Programlama Laboratuvarı Gezgin Kargo Problemi

Özge POYRAZ
Kocaeli Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği
180202025@kocaeli.edu.tr

Burak Can TEMİZEL
Kocaeli Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği
180202024@kocaeli.edu.tr

***Özet*—Bu çalışmada en kısa yol algoritması olan Dijkstra Algoritması ve sezgisel bir yaklaşım olan Genetik algoritma ile belirtilen gezgin kargo problemini çözüme ulaştırdık. Bu algoritmaların implementasyonunda Java programlama dilini ve Collections sınıfındaki veri yapılarını kullandık. Ayrıca kullanıcı için dinamik bir Kullanıcı Arayüzü oluşturduk.**

Anahtar kelimeler— En kısa yol Algoritması, Dijkstra Algoritması, Sezgisel Algoritma, Genetik Algoritma, Graf Yapısı, Veri Yapıları, Kullanıcı Arayüzü

I. GIRIŞ

Gerçekleştirdiğimiz proje, temel olarak bir graf problemi ile ilgiliydi. Genel olarak birtakım kurallar çerçevesinde bir rotanın çizilmesini gerektiriyordu. Bu rotada seyahat, bir başlangıç şehrinden başlanarak tekrar aynı şehirde bitirilmeli ve diğer tüm hedef şehirlere de uğranmalıydı.

İlk etapta problem direkt olarak en kısa yol problemi gibi gözükse de aslında biraz daha kompleks bir haldeydi. Öncelikle hareketler komşu şehirler üzerinden gerçekleşmekte ve hedef şehir girdilerimiz on şehre kadar çıkabilmekteydi. Çoğu zaman bu şehirlerin komşu olmayacağını da düşünürsek, harekete başladığımız şehirden yapacağımız komşu seçimi ve daha sonraki hareketler oldukça karmaşık bir kombinasyon problemine dönüşüyordu. Her komşu şehir için yapılması gereken hesaplama oldukça hızlı bir şekilde artıyordu. Kombinasyonlar sıralı bir şekilde denense bile polinom zamanda çözülemeyecek bir hal almaktaydı. Bir sonraki adım olarak buradaki problemi basite indirgedik. Elimizde hedef şehirler mevcuttu. Eğer başlangıç şehrimizden başlayarak bu hedef şehirlerin permütasyonlarını alırsak ve tekrar başlangıç şehrimize dönersek, yani dairesel bir permütasyon oluştururursak, problemin artış miktarını tüm şehirlerde gezinmeye göre azaltabilirdik. Hedef şehirler arasındaki yolları da en kısa yol algoritmasından hesaplayarak yerine yerleştirip rota elde edebilirdik. Tüm şehirlerde gezinmek kadar olmasa da yine burada on şehrin permütasyonu oldukça fazlalaşmakta bu da tekrardan problemi kompleks bir hale sokmaktaydı. Öncelikle problemi daha iyi anlamak ve bir takım deneysel sonuçlar elde etmek için birkaç tane ön prototip hazırladık. İlk olarak on şehrin permütasyonlarını rastgele bir şekilde kaba kuvvet yöntemiyle elde edip mesafeleri birbiriyle kıyaslayarak bazı rotalar elde ettik. Düşük girdilerde ihtimaller daha az olduğu için kaba kuvvet algoritması, sonucu hızlı verse de faktöriyel artışı arttıran her bir girdi için sonucu doğru bir şekilde hesaplaması sadece şansa kalmaktaydı. Daha sonra ikinci bir yöntem olarak yollar hesaplanmadan önce tüm permütasyonları listeleyip tek tek deneme yöntemini inceledik. Bu şekilde şans faktörü bir nebze azaldı çünkü olası tüm ihtimaller tek tek denenecekti. Fakat bu da çalışma süresini yüksek girdiler için oldukça arttırmakta ve bizi optimum süreden uzaklaştırmaktaydı. Bu noktadan sonra problemin neden polinom zamanda çözülemediği neden sezgisel yöntemlerin kullanıldığı kafamızda daha net bir hale bürünmüştü. Bu problemi çözmek için kullanacağımız sezgisel algoritmayı, Genetik algoritma olarak belirledik. Genetik algoritma gerçek hayatta gözlemleyebildiğimiz bilimsel bir modelin bilgisayar ortamına aktarılması gibiydi ve fark ettik ki aslında problemimiz de bu yapıya oldukça uyuyordu. Bir sürü rastgele yolu kullanarak en iyi yola varabilirdik. Genetik algoritma yardımı ile hedef şehirlerimizi sıralayacak ve diğer yardımcı yapılarla birlikte yolları ortaya çıkaracaktık. Daha sonrasında prototip aşamasını sonlandırarak geliştirme aşamasına geçtik.

II. YÖNTEMLER VE PROGRAM MIMARISI

Bu kısımda programın farklı özelliklerini oluşturmak için kullandığımız araçlar ve yöntemler üzerinde durularak ayrıntılı olarak bilgi verilecektir. Program mimarisi daha detaylı bir şekilde açıklanacaktır.

A. En Kısa Yol Algoritması

Her ne kadar hedef şehirleri sıralamaktan bahsetsek de aslında burada hedef şehirler arasındaki en kısa yollar sıralanmaktaydı. Bu şekilde yollar oluşturulabilecekti. En kısa yolları bulmak için çok fazla algoritma buluyordu. Biz prototip aşamasında basit uygulanabilir olmasından dolayı Dijkstra algoritmasını kullanmaya karar verdik daha sonraki aşamalarda da değiştirme gereği duymadık. Floyd-Warshall algoritması ise üzerinde durduğumuz bir diğer algoritmaydı. Dijkstra algoritmasının implementasyonunu gerçekleştirmenin iki yöntemi vardı. Ya matris yapısını kullanacaktık ve modellerimizi matrisler biçiminde gerçekleştirecektik ya da liste yapılarıyla gerçekleştirecektik. Biz burada listelerle modellemeyi tercih ettik çünkü problemi nesneye yönelik problem paradigması içerisinde küçük parçalara bölerek ilerleme fikri oldukça mantıklıydı. Ayrıca gerçek hayata yakın modellemeler programlama aşamasında işimizi bir hayli kolaylaştırmaktaydı. Matematiksel bir halden ziyade daha net ve somut bir şekilde gerçekleştirebilmekteydik. Veri yapıları kısmında bunlara daha ayrıntılı değineceğiz. Başlangıçta en kısa yol algoritmasını kullanma amacımız bahsettiğimiz gibi hedef şehirler arasındaki en kısa yolları elde etmekti. Bu şekilde de kullandık. Buradan aldığımız bilgileri çeşitli veri yapılarında tekrar tekrar kullanmak için sakladık. Fakat bizden beklenen bir takım matris çıktıları için tüm şehirler arasındaki uzaklık hesaplamalarını yapmamız ve ilişkisel listelerimiz üzerinden göstermemiz gerekliydi. Burada performans olarak Floyd-Warshall algoritmasının daha ön plana çıktığını biliyorduk fakat ikinci bir implementasyon gerçekleştirmedik. Bu daha sonra çözeceğimiz bazı küçük bellek kayıplarına yol açtı. Bu şekilde program genelinde iki farklı amaçla en kısa yol algoritmasını kullandık. Baeldung[1] sitesinde bulunan implementasyondan referans aldık. Dijkstra algoritması iki hedef arasında en kısa yol bulmaktaydı. Uygun veri

yapılarını kullanarak ve uygun ana fonksiyonu oluşturarak iki kullanımda da ihtiyacımızı görecek hale getirdik. Birden fazla şehir arasında hesaplamaları gerçekleştirdik.

B. Veri Yapıları

Programda kullandığımız Dijkstra algoritması, en kısa yol algoritması ve Genetik algoritma implementasyonlarında çok çeşitli veri yapıları kullandık. Genel olarak Java Collection sınıfına ait yapılarla modellemelerimizi gerçekleştirdiğimizi söylemiştik. Bu kısımda burayı biraz daha detaylandırıp yaşadığımız bazı problemlere de değineceğiz. Öncelikle bilgileri saklamak için çoğu yerde liste yapılarını kullandık. Bunun en büyük artısı daha nesneye yönelik bir anlayış içinde geliştirmeye devam edebilme imkânı sağlaması. Listelerimiz sayesinde verilere gerçek nesneleri kontrol eder gibi hakimdik fakat listelerin iç içe girmesiyle çok sayıda karmaşık yapılar oluştu kimi zaman bu yapıların hiyerarşisini kontrol etmek zorlaştı kimi zaman ise fazladan bellek tüketimine sebep oldu. Bunu engellemek için nesnelerin referanslarını birden fazla yerde geçirmemeye ve Garbage Collector tarafından toplanmasını sağlamaya odaklandık. Bellek tüketimi liste işlemleri sırasında anlık olarak artmakta ve bir müddet azalmamaktaydı. Bunu engellemek için bazı noktalarda Garbage Collectoru kendimiz tetikleyerek referansını kaybetmiş listeleri sildik. Genetik algoritmada da aynı şekilde listeleri kullandık. Bu kısımda amacımız hedef şehirleri uygunluk seviyelerine göre çaprazlama işlemi uygulayarak yeni nesiller elde etmek olduğu için onları temsilen integer değerler kullandık ve integer listelerle yapıyı kurduk. Standart dizileri kullanmadık.

C. Kullanıcı Arayüzü

Arayüzü Java Awt ve Swing kullanarak oluşturduk. Arayüzde etkileşimi ve kullanıcı deneyimini arttırmak için çok çeşitli komponentler kullandık. Proje kapsamında öncelikli amacımız arayüz olmadığı için bazı şeylerden ödün vererek yine de iyi bir arayüz elde etmeye çalıştık. Sabit bir yerleşimde çoklu çözünürlük desteklemeyen bir pencere oluşturduk. Fakat pencere içerisinde öge yerleşimini iyi bir şekilde gerçekleştirdik. Arayüzü dinamik bir hale getirdik. Arayüzde gerçekleşen her işlem dinamik olarak bazı fonksiyonları tetikleyecek hale geldi. Bir thread üzerinde canvas çizimi gerçekleştirip hem de eventları kullandık. Ayrıca Swing için harici bir kütüphane olan Darcula[2] ile LookAndFeel modülü kullandık ve arayüze karanlık tema yaptık. Arayüz dinamik boyutlanan bir pencere içerisinde temel olarak iki ekran barındırıyor. İlk açılışta kullanıcıyı Ayar Ekranı karşılıyor. Burada gerekli seçimleri yaptıktan sonra dinamik olarak çalışan ve canlı ön izlemeye sahip Sunum Ekranına gidiyor. Bu ekranda anlık olarak haritayı takip edip bilgileri ediniyor ayrıca çıktıları da alabiliyor. Ara yüzün görünüşü programın genel yapısı ve tasarımı başlığında verilmiştir.

D. Genetik Algoritma

Şimdiye kadar program ve problem çözümü hakkında çok şeyden bahsettik fakat projenin esas noktası Genetik algoritma. Genetik algoritmayı seçme sebebimizden genel olarak bahsetmiştik. Bunun dışında kişisel bilgi birikimi de edinmek istiyorduk. Algoritmayı implement etmeden önce uzun süreli bir araştırma ve öğrenme gerçekleştirdik. Genel olarak Daniel Shiffman'ın Nature Of Code[3] kitabından ve çeşitli Travelling Salesperson problem çözümlerinden faydalandık. Bu kısımda biraz Genetik algoritmadan bahsetmek istiyoruz.

Genetik algoritmamızı oluştururken Darwin evriminin üç temel prensibini kullandık. Algoritmamızın çalışma mantığı doğal seleksiyona dayanmakta ve doğal seleksiyonun çalışması için bu üç prensibin de mevcut olması gereklidir.

Algoritmamız popülasyonlar üzerinden ilerlemekte aslında seçilen her yol popülasyon içerisinde bulunan bir birey gibidir. Gerçek hayatta olduğu gibi bu bireylerden yeni bireyler elde etmek istiyoruz. Bu yüzden ilk prensibimiz kalıtım ilkesi olmaktadır. Eğer doğru bireyleri yani yolları hayatta tutabilirsek daha sonraki nesillere genlerini aktarabilecek ve ortaya çıkacak yeni bireyler bizi istediğimiz sonuca götürecektir. Yukarda bahsettiğimiz gibi gen aktarımıyla yeni bireyler elde etmek istiyoruz fakat eğer bir popülasyonda tüm bireyler aynı olursa bir sonraki nesilde çeşitlilik olmayacak ve önceki neslin birebir kopyası olacaktır. Böylece bizi istediğimiz sonuca götürmeyecek tam aksine başladığımız noktada saymış olacağız. Bu yüzden çeşitliliğe ihtiyacımız var.

Genetik bilgiyi aktararak yeni bireyleri ortaya çıkarmaktan bahsetmiştik. Ortaya çıkartacağımız birey bizim isteklerimizi karşılamalı. Bir nevi en uygun olan bireyi yaşatmalı ve onların genlerini aktarmalıyız. Başlangıç popülasyonumuzda bulunan yollardan rastgele seçimler yaptığımızda bize en uzak mesafeleri verenleri seçip kalıtım almamalıyız. Bu yüzden doğru bir matematiksel modelleme yapmalıydık. Bu noktada artık algoritmanın nasıl işlediğinden bahsedebiliriz. Öncelikle başlangıçta bize ulaşan hedef şehir listesinden farklı farklı yollar oluşturup bunları bir popülasyon içerisine atıyoruz. Yukarda bahsettiğimiz nedenlerden ötürü popülasyonda yol çeşitliliğini sağlamak çok önemliydi.

Artık elimizde farklı yollardan oluşan bir popülasyon var doğal seleksiyon mantığıyla ilerlemekten bahsetmiştik ve doğru olanı hayatta tutmamız gerektiğini söylemiştik. Bu noktada bu seçimi yapmak çok önemliydi. Her yol için bir uygunluk değeri elde ediyoruz bunu da çok basit bir matematiksel ifadeyle yapıyoruz.

$$uygunluk = 1 / (uzunluk^{10})$$

Burada üstel fonksiyonun artış hızını kullanarak uzunluk değerimizin kuvvetini alıyoruz ve daha sonra uygunlukla ters orantılı hale getiriyoruz. Çünkü bir yol bizim için ne kadar uzunsa o kadar uygun değil. Ve bir yol ne kadar uzunsa o kadar giderek uygunsuz hale gelecek. Burada her yola uygunluk dizisinde bir karşılık veriyoruz fakat hala seçim yapmadık. Bu aşamadan sonra her popülasyonun kendine ait bir uygunluk değeri var. Örneğin; A yolu için 4, B yolu için 4, C yolu için 2 olsun şimdi uygunluk değerlerimizi normalize etmeliyiz. Bu işlemi toplamlarına bölerek yapıyoruz. Böylece normalize edilmiş A değerimiz 0.4, normalize edilmiş B değerimiz 0.4, ve C değerimiz 0.2 oluyor. Değerlerin toplamı 1 olacak şekilde normalizasyonu yapıyoruz. Ardından seçim yapabiliriz. Seçim yapmak için çok fazla seçenek var. Biz servet çarkı yöntemini kullanıyoruz ve bu uygunluk değerleri arasından rastgele seçim yapıyoruz. Böylece hem iyi gene sahip elemanın çoğalma ihtimali daha fazla oluyor hem de kötü genlerin de seçilebilme ihtimali daha önce bahsettiğimiz çeşitlilik ilkesini sağlıyor.

Yaptığımız seçim işlemiyle birlikte artık yeni bir popülasyon üretebiliriz. Sahip olduğumuz iyi genleri birleştirerek bu işlemi gerçekleştiriyoruz. Bu yöntemle yeni popülasyon oluştuktan sonra varyasyonu sağlamak için küçük bir mutasyon faktörü ekliyoruz. Böylece sistemin işleyişini büyük ölçüde etkilemeyecek fakat şans faktörüne bağlı olarak çok fazla iyi genden kalıtım alınıp yeni nesil türetilirse bu küçük mutasyon faktörüyle hiçbir zaman ortaya çıkamayacak genlerin ortaya çıkması sağlanabilir.

Tüm bu adımlar bir döngü içerisinde tekrarlanarak Genetik algoritma çalıştırılır. Genetik algoritma ayarlarına bağlı olmakla birlikte örnek olarak düşük seviye girdilerde popülasyonlar oluştukça ideal uygunluk yayılacak ve bir müddet sonra tüm popülasyon elemanları aynı uygunluğa sahip olacak fakat Genetik algoritma yeni aramalar yapacak. Bu yüzden Genetik algoritma, sezgisel ve sonu olmayan bir algoritma bazı noktalarda tetiklenerek durdurulması gerekiyor. Ayrıca her oluşan popülasyonda iyi genlerin aktarılma şansı fazla olduğu için bu problemde yaşadığımız en büyük sıkıntı alternatif rotalar oluşturmaktı çünkü ilk şanssız seçimden sonra genetik yasaları ile birlikte bir anda popülasyonlarda yüksek uygunluklu bireyler kalıtım veriyor. Yol verisi oldukça hızlı bir şekilde iyileşiyordu. Bu da az sayıda ve birbiri arasında çok hızlı bir azalmış uzunluk olan yol verileri elde etmemizi sağlıyordu. Genetik algoritma en iyi sonucu bulmak için ise çok iyi çalışıyordu. En iyi alternatif yolları elde edebilmek için bazı girdiler için özel olarak thread zamanlarında ve popülasyon sayılarında özel ayarlamalar yaparak yol sayısını minimumun üzerinde tutmaya çalıştık.

Genetik sınıfı birçok ana ve yardımcı metoda sahip olmakla birlikte kabaca algoritmanın çalışma döngüsü:

```
uygunluguHesapla();
uygunluğuNormalizeEt();
sonrakiNesliOlustur();
```

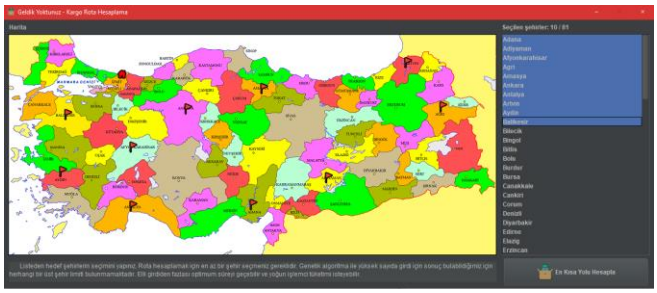
şeklinde devam etmektedir. sonrakiNesliOlustur() metodu kendi içerisinde seçim fonksiyonu çaprazlama fonksiyonu ve mutasyon fonksiyonu gibi diğer yardımcı fonksiyonları da kullanmaktadır.

III. Geliştirme Ortamı, Programın Derlenmesi ve Çalıştırılması

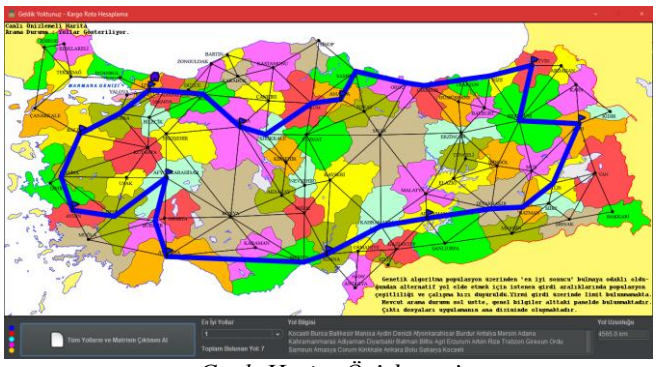
Projeyi Java Programlama dilinde JDK 13.0.1 ile Windows işletim sistemi üzerinde gerçekleştirirken, geliştirme ortamı olarak Intellij Idea'yı kullandık. Son proje dosya yapısı Intellij Idea dosya formatındadır ve javac derleyicisi ile derlenmiştir. İçerisinde harici kütüphanenin de gömüldüğü çalıştırılabilir jar dosyası bulunmaktadır ve kaynaklar klasörüne bağımlılık duymaktadır.Programın derlenmesi ya da jar dosyası üzerinden çalıştırılması ve kullanılması raporun son kısmında görseller ile detaylı olarak anlatılmaktadır.

IV. PROGRAMIN GENEL YAPISI VE TASARIMI

Program temel olarak iki ekrandan oluşuyor ve kullanıcının bu ekranlar üzerinden seçtiği çeşitli işlemleri gerçekleştiriyor. Bu işlemler arasında hedef şehirleri seçme, gösterilecek yolları seçme, çıktı alma gibi işlemler bulunuyor.



Hedef Şehir Seçim Ekranı



Canlı Harita Önizlemesi

V. Deneysel Sonuçlar

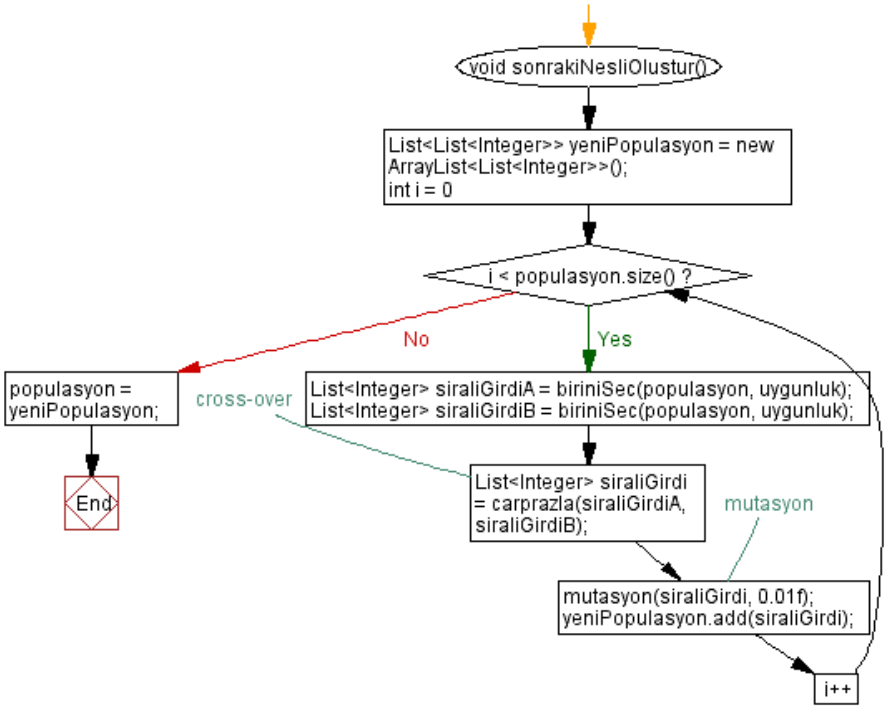
Projeyi gerçekleştirmeden önce sezgisel algoritma mantığını daha iyi anlamak için zorlama yöntemi ve permütasyonların sıralanması gibi yöntemleri deneyerek bu konularda çeşitli tecrübeler edinmiş olduk. Daha sonrasında en kısa yolları bulmak için çeşitli kısa yol bulma algoritmalarını inceleme fırsatı elde ettik. Dinamik arayüz elementlerini oldukça çeşitli bir şekilde kullandık. Genetik algoritma hakkında çok fazla bilgi edindik. En önemlisi gündelik hayattaki ögelerle problemlerin nasıl çözümleneceğine dair tecrübeler edindik. Diğer bilimlerin bilgisayar bilimlerindeki kullanımlarını çok net bir şekilde gördük. Çeşitli algoritmaların implementasyonlarını gerçekleştirirken daha önce kullanmadığımız yeni liste tipleri ve kümeler gibi çok sayıda veri yapısını kullanma fırsatı elde ettik. Javada harici kütüphaneler ile çalışmayı deneyimleme imkanını elde ettik. Programın çeşitli kısımlarında farklı optimizasyon işlemlerini gerçekleştirdik.

VI. Sonuç

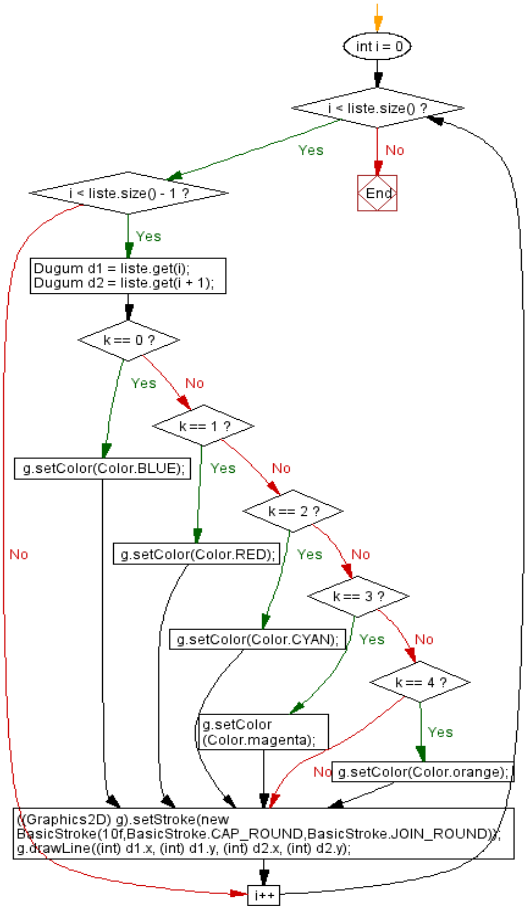
Bu projeyi gerçekleştirerek polinom zamanda çözülemeyen np-hard bir problem sezgisel yöntemlerle çözüme ulaştırdık ayrıca en kısa yol algoritmaları ve Genetik algoritma gibi konularda oldukça önemli bilgiler ve tecrübeler edindik.

VII. Kaynaklar

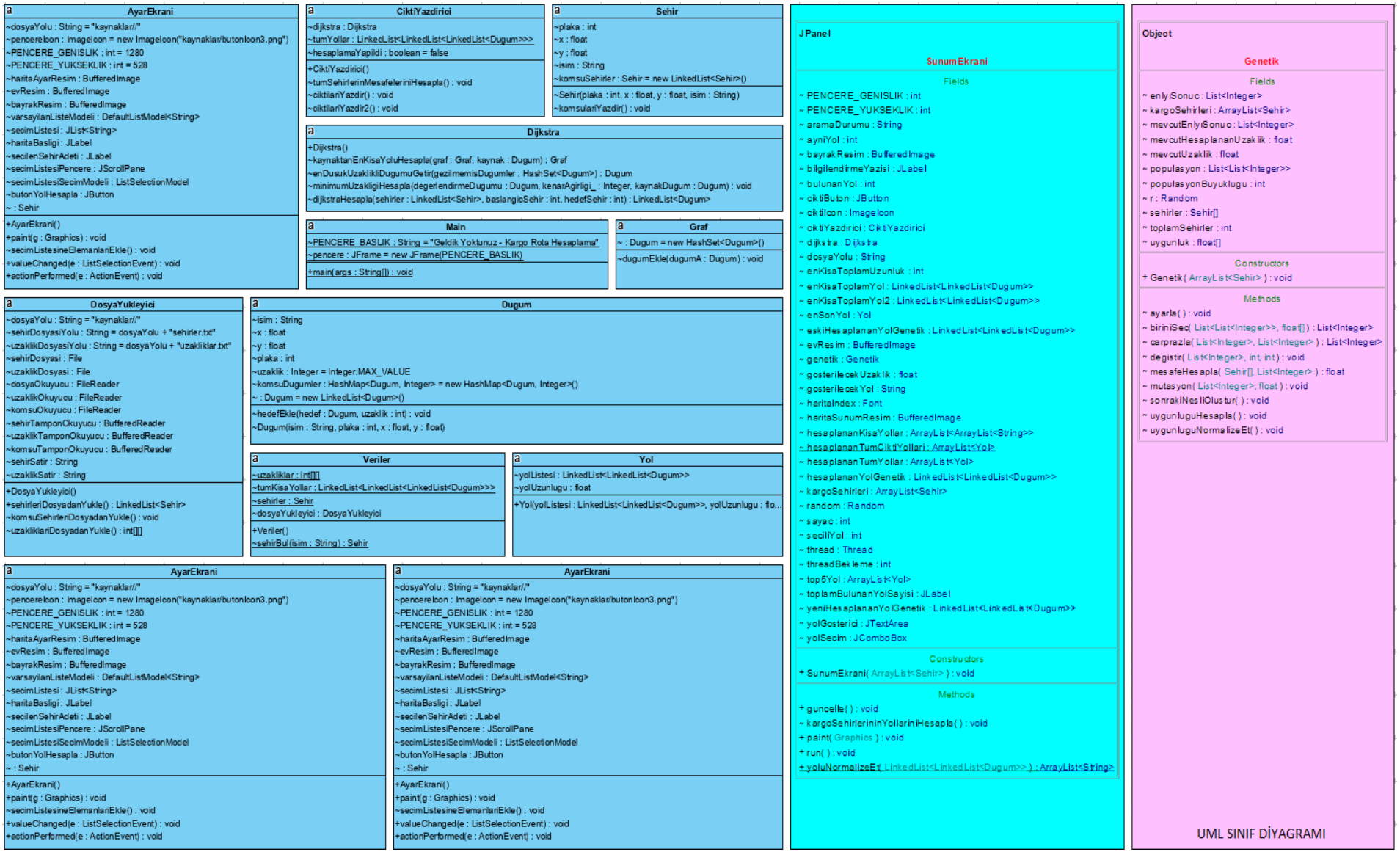
(1) https://www.baeldung.com/java-dijkstra
(2) https://github.com/bulenkov/Darcula
(3) Nature Of Code, Daniel Shiffman



Genetik Algoritmanın Temel Yeni Nesil Oluşturma Fonksiyonunun Akış Şeması

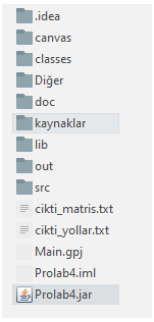


Yol Çizim Fonksiyonunun Akış Şeması


AyarEkrani
~dosyaYolu : String = "kaynaklar//"
~pencereIcon : ImageIcon = new ImageIcon("kaynaklar/butonIcon3.png")
~PENCERE_GENISLIK : int = 1280
~PENCERE_YUKSEKLIK : int = 528
~haritaAyarResim : BufferedImage
~evResim : BufferedImage
~bayrakResim : BufferedImage
~varsayilanListeModeli : DefaultListModel<String>
~secimListesi : JList<String>
~haritaBasligi : JLabel
~secilenSehirAdeti : JLabel
~secimListesiPencere : JScrollPane
~secimListesiSecimModeli : ListSelectionModel
~butonYolHesapla : JButton
~ : Sehir
+AyarEkrani()
+paint(g : Graphics) : void
~secimListesineElemanlariEkle() : void
+valueChanged(e : ListSelectionEvent) : void
+actionPerformed(e : ActionEvent) : void
CiktiYazdirici
~dijkstra : Dijkstra
~tumYollar : LinkedList<LinkedList<LinkedList<Dugum>>>
~hesaplamaYapildi : boolean = false
+CiktiYazdirici()
~tumSehirlerinMesafeleriniHesapla() : void
~ciktilariYazdir() : void
~ciktilariYazdir2() : void
Sehir
~plaka : int
~x : float
~y : float
~isim : String
~komsuSehirler : Sehir = new LinkedList<Sehir>()
~Sehir(plaka : int, x : float, y : float, isim : String)
~komsulariYazdir() : void
Dijkstra
+Dijkstra()
~kaynaktanEnKisaYoluHesapla(graf : Graf, kaynak : Dugum) : Graf
~enDusukUzaklikliDugumuGetir(gezilmemisDugumler : HashSet<Dugum>) : Dugum
~minimumUzakligiHesapla(degerlendirmeDugumu : Dugum, kenarAgirligi_ : Integer, kaynakDugum : Dugum) : void
~dijkstraHesapla(sehirler : LinkedList<Sehir>, baslangicSehir : int, hedefSehir : int) : LinkedList<Dugum>
Main
~PENCERE_BASLIK : String = "Geldik Yoktunuz - Kargo Rota Hesaplama"
~pencere : JFrame = new JFrame(PENCERE_BASLIK)
+main(args : String[]) : void
Graf
~ : Dugum = new HashSet<Dugum>()
~dugumEkle(dugumA : Dugum) : void
DosyaYukleyici
~dosyaYolu : String = "kaynaklar//"
~sehirDosyasiYolu : String = dosyaYolu + "sehirler.txt"
~uzaklikDosyasiYolu : String = dosyaYolu + "uzakliklar.txt"
~sehirDosyasi : File
~uzaklikDosyasi : File
~dosyaOkuyucu : FileReader
~uzaklikOkuyucu : FileReader
~komsuOkuyucu : FileReader
~sehirTamponOkuyucu : BufferedReader
~uzaklikTamponOkuyucu : BufferedReader
~komsuTamponOkuyucu : BufferedReader
~sehirSatir : String
~uzaklikSatir : String
+DosyaYukleyici()
+sehirleriDosyadanYukle() : LinkedList<Sehir>
~komsuSehirleriDosyadanYukle() : void
~uzakliklariDosyadanYukle() : int[][]
Dugum
~isim : String
~x : float
~y : float
~plaka : int
~uzaklik : Integer = Integer.MAX_VALUE
~komsuDugumler : HashMap<Dugum, Integer> = new HashMap<Dugum, Integer>()
~ : Dugum = new LinkedList<Dugum>()
~hedefEkle(hedef : Dugum, uzaklik : int) : void
~Dugum(isim : String, plaka : int, x : float, y : float)
Veriler
~uzakliklar : int[][]
~tumKisaYollar : LinkedList<LinkedList<LinkedList<Dugum>>>
~sehirler : Sehir
~dosyaYukleyici : DosyaYukleyici
+Veriler()
~sehirBul(isim : String) : Sehir
Yol
~yolListesi : LinkedList<LinkedList<Dugum>>
~yolUzunlugu : float
+Yol(yolListesi : LinkedList<LinkedList<Dugum>>, yolUzunlugu : flo...
AyarEkrani
~dosyaYolu : String = "kaynaklar//"
~pencereIcon : ImageIcon = new ImageIcon("kaynaklar/butonIcon3.png")
~PENCERE_GENISLIK : int = 1280
~PENCERE_YUKSEKLIK : int = 528
~haritaAyarResim : BufferedImage
~evResim : BufferedImage
~bayrakResim : BufferedImage
~varsayilanListeModeli : DefaultListModel<String>
~secimListesi : JList<String>
~haritaBasligi : JLabel
~secilenSehirAdeti : JLabel
~secimListesiPencere : JScrollPane
~secimListesiSecimModeli : ListSelectionModel
~butonYolHesapla : JButton
~ : Sehir
+AyarEkrani()
+paint(g : Graphics) : void
~secimListesineElemanlariEkle() : void
+valueChanged(e : ListSelectionEvent) : void
+actionPerformed(e : ActionEvent) : void
AyarEkrani
~dosyaYolu : String = "kaynaklar//"
~pencereIcon : ImageIcon = new ImageIcon("kaynaklar/butonIcon3.png")
~PENCERE_GENISLIK : int = 1280
~PENCERE_YUKSEKLIK : int = 528
~haritaAyarResim : BufferedImage
~evResim : BufferedImage
~bayrakResim : BufferedImage
~varsayilanListeModeli : DefaultListModel<String>
~secimListesi : JList<String>
~haritaBasligi : JLabel
~secilenSehirAdeti : JLabel
~secimListesiPencere : JScrollPane
~secimListesiSecimModeli : ListSelectionModel
~butonYolHesapla : JButton
~ : Sehir
+AyarEkrani()
+paint(g : Graphics) : void
~secimListesineElemanlariEkle() : void
+valueChanged(e : ListSelectionEvent) : void
+actionPerformed(e : ActionEvent) : void
JPanel
SunumEkrani
Fields
~ PENCERE_GENISLIK : int
~ PENCERE_YUKSEKLIK : int
~ aramaDurumu : String
~ ayniYol : int
~ bayrakResim : BufferedImage
~ bilgilendirmeYazisi : JLabel
~ bulunanYol : int
~ ciktiButon : JButton
~ ciktiIcon : ImageIcon
~ ciktiYazdirici : CiktiYazdirici
~ dijkstra : Dijkstra
~ dosyaYolu : String
~ enKisaToplamUzunluk : int
~ enKisaToplamYol : LinkedList<LinkedList<Dugum>>
~ enKisaToplamYol2 : LinkedList<LinkedList<Dugum>>
~ enSonYol : Yol
~ eskiHesaplananYolGenetik : LinkedList<LinkedList<Dugum>>
~ evResim : BufferedImage
~ genetik : Genetik
~ gosterilecekUzaklik : float
~ gosterilecekYol : String
~ haritaIndex : Font
~ haritaSunumResim : BufferedImage
~ hesaplananKisaYollar : ArrayList<ArrayList<String>>
~ hesaplananTumCiktiYollari : ArrayList<Yol>
~ hesaplananTumYollar : ArrayList<Yol>
~ hesaplananYolGenetik : LinkedList<LinkedList<Dugum>>
~ kargoSehirleri : ArrayList<Sehir>
~ random : Random
~ sayac : int
~ seciliYol : int
~ thread : Thread
~ threadBekleme : int
~ top5Yol : ArrayList<Yol>
~ toplamBulunanYolSayisi : JLabel
~ yeniHesaplananYolGenetik : LinkedList<LinkedList<Dugum>>
~ yolGosterici : JTextArea
~ yolSecim : JComboBox
Constructors
+ SunumEkrani(ArrayList<Sehir>) : void
Methods
+ guncelle( ) : void
~ kargoSehirleriniYollariniHesapla( ) : void
+ paint(Graphics) : void
+ run( ) : void
+ yoluNormalizeEt(LinkedList<LinkedList<Dugum>>) : ArrayList<String>
Object
Genetik
Fields
~ enIyiSonuc : List<Integer>
~ kargoSehirleri : ArrayList<Sehir>
~ mevcutEnIyiSonuc : List<Integer>
~ mevcutHesaplananUzaklik : float
~ mevcutUzaklik : float
~ populasyon : List<List<Integer>>
~ populasyonBuyuklugu : int
~ r : Random
~ sehirler : Sehir[]
~ toplamSehirler : int
~ uygunluk : float[]
Constructors
+ Genetik(ArrayList<Sehir>) : void
Methods
~ ayarla( ) : void
~ biriniSec(List<List<Integer>>, float[]) : List<Integer>
~ carprazla(List<Integer>, List<Integer>) : List<Integer>
~ degistir(List<Integer>, int, int) : void
~ mesafeHesapla(Sehir[], List<Integer>) : float
~ mutasyon(List<Integer>, float) : void
~ sonrakiNesliOlustur( ) : void
~ uygunluguHesapla( ) : void
~ uygunluguNormalizeEt( ) : void
UML SINIF DİYAGRAMI

PROGRAMIN ÇALIŞTIRILMASI

Direkt Olarak Çalıştırma



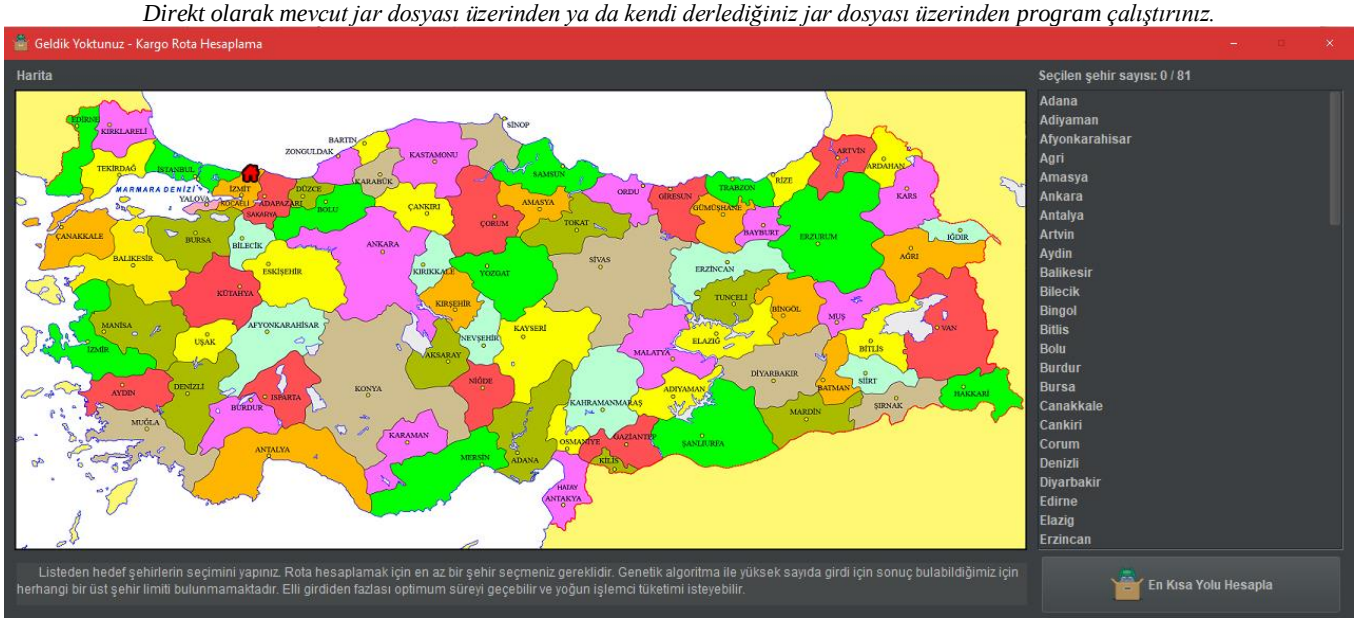
Program 2_180202024_180202025 içerisindeki Prolab4.jar üzerinden direkt olarak çalıştırılabilir.(Bunu tavsiye ediyoruz.) Bu jar dosyası sadece kaynaklar klasörüne bağımlılık duyar. Yani bu iki klasörü herhangi farklı bir yere kopyalayıp ya da taşıyarak da kullanabilirsiniz yine aynı şekilde o klasör altında çalıştırabilirsiniz.Sol tarafta 2_180202024_180202025 klasörünün genel yapısı ve direkt çalıştırma için gerekli dosya ve klasör işaretlidir. Proje Jdk 13.0.1 ile geliştirilmiştir. Önceki java sürümlerinde uyumluluk sorunları ve hatalar test edilmemiştir. 13.0.1 de sorunsuz çalışmaktadır. Sisteminizde birden fazla java sürümü bulunuyorsa 13.0.1 e yakın mümkünse daha güncel bir jvm ile çalıştırmalısınız.

Proje Üzerinden Derleme

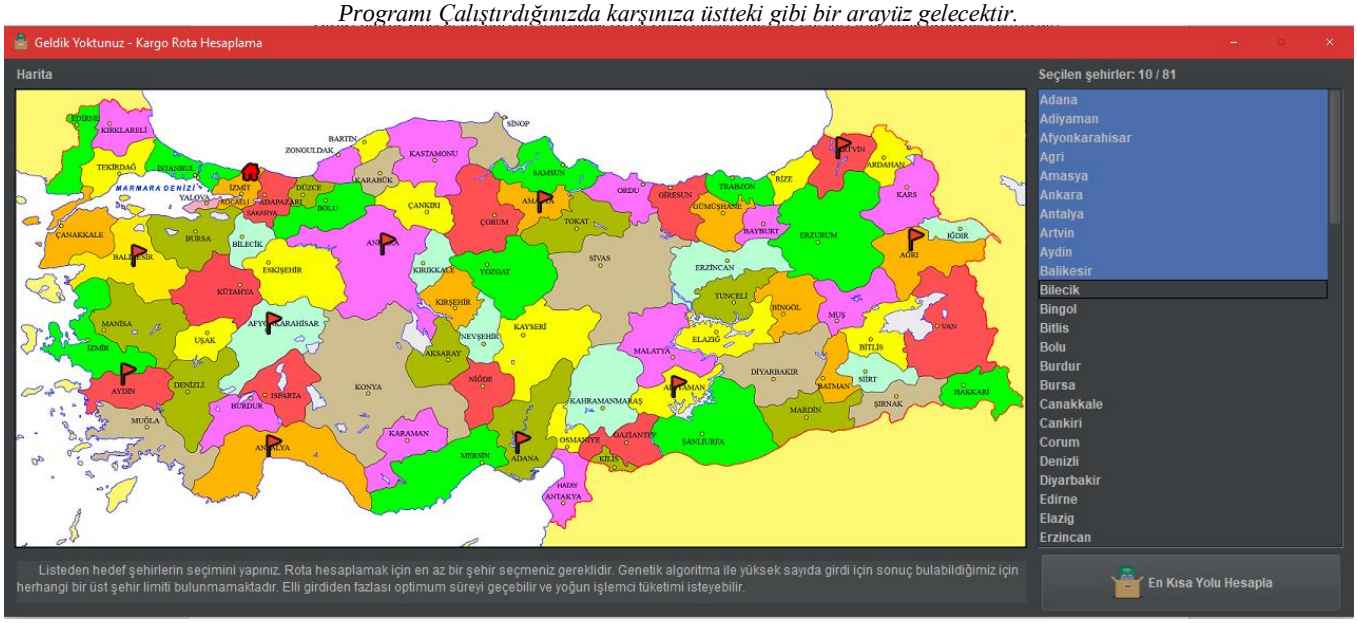
Projenin dosya yapısı intellij idea formatındadır. Yani 2_180202024_180202025 klasörünü direkt olarak intellij ideaya import ederseniz proje ayarları yapılı bir şekilde gelecektir. Eğer başka bir ide üzerinden tekrar derleme işlemi yapmak istiyorsanız ekstradan yapmanız gereken bir adım daha bulunmaktadır. Proje arayüz için bir adet harici görünüş kütüphanesi kullanmaktadır. Lib klasörü altında Darcula.jar'ın projeye dahil edilmesi gerekmektedir. Kendi proje yapımızda bu harici kütüphane ana jar dosyasının içine gömülmüştür. Projeye bu harici kütüphaneyi ekleyerek ya da koyu temayı devre dışı bırakmak için Ana Sınıf altında gerekli import satırını ve temayı yükleyen try-catch bloğunu kaldırarak temaya olan bağımlılığı da kaldırıp derleme işlemini gerçekleştirebilirsiniz.Intellij ideadan yapılan derlemeler out klasörü altında artifacts klasöründe oluşur burada oluşan jar dosyasının yanına aynı şekilde kaynaklar klasörünü verip kullanabilirsiniz.

Programın Kullanılması

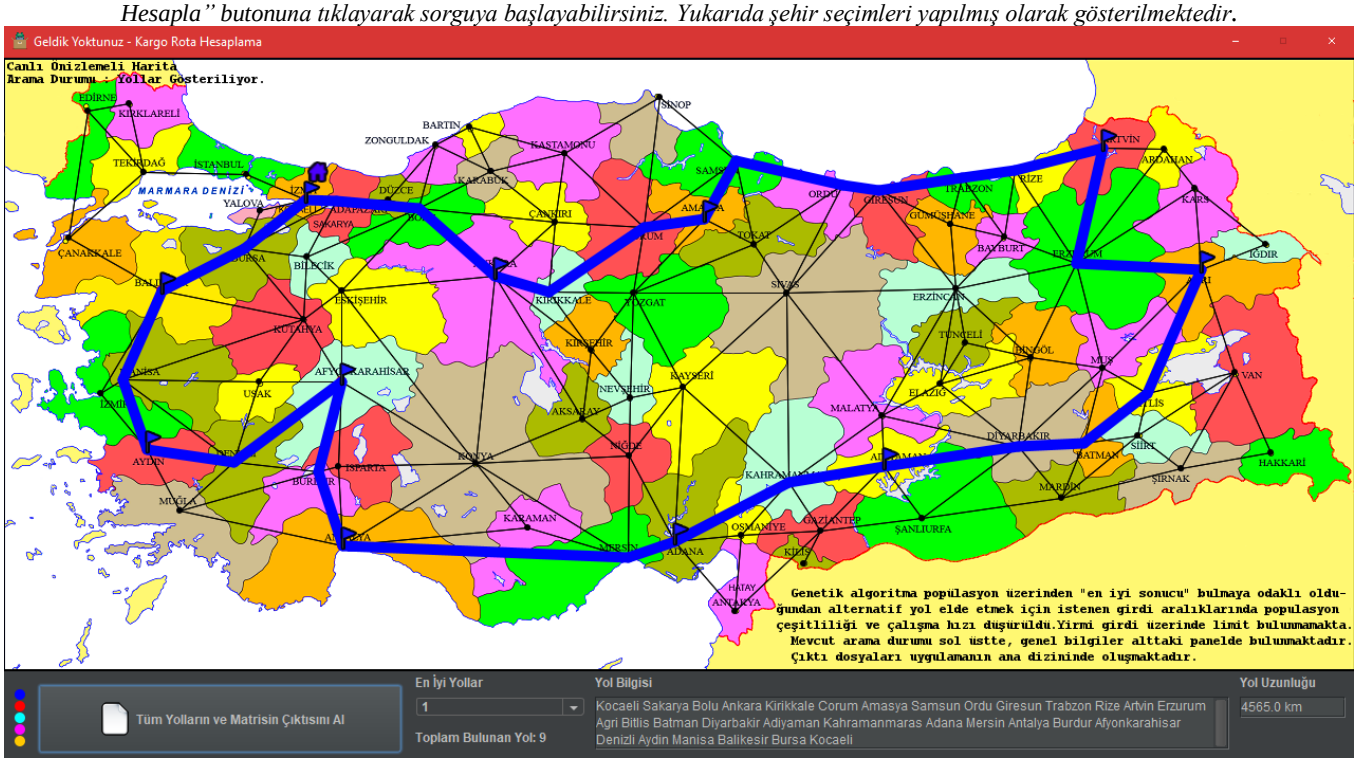
Direkt olarak mevcut jar dosyası üzerinden ya da kendi derlediğiniz jar dosyası üzerinden program çalıştırınız.



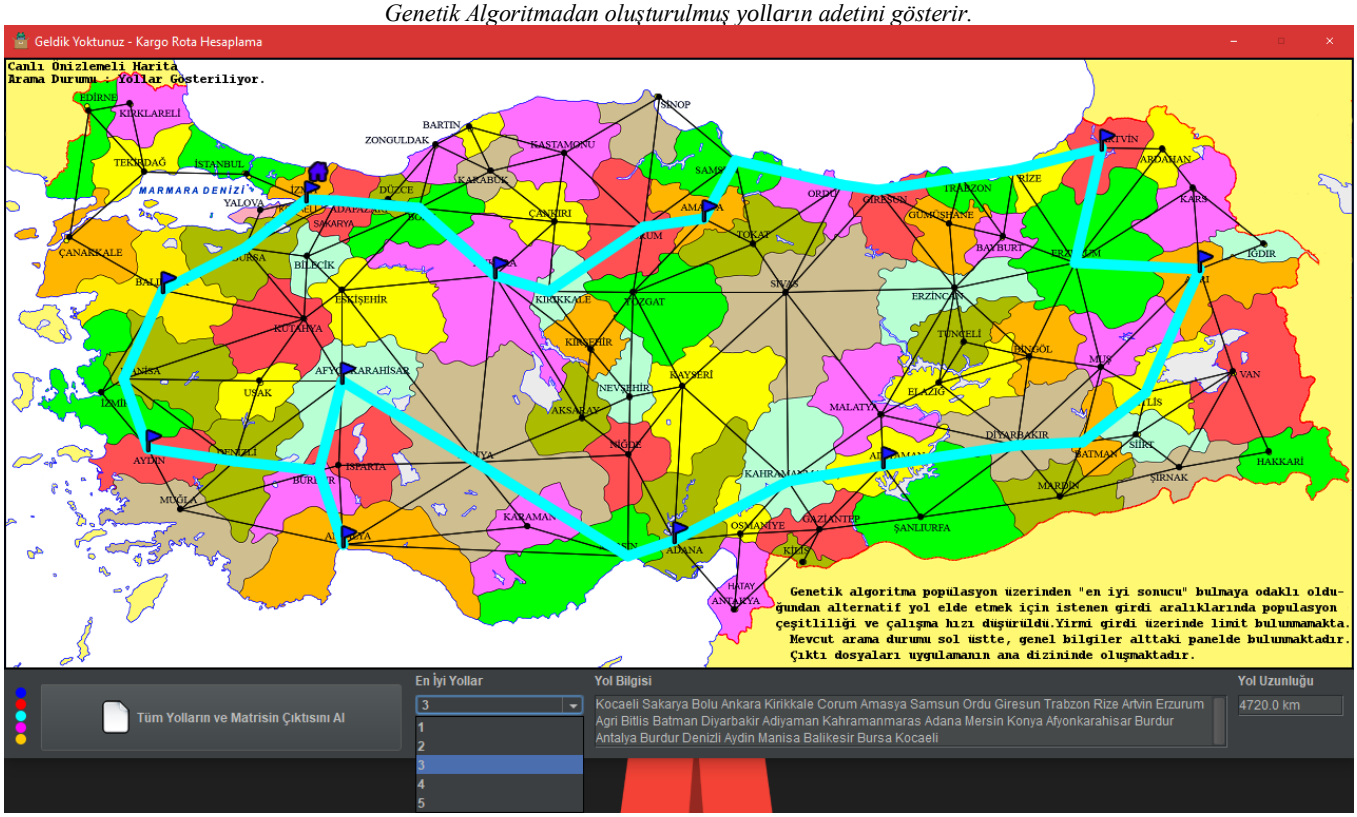
Programı Çalıştırdığınızda karşınıza üstteki gibi bir arayüz gelecektir.

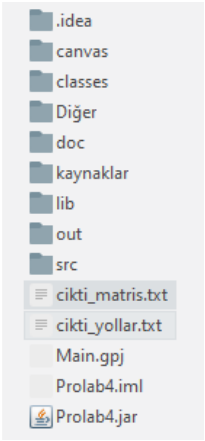


Yanda bulunan şehir listesine tıklayarak şehir seçimlereini yapmalısınız. Sorgu gerçekleştirmek için en az 1 adet şehir seçmelisiniz. Aksi halde butona tıkladığınızda işlem gerçekleşmeyecektir. Listeden seçtiğiniz şehirler canlı olarak harita üzerinde size işaretlenip görsel geri bildirim verilmektedir. Aynı şekilde problemimizin Merkez üssü olan Kocaeli de haritada belirtilmektedir. Liste üzerinde seçimler tıklama ile yapılmaktadır. Tıkladığınız şehir seçilip mavi renge dönmektedir. Tekrar tıklamanız halinde seçili olmayan durumuna dönmektedir. Toplam seçtiğiniz hedef şehir sayısı listenin üstünde canlı olarak gözükmektedir. Seçim işlemi tamamlandığı zaman "En Kısa Yolu Hesapla" butonuna tıklayarak sorguya başlayabilirsiniz. Yukarıda şehir seçimleri yapılmış olarak gösterilmektedir.



Butona tıklağınız zaman canlı harita önizleme penceresi açılacak ve algoritma çalışmaya başlayacaktır burada hesaplamalar size harita üzerinde canlı olarak gösterilmektedir. Sonuçların görülebilmesi için algoritma yavaşlatılmıştır. Sol üstte arama durumunu görebilirsiniz. 10 girdi için genelde arama bir kaç saniyede sonlanacak ve sol üstteki arama devam ediyor bildirimi ortadan kalkarak yollar gösteriliyor biçimini alacaktır. Bu esnada arama işleminiz tamamlanmıştır. Her şey dinamik olarak çalıştığı için o anki duruma göre arayüzdeki her şey güncellenmektedir.Ekranın alt kısmındaki bilgi panelinde En İyi Yollar seçim menüsünün altında Toplam Bulunan Yol size o ana kadar Genetik Algoritmadan oluşturulmuş yolların adetini gösterir.





Yukarıda gösterildiği gibi bilgi göstergelerinin bulunduğu panelde "En İyi Yollar" seçim menüsünden eğer hesaplanmışsa 5 adete kadar olan en kisa yollari harita üzerinden canlı olarak görüntüleyebilirsiniz. Yandaki bilgi panellerinden yolun il il sıralanışını ve yolun toplam uzunluğunu görebilirsiniz.Tüm bu gösterimler canlı olarak harita üzerinden gerçekleşmektedir. Eğer tüm hesaplanmış yolları görmek istiyorsanız Tüm Yolların ve Matrisin Çıktısını Al butonuna basmalısınız. Program dinamik olarak çalıştığı için tıkladığınız andaki tüm hesaplanan yolların çıktısını alacaktır.

Yanda çıktı dosyalarını görmektesiniz. Ana jarın dizininde oluşmaktadırlar. Dosyalar mevcut değilse bile çıktı işlemi sırasında oluşturulurlar

.

cikti_yollar.txt size en iyi 5 şehir haricinde mevcutsa diğer şehirlerinde uzunluklarını ve yolları gösterir. cikti_matris.txt size tüm illerin mesafe matrisini Dijksta en kısa yol algoritmasından elde ettiğimiz uzunluk verilerine göre verir.