehirler graphına ekle.
      }
}
```

public void rotaOlusturma() {

Seçilen şehirler için 35 saniyenin altında alternatif rotalar oluşturan, tamamiyle kendi ürettiğimiz algoritmanın kod parçacığı:

```
ArrayList<Integer> a = new ArrayList<>();
      int sayi = gidilecekSehirler.size() - 1;//kocaeli
dahil olmadığı için 1 çıkarttık.
      String[] s = new String[fak(sayi)];//tüm rotaları
s dizisinde tutuyoruz.
      for (int i = 1; i <= sayi; i++) {
         a.add(i);
      }
      for (int i = 0; i < fak(sayi); i++) {//güzergahların
başına kocaeliyi ekler
         s[i] = "";
         s[i] += 0;
      }
      for (int i = 1; i <= sayi; i++) {
         for (int j = 0; j < fak(sayi - 1); j++) {
            if (i == 10) {
               s[j + (fak(sayi - 1) * (i - 1))] += ":";
            } else {
               s[j + (fak(sayi - 1) * (i - 1))] += i;
            }
         }       }
      int sayi3 = sayi - 1;
      int sayi2 = sayi - 2;
      int m = 999;
      int n = 0;
   ArrayList<Integer>iceren=newArrayList<>();
      for (int i = 0; i < sayi - 1; i++) {//sütunlar için
         for (int j = n; j < fak(sayi);) {//satırlar için
            for (int k = 0; k < fak(sayi2); k++) {
               for (int u = 1; u <= sayi; u++) {
                  if (s[j].contains(String.valueOf(u)))
                     iceren.add(u);
                  }
               }
               if (m == 999) {
                  for (int o = 1; o <= sayi; o++) {
                     if (iceren.contains(o) == false) {
                        m = o;
                        iceren.add(o);
                        break;
                     }
                  }
               }
               if (m == 10) {
                  s[j] += ":";      //10. şehir için ascii
tablodan 9 dan sonraki değeri aldık.
               } else { s[j] += m;    }
               j++;
               n++;
               if (n % fak(sayi3) == 0) {
                  iceren.clear();
               }
            }
            m = 999;
         }
         n = 0;
         sayi2 -= 1;
         sayi3 -= 1;
      }
      for (int i = 0; i < fak(sayi); i++) {//güzergahların
sonuna kocaeliyi ekler.
         s[i] += 0;
 }
```

İskelet rotadan alternatif rotaları türeten kod parçacığı:

```
index1=0;//gidilecek sehirlerin boyutu-1'e kadar gidecek
for (int i = 1; i < rotaIskelet.size() - 2; i++) {
  aktifSehir = rotaIskelet.get(i);
   if (aktifSehirin komşusunda gidilecekSehir var mı)
   {
      if (index1 küçük mü gidilecek şehirlerin sayısından)
      {
         ++index1;
       }
       continue;
   } else {
       for (int j = 0; j < aktifSehir.komsular.size(); j++)
      {
          for (int k = 0; k < aktifSehir.komsular.get(j).hedefVertex.komsular.size(); k++)  {
              if(aktifSehirin komşusunun komşusunda aktifSehrin iki sonraki şehri var mı)
              {
                  iskeletin kopyasında iki şehrin arasındaki mevcut şehri sil ve yerine alternatif şehri koy.
                  Rotanın mesafelerini ve gidişatını güncelle
                  alternatifRotalara iskelet rotadan türetilip güncellenen rotayı ekle.
              }
          }
      }
   }
}
```

Oluşturulan alternatif rotaları en az maliyetliden başlayarak sıralayan kod parçacığı:

```
for (int i = 1; i < alternatifRotalar.size(); i++) {     // alternatif rotaları en az maaliyetli olacak şekilde sıralar.
     for (int j = i; j < alternatifRotalar.size(); j++) {
        if (alternatifRotalar.get(i).get(0).minMesafe > alternatifRotalar.get(j).get(0).minMesafe) {
              ArrayList<Vertex> tmp = (ArrayList<Vertex>) alternatifRotalar.get(i).clone();
              alternatifRotalar.set(i, alternatifRotalar.get(j));
              alternatifRotalar.set(j, tmp);
        }
     }
}
```

V. SONUÇ

Projeyi geliştirirken uzun bir süre piyasada bulunan en kısa yol geliştirme algoritmalarından Greedy, Arı kolonisi, A*, genetik algoritmasını ve Dijkstra gibi pek çok algoritmayı inceledik ve bu algoritmalar içerisinden Dijkstra algoritması hariç diğer algoritmaları kullanarak tam olarak istenilen sonucu alamayacağımızı fark ettik. Bu yüzden içlerinden en uygun bulduğumuz Dijkstra algoritmasından faydalanarak kendi rota bulma algoritmamızı geliştirdik.

Bu algoritmayı geliştirirken random sınıfını ve pek çok hazır fonksiyon kullanmayı denedik. Fakat bunlar rotaların 40 saniyenin altında hesaplanabilmesi için yeterli olmadı. Bu yüzden kendi algoritmamızı geliştirmeye karar verdik.

Kendi geliştirdiğimiz algoritma sayesinde, diğer rota hesaplama algoritmaları ile 2 saatin üzerinde sürecek işlemleri 35-40 saniye gibi kısa süreler içerisinde minimim işlem gücü ile gerçekleştirebildik. Projede en çok zorlandığımız kısım bu algoritmayı geliştirmek oldu.

Her şeye rağmen bu algoritma da çok fazla işlem gücü gerektirdiğinden 9 şehirden sonrası için Ara yüzde bu işlem gerçekleşirken ara yüzümüz kitleniyor ve ara yüz üzerinde işlem gerçekleştiremiyoruz. Bu sorunu çözmek için "Thread", "SwingWorker" gibi pek çok sınıfı ve fonksiyonlarını kullanmayı denedik fakat başarılı olamadık. Bu yüzden hesaplamalar başlamadan önce kullanıcıya panel üzerinden bilgi verdik.

Önce kullanacağımız fonksiyonları bir sınıfta toplayıp daha sonra ara yüz üzerinde bu sınıftan bir nesne oluşturup bu fonksiyonları kullandık. Bu sayede ara yüzümüz daha stabil ve çalıştırılabilir oldu. Ara yüzde swing materyalleri kullanarak programımızı tamamladık.

VI. KAYNAKÇA

[1] Algoritma Uzmanı (n.d.). Retrieved from https://www.algoritmauzmani.com/algoritmalar/dijkstra-algoritmasi-nedir-dijkstra-ornekli-anlatim-c-kodu/

[2] Kodlab JAVA-10 Mehmet Kirazlı – Sezer Tanrıverdioğlu

[3] T.C Ulaştırma ve Altyapı Bakanlığı (n.d). Retrieved from https://www.kgm.gov.tr/Sayfalar/KGM/SiteTr/Root/Uzakliklar.aspx

[4] Yazılım Bilimi Java programlama

[5] Message Dialogs in Java (GUI) (n.d.). Retrieved from https://www.geeksforgeeks.org/message-dialogs-java-gui/