T.C.
HALİÇ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSİLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSİLİĞİ PROGRAMI

MOBİL ANDROİD UYGULAMASI İLE FARKLI KISA YOL ALGORİTMALARININ SİMÜLASYONU VE KARŞILAŞTIRILMASI

YÜKSEK LİSANS TEZİ

Hazırlayan
Giray BAL

Danışman
Yrd. Doç. Dr. ÜLVİYE HACIZADE

İstanbul – 2018

FEN BİLİMLERİ ENSTİTÜSÜ MÜDÜRLÜĞÜNE

Bilgisayar Mühendisliği Anabilim Dalı Yüksek Lisans Programı Öğrencisi Giray BAL tarafından hazırlanan **"Mobil Android Uygulaması ile Farklı Kısa Yol Algoritmalarının Silmülasyonu ve Karşılaştırılması"** adlı tez çalışma jürimizce Yüksek Lisans tezi olarak kabul edilmiştir.

Tez Savunma Tarihi: 26.01.2018

Jüri Üyesinin Unvanı, Adı, Soyadı ve Kurumu		İmzası
Jüri Üyesi	: Yrd. Doç. Dr. Ulviye HACIZADE : Haliç Üniv./Danışmanı	
Jüri Üyesi	: Prof. Dr. Mübariz EMİNLİ : Haliç Üniv./ Üye	
Jüri Üyesi	: Yrd. Doç. Dr. Alev MUTLU : Kocaeli Üniv./ Üye	

Bu tez Enstitü Yönetim Kurulu tarafından belirlenen jüri üyeleri tarafından uygun görülmüş ve Enstitü Yönetim Kurulunun kararıyla kabul edilmiştir.

Prof. Dr. Temel SAVAŞKAN
Fen Bilimleri Enstitüsü
Vekil Müdür

MOBİL ANDROİD UYGULAMASI İLE FARKLI KISA YOL ALGORİTMALARININ SİMÜLASYONU VE KARŞILAŞTIRILMASI

ORIJINALLIK RAPORU

%5	%4	%1	%
BENZERLIK ENDEKSI	İNTERNET KAYNAKLARI	YAYINLAR	ÖĞRENCI ÖDEVLERI

BIRINCIL KAYNAKLAR

#	Kaynak	Oran
1	tr.wikipedia.org İnternet Kaynağı	%1
2	www.ofelio.com İnternet Kaynağı	<%1
3	emyo.beun.edu.tr İnternet Kaynağı	<%1
4	polen.itu.edu.tr İnternet Kaynağı	<%1
5	webdoc.sub.gwdg.de İnternet Kaynağı	<%1
6	etv.emb.gov.hk İnternet Kaynağı	<%1
7	alshamess.ifa.hawaii.edu İnternet Kaynağı	<%1
8	www.android-dev.fr İnternet Kaynağı	<%1

ÖNSÖZ

Üniversite eğitim hayatımda bana hep olumlu ve yapıcı destek olan hocam Yrd. Doç. Dr. Ülviye Hacızade hocama bana bugüne kadarki lisans ve yüksek lisansta olan desteklerinden dolayı teşekkürlerimi borç bilirim.

Sevgili Bölüm Başkanı ve hocamız Sayın Prof. Dr. Mübariz Eminli hocama ders ve ders dışındaki konularda lisans ve yüksek lisansta vermiş olduğu katkı, destek ve yardım severliğinden dolayı teşekkürlerimi borç bilirim.

Yüksek lisans tezimi yazarken bana özellikle teknik konularda yardımcı olan ve tezimi yazma konusunda yeterli zamanı ve rahatlığı sağlayan sevgili Erkan Yiğiter, Onur Çelik ve Miops Teknoloji şirketine teşekkürlerimi borç bilirim.

Beni bugünlere getiren babam Yüksel BAL, annem Nuray BAL ve kardeşim Aybike BAL'a bana verdikleri maddi ve manevi destekten dolayı çok teşekkür ederim.

İstanbul, 2017 Giray BAL

İÇİNDEKİLER

Sayfa No.

KISALTMALAR .... **IV**

TABLOLAR .... **V**

ŞEKİLLER .... **VII**

ÖZET .... **IX**

SUMMARY .... **X**

1. GİRİŞ **1**

2. LİTERATÜR TARAMASI **3**

2.1. En Kısa Yol Algoritmalarının Kullanıldığı Video Oyunlarının İncelenmesi 3

2.2. En Kısa Yol Algoritmalarının Kullanıldığı Uygulama ve Navigasyon Programlarının İncelenmesi 10

3. ALGORİTMALAR **12**

3.1. Algoritmaların Tarihi 12

3.2. Algoritmaların Mantığı 12

3.3. Algoritmaların Yapısı 13

3.4. Sözde (Pseudo) Kod 14

3.5. Karmaşıklık (Complexity) 14

3.6. Açgözlü Yaklaşımı (Greedy Approach) 15

3.7. Bazı Algoritma Türleri 15

3.7.1. Arama Algoritmaları 15

3.7.2. Kriptografik Algoritmalar 16

3.7.3. Sıralama Algoritmaları 16

3.7.4. Genetik Algoritmalar 16

3.7.5. Veri Sıkıştırma Algoritmaları 16

3.7.6. En Kısa Yol Bulma Algoritmaları 16
3.8. En Kısa Yol Bulma Algoritmalarının İncelemesi 17
3.8.1. Dijkstra Algoritması 18
3.8.1.1. Nedir? 18
3.8.1.2. Çalışma Mantığı 19
3.8.2. Floyd-Warshall Algoritması 29
3.8.2.1. Nedir? 29
3.8.2.2. Çalışma Mantığı 30
3.8.3. Bellman-Ford Algoritması 36
3.8.3.1. Nedir? 36
3.8.3.2. Çalışma Mantığı 37
3.8.4. A* Algoritması 42
3.8.4.1. Nedir? 42
3.8.4.2. Çalışma Mantığı 43
3.9. En Kısa Yol Bulma Algoritmalarının Analizi 50
3.10. Algoritmaların Çalışma Mantığı ve Kaba Kodları 51
3.10.1. Dijkstra Algoritması 51
3.10.2. Floyd-Warshall 52
3.10.3. Bellman-Ford 53
3.10.4. A* Algoritması 54
4. KULLANILAN TEKNOLOJİLER 55
4.1. Java 55
4.1.1. Bileşenleri 55
4.1.2. Veri Tipleri 57
4.1.3. Erişim Belirleyiciler (Access Modifiers) 59
4.1.4. Değişken Türleri (Variable Types) 60
4.1.5. Erişim Dışı Belirleyiciler (Non Access Modifiers) 61

4.1.6. Aritmetik Operatörler ... 62

4.1.7. Döngüler ... 62

4.1.8. Nesne Yönelimli Programlama ... 64

4.2. Android ... 66

5. MOBİL ANDROİD UYGULAMASININ TASARIMI VE GELİŞTİRİLMESİ ... 69

5.1. Giriş Ekranı ... 69

5.2. Ayarlar Ekranı ... 70

5.3. Harita Seçme Ekranı ... 71

5.4. Algoritma Seçme Ekranı ... 72

5.5. Simülasyon Ekranı ... 73

5.6. Sonuçlar Ekranı ... 74

6. SONUÇLAR ... 78

7. KAYNAKLAR ... 80

8. EKLER ... 83

9. ÖZGEÇMİŞ ... 88

KISALTMALAR

OOP : Object Oriented Programming

JVM : Java Virtual Machine

JDK : Java Development Kit

SDK : Software Development Kit

OHA : Open Handset Alliance

VM : Virtual Machine

API : Application Programming Interface

CPU : Central Processing Unit

ABD : Amerika Birleşik Devletleri

IP : Internet Protocol Address

TABLOLAR

Sayfa No.

Tablo 3.1. Dijkstra Başlangıç Tablosu 20
Tablo 3.2. İkinci Satıra Aktarma İşlemi 20
Tablo 3.3. İkinci Satırı Doldurma İşlemi 21
Tablo 3.4. İkinci Satırı Hesaplama İşlemi 22
Tablo 3.5. Üçüncü Satıra Aktarma İşlemi 23
Tablo 3.6. Üçüncü Satırı Hesaplama İşlemi 23
Tablo 3.7. Dördüncü Satırı Hesaplama İşlemi 24
Tablo 3.8. Beşinci ve Altıncı Satırları Hesaplama İşlemi 24
Tablo 3.9. Dijkstra Tablosunun Son Hali 25
Tablo 3.10. Altıncı ve Beşinci Satırların Karşılaştırılması 26
Tablo 3.11. Beşinci ve Dördüncü Satırların Karşılaştırılması 26
Tablo 3.12. Dördüncü ve Üçüncü Satırların Karşılaştırılması 27
Tablo 3.13. Üçüncü ve İkinci Satırların Karşılaştırılması 27
Tablo 3.14. Floyd-Warshall Maliyet ve Sıra Tablosu 31
Tablo 3.15. Maliyet ve Sıra Tablosunun Doldurulmuş Hali 31
Tablo 3.16. Birinci İterasyon Tablosu 32
Tablo 3.17. H23 Hücresinin Doldurulması 33
Tablo 3.18. İkinci İterasyon Tablosu 33
Tablo 3.19. Üçüncü İterasyon Tablosu 34
Tablo 3.20. Dördüncü İterasyon Tablosu 34
Tablo 3.21. Birinci Noktadan İkinci Noktaya Giden Kısayol ve Maliyeti 35
Tablo 3.22. Bütün Yolların Kısayol ve Maliyetleri 35
Tablo 3.23. Yol ve Ağırlık Tablosu 38
Tablo 3.24. Maliyet ve Öncelik Tablosu 38
Tablo 3.25. Birinci Sütunun Hesaplanması 39
Tablo 3.26. İkinci Sütunun Hesaplanması 39
Tablo 3.27. Birinci İterasyonun 4. Satırının Hesap Sonucu 40
Tablo 3.28. Birinci İterasyonun 5. Satırının Hesap Sonucu 40
Tablo 3.29. Son Değerler Tablosu 40

Tablo 3.30. En Kısa Yol Tablosu 41
Tablo 3.31. A* Harita Örneği 46
Tablo 3.32. Birinci Noktanın Çevresindeki Hesaplanan Değerler 47
Tablo 3.33. Diğer Noktaların Hesaplanan Değerleri 48
Tablo 4.1. Java'daki Erişim Belirleyiciler ve Yetkileri 60
Tablo 4.2. Java'daki Operatörler ve Açıklamaları 62
Tablo 4.3. Android versiyon kullanım oranları (Aralık 2017) 67

ŞEKİLLER

Sayfa No.

Şekil 2.1. Dune 2000 Oyunu Ekran Görüntüsü 4
Şekil 2.2. Haritanın Izgaralara Bölünmüş Hali 5
Şekil 2.3. Haritanın Engeller Koyulmuş Hali 6
Şekil 2.4. Haritanın Kısayol Çizilmiş Hali 7
Şekil 2.5. Age of Empires Oyun Görüntüsü 8
Şekil 2.6. Age of Empires Oyununun Maliyetli Görüntüsü 9
Şekil 2.7. Yandex Uygulamasının Ekran Görüntüsü 11
Şekil 3.1. Algoritmaları oluşturma aşamaları 13
Şekil 3.2. Dijkstra Örnek Haritası 19
Şekil 3.3. Dijkstra Sonuç Haritası 28
Şekil 3.4. Floyd-Warshall Harita Örneği 30
Şekil 3.5. Bellman-Ford Örnek Haritası 37
Şekil 3.6. A* Manhattan Uzaklığı 43
Şekil 3.7. A* Çapraz Uzaklık 44
Şekil 3.8. A* Öklid Uzaklığı 45
Şekil 3.9. Bir Noktadan Birçok Noktaya Gidiş Grafı 51
Şekil 3.10. Dijkstra Algoritması Kaba Kodu 51
Şekil 3.11. Birçok Noktadan Birçok Noktaya Gidiş Grafı 52
Şekil 3.12. Floyd-Warshall Algoritması Kaba Kodu 52
Şekil 3.13. Bir Noktadan Birçok Noktaya Gidiş Grafı 53
Şekil 3.14. Bellman-Ford Algoritması Kaba Kodu 53
Şekil 3.15. Bir Noktadan Bir Noktaya Gidiş Grafı 54
Şekil 3.16. A* Algoritması Kaba Kodu 54
Şekil 4.1. JVM Diagramı 56
Şekil 4.2. Java Veri Tipleri 58
Şekil 4.3. Java Polimorfizm Örneği 65
Şekil 4.4. Android Logosu 66
Şekil 5.1. Giriş Ekranı 69
Şekil 5.2. Ayarlar Ekranı 70

Şekil 5.3. Harita Seçme Ekranı 71
Şekil 5.4. Algoritma Seçme Ekranı.. 72
Şekil 5.5. Simülasyon Ekranı 73
Şekil 5.6. Harita1'in Hesaplanan Sonuçları 74
Şekil 5.7. Harita2'nin Hesaplanan Sonuçları 75
Şekil 5.8. Harita3'ün Hesaplanan Sonuçları 77
Şekil 5.7. Harita4'ün Hesaplanan Sonuçları 77

GENEL BİLGİLER

Adı ve Soyadı	:	Giray BAL
Anabilim Dalı	:	Bilgisayar Mühendisliği
Programı	:	Bilgisayar Mühendisliği
Tez Danışmanı	:	Yrd. Doç. Dr. ÜLVİYE HACIZADE
Tez Türü ve Tarihi	:	Yüksek Lisans - Ocak 2018

ÖZET

MOBİL ANDROİD UYGULAMASI İLE FARKLI KISA YOL ALGORİTMALARININ SİMÜLASYONU VE KARŞILAŞTIRILMASI

Artık dünyadaki bütün insanların en az bir teknolojik cihaza sahip olduğu günümüzde teknoloji çok hızlı bir biçimde ilerlemektedir. Buna bağlı olarak taşınabilir cihaz denilen küçük boyutlu cep telefonları, tabletler, giyilebilir teknoloji olan kol saatleri gibi cihazların kullanımları eskiye nazaran kat ve kat artmıştır. Bu hızlı artışın özellikle bilgisayar bilimleri alanında katlanarak arttığı görülmektedir. Özgür yazılım grupları ile birlikte gelişen bu teknolojilerle beraber günümüzde milyarlarca insanın kullandığı Android tabanlı telefonlar da hızlı bir kullanım artışı göstermektedir. Google Play Market ile beraber indirilen oyun ve uygulamalar günümüzde milyarlarca dolar gelir getirmektedir. Bu talepten dolayı kullanılan cihazların performanslarının doğru kullanılması önemli bir ihtiyaç olarak doğmuştur. Günümüzde özellikle birçok strateji oyunların ve özellikle navigasyon tabanlı uygulamaların telefonlarda performanslı ve hızlı bir şekilde çalışması için araştırmalar yapılmaktadır. Bu çalışmada; bu oyunlar ve uygulamalarda en çok kullanılan en kısa yol bulma algoritmalarının uygulamalara nasıl entegre edilebileceği gösterilmiş ve performansları karşılaştırılmıştır. Bazı en kısa yol bulma algoritmalarının Android tabanlı telefon üzerindeki verimleri çıkartılmış ve detaylı bir biçimde anlatılarak algoritmaların avantaj ve dezavantajlarına bağlı olarak sonuçları nasıl etkilediği belirtilmiştir. Ayrıca Android tabanlı bir mobil uygulama geliştirilmiş ve kullanıcıların bu uygulama ile algoritmaları nasıl kullanacakları ve tercih edecekleri hakkında bilgi sahibi olmaları amaçlanmıştır.

GENERAL INFORMATION

Name and Surname	:	Giray BAL
Field	:	Computer Engineering
Program	:	Computer Engineering
Supervisor	:	Yrd. Doç. Dr. ÜLVİYE HACIZADE
Degree Awarded and Date	:	Master of Science - January 2018

SUMMARY

COMPARISON AND SIMULATION OF DIFFERENT SHORTEST PATH ALGORITHMS WITH MOBILE ANDROID APPLICATION

Nowadays, technology is advancing very quickly and all people in the world have at least one technological device. Thanks to these developments, small-size mobile phones, tablets, wearable technology devices like wrist watches which we call portable devices have increased in usage compared to the last decade. This rapid increase appears to have increased exponentially, especially in the field of computer science. Technology has developed along with Free Software groups. Now billions of people are using Android based mobile phones and the number of these people is increasing. Games and applications downloaded from Google Play Market are generating billions of dollars in revenue. For this reason, the optimization of the performance of the used devices has become an important necessity. Research is being conducted on the performance of games and navigation-based applications for a fast and efficient way, especially in many strategy games. In these studies, it is shown how to implement the most used shortest path finding algorithms in games and applications and their performances have been compared. The performance of shortest path algorithms on Android phones is shown. Depending on the advantages and disadvantages of the algorithms, how the results are affected has been specified in detailed way. Also an Android application developed, and with this application, it is aimed to have knowledge about how users will use these algorithms and how they will prefer.

1. GİRİŞ

Teknolojinin gelişmesiyle birlikte günümüzde birçok alanda insanların yaptıkları işleri artık teknolojiye bıraktıkları görülmektedir. Bilgisayarlar, robotlar, akıllı telefonlar ve eşyalar artık günümüzde insanların becerilerinin ulaşamayacağı hafıza ve hızlara ulaşabilmektedir. Günümüzde çalışan birçok robot veya akıllı cihaz yapılacak işin algoritması ne kadar karmaşık olursa olsun insana göre çok daha kısa sürede sonuçlara ulaşabilmekte ve bu ulaştığı sonuçlara nasıl eriştiğini algoritmanın tasarlanma biçimine bağlı olarak bizlere sunabilmektedir.

Yapay zekanın da kullanım alanı olan algoritmalar günümüzde işlerin belli istenenlerle ne gibi sonuçlar beklediğimizi belirttiğimiz ve buna göre tasarladığımız algoritmalar insan hayatını kolaylaştıran en önemli teknolojik konulardan bir tanesidir. Algoritmaların yapabilecekleri algoritmayı tasarlayan kişinin algoritmaya verdiği özellikler dahilindedir. Performansları da yine algoritmanın tasarlanmasına ve hangi konuda taviz verip hangi konuda tutumlu olmasına bağlı olarak değişiklik gösterir.

Sanılanın aksine algoritmalar sadece teknoloji ile kullanılabilen birer program parçacığı değildir. Günümüzde birçok alanda farkında olmadan kullanılan ufak ta olsa algoritmalar bulunur. Trafik ışıklarında kırmızı yandığında durulduğunda ve yeşil yandığında geçildiğinde farkında olmadan diğer sürücüler ile birlikte bir algoritmanın parçası olunmaktadır.

Bu örneklerde de görüldüğü gibi, algoritmaların ne kadar karmaşık olabilecekleri anlaşılmaktadır. Bu tezde de açıklanacak olan bazı algoritmalar kendi alanlarına bağlı olarak ufak boyutlu, ufak kapasiteli ve az karmaşık veya büyük boyutlu, büyük kapasiteli ve çok karmaşık olabileceklerdir.

Tezin konusu olan en kısa yol bulma algoritmasında verimlik ön plandadır. Örneğin, her gün arabayla işe giderken ve işten dönerken kullanılan navigasyon cihazı tamamen en kısa yol bulma algoritmaları üzerine kuruludur. Trafik olan yollar, kapalı olan yollar, alternatif veya daha uzun olan yollar kullanılan programın tasarımına bağlı olarak bir algoritmaya sokulur ve gidilebilecek en kısa yol, içinde bilerek eklenen bazı

hata payları ile birlikte kullanıcılara sunulur. Bu algoritmanın amacı maliyetten ve zamandan tasarruf sağlayabilmektir.

Her gün milyarlarca kez kullanılan Google Arama motorunun sahibi Google Inc. hem kendi arama algoritmasında hem de 2013 yılında satın aldığı Boston Dynamics ile ürettiği yapay zekalı robotlar olan "LittleDog", "SandFlea", "BigDog", "Cheetah", ve "Petman" robotlarında en kısa yol bulma algoritmalarını kullandığı bilinmektedir [1].

En kısa yol bulma algoritmasının üzerinde çalıştığı aygıtın teknik donanım özelliklerine bağlı işlemci hızı ve bellek kapasitesine bağlı olarak çıkacak sonuçlar hızlı aygıtlarda daha hızlı veya yavaş aygıtlarda daha yavaş çalışacaktır. Tezin içeriğinde yer alan en kısa yol bulma algoritmalarının karşılaştırmaları ve faklı aygıtlara bağlı olarak algoritmanın hız ve iterasyon sayısının hangi değerlerde bulunduğu anlatılmıştır. Bu karşılaştırma sonuçlarına bağlı olarak kullanıcıların hangi algoritmayı nerede kullanmaları gerektiği sonuçlarına varılmıştır. Algoritmalar birbirleriyle karşılaştırılıp çıkan sonuçlar tablolarda verilmiş ve birbirleri üzerinde üstün özellikleri ve dezavantajları anlatılmıştır. Ayrıca bu karşılaştırmaları kullanıcılara gösterebilen Android tabanlı bir mobil uygulama geliştirilmiştir. Bu uygulamada Dijkstra, Bellman-Ford, Floyd-Warshall ve A* en kısa yol bulma algoritmalarının her biri birbiri ile karşılaştırılmıştır ve sonuçlar değerlendirilip belli çıkarımlarda bulunulmuştur.

2. LİTERATÜR TARAMASI

Literatür taraması yapılırken Google Akademik ve çeşitli algoritma konulu kitaplar incelenmiştir. Üniversitelerin web sitelerindeki makaleler incelenmiş ve konu ile alakalı çeşitli makalelerden bilgiler derlenmiştir. En kısa yol algoritmalarının kullanım alanları hakkında incelemeler yapılmıştır. En kısa yol algoritmaları konusunun en çok kullanıldığı alanların başında video oyunları, navigasyon uygulamaları robotik ve uzay araştırmaları gelmektedir.

2.1. En Kısa Yol Algoritmalarının Kullanıldığı Video Oyunlarının İncelenmesi

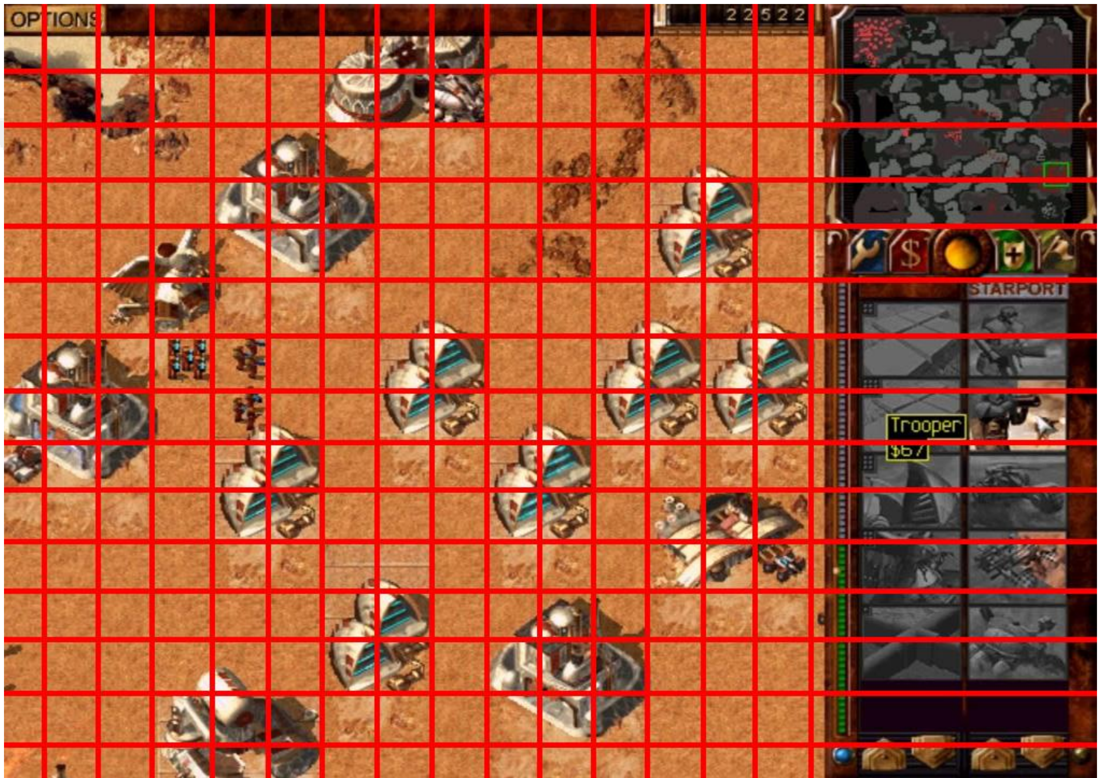
Video oyunları, bilgisayarların evlere girdiği tarihten itibaren özellikle çocukların ve gençlerin kullandığı eğlence aktivitelerinden biridir. En kısa yol bulma algoritmaları özellikle strateji oyunlarında ızgara mantığı ile oyun haritalarının belli karelere bölünerek, oyunun amacı doğrultusunda ağırlıklı veya ağırlıksız graflar ile başta A* algoritması olmak üzere Dijkstra ve diğer algoritmaların da kullanıldığı algoritmalardır. Günümüzde Supercell şirketinin Clash of Clans oyunu yaklaşık 917 milyon Euro yaklaşık 4 trilyon kar elde etmiştir. Bu oyunların başarılı olmasının nedeni tasarım ve oynayış biçiminden sonra yazılan oyunların takılmadan ve hızlı bir biçimde oynanabilmesidir [2].

Mobil telefonlarda algoritmaların doğru seçimi telefonların işlemcilerinin kullanım oranını doğrudan etkileyen bir faktördür. İşlemci ne kadar fazla çalışırsa ve işlem yaparsa telefon o kadar fazla pil tüketecektir. Mobil aygıtlarındaki en önemli etken pil faktörüdür. Yapılan uygulama veya oyun ne kadar fazla pil tüketiyorsa kullanıcı o kadar memnuniyetsiz olacaktır. Algoritmaları yazan ve programlayan kişilerin verimli bir şekilde kodu yazıp en doğru algoritmaları seçmeleri gerekmektedir.

Şekil 2.1'de Westwood firmasının 1998 yayınladığı bir video strateji oyunu olan Dune2000 görülmektedir [3]. Dune2000'deki üniteler fare ile kontrol edilmektedir. Bir üniteyi seçtiğinizde ve bir başka noktaya tıkladığınızda seçilen ünite en kısa yol bulma algoritması olan A* algoritmasına sokularak hedef gösterilen noktaya engelleri aşarak en kısa mesafeden gitmeye çalışacaktır. Şekil 2.1'deki kırmızı yuvarlak içindeki objeye tıklandığında ve yine aynı şekilde mavi yuvarlak ile gösterilen yere tıklandığında ünite, binalara çarpmadan gidilebilecek en kısa yolu bulmak için en kısa yol bulma algoritmasını çalıştıracaktır.

Şekil 2.1. Dune 2000 Oyunu Ekran Görüntüsü

Algoritma ekrandaki görselleri anlamlandıramadığı için ilk olarak yapılması gereken oyunun içindeki objelerin bir dizi şeklinde algoritmaya verilmesidir. Bunun için oyun ilk olarak haritayı Şekil 2.2'deki gibi bir ızgara şeklinde böler. Bölünen bu harita 3 farklı nokta olarak doldurulur. Bunlardan birincisi kaynak yani en kısa yol algoritmasının hangi noktadan başlayacağı, ikincisi hedef nokta, yani objenin başlangıç noktasından başlayarak hangi notaya kadar gideceğini belirten nokta, üçüncüsü ise başlangıçtan bitişe gidene kadar aralarda var olan oyun içinde evler, üniteler, kayalar, taşlar şeklinde görülebilen engellerdir.

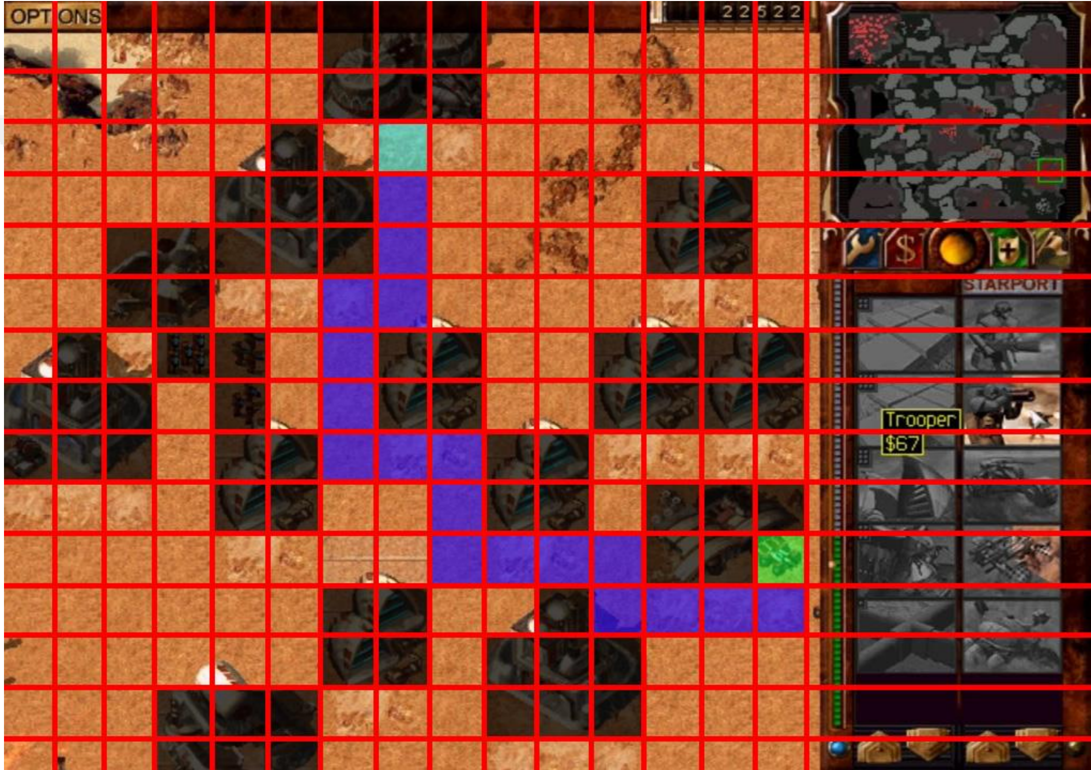


Şekil 2.2. Haritanın Izgaralara Bölünmüş Hali

Şekil 2.3'te görüldüğü gibi başlangıç noktası yeşil ve bitiş noktası turkuaz boyanmıştır. Başlangıçtan bitişe giderken ki engeller ise siyah renkte gösterilmektedir. Haritada bu tanımlar yapıldıktan sonra bu değerler yazılan programlama dilinin tasarımına bağlı olarak sayılara veya objelere dönüştürülür. Örneğin haritadaki boş alanlar 0 ile, engel olan alanlar ise 1 ile oluşturulmuş bir dizi oluşturulup bu dizi en kısa yol bulma algoritmasına sokulur.

Şekil 2.3. Haritanın Engeller Koyulmuş Hali

Burada dikkat edilecek olan kısım bu oyunda ağırlık değerleri tüm boş hücreler için 0 olarak kabul edilmiştir. Buna bağlı olarak herhangi iki hücrenin maliyeti dikkate alınmaksızın herhangi bir boş hücreden seçim yapılabilir. Algoritma çalıştırıldığında dizideki başlangıç, bitiş ve engeller hesaba katılarak, oyunun içinde kullanılan en kısa yol algoritmasına göre bir noktalar dizisi elde edilir. Bu noktalar dizisi Şekil 2.4'te mavi renklerle gösterilmiştir. Bu hesaplamadan sonra ünitenin bu en kısa yol noktalarına sıra sıra gitmesi sağlanır. Böylece oyun içinde bir noktadan diğer noktaya engeller üzerinden geçmeden gidilebilecek en kısa yol hesaplanmış olur.

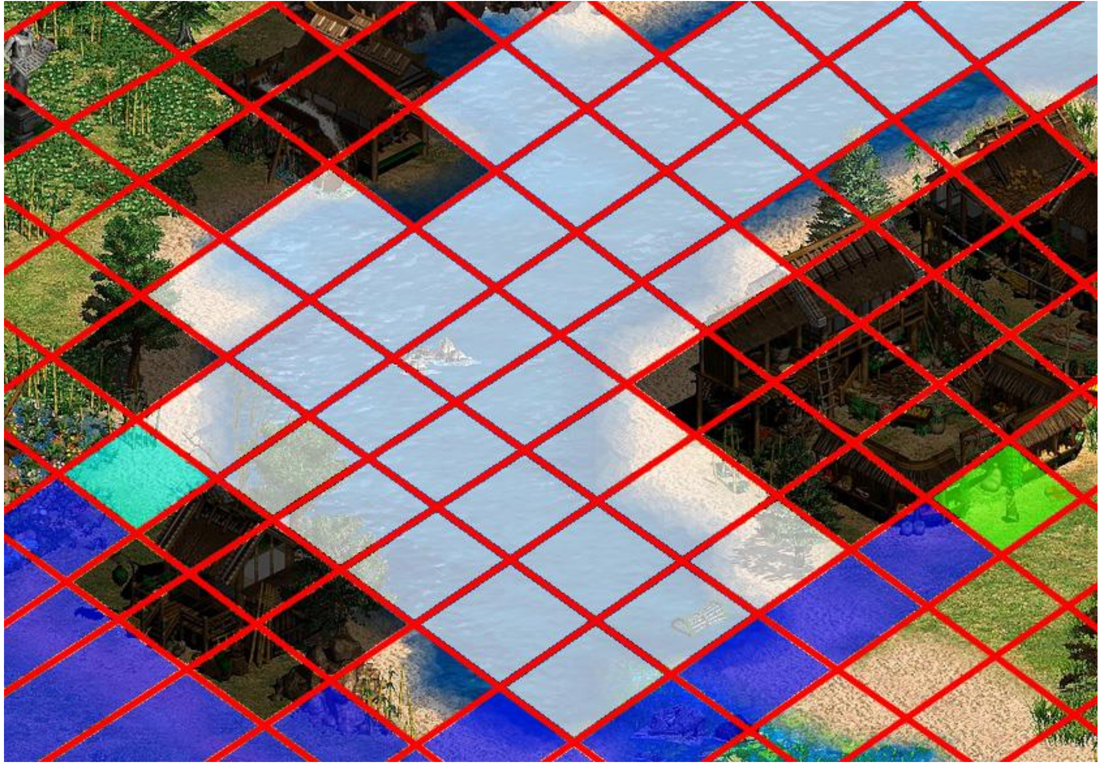


Şekil 2.4. Haritanın Kısayol Çizilmiş Hali

Şekil 2.5'teki örnekte ise Microsoft Game Studios tarafından 1998 yılında geliştirilen Age Of Empires versiyon 2 video oyunundan bir görüntü vardır [4]. Şekilde görüldüğü gibi oyun, algoritmanın anlayabileceği şekilde ızgaralara bölünür. Haritada hiçbir türlü geçilemeyen yerler siyah renkli olarak belirtilmiştir. Burada yeşil noktadan turkuaz renkli noktaya gidilmek istendiğinde önceki oyundan farklı olarak ağırlıklı graflar kullanılmıştır.

Şekil 2.5. Age of Empires Oyun Görüntüsü

Şekil 2.6’da beyaz ile gösterilen yerler suyu temsil etmektedir. Çim üzerinden geçme maliyeti 1 ve su üzerinden geçme maliyeti 2 olarak belirtildiğinde ve algoritma çalıştırıldığında ünitenin şekilde gösterilen mavi yoldan gittiği görülmektedir. Burada dikkat edilecek olan kısım aslında en kısa yolun mavi değil yeşil noktadan başlayarak ve beyaz noktalar üzerinden geçerek turkuaz renge giden yolun en kısa yol olmasıdır. Maliyet tabanlı algoritma burada devreye girerek beyaz noktalar üzerinden gidilen yol daha kısa olsa bile beyaz noktaların maliyetinin fazla olmasından dolayı toplamda en az maliyetli en kısa yolu bulduğundan mavi yolu en kısa yol olarak hesaplayacaktır.

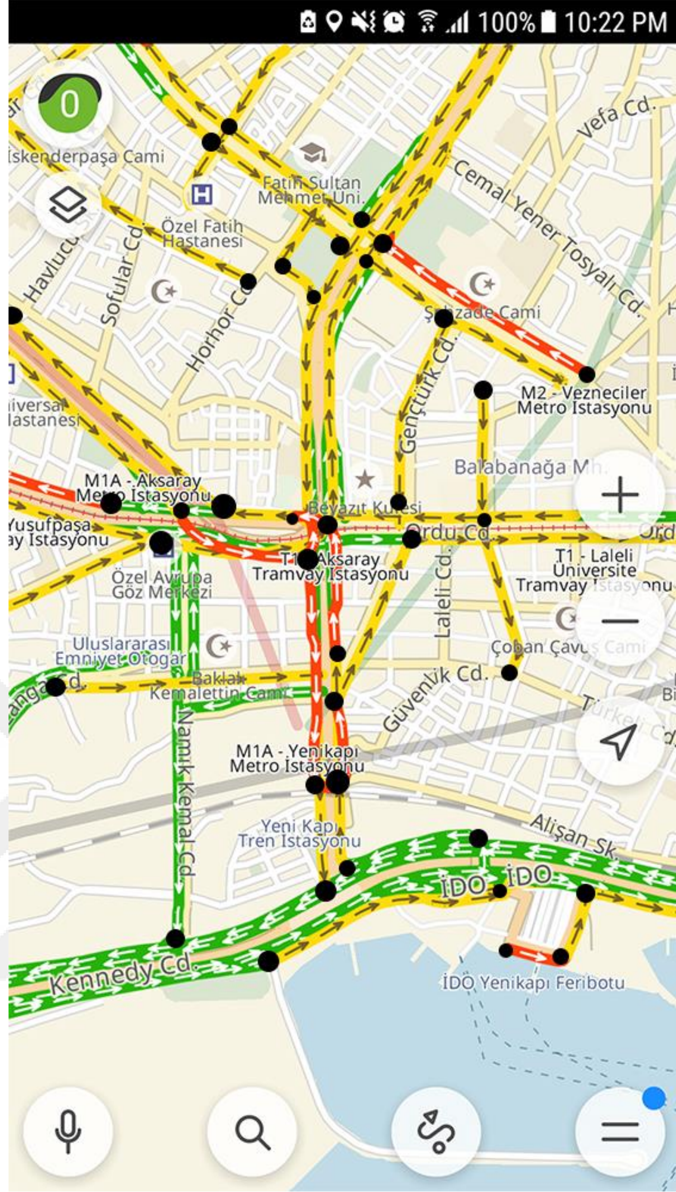


Şekil 2.6. Age of Empires Oyununun Maliyetli Görüntüsü

2.2. En Kısa Yol Algoritmalarının Kullanıldığı Uygulama ve Navigasyon Programlarının İncelenmesi

En kısa yol algoritmalarının bir başka kullanıldığı alan ise navigasyon uygulamalarıdır. Bu uygulamalarda kullanıcıların bulundukları koordinat harita üzerinde gösterilir. Kullanıcılar gitmek istedikleri bir yer seçtiklerinde uygulama kullanıcının bulunduğu noktadan hedef gösterdiği noktaya doğru bir yol çizer. Bu yol, belli maliyetler verilmiş yollar ile bu yolların kesiştiği noktalar üzerinde bulunur. Graflarda maliyet verilebilmesi navigasyon uygulamalarının vazgeçilmezidir. Özellikle trafik olan yollarda bu maliyet değeri daha fazla olacaktır. Kullanıcılar uygulama baktıklarında haritada herhangi bir nokta ve maliyet değerleri görmezler, fakat arka planda bir noktadan başka bir noktaya en kısa yoldan gidebilmek için en kısa yol bulma algoritmaları kullanılır [5].

Şekil 2.7'de Yandex şirketinin 2004 yılında yayınladığı Android uygulaması Yandex Map'ın ekran görüntüsü gösterilmektedir. Bu uygulamada boş yollar yeşil renkte, orta trafikli yollar sarı renkte, çok trafikli yollar kırmızı renkte gösterilmektedir [6]. Siyah noktalar ise sonradan çizilmiştir ve harita üzerindeki düğüm noktalarını belli etmek için çizilmiştir Örnek verilecek olursa yeşil yollara 10 değerinde bir maliyet, sarı yollara 20 değerinde bir maliyet ve kırmızı yollara 30 değerinde bir maliyet verilirse ve muhakkak bu yollardan geçen bir hedef verildiğinde maliyet değeri fazla olan yollar algoritmada fazla maliyetli çıkacağından en kısa yol hesaplanırken bu yollar tercih edilmeyecektir. Tatbikî bu sonuçlar kullanıcıya maliyet değerleri ile değil bu kısımlarda trafik olduğunu gösteren kırmızı çizgiler ile belirtilir. Böylelikle kullanıcı en kısa yolu harita üzerinde görmüş olur.



Şekil 2.7. Yandex Uygulamasının Ekran Görüntüsü

3. ALGORİTMALAR

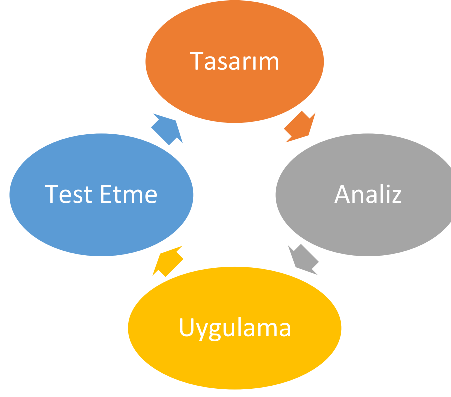
3.1. Algoritmaların Tarihi

Algoritma kelimesi 780 yılında bugünkü Türkmenistan'ın Hive kentinde doğmuş olan Ebu Abdullah Muhammed İbn Musa el Harezmi'den gelmektedir. Bilinen ilk algoritmalar, el Harezmi tarafından "Hisab el-cebir ve el-mukabala" kitabında sunulmuştur [7]. Bu alim dokuzuncu yüzyılda cebir alanındaki algoritmik çalışmalarını bu kitaba yazarak matematiğe çok büyük bir katkı sağlamıştır. "Hisab el-cebir ve el-mukabala" kitabı dünyanın ilk cebir kitabı olarak bilinir ve aynı zamanda ilk algoritma koleksiyonunu oluşturur.

Algoritma sözcüğü Avrupalıların telaffuzundan oluşmuştur. Kitabın Latince çevirisi Avrupa'da büyük ilgi görür ve alimin ismini telaffuz edemeyen Avrupalılar "algorizm" sözcüğünü "Arap sayıları kullanarak aritmetik problemler çözme kuralları" anlamında kullanırlar. Bu sözcük daha sonra "algoritma" ya dönüşür ve günümüzde bu şekilde kullanılmaktadır [7].

3.2. Algoritmaların Mantığı

Algoritma, bir sorunu veya problemi çözmek için izlenecek yol veya belirli bir sonuca ulaşmak için oluşturulan mantıksal çözümlere verilen ortak isimdir. Matematikte ve bilgisayar bilimlerinde çok fazla kullanılan algoritmalar bir başlangıç durumundan başlayıp, bir son durumunda sonlanan, yapısına bağlı olarak farklı yollar kullanabilen sonlu işlemler kümesidir. Algoritmaların en önemli kısmı problemin çözümü için en uygun işlevler akışını hazırlayabilmek ve geliştirmektir. Problemin çözümü adımlar halinde yazılmasıyla oluşturulur. Genel olarak algoritma Şekil 3.1'de gösterilen aşamaları izleyerek oluşturulur.



Şekil 3.1. Algoritmaları oluşturma aşamaları

Her adım olabildiğince belirleyici olmalıdır ve adımlar şansa bağlı olmamalıdır. Belirli bir adım sonunda algoritma sonlanmalıdır. Algoritmalar karşılaşılabilecek tüm ihtimalleri ele alabilecek kadar genel olmalıdır. Ne kadar karmaşık olursa olsun, birim zamanda hızlı işlemler yapabildiği için genellikle bilgisayar bilimlerinde ve programlamada kullanılır. Algoritmalar tek bir problemi çözmek için temel işleri yapan işlemlerin veya deyimlerin sıra sıra ortaya çıkmasıdır ve bu adımların sıralaması önemlidir.

3.3. Algoritmaların Yapısı

Günümüzde bilgisayarların işlemcileri (CPU - Central Processing Unit) ve bellekleri yeterince geliştiği için büyük veya çok adımlı algoritmaları kolay bir şekilde bir insana göre çok çok kısa sürelerde derleyip sonuçları çıktılar veya karşılaştırmalar ile ortaya koyabilir. Algoritmalar programlama diline bağlı bir yapı değil sadece bir düşünce biçimidir. Farklı programlama dillerinde yazılabilir, bu kısımda algoritmanın akışı aynı olacaktır sadece program dili biçimi değişecektir.

Algoritmalarda bir problemi çözerken algoritmik ve sezgisel (heuristic) denilen iki farklı yaklaşım söz konusudur [8]. Algoritmik yaklaşımda, çözüm için gerçekleşebilen yöntemlerden en uygun olanı seçilir ve yapılması gerekenler adım adım ortaya konulur. Sezgisel yaklaşımda ise, çözüm belirli bir şekilde ortada değildir. Tasarımcının deneyimi, birikimi ve o andaki düşüncesine göre problemi çözecek bilgilerin şekillendirilmesiyle yapılır. Program tasarımcısı, algoritmik yaklaşımla çözemediği sorunları sezgisel yaklaşımlarla çözmek için bu yöntemi kullanır.

Algoritmalar genellikle tek başlarına patent verilebilir özellikte bir buluş olarak değerlendirilmez. ABD (Amerika Birleşik Devletleri)'nde soyut kavramlar, sayılar ve işaretler yalnızca basit yönlendirmelerinden oluşan bir iddia "süreç" olduğundan dolayı algoritmalar patent verilebilir değildir (bkz: Gottschalk v.Benson'da olduğu gibi USPTO 2006)[9]. Bunların dışında, algoritmaların bazı pratik uygulamalarına zaman zaman patent verilmiştir. Örneğin, Diamond v.Diehr'da, sentetik kauçuğun muhafaza edilmesine yardımcı olmak için kullanılan basit geri bildirim algoritmasının uygulamasına patent verilmiş ve bu algoritma kabul edilebilir sayılmıştır [10]. Günümüzde özellikle yazılım patentleri son derece tartışmalı bir konudur. Veri sıkıştırma algoritmaları, Unix dosya sıkıştırmada ve GIF resim formatında kullanılan Unisys'LZW patentinde olduğu gibi [11].

3.4. Sözde (Pseudo) Kod

Sözde kod, bilgisayar bilimlerinde program yazarken veya algoritma oluştururken herhangi bir programlama dili kullanmadan genel konuşma dili ile yazılan, derlendiğinde bir sonuç çıkarmayan ve basit kelimelerle anlatılmış akış cümlecikleridir. Sözde kod için mutlaka uyulması gereken herhangi bir standart veya kural yoktur. Bu yazım tarzı ile amaç platformdan bağımsız olarak kişiye istenilen akışı basit bir şekilde anlatmaktır.

3.5. Karmaşıklık (Complexity)

Algoritmaların önemli bir parçası olan karmaşıklık, algoritmanın verilen değer sonsuza giderken ki maliyetini verir. Algoritmaya sokulacak veri küçük ise hafıza (space) ve zaman (time) karmaşıklığı dikkate alınmayacak kadar küçük olacaktır. Örneğin 3 noktalı bir graf üzerinde en kısa yol bulma hesaplanırken hangi algoritmanın çalıştırıldığı ve ne kadar zamanda ne kadar bellek harcadığı dikkate alınmayacak kadar küçük farklar ile sonuçlanır. Algoritmalarda fonksiyonların büyüme hızını matematikte "big-o" olarak ifade edilen fonksiyon gösterir. Örneğin "n" sonsuza giderken bazı fonksiyonların büyüklükleri şu şekilde sıralanabilir $1 < \log(n) < n < n * \log(n) < n^2 < n^3 < 2^n < n!$.

3.6. Açgözlü Yaklaşımı (Greedy Approach)

Açgözlü yaklaşım bir algoritma geliştirme yaklaşımı olarak yorumlanabilir. Genellikle çok hızlı karar verme ile çalışan basit bir mantıktır. Karşılaştırma yapılıp karar verilirken sonuca en yakın olan seçim yapılır. Bir seçim yapılması istendiğinde bizi sonuca en çok yaklaştıracak seçimi yapmaya çalışır. Fakat tüm yolları denemeden hızlı karar veren bu seçim toplamda her zaman en iyi seçim olmayabilir. Örneğin, para üstü probleminde (Coin Exchange Problem) 18 liralık bir para üstü verilecekse ve bozuk para birimleri 15 - 14 - 9 - 4 - 1 ise açgözlü yaklaşım kullanıldığında aç gözlü yaklaşım sonuca en yakın sayıları seçecektir. 18'e en yakın olan 15'i ardından geri kalan 3TL yi ise 1TL lik 3 adet bozuklukla toplam 4 bozuk para vererek sonuca ulaşacaktır. Fakat aynı problem 14 + 4 şeklinde sadece 2 bozuk para vererek daha az sayıda çözülebilirdi.

3.7. Bazı Algoritma Türleri

Algoritmalar kullanım alanlarına göre birçok alt başlığa ayrılırlar. Algoritmaların belli bir kuralı ve sayısı bulunmamaktadır. Günümüzde yapay zekâ alanının da bir alt konusu olan algoritmalar robotik ve harita alanlarında da çokça kullanılmaktadır. Algoritmaları çokça kullandığı bilinen Google'ın da kendi içinde çok miktarda algoritma oluşturduğu bilinmekte ve Google arama motorunun arama faktörünü etkileyen yaklaşık 200 farklı algoritmanın çalıştığı söylenmekte [12].

3.7.1. Arama Algoritmaları

Bir veri yapısı üzerinde, örneğin bir yazı içeriğinde veya bir dizi içinde istenilen karakter veya kelimenin bulunması için kullanılan algoritmalardır. Doğrusal Arama ve İkili arama örnek olarak verilebilir.

3.7.2. Kriptografik Algoritmalar

Verilerin üçüncü şahıslar tarafından elde edilememesi için verilerin şifrelenmesi için kullanılan algoritmalardır. Bu algoritmalar anahtarlı veya anahtarsız, eğer anahtarlı ise simetrik ve asimetrik olarak alt dallara ayrılırlar.

3.7.3. Sıralama Algoritmaları

Verilen karışık veya ters sıralı sayıları, sıralı bir şekilde sıralamaya yarayan algoritmalardır. Kabarcık Sıralaması, Birleştirme Sıralaması, Yığın Sıralaması, Kabuk Sıralaması örnek olarak verilebilir.

3.7.4. Genetik Algoritmalar

Bu algoritma türü problemleri tek bir çözüm ile çözmek yerine farklı çözümlerden oluşan bir çözüm kümesi üretir. Genetik algoritmalar problemlerin çözümü için evrimsel olan süreci bilgisayar ortamında simüle ederek çalışırlar.

3.7.5. Veri Sıkıştırma Algoritmaları

Veri yığınlarının veya bilgisayar terimlerinde dosyaların, veri yığını içindeki karakterlerin belli tekrarlarını azaltarak yığının veya dosyanın kapladığı alanı küçültmeye yarayan algoritmalardır. Huffman algoritması ve LZW algoritması örnek olarak verilebilir.

3.7.6. En Kısa Yol Bulma Algoritmaları

Belirli bir harita üzerinde ağırlıklı veya ağırlıksız yollar üzerinden bir noktadan başlayıp engeller ve tek veya çift yönler ile beraber gidilebilen en kısa yolu hesaplamaya yarayan algoritmalardır.

3.8. En Kısa Yol Bulma Algoritmalarının İncelemesi

Örneğin Google'ın bir servisi olan "Google Map", bir kullanıcıyı bulunduğu konumdan, kullanıcının koordinatını verdiği başka bir yere giderken ki yolu en kısa yol bulma algoritmaları ile bulmaktadır. En kısa yolu bulma algoritması, verilen iki nokta arasındaki en kısa yolu bulan bir algoritmadır.

Dijkstra, Bellman-Ford, Floyd-Warshall ve A* en kısa yol bulma algoritmalarının çalışma mantığı örnekler üzerinde anlatılacaktır. Algoritmaların kendi özelliklerine bağlı olarak avantaj ve dezavantajları belirtilecektir.

3.8.1. Dijkstra Algoritması

3.8.1.1. Nedir?

Dijkstra algoritması, belli metrik ağırlıklı değerlere sahip yollar üzerinde bir noktadan diğer noktaya en kısa mesafeyi bulan bir algoritmadır. Hollandalı bilgisayar bilimcisi ve matematikçi 1930 doğumlu Hollandalı Edsger W. Dijkstra tarafından 1956 yılında geliştirilmiştir. Algoritma başta harita üzerinde yol bulma olmak üzere, navigasyonlarda, bilgisayar oyunlarında, robotların kendi kısa yollarını bulmasında, askeri alanlarda kullanılmaktadır [13]. Algoritma oluşturulduktan sonra birçok türevi yapılmış olsa da en çok bilinen ve kullanılan versiyonu; bir "kaynak" noktasını belirlenip bu kaynak noktasından tüm diğer noktalara maliyetleri üzerinden geçerek hesaplanan versiyonudur [13].

Algoritmanın bir diğer çok kullanılan alanı Ağ Yönlendirme Protokolleri olarak verilebilir. Bu ağ sistemi, internet üzerinde IP (Internet Protocol Address) adresi alan bir bilgisayarın internete çıktığında yönlendirici sunucuları üzerinden internette istenilen adrese en kısa yoldan gidebilmesi için Dijkstra algoritmasından yararlanır [13]. Bu algoritma ile internete çıkan veri paketi, dünya üzerindeki bütün yönlendiricileri dolaşmadan gideceği hedefe algoritmadan çıkan sonuçlara göre sadece gerekli yönlendiriciler üzerinden giderek hem internet trafiğini gereksiz yere yormayacak hem de kullanıcının gönderdiği veri paketinin hızlı bir şekilde hedefe varmasını sağlayarak kullanıcıyı bekletmeyecektir [13].

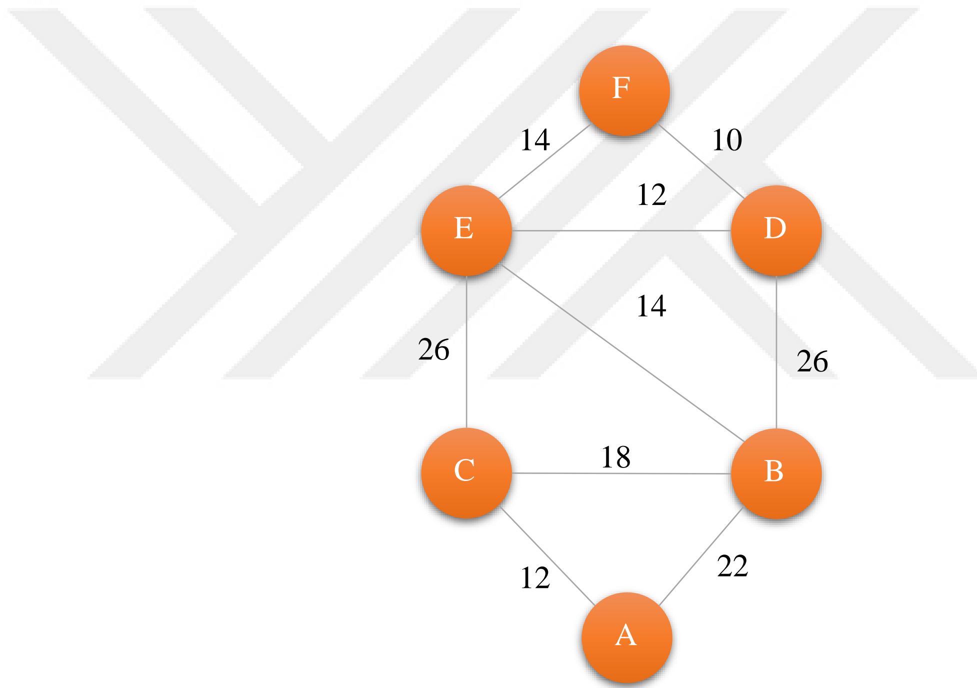
Algoritmanın sağlıklı bir şekilde kullanılabilmesi için bazı koşulların sağlanması gerekmektedir. Eğer maliyetlerden bir tanesine negatif bir değer verilirse algoritmadaki sonuçlar hatalı ve tutarsız olacaktır ve buna bağlı olarak sonuçlardan çıkan yollar en kısa yol olmayacaktır. Yollardaki maliyetlere yine de negatif değerler verilmek isteniyorsa "Bellman-Ford" algoritması kullanılabilir. Dijkstra algoritması Aç Gözlü (Greedy) bir algoritmadır. Algoritmanın zaman karmaşıklığı yani büyük o notasyonu O(ElogV)'dir. Buradaki "V" noktaların toplam sayısını, "E" ise yolların toplam sayısını belirtmektedir [14].

Örneğin bir ilin doğalgaz veya su boru döşeme yolları yapılacak ise Dijkstra algoritması kullanılarak yollar ne kadar karmaşık olursa olsun alternatif yollardaki maliyetler hesaplanarak en kısa yoldan en ucuz maliyetli yol bulunabilir. Algoritma, değerlendirilmiş bir noktalar dizisi ve diğer tüm değerlendirilmemiş noktalara olan

mesafeleri sürekli olarak güncelleyerek çalışır. Her bir adımda değerlendirilmemiş en yakın nokta, değerlendirilmiş noktalar kümesine eklenir ve daha sonra bu noktanın direk bağlı ve değerlendirilmemiş olan noktalara uzaklıkları güncellenir.

3.8.1.2. Çalışma Mantığı

Dijkstra algoritmasının çalışma mantığı Şekil 3.2'de gösterilen graf üzerinden anlatılacaktır. Bu graf 6 adet noktadan ve 9 adet yoldan meydana gelmektedir. Amaç A noktasından F noktasına gitmek olsun. Dijkstra algoritması kullanılarak A noktasından F noktasına en kısa yoldan nasıl gidileceği hesaplanacaktır. Yolların üzerindeki maliyet sayıları o yolun hangi maliyette olacağını göstermektedir.



Şekil 3.2. Dijkstra Örnek Haritası

İlk bakışta en kısa yol, birbirine en yakın noktaların oluşturduğu yollar üzerinden çıkacağı zannedilebilir. Fakat bu yollar üzerindeki maliyetler gidilen yolu çok farklı şekillerde değiştirebilir. Bunun için noktaları, noktalar arasında gidilebilen yolları, bu gidilen yol üzerindeki maliyetleri hesaplayarak bir tablo oluşturulmalıdır. Daha sonra bu tablodan Dijkstra algoritmasına özel bazı hesaplamalar yapılıp çıkan değerler değerlendirdikten sonra gidilebilen en kısa yol elde edilebilir [15].

Tablo 3.1'deki gibi bir tablo oluşturulması ve bu tablonun kolonları Şekil 3.2'de bulunan noktalar olacak şekilde doldurulması gerekir. "Seçili" olarak belirtilen tablo kolonu, ileride hangi noktanın seçilmiş olduğunu belirtecektir. Kolon tanımlamaları yapıldıktan sonra birinci satırın oluşturulmalıdır. Bu satırda, yolun hangi noktadan başlaması gerekiyor ise o noktaya "0" değeri verilmeli ve diğer tüm noktalara "∞" değeri verilmelidir. Bu sayısal değerler sayısal karşılaştırmalarda kullanılacaktır.

Tablo 3.1. Dijkstra Başlangıç Tablosu

	Seçili	**A**	**B**	**C**	**D**	**E**	**F**
1.		0	∞	∞	∞	∞	∞

Tablo 3.1'deki gibi bir satırın tüm değerleri tamamlandığında bir sonraki satıra geçilir. Burada önemli olan nokta bir satır tamamlandığında, o satırdaki tüm değerlere bakılır ve aralarında en küçük değerli nokta bir alt satıra aynı değer ile geçirilir. Tablo 3.1'deki değerlere bakıldığında en küçük değerli noktanın "A" noktası olduğu ve diğer tüm noktaların "∞" olduğu görülmektedir. Bundan dolayı yeni bir satır oluşturulur ve "A" noktası "Seçili" noktadır denir. "Seçili" kolonuna "A" yazılır. "A" noktası dışındaki değer noktalar ise ikinci satıra Tablo 3.2'deki gibi aynen aktarılır.

Tablo 3.2. İkinci Satıra Aktarma İşlemi

	Seçili	**A**	**B**	**C**	**D**	**E**	**F**
1.	A	0	∞	∞	∞	∞	∞
2.		0					

Seçili noktanın alt satıra aktarma işleminden hemen sonra diğer değerler hesaplanır. Bu kısımda ilk önce yapılacak işlem seçilen noktanın direk olarak bağlı olduğu noktaları bulmaktır. Şekil 3.2'de "A" noktasının direk olarak bağlandığı noktaların "B" noktası ve "C" noktası olduğu görülmektedir. "D", "E" ve "F" noktalarının "A" noktasına direk olarak bağlantısı yoktur. Bundan dolayı Dijkstra

algoritması “D”, “E”, “F” noktalarının bağlantı değerinin “∞”’ olması gerektiğini söylemektedir. Bu yüzden bu noktaların değerleri Tablo 3.3’teki gibi “∞” yapılmalıdır.

Tablo 3.3. İkinci Satırı Doldurma İşlemi

	Seçili	**A**	**B**	**C**	**D**	**E**	**F**
1.	A	0	∞	∞	∞	∞	∞
2.		0			∞	∞	∞

“B” ve “C” noktalarında ise minimum fonksiyonu kullanılmalıdır. “Minimum değer fonksiyonu” şu şekilde hesaplanmaktadır:

f(x) = min(Hedef Değer, Seçili Değer + Yol Maliyeti)

Buradaki değişkenlerin açıklamaları aşağıda verilmiştir.
X: Maliyeti hesaplanacak olan nokta
Hedef Değer: Hesaplamayı yaparken sırası gelen noktanın değeri.
Seçili Değer: Bir üst satırdan gelirken “Seçili” kolonuna yazılan noktanın değeri.
Yol Maliyeti: Hesaplanan iki nokta arasında verilmiş olan maliyet.

Matematiksel fonksiyondan çıkartılacak sonuç şu şekildedir; “Hedef Değer” ile “Seçili Değer” ve “Yol Maliyeti” değerlerinin toplamının, karşılaştırılıp bu iki değerden en küçüğü hangisi ise onun sonuç olarak alınmasıdır.

Bu fonksiyona bağlı kalarak “A” ve “B” noktalarının değerleri hesaplanırsa, “B” noktasından gelen değerin “∞” olduğunu görülür ve “Hedef Değer”, “∞” olarak yazılır. “Seçili Değer” ise “A” noktası olduğundan ve bu noktanın değerinin bir üst satırdan gelen “0” olduğu görülmektedir. “Yol Maliyeti” ise Şekil 3.2’de görülen “A” noktası ile “B” noktasının arasındaki maliyet olan “22” olacaktır. Böylelikle fonksiyon *min(∞, 0+22)* şeklinde tanımlanabilir. Bu fonksiyondan “∞” ile “22” değerini karşılaştırıldığında “22” değerinin “∞” değerinden daha küçük olduğu görülür ve “B” noktasının hesaplanan yeni değeri “22” olarak bulunur.

Bulunan bu değerler Tablo 3.4'teki gibi "B" kolonunun altına yazılır. Bu hesaplamadan sonra ikinci satırın tüm hesaplamaları tamamlanmış olur. Bu kısımda yapılacak işlem tabloda hiç seçilmemiş nokta olup olmadığına bakmaktır. Tabloda sadece "A" noktasının seçilmiş, diğer noktaların hiç seçilmemiş olduğunu görülmektedir. Dijkstra algoritmasına göre yapılacak işlem tabloda tüm noktalar seçilmiş olana kadar işlemlerin devam etmesidir. "B", "C", "D", "E" ve "F" noktalarının hiçbiri daha seçilmiş olarak işaretlenmediğinden yeni bir satır oluşturup hesaplamalara devam edilmelidir.

Tablo 3.4. İkinci Satırı Hesaplama İşlemi

	Seçili	**A**	**B**	**C**	**D**	**E**	**F**
1.	A	0	∞	∞	∞	∞	∞
2.		0	min(∞, 0+22) 22	min(∞, 0+12) 12	∞	∞	∞

Daha önce hiç seçilmemiş olan noktalar karşılaştırılır ve içlerinden en küçük olanı bulunur. Tablo 3.4'e bakıldığında en küçük değerin "C" noktasının "12" değeri olduğunu görülmektedir. Bundan dolayı "C" noktası seçilmiş yeni nokta olarak belirlenir.

Tablo 3.5'deki gibi 2. satırın seçili değeri olarak "C" noktası yazılmalıdır ve daha önceden seçilmiş tüm seçili değerler 3. satıra aynen yazılır. Bu kısımda seçili nokta "C" noktası olduğu için sadece "C" noktası ile direk bağlantısı olan noktaların değerleri "Minimum değer fonksiyonu" ile hesaplanır ve diğer noktalara önceden yapıldığı gibi "∞" değeri yazılır.

Tablo 3.5. Üçüncü Satıra Aktarma İşlemi

	Seçili	**A**	**B**	**C**	**D**	**E**	**F**
1.	A	0	∞	∞	∞	∞	∞
2.	C	0	min(∞, 0+22) 22	min(∞, 0+12) 12	∞	∞	∞
3.		0		12			

Tablo 3.6'da "C"nin "B" noktasına ve "E" noktasına olan maliyetleri de hesaba katılarak 2. satırda yapıldığı gibi "Minimum Değer Fonksiyonu" ile "22" ve "38" sonuçları bulunur.

Tablo 3.6. Üçüncü Satırı Hesaplama İşlemi

	Seçili	**A**	**B**	**C**	**D**	**E**	**F**
1.	A	0	∞	∞	∞	∞	∞
2.	C	0	min(∞, 0+22) 22	min(∞, 0+12) 12	∞	∞	∞
3.		0	min(22, 12+18) 22	12	∞	min(∞, 12+26) 38	∞

Bütün noktaların değerleri tamamlandığında seçili (gri ile boyanan) değerler dışındaki değerler kontrol edilir ve en düşük değerli nokta, yeni seçili nokta olarak belirlenir. Tablo 3.6'dan görülebileceği gibi "B" noktası en düşük değerli nokta olduğu için yeni seçili değer "B" noktası olacaktır. Aynı şekilde 4. satır oluşturulup değerleri "Minimum Değer Fonksiyonu" ile hesaplandıktan sonra Tablo 3.7 elde edilir.

Tablo 3.7. Dördüncü Satırı Hesaplama İşlemi

	Seçili	**A**	**B**	**C**	**D**	**E**	**F**
1.	A	0	∞	∞	∞	∞	∞
2.	C	0	min(∞, 0+22) 22	min(∞, 0+12) 12	∞	∞	∞
3.	B	0	min(22, 12+18) 22	12	∞	min(∞, 12+26) 38	∞
4.		0	22	12	min(∞, 22+26) 48	min(38, 22+14) 36	∞

5. ve 6. satırlar da aynı şekilde hesaplandıktan sonra Tablo 3.8'de belirtildiği gibi 6. Satırda tüm noktaların seçilmiş olduğu görülmektedir. Noktaların seçili olarak geldiği tüm adımlar gri rengine boyanmıştır. Son adımda tüm adımlar halihazırda dolaşıldığı için artık yeni bir satıra gerek yoktur ve aynı zamanda "∞" değerli bir nokta kalmamıştır.

Tablo 3.8. Beşinci ve Altıncı Satırları Hesaplama İşlemi

	Seçili	**A**	**B**	**C**	**D**	**E**	**F**
1.	A	0	∞	∞	∞	∞	∞
2.	C	0	min(∞, 0+22) 22	min(∞, 0+12) 12	∞	∞	∞

3.	B	0	min(22, 12+18) 22	12	∞	min(∞, 12+26) 38	∞
4.	E	0	22	12	min(∞, 22+26) 48	min(38, 22+14) 36	∞
5.	D	0	22	12	48	36	min(∞, 36+14) 50
6.	F	0	22	12	48	36	min(50, 48+10) 50

Bu kısımda artık işlemlerin başında istenilen “A” noktasından “F” noktasına en kısa yoldan gelme işlemi tamamlanmış ve “F” noktasına ne kadar maliyet ile gelindiği belirlenmiştir. Öğleki “F” noktasına Dijkstra algoritması kullanılarak tabloda belirtilen değeri “50” olan maliyet ile gelinmiştir.

Şimdi ise Tablo 3.9 kullanılarak yolun hangi noktalarından geçilerek “F” noktasına varıldığı tespit edilmelidir. Bunun için Dijkstra için “Geri İzleme” denilen metot kullanılır. Varılan son adımdan başlayarak tablo üzerinde adım adım ilerlenmesi gerekmektedir.

Tablo 3.9. Dijkstra Tablosunun Son Hali

	Seçili	A	B	C	D	E	F
1.	A	0	∞	∞	∞	∞	∞
2.	C	0	min(∞, 0+22) 22	min(∞, 0+12) 12	∞	∞	∞
3.	B	0	min(22, 12+18) 22	12	∞	min(∞, 12+26) 38	∞
4.	E	0	22	12	min(∞, 22+26) 48	min(38, 22+14) 36	∞

5.	D	0	22	12	48	36	min(∞, 36+14) 50
6.	F	0	22	12	48	36	50

En kısa yol adımlarının en son noktası “F” noktası olarak görülmektedir. Tablo 3.10 üzerinde gösterildiği gibi “F” noktasının son satırı 6. satırında “50” olan değer ile bir üst satırındaki değer karşılaştırılır. Eğer karşılaştırılan iki değer eşit ise tekrar bir üst satıra gidilir (4. satır) ve 6. Satır pas geçilerek 5. Satırdaki değer ile 4. Satırdaki değer karşılaştırılır.

Tablo 3.10. Altıncı ve Beşinci Satırların Karşılaştırılması

5.	D	0	22	12	48	36	50
6.	F	0	22	12	48	36	50

Tablo 3.11’de sarı ile işaretlenen 4. ve 6. satır değerleri “50” ve “∞” değerleri görülmektedir. Bu iki değeri karşılaştırıldığında farklı olduğu görülmektedir. Bu durumda 4. Satırda “Seçili” noktaya gidilmesi gerekir. Bu satırdaki seçili nokta, daha önce tablodaki “Seçili” kolonuna yazılan değerlere bakılarak bulunur. 4. satıra bakıldığında “Seçili” noktanın “E” noktası olduğu görülmektedir. Bundan dolayı bir sonraki en kısa yol noktalarının sondan 2. Noktası “E” noktası olarak belirlenir. Son nokta olarak bulunan ilk nokta “F” olduğundan en kısa yolun sondan ikinci noktası “E” > “F” olarak en kısa yol şekillenmeye başlayacaktır.

Tablo 3.11. Beşinci ve Dördüncü Satırların Karşılaştırılması

4.	E	0	22	12	48	36	∞
5.	D	0	22	12	48	36	50
6.	F	0	22	12	48	36	50

Tablo 3.12’de “E” kolonunda 3. satıra bakılıp 4. Satır ile karşılaştırıldığında, “36” ve “38” değerleri görülmektedir. Bu iki değer birbirinden farklı olduğu için bir üst satıra geçilmez ve o satırdaki “Seçili” değere gidilir. 3. Satırdaki seçili değer “B” olduğundan en kısa yolun sondan 3. Noktası “B” olarak belirlenir ve noktalar dizisinin son üç noktası “B” > “E” > “F” olarak belirlenir.

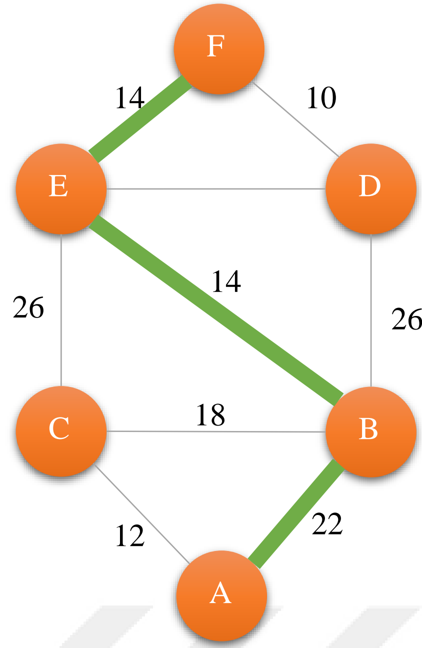
Tablo 3.12. Dördüncü ve Üçüncü Satırların Karşılaştırılması

2.	C	0	22	12	∞	∞	∞
3.	B	0	22	12	∞	38	∞
4.	E	0	22	12	48	36	∞
5.	D	0	22	12	48	36	50

Tablo 3.13’te “B” kolonunun 2. Satırına bakılır. Burada 3. Satır ile 2. Satırın değerleri “22” aynı değer olduğundan bir üst satır olan 1. satıra geçilir. Bir üst satırdaki değer “∞” olduğundan ve “22” değerinden farklı olduğundan 1. Satırdaki “Seçili” değere gidilir. Bu satırdaki seçili değer “A” noktası olduğundan ve başka bakılacak bir satır kalmadığından en kısa yolun ilk değeri “A” noktası olarak belirlenmiş olur. Zaten “A” noktası algoritmanın başlangıç değeri olduğundan, Dijkstra algoritmasını kullanarak “A” noktasından “F” noktasına gidilen yol “A” > “B” > “E” > “F” olarak belirlenmiş ve algoritma tamamlanmış olur.

Tablo 3.13. Üçüncü ve İkinci Satırların Karşılaştırılması

	Seçili	**A**	**B**	**C**	**D**	**E**	**F**
1.	A	0	∞	∞	∞	∞	∞
2.	C	0	22	12	∞	∞	∞
3.	B	0	22	12	∞	38	∞



Şekil 3.3. Dijkstra Sonuç Haritası

Son olarak Dijkstra en kısa yol algoritmasından çıkan yol, şekil üzerinde gösterilecek olursa Şekil 3.3'de görüldüğü gibi bir sonuç elde edilecektir. Buradaki yeşil yollar algoritmadan bulunan noktalar olan "A" > "B" > "E" > "F" üzerinden geçmektedir. Bu yollardaki maliyet değerleri toplandığında 22 + 14 + 14 = 50 değeri bulunmuş olacaktır. Dikkat edilecek olunursa bu değer tablonun "F" kolonunun son satırında algoritmanın sonlandığı satırdaki değer ile aynı değerdir. Öyleyse "A" noktasından "F" noktasına en az maliyetle gidilecek yol 50 maliyeti ile "A" > "B" > "E" > "F" noktalarıdır denir ve sonuç bulunmuş olur.

3.8.2. Floyd-Warshall Algoritması

3.8.2.1. Nedir?

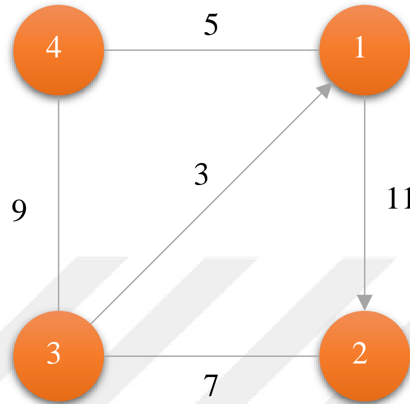
Bir başka algoritma olan Floyd-Warshall algoritması bir graf üzerinde en kısa yolu bulmaya yarayan bir algoritmadır. Algoritma 1962 yılında 1936 doğumlu Amerikalı Robert Floyd tarafından yayınlanmıştır. Ondan birkaç sene önce 1959 yılında Bernard Roy ve 1962 yılında Stephen Warshall tarafından'da yayınlanan algoritma bugün daha çok Floyd-Warshall algoritması olarak bilinmektedir. Algoritma aynı zamanda Floyd algoritması, Roy-Warshall algoritması, Roy-Floyd algoritması ve WFI algoritması olarak ta isimlendirilmektedir [16].

Bu algoritmanın Dijkstra algoritmasından farkı yolların pozitif veya negatif değer alabiliyor olmasıdır. Fakat algoritmada kullanılacak yolun aynı noktadan aynı noktaya, örneğin "A" noktasından "A" noktasına giden döngü şeklindeki bir yolun negatif değer almaması gerekir. Bunun dışından bir noktadan başka bir noktaya giden bir yol negatif değer alabilir. Algoritma yöntem açısından 1956 yılında yayınlanan Kleene'nin algoritmasıyla da benzerlik göstermektedir. Algoritma özetle bir kaynak noktasından gidilebilen noktaların komşuluk listesini çıkartıp, bu hedef noktalara olan maliyeti tutan bir matris üzerinde sonuçları bulur. Algoritmanın her bir iterasyonunda oluşturulan "Maliyet" ve "Sıra" tabloları üzerinde işlemler yapılarak sonuçlar bu tablolardan belirli koşullardan alınıp yeni oluşturulan sonraki iterasyon tablolarına aktarılır.

Algoritma çoğunlukla bilgisayar bilimlerinde çok kullanılan karakterler dizileri üzerinde belirli isterleri bulan Regular Expression'da, dinamik programlamada, internet ağı yönlendirme protokollerinde kullanılır. Algoritmanın zaman karmaşıklığı $O(V^3)$ 'tür [17].

3.8.2.2. Çalışma Mantığı

Floyd-Warshall algoritmasının çalışma mantığı Şekil 3.4'de gösterilen graf üzerinden anlatılacaktır. Bu grafta 4 adet nokta ve yönlü yollar bulunmaktadır. Bu algoritma tüm noktalardan tüm noktalara en kısa yolları bir seferde hesaplayacaktır.



Şekil 3.4. Floyd-Warshall Harita Örneği

Algoritma üzerinde kullanılan değerler Şekil 3.4 üzerindeki gibidir. Bu değerleri "Floyd-Warshall" algoritması ile çözmek için "Maliyet Tablosu" ve "Sıra Tablosu" olarak adlandırılan 2 tabloya ihtiyaç vardır. Bu tablolardan birincisi, harita üzerindeki her bir noktanın diğer noktaya gidişinde gerekli olan toplam maliyeti gösterecektir. İkinci tablo ise algoritma adım adım ilerledikten sonra algoritmanın sonunda en kısa yol bu tablodan çıkartılacak bir noktalar sırasını içeren tablodur.

"Maliyet" tablosu oluşturulurken haritadaki toplam nokta sayısı kadar satır ve sütun eklenmelidir. Tablo 3.14 oluşturulduktan sonra bir noktanın kendine olan uzaklığı olmadığı için satır ve sütunlarda aynı noktaya karşılık gelen yerler "-" işareti ile doldurulur. "Sıra" tablosunda ise yine aynı şekilde tüm noktaların yine kendi ile eşleştiği satır ve sütunlara karşılık gelen yerler "-" ile doldurulur. Bu kısımda yapılacak ilk işlem, "Sıra" tablosunda tablonun sütunlarının yukarıdan aşağıya sütunların karşılık geldiği nokta isminin, tüm sütunlar için boş yerlerinin aynı sütun değeri ile doldurulmasıdır. Yani "1" sütunun tüm boş satırları yine "1", "2" sütununun boş satırları yine "2" olarak şekilde tüm sütunlar yukarıdan aşağıya doldurulur.

Tablo 3.14. Floyd-Warshall Maliyet ve Sıra Tablosu

M_0	1	2	3	4	S_0	1	2	3	4
1	-				1	-	2	3	4
2		-			2	1	-	3	4
3			-		3	1	2	-	4
4				-	4	1	2	3	-

"Sıra" tablosunu doldurduktan sonra "Maliyet" tablosundaki boşluklar tek tek doldurulur. Bunun için herhangi bir sıra gerekli değildir. Tabloda görülen herhangi bir boş hücreden başlanılabilir. Örneğin satır 1 ve sütun 2'yi H_{12} olarak belirtecek olursak, H_{12}, birinci noktadan ikinci noktaya gidişi temsil etmektedir. H_{12} ye Şekil 3.4'ten bakıldığında 1. noktadan 2. noktaya gidiş maliyeti "11" olarak görülmektedir. Bundan dolayı 1. Satırın 2. Sütununa "11" yazılmalıdır. Hemen arkasından H_{21} e bakıldığında yönlü bir graf olduğundan 2 noktasından 1 noktasına direk gidiş olmadığı için H_{21} e yani 2. Satırın 1. Sütununa "∞" değeri yazılmalıdır.

H_{13} e bakıldığında 1 noktasından 3 noktasına direk gidiş olmadığı için bu hücreye "∞" değeri yazılmalıdır. H_{31} e bakıldığında 1 noktasına direk gidiş olduğu ve bu değerin "3" olduğu görülmektedir. Aynı şekilde H_{23} ve H_{32} çift yönlü ve "7" değerinde görülmektedir. H_{41} ve H_{14} maliyetleri "5" olarak tabloya yazılır. H_{42} ve H_{24} "∞" olarak yazılır ve son olarak H_{43} ve H_{34} "9" maliyeti ile tabloya yazılır ve 0. iterasyon "Maliyet" tablosu tamamlanmış olur. Tablonun son hali Tablo 3.15'te gösterilmektedir.

Tablo 3.15. Maliyet ve Sıra Tablosunun Doldurulmuş Hali

M_0	1	2	3	4	S_0	1	2	3	4
1	-	11	∞	5	1	-	2	3	4
2	∞	-	7	∞	2	1	-	3	4
3	3	7	-	9	3	1	2	-	4
4	5	∞	9	-	4	1	2	3	-

Tablo 3.16'daki gibi 1. iterasyon tabloları oluşturulurken 0. iterasyonun 1. Satır ve 1. Sütun değerleri 1. iterasyon tablosunun 1. Satır ve 1. Sütun hücrelerine Tablo 3.16'da sarı renkle gösterildiği gibi aynen aktarılır. Aktarma işlemi tamamlandığında 2. iterasyona başlanır. 2. iterasyonda 1. iterasyondan farklı olarak "Sıra" tablosundaki değerler sütundaki değerler ile doldurulmaz. "Sıra" tablosunu doldurabilmek için "Maliyet" tablosunda bazı işlemler yapıp bazı koşulları sağlayıp ondan sonra "Sıra" tablosuna değerler girilir.

Tablo 3.16. Birinci İterasyon Tablosu

M_1	1	2	3	4	S_1	1	2	3	4
1	-	11	∞	5	1	-	2	3	4
2	∞	-			2	1	-		
3	3		-		3	1		-	
4	5			-	4	1			-

"Maliyet" tablosu doldurulurken artık önceki iterasyon tablosundaki değerler işin içine girmektedir. Algoritma sırasında kullanılacak koşul aşağıdaki gibidir:

$H_{ij} > H_{ik} + H_{kj}$ (H_{ij}, H_{ik} ve H_{kj} hücreleri hesaplanırken önceki tabloya bakılmalıdır)

Buradaki "H" hücredir, i satır numarasıdır, j sütun numarasıdır, k ise iterasyon numarasıdır.

Eğer yukarıdaki koşul sağlanıyorsa M_k tablosundaki H_{ij} hücresi bir önceki iterasyondaki $M_{(k-1)}$ tablosundaki $H_{ik} + H_{kj}$ değeri ile doldurulur. Eğer koşul sağlanmıyorsa bir önceki iterasyondaki $M_{(k-1)}$ tablosundaki H_{ij} değeri M_k tablosundaki H_{ij} hücresine aynen yazılır. Yani H_{ij} tablosuna her zaman bu iki maliyet karşılaştırmasından en küçük olan maliyet yazılır.

H_{23} doldurmak istenirse $H_{ij} > H_{ik} + H_{kj}$ koşuluna göre değerler doldurulduğunda k = 1, i = 2, j = 3 olduğunda $H_{23} > H_{21} + H_{13}$ koşulundan değerler doldurulduğunda $7 > \infty + \infty$ koşulu sağlanmadığı görülmektedir. Koşul sağlanmadığı için H_{23} hücresine "7" değeri Tablo 3.17'deki gibi aynen yazılmalıdır.

Tablo 3.17. H23 Hücresinin Doldurulması

M_1	1	2	3	4	S_1	1	2	3	4
1	-	11	∞	5	1	-	2	3	4
2	∞	-	7	∞	2	1	-	3	4
3	3	7	-	8	3	1	2	-	1
4	5	16	9	-	4	1	1	3	-

"Sıra" tablosunda ise M_k tablosundaki H_{23} hücresi ile $M_{(k-1)}$ tablosundaki H_{23} hücresinin karşılaştırılması gerekmektedir. Bu iki hücre karşılaştırıldığında iki hücrenin değerinin de "7" olduğu görülmektedir. Eğer iki hücre aynı ise S_0 tablosundaki değer S_1 tablosundaki hücreye aynen aktarılmalıdır. Eğer iki hücre farklı ise S_1 tablosundaki değer direk olarak iterasyon numarası olarak güncellenir. İki hücrenin de değeri "7" olduğundan S_1 tablosundaki hücrenin değeri S_0 tablosundaki değer olan "3" ile aynı şekilde Tablo 3.17'deki gibi güncellenir.

M1 tablosunu oluştururken bir önceki varyasyon olan M0 tablosu referans alınmıştır. M2 tablosu doldururken ise Tablo 3.18'deki gibi onun bir önceki iterasyonu olan M1 tablosu referans alınmalıdır. Her tablo doldurulurken kendinden önceki tablo referans alınmaz ise sonuçlar hatalı veya eksik çıkacaktır.

Tablo 3.18. İkinci İterasyon Tablosu

M_2	1	2	3	4	S_2	1	2	3	4
1	-	11	18	5	1	-	2	2	4
2	∞	-	7	∞	2	1	-	3	4
3	3	7	-	8	3	1	2	-	1
4	5	16	9	-	4	1	1	3	-

Aynı şekilde 3. ve 4. iterasyonu da Tablo 3.18’deki gibi hesaplanacaktır. Üçüncü iterasyon Tablo 3.19’da, dördüncü iterasyon Tablo 3.20’de gösterilmiştir. Tüm noktalar için iterasyonlar tamamlandığında 0. iterasyon dahil toplam 5 iterasyonda son tablo Tablo 3.20 elde edilmektedir. Bu kısımdan sonra yapılacak işlem son tabloya bakılarak en kısa yolu çıkartmaktır. Örneğin 1 noktasından 2 noktasına giderken ki toplam maliyeti bulmak için “Maliyet” tablosuna bakılması yeterlidir. Bunun için M_4 tablosundaki H_{12} ye bakılırsa toplam maliyetin “11” olduğu görülebilir. S_4 tablosunda H_{12} ye bakıldığında bu değerin “2” olduğu görülmektedir. Ulaşmak istenilen nokta “2” olduğundan ve bu tabloda da bu hücredeki değerin “2” olmasından dolayı, 1 noktasından 2 noktasına direk gidilebilir denilir.

Tablo 3.19. Üçüncü İterasyon Tablosu

M_3	1	2	3	4	S_3	1	2	3	4
1	-	11	18	5	1	-	2	2	4
2	10	-	7	15	2	3	-	3	3
3	3	7	-	8	3	1	2	-	1
4	5	16	9	-	4	1	1	3	-

Tablo 3.20. Dördüncü İterasyon Tablosu

M_4	1	2	3	4	S_4	1	2	3	4
1	-	11	14	5	1	-	2	4	4
2	10	-	7	15	2	3	-	3	3
3	3	7	-	8	3	1	2	-	1
4	5	16	9	-	4	1	1	3	-

Bu bilgiler bir tabloda toplanacak olursa. Tablo 3.21’deki gibi bir sonuç çıkacaktır.

Tablo 3.21. Birinci Noktadan İkinci Noktaya Giden Kısayol ve Maliyeti

Kaynak Nokta	Hedef Nokta	Maliyet	Yol
1	2	11	1 > 2

1 noktasından 3 noktasına gidilmeye çalışıldığında (1 > 3) Şekil 3.4’te görüldüğü gibi bu yolun tek yönlü olduğu görülmektedir. “Maliyet” tablosuna H_{13} e bakıldığında 1 noktasından 3 noktasına gitmenin maliyetinin “14” olduğu görülmektedir. “Sıra” tablosuna H_{13} e bakıldığında hücredeki değerin bir önceki örnek gibi hedef nokta “3” değil başka bir nokta olan “4” olduğu görülmektedir. Bu durumda yapılacak işlem bu yeni noktayı en kısa yolun arasına 1 > 4 > 3 şeklinde eklemek olacaktır. Bu işlemden sonra “Sıra” tablosunda H_{43} hücresine bakıldığında oradaki değerin “3” olduğunu görülmektedir. Bu değer hedef noktası ile aynı değer olduğundan yol işlemi tamamlanmış ve 1 > 4 > 3 şeklinde belirlenmiştir.

Diğer noktalar arasındaki maliyetler ve yollar yine aynı şekilde bulunacaktır. Tablo 3.22’de diğer tüm noktaların en kısa yolları “Floyd-Warshall Algoritması” kullanılarak son iterasyon olan M_4 ve S_4 tablolarına bakılarak yazılmıştır.

Tablo 3.22. Bütün Yolların Kısayol ve Maliyetleri

Kaynak Nokta	Hedef Nokta	Maliyet	Yol
1	2	11	1 > 2
1	3	14	1 > 4 > 3
1	4	5	1 > 4
2	1	10	2 > 3 > 1
2	3	7	2 > 3
2	4	15	2 >3 >1 > 4
3	1	3	3 > 1
3	2	7	3 > 2
3	4	8	3 > 1 > 4
4	1	5	4 > 1
4	2	16	4 > 1 > 2

3.8.3. Bellman-Ford Algoritması

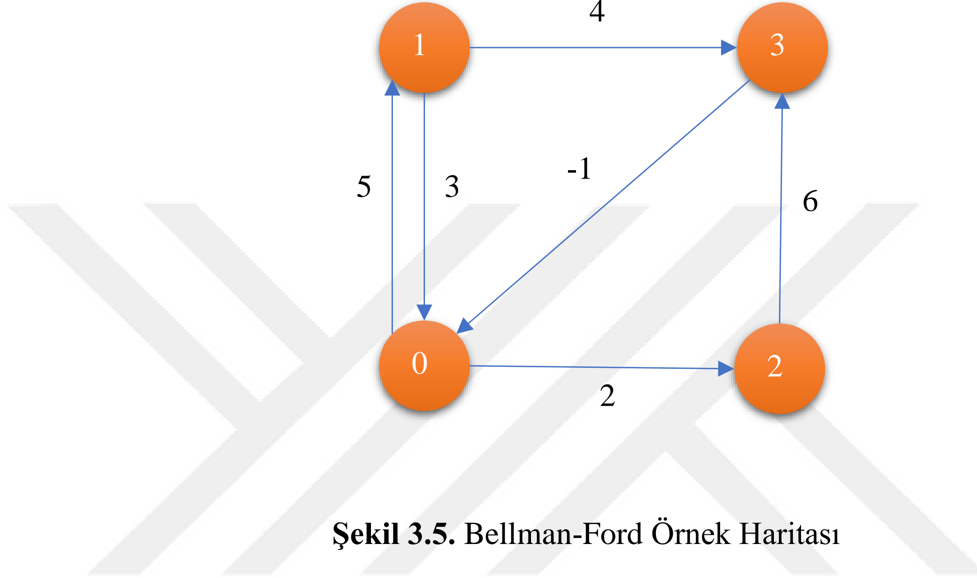
3.8.3.1. Nedir?

Bellman-Ford Algoritması, kaynak bir noktadan diğer tüm hedef noktalara giden kısa yolları bulan bir ağırlıklı graf algoritmasıdır. 1955 yılında ilk defa Alfonso Shimbel tarafından ortaya çıkarılan bu algoritma daha sonra 1956 ve 1958 yıllarında 1920 doğumlu Amerikalı Richard Bellman ve 1886 doğumlu Lester Ford tarafından yayınlanmıştır. 1957 yılında Edward F. Moore tarafından da yayınlanan algoritma bazı kaynaklarda Bellman–Ford–Moore algoritması olarak ta geçmektedir [18]. Bu algoritma Dijkstra algoritmasına göre daha yavaş çalışır. Fakat daha yavaş çalışmasının yanında artıları da vardır, örneğin algoritmaya verilen yollardan bazıları negatif değerler alabilir. Negatif ağırlıklı yollar birçok uygulamada kullanılır ayrıca algoritma graf üzerinde negatif döngülerin var olup olmadığını saptayabilir. Performansı nokta sayısı V yol sayısı E olarak kabul edildiğinde O(VE) şeklinde hesaplanır [19].

Algoritmada bir noktadaki değerin güncellenmesi sırasında o noktaya hangi noktadan gelindiği bilgisi tutulmaktadır. Bu avantajından dolayı eksi yüklü yolların hesaplanması sağlanabilmektedir. Dijkstra algoritmasında negatif değerli yollardan kaynaklanan döngüye girilme ihtimali Bellman-Ford Algoritmasında yoktur. Bellman-Ford Algoritması, Dijkstra algoritmasındaki gibi aç gözlü yaklaşım kullanarak en küçük değerli yoldan gitme mantığını kullanmaz, bunun yerine tüm yolları tek tek dener ve her düğüme sadece 1 kez bakarak en kısa yolu bulur. Algoritma Dijkstra algoritmasına göre negatif kenar konusunda daha verimli olsa da Dijkstra algoritmasına göre daha yavaş çalışır [19].

3.8.3.2. Çalışma Mantığı

Bellman-Ford algoritmasının çalışma mantığı Şekil 3.5'te gösterilen graf üzerinden anlatılacaktır. Bu grafta 4 adet nokta ve yönlü yollar bulunmaktadır. Bu algoritmanın 0 noktasından diğer tüm noktalara giden en kısa yolu bir seferde nasıl hesapladığı anlatılacaktır.



Şekil 3.5. Bellman-Ford Örnek Haritası

Algoritmanın çalışma mantığı anlatılırken aşağıdaki ifadeler kullanılacaktır.

s: Başlangıç noktası

f: Bitiş noktası

s > f: "s" noktasından "f" noktasına direk giden yol

A(s, f): "s" noktasından "f" noktasına gidiş maliyeti

Gerekli tanımlamaları yaptıktan sonra yapılacak ilk iş "Yol", "Ağırlık" tablosunun oluşturulmasıdır. Bu tablo Şekil 3.5'e bakarak oluşturulur. Yol sütununa bir noktadan başka bir noktaya direk gidilebilen yollar yazılır. Ağırlık sütununa ise bu yolda gidilirken verilen maliyet değeri yazılır. Tüm bu veriler tabloya girdikten sonra Tablo 3.23'deki tablo elde edilecektir.

Tablo 3.23. Yol ve Ağırlık Tablosu

Yol	Ağırlık
0 > 1	5
0 > 2	2
1 > 0	3
1 > 3	4
2 > 3	6
3 > 0	-1

Tablo 3.23 oluşturulduktan sonra iterasyon sayısı bulunmalıdır. Bunun için toplam nokta sayısı referans alınır. Şekil 3.5'e bakıldığında toplam nokta sayısının 4 olduğu görülmektedir. Buna göre toplam nokta sayısına "N" denildiğinde "N-1" kadar iterasyon yapılmalıdır denir. Yani toplam 4 nokta olan graftaki toplam iterasyon sayısı 4-1 = 3 olmalıdır.

İterasyon sayısını belirledikten sonra "Maliyet ve Öncelik" tablosu oluşturulmalıdır. "Maliyet" tablosunda her bir iterasyon için tüm noktalar Tablo 3.23 ile belli bir koşul içerisinde karşılaştırılacak ve çıkan sonuçlar bu tabloya yazılacaktır. "Öncelik" tablosuna ise "Maliyet" tablosunda çıkan değerler üzerinden bir çıkarım yapılacak ve buradaki değerin değiştirilip değiştirilmeyeceğine bakılacak. Tablo 3.24 oluşturulduktan sonra kaynak noktası "0" hariç tüm satırlar "∞" ile doldurulmalıdır. Kaynak noktası olan 0 noktası ise "0" değeri ile doldurulmalıdır. "Öncelik" tablosunda ise tüm hücreler "-" ile doldurulmalıdır.

Tablo 3.24. Maliyet ve Öncelik Tablosu

	0	1	2	3		0	1	2	3
Maliyet	0	∞	∞	∞	Öncelik	-	-	-	-

"M" Maliyet tablosu ve "O" Öncelik tablosu olmak üzere maliyet tablosu doldurulurken aranacak koşul şöyle tanımlanır $Mf > Ms + A(s,f)$ koşulu sağlandığında $M_f = Ms + A(s, f)$ olarak değiştirilir. "O_f" hücresi ise "s" değerini alır.

İterasyon 1 için;

Koşul, Tablo 3.23'te bulunan tüm satırlar için sorgulanır ve çıkan değerler "Maliyet" ve "Öncelik" tablosuna aktarılır. 0 > 1 için, s = 0, f = 1 ve Mf > Ms + A(s,f) koşulu bu değerler için Tablo 3.24'e bakıldığında M0 = 0, M1 = ∞, A(0, 1) = 5 olduğundan bu değerler koşula uygulandığında ∞ > 0 + 5 koşulu doğru olduğundan, Mf = Ms + A(s, f) olur denebilir. Buna göre M1 = 0 + 5 = 5 olur. Yine aynı koşul sağlandığından "Öncelik" tablosunda da Of = s olacağından O1 = 0 değeri hesaplanmış olur.

Tablo 3.25'ten görüldüğü gibi iki güncelleme ile yeşille belirtilen yerlere "5" ve "0" değerleri yazılmıştır. Böylelikle 1. iterasyonun 1. Satırı tamamlanmış olacaktır ve 1. iterasyonun 2. Satırına geçilmelidir. s = 0, f = 2, A(0, 2) = 2 olduğundan koşula bakıldığında M2 > M0 + A(0, 2) den ∞ > 0 + 2 koşulu sağlandığından Mf = M2 = 2 ve Of = O2 = s = 0 olacaktır.

Tablo 3.25. Birinci Sütunun Hesaplanması

	0	1	2	3		0	1	2	3
Maliyet	0	5	∞	∞	Öncelik	-	0	-	-

Tablo 3.25'te görüldüğü gibi "Maliyet" ve "Öncelik" tabloları güncellenecektir. 3. Satıra geçildiğinde s = 1, f = 0, A(1, 0) = 3 olduğundan koşula bakıldığında 0 > 5 + 3 koşulu sorgulandığında koşulun sağlanmadığı görülmektedir. Bu durumda tablolarda hiçbir değişiklik yapılmadan direk olarak sonraki adıma yani Tablo 3.26 daki sonraki sütuna geçilmelidir.

Tablo 3.26. İkinci Sütunun Hesaplanması

	0	1	2	3		0	1	2	3
Maliyet	0	5	2	∞	Öncelik	-	0	0	-

1. iterasyonun 4. Satırında Tablo 3.27 ve 5. Satırında Tablo 3.28 elde edilecektir. 6. Satırda ise 0 > 8 + (-1) koşulu sağlanmadığından tablo güncellenmeyecektir. Tüm satırlar tamamlandığında 1. iterasyon bitmiştir ve 2. iterasyona başlanmalıdır.

Tablo 3.27. Birinci İterasyonun 4. Satırının Hesap Sonucu

	0	1	2	3		0	1	2	3
Maliyet	0	5	2	9	Öncelik	-	0	0	1

Tablo 3.28. Birinci İterasyonun 5. Satırının Hesap Sonucu

	0	1	2	3		0	1	2	3
Maliyet	0	5	2	8	Öncelik	-	0	0	2

İkinci iterasyona başlarken Tablo 3.27'deki satırlar sırası ile kullanılarak 1. İterasyondaki gibi "Maliyet" ve "Öncelik" tabloları güncellenmelidir. 2. ve 3. iterasyonları tamamlandığında değerler Tablo 3.29'daki gibi olacaktır. Algoritmanın başında belirtilen N-1 iterasyonuna 3. İterasyonda varıldığından bu kısımda iterasyon tamamlanmış olur.

Tablo 3.29. Son Değerler Tablosu

	0	1	2	3		0	1	2	3
Maliyet	0	5	2	8	Öncelik	-	0	0	2

Algoritma iterasyonları dolaşması tamamlandığında 1 tane daha iterasyon yaparak bu hesaplamalar sonucu maliyetlerde negatif değer olup olmadığı kontrol edilmelidir. Eğer bu hesaplardan sonra maliyetlerde negatif bir değer çıkar ise grafta negatif döngü vardır denilir ve en kısa yol hesaplanamaz. Negatif döngüyü hesaplamak için son bir kez iterasyon tekrarlanır.

s = 0, f = 1, A(0, 1) = 5 için 5 > 0 + 5 koşulu yanlıştır ve tablo değişiklik yapılmaz. s = 0, f = 2, A(0, 2) = 2 için 2 > 0 + 2 koşulu yanlıştır. S = 1, f = 0, A(1, 0) = 3 için 0 > 5 + 3 koşulu yanlıştır. S = 1, f = 3, A(1, 3) = 4 için 8 > 5 + 4 koşulu

yanlıştır. S = 2, f = 3, A(2, 3) = 6 için 8 > 2 + 6 koşulu yanlıştır. Son olarak s = 3, f = 0, A(3, 0) = -1 için 0 > 8 + (-1) koşulu yanlıştır ve bu koşullara göre tabloda hiçbir güncelleme yapılmamıştır. Son iterasyon tamamlanmıştır ve tabloda değişen hiçbir değer yoktur. Bu kısımdan sonra en kısa yolların üzerinden geçtiği noktalar hesaplanmalıdır, bunun için Tablo 3.29 kullanılmalıdır.

0'dan 1'e giderken ki maliyet M1'den 5, "Öncelik" tablosuna bakıldığında O1 = 0 olarak göründüğü için 0'dan 1'e doğrudan gidilebilir ve maliyeti 5'tir sonucu çıkacaktır ve en kısa yol 0 > 1 olarak bulunur. 0'dan 2'ye gidişin maliyeti M2'den 2 ve O2'ye bakıldığında 0 değeri görülmektedir. Yani 0'dan O2'ye direk gidiş var demektir en kısa yolu 0 > 2 olarak bulunur. 0'dan 3'e gitmenin maliyeti M3'ten 8 olarak bulunur ve O3'ten öncelik noktasının 0 değil 2 olduğu görülmektedir. Bu 0'dan 3'e direk yol yok demektir. O2'ye bakıldığında ise 0 noktası görülmektedir, bundan dolayı 0'den 3'e en kısa yol 0 > 2 > 3 olarak hesaplanacaktır. Bu değerlerin tabloya aktarılmış hali Tablo 3.30'da gösterilmiştir.

Tablo 3.30. En Kısa Yol Tablosu

Kaynak Nokta	**Hedef Nokta**	**En Kısa Yol**
0	1	0 > 1
0	2	0 > 2
0	3	0 > 2 > 3

3.8.4. A* Algoritması

3.8.4.1. Nedir?

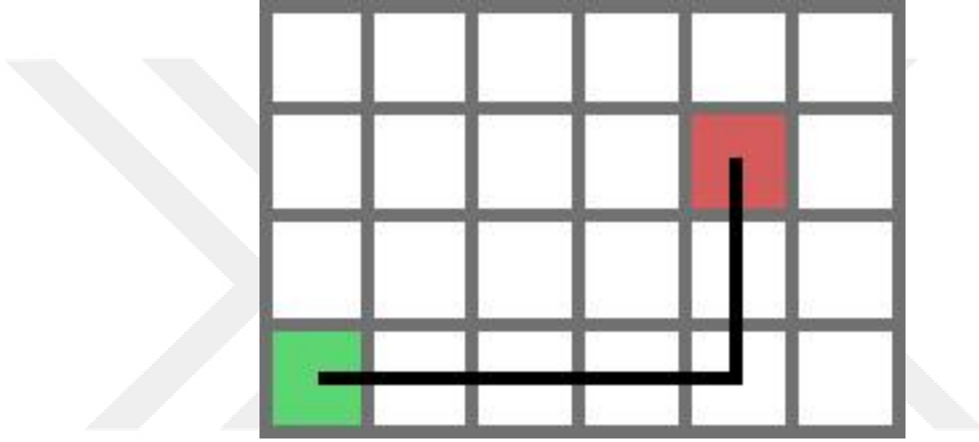
Peter Hart, Nils Nilsson ve Stanford Araştırma Enstitüsü'nden Bertram Raphael tarafından 1968 yılında yayınlanan bu algoritma çok hızlı olması sebebiyle en popüler en kısa yol bulma algoritmalarının başında gelir. Video oyunlarında çok fazla tercih edilen bu algoritma diğer algoritmalara göre problem çözme tekniğinden kaynaklanan çözüm tarzına göre diğer algoritmalardan daha zeki olarak adlandırılır. Algoritma her noktada elinde tuttuğu verileri verimli bir şekilde kullanır. Algoritma çalıştırıldığında her adımda iki farklı parametre olan "G" ve "H" değerlerini toplayarak ortaya bir "F" değeri çıkartır. Her bir adımda bu ortaya çıkan "F" değerlerinden en küçük değerli olanını seçer ve bir sonraki adıma geçer ve algoritmanın zaman karmaşıklığı "d" nokta sayısı ve "b" iterasyon numarası olmak üzere $O(b^d)$ şeklinde tanımlanır [20].

Buradaki "G" değeri o anki seçili nokta ile algoritmanın bir sonraki adım için hedeflediği noktanın arasındaki maliyete karşılık gelir. "H" değeri ise o anki seçili nokta ile bitiş noktasının arasındaki tahmini maliyet değerine karşılık gelir. "H" değeri sezgisel bir değerdir ve farklı şekillerde hesaplanabilir. Bu değer olabilecek en akıllı şekilde tahmin edilmiş en düşük maliyettir. Gerçek en kısa yol bulunana kadar maliyet hiçbir zaman bilinemez çünkü tahmini en kısa yol içinde engel teşkil edecek noktalar bulunabilir.

3.8.4.2. Çalışma Mantığı

A* algoritması hesaplanırken F(n) = G(n) + H(n) fonksiyonu kullanılır. Bu fonksiyonda H(n) değeri Manhattan Uzaklığı veya Çapraz Uzaklık veya Öklid Uzaklığı olmak üzere üç farklı hesaplamadan biri seçilerek hesaplanabilir. Bu uzaklık maliyetleri hesaplanırken haritadaki engeller hesaplamaya dahil edilmeden hesaplanır.

Manhattan Uzaklığı hesaplanırken Şekil 3.6'da görüldüğü gibi beyaz noktalar gidilebilecek noktalar, yeşil nokta başlangıç noktası, kırmızı nokta bitiş noktası kabul edilmiştir. Manhattan Uzaklığı Şekil 3.6'da siyah çizgi ile gösterilmiş uzaklıktır.



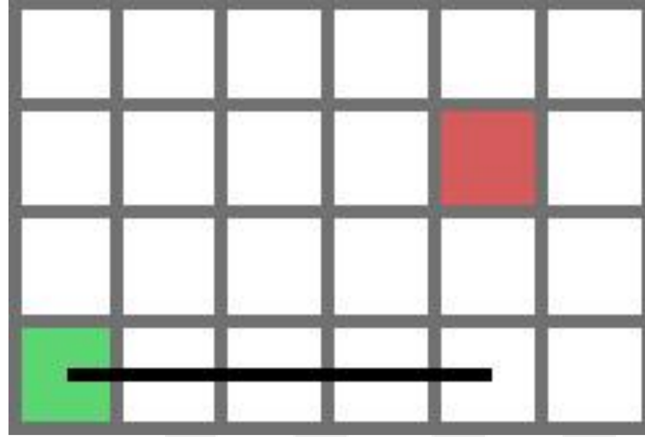
Şekil 3.6. A* Manhattan Uzaklığı

Bu uzaklık hesaplanırken şu matematiksel fonksiyon kullanılır:

H(n) = |mevcut X koordinatı - hedef X koordinatı| + | mevcut Y koordinatı - hedef Y koordinatı|

Mevcut ve hedef koordinatlar hesaplanırken mutlak değer unutulmamalıdır. Buradaki mutlak değer, vektörel değeri kaldırdığından mutlak uzaklık değeri ortaya çıkacaktır. Bu fonksiyon harita üzerinde çapraz gidiş olmayan sadece yukarı, aşağı, sağ ve sol şeklinde gidilebildiğinde daha verimli çalışacaktır.

Çapraz Uzaklık Şekil 3.7'de gösterilmiştir. Beyaz noktalar gidilebilecek noktalar, yeşil nokta başlangıç noktası, kırmızı nokta bitiş noktası kabul edilmiştir. Çapraz Uzaklık Şekil 3.7 'de siyah çizgi ile gösterilmiştir.



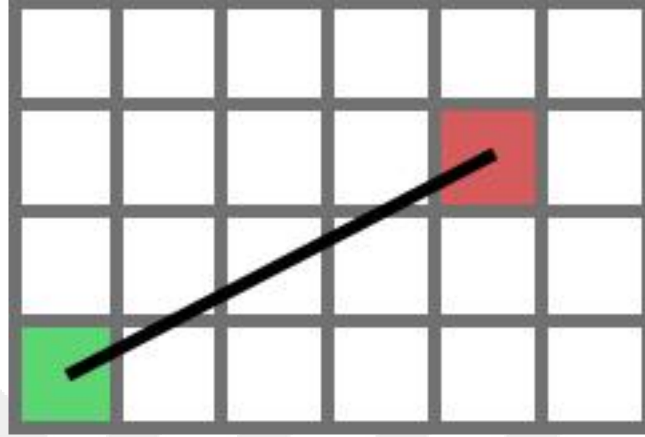
Şekil 3.7. A* Çapraz Uzaklık

Bu uzaklık hesaplanırken şu matematiksel fonksiyon kullanılır:

H(n) = max(|mevcut X koordinatı - hedef X koordinatı|, | mevcut Y koordinatı - hedef Y koordinatı|)

Burada x ve y değerlerinin arasındaki uzaklık hesaplandıktan sonra en büyük değerli olan uzaklık dikkate alınır. Bu fonksiyon yukarı, aşağı, sağ, sol ve dört yöne çapraz gidebilen haritalarda verimli çalışacaktır.

Öklid Uzaklığı hesaplanırken Şekil 3.8'de görüldüğü gibi beyaz noktalar gidilebilecek noktalar, yeşil nokta başlangıç noktası kırmızı nokta bitiş noktası kabul edilmiştir. Öklid Uzaklığı Şekil 3.8'de siyah çizgi ile gösterilen çapraz çizilen uzaklıktır.



Şekil 3.8. A* Öklid Uzaklığı

Bu uzaklık hesaplanırken aşağıdaki matematiksel fonksiyon kullanılır:
H(n) = karekök ((mevcut X koordinatı - hedef X koordinatı)^2 + (mevcut Y koordinatı - hedef Y koordinatı)^2)

Bu öklid fonksiyonu başlangıç ve bitiş notaları arasındaki kuş uçumu uzaklığı vermektedir. Bu fonksiyon ızgara haritalarından ziyade graf haritalarında her doğrultuya gidebilen noktalar olduğunda verimli çalışacaktır.

A* algoritması Tablo 3.31'de gösterilen harita üzerinden anlatılacaktır. Harita bir ızgara haritası olduğu için sezgisel fonksiyonlardan "Çapraz Uzaklık" fonksiyonu kullanılacaktır. Tablo 3.31 üzerindeki siyah hücreler haritadaki engel noktalarını göstermektedir. En kısa yol bulunurken bu noktalar üzerinden hiçbir zaman geçilemez. Tablodaki yeşil nokta başlangıç noktasıdır ve kırmızı nokta gitmek istenilen hedef noktayı temsil etmektedir. Algoritma çalışırken üzerinden geçtiği tüm noktalara "G" değeri "H" değeri, "F" değeri ve o noktaya hangi bir önceki noktadan geldiğini belirten bir hücre numarası verilir. Bu "G", "H", "F" ve geldiği nokta Tablo 3.31'deki her bir hücre içerisinde sırasıyla 1, 2, 3 ve 4. Satırlarda gösterilmektedir.

Bu noktaların "G" değerleri hesaplanırken yukarı, aşağı, sağ veya sola giderken 10, çapraz gidilirken 14 eklenir. Noktalar dolaşılırken, dolaşılan noktalar "Açık Liste" ve "Kapalı Liste" denilen 2 farklı listede tutulur. Noktalar dolaşılırken o noktaya ilk defa işlem yapılacaksa "Açık Liste" ye eklenir. Gerekli işlemler bittiğinde ve sıra o noktanın çevresindeki noktalara bakmaya geldiğinde o nokta açık listeden silinip kapalı listeye eklenir ve noktanın çevresindeki yeni noktalar açık listeye eklenir.

Algoritma yeşil ile gösterilen başlangıç değerinden başlar ve o noktanın 4 tarafındaki 8 noktaya değerler vermeye başlar. Tablo 3.31'deki başlangıç noktası en alt satırdan başladığı için etrafında toplam 8 değil 6 hücre bulunmaktadır. D2 noktasının çevresindeki noktalar C1, C2, C3, D1, D3'tür. Bu noktalar için tek tek "G", "H", "F" hesaplanmalıdır. Başlangıç noktası için "G" sıfır kabul edilir. "H" ve buna bağlı "F" değeri ise olmadığından "-" yazılmıştır.

Tablo 3.31. A* Harita Örneği

	1	2	3	4	5	6
A						
B						
C						
D		G:0 H:- F:-				

“G” değeri hesaplanırken, o noktaya hangi noktadan gidiliyorsa o noktanın “G” değeri ile, hedef noktaya çapraz gidiliyor ise 14; yukarı, sağ, sol veya aşağı gidiliyor ise 10 sayısı toplanır ve yeni noktanın “G” değeri hesaplanmış olur.

Buna göre C1 noktası için “G” değeri hesaplanırken, geldiği noktanın “G” değeri olan 0 ve bu noktaya çapraz gidildiği için 14’ün toplamından 0 + 14 = 14 olarak hesaplanır. “H” değeri ise sezgisel fonksiyonlardan “Çapraz Uzaklık” fonksiyonu kullanıldığından C2 noktasının bitiş noktası olan A5 noktasına X uzaklığı 4 ve Y uzaklığı 2 olduğundan ve en büyük değer seçilmesi gerektiğinden “H” noktasının değeri 4 olarak hesaplanır. “F” değeri hesaplanırken ise F(n) = G(n) + H(n) denkleminden F(n) = 14 + 4 = 18 olarak hesaplanacaktır. Hücrenin en alt satırına ise hangi noktadan geldiği yazılması gerektiğinden ve gelinen nokta “D2” olduğundan “D2” yazılır ve C1 noktasının tüm hesaplamaları tamamlanmış olur.

C2 noktası için “G” değeri, D2’den gelen 0 ile D2’den C2’ye yukarı doğru gidildiği için 10 değerinin toplamıdır ve G(n) = 0 + 10 = 10 olarak bulunacaktır. “H” değeri ise bitiş noktasına olan X uzaklığı ile Y uzaklığının hesaplanmasından sonra bu iki değer arasındaki en büyük değer olacağından X uzaklığı 3 ve Y uzaklığı 2 olduğundan “H” değeri 3 olarak hesaplanacaktır. “F” değeri ise G + H = 13 + 3 = 13 olarak hesaplanır. C2 noktasının geldiği nokta ise D2’dir. Aynı şekilde C3, D1, D3’te hesaplanır, bu hesaplanmış değerler Tablo 3.32’de görülmektedir.

Tablo 3.32. Birinci Noktanın Çevresindeki Hesaplanan Değerler

	1	2	3	4	5	6
A						
B						
C	G:14 H:4 F:18 D2	G:10 H:3 F:13 D2	G:14 H:2 F:16 D2			
D	G:10 H:4 F:14 D2	G:0 H:- F:-	G:10 H:3 F:13 D2			

Açık listede ekli olan tüm noktalar hesaplandığında her bir noktanın “F” değerine bakılır ve en düşük “F” değeri olan nokta bir sonraki hesaplama için kaynak nokta olarak seçilir. Tablo 3.32’ye bakıldığında D1, C1, C2 C3 ve D3 noktalarından en düşük “F” değeri olan nokta 13 değeri ile C2 ve D3 noktalarıdır. Eğer aynı değerli birden fazla nokta ile karşılaşılırsa rastgele olarak herhangi birinden başlanır.

C2 noktası kaynak nokta olarak seçildiğinde bu nokta Açık Liste’den silinir ve Kapalı Liste’ye eklenir. C2 noktasının Kapalı Liste’de bulunmayan komşu hücreleri ise Açık Listeye eklenir. Bu durumda Açık Liste: B1, B3, C1, C3, D1, D3 ve Kapalı Liste: D2, C2 olacaktır. C2 hücresinin çevresindeki daha önceden hesaplanmış hücreler bu kez C2 noktası seçili nokta olacak şekilde “F” değeri hesaplanır. Bu çıkan “F” değeri eski değerden küçükse noktanın “F” değeri bu daha küçük olan değer ile güncellenir. Bu şekilde bitiş noktasına kadar hesaplamalar yapıldığında Tablo 3.33 elde edilecektir.

Tablo 3.33. Diğer Noktaların Hesaplanan Değerleri

	1	2	3	4	5	6
A	G:34 H:4 F:38 B1	G:48 H:3 F:51 B3	G:34 H:2 F:36 B3	G:38 H:1 F:39 B3	A4	
B	G:24 H:4 F:28 C2		G:24 H:2 F:26 C3			G:48 H:1 F:49 C5
C	G:14 H:4 F:18 D2	G:10 H:3 F:13 D2	G:14 H:2 F:16 D2	G:24 H:2 F:26 C2	G:34 H:2 F:36 D4	G:44 H:2 F:46 C5
D	G:10 H:4 F:14 D2	G:0 H:- F:-	G:10 H:3 F:13 D2	G:20 H:3 F:23 D3	G:30 H:3 F:33 D4	G:40 H:3 F:43 D5

Algoritma çalışırken hesaplama sırası A4 noktasına geldiğinde ve bu nokta çevresindeki noktalara baktığında varılmak istenen nokta olan A5 noktasıyla karşılaşılmıştır. Bu durumda algoritma döngülerden çıkar ve çalışmayı durdurur. Noktaların üzerine her hesaplamada yazılan hangi noktadan geldiği bilgisi kullanılarak bitiş noktasından başlangıç noktasına ulaşılır. Bu noktalar dizisi bitiş noktasından başlangıç noktasına gittiği için daha sonra dizi ters çevrilerek başlangıç noktasından bitiş noktasına giden gerçek yol bulunur.

A5 noktasına A4 noktasından gelindiği için en kısa yol ters şekilde A5 > A4 şeklinde olacaktır. A4 noktasına bakıldığında geldiği yolun B3 olduğu görülmektedir B3 noktası da diziye eklendiğinde dizi A5 > A4 > B3 şeklinde olacaktır. B3 noktasına bakıldığında C3, C3 noktasına bakıldığında ise başlangıç noktası olan D2 olduğu görülmektedir. Böylelikle başlangıç noktasına ulaşılmış olur ve dizinin son hali A5 > A4 > B3 > C3 > D2 şeklinde olacaktır. Gerçek yolu görmek için bu dizi ters çevrilir ve ters çevrildiğinde D2 > C3 > B3 > A4 sonucu bulunur. Tablo 3.33'te, bulunan en kısa yol mavi renk ile gösterilmiştir.

3.9. En Kısa Yol Bulma Algoritmalarının Analizi

Algoritmalar özellikle kaynakların az ve kısıtlı olduğu yerlerde kaynağın kapasitesi ve hızı dikkate alınarak etkin bir şekilde geliştirilmelidir. Yapılacak işin önemine göre, örneğin otonom giden bir arabanın gideceği yolu en güvenli olan yolu seçerek gitmesi gibi, algoritmanın şekillendirilmesi gerekir.

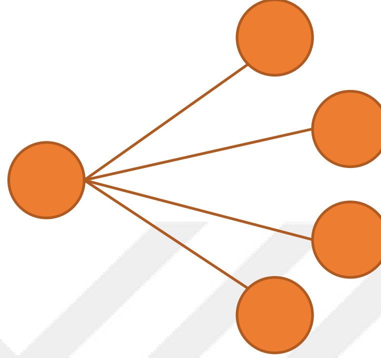
Algoritma analiz edilirken yazılan programın kaç adımda bittiği (yürütme süresi) ve buna bağlı olarak ne kadar sürede bittiği hesaplanır. Algoritmanın analiz edilmesinin nedeni algoritmanın tutarlı çalışıp çalışmadığını kontrol etmek, diğer algoritmalarla performansını karşılaştırmak ve en önemlisi yazılan algoritmanın daha iyisinin yazılıp yazılamayacağını test etmektir.

Zaman karmaşıklığı ve bellek karmaşıklığı olarak adlandırılan incelenmesi gereken 2 önemli özellik vardır. Zaman karmaşıklığı verinin algoritmaya verilmesinden sonra programdaki dallanma, toplama, çıkarma, çarpma, bölme gibi işlemlerin sayısı üzerinden ortaya çıkar. Bellek karmaşıklığı ise platforma bağlı olarak bellekte kapladığı toplam yer üzerinden hesaplanır [21].

3.10. Algoritmaların Çalışma Mantığı ve Kaba Kodları

3.10.1. Dijkstra Algoritması

Dijkstra algoritması Şekil 3.9'da gösterildiği gibi bir noktadan birçok noktaya olan uzaklıkları tek seferde bulabilen bir algoritmadır.



Şekil 3.9. Bir Noktadan Birçok Noktaya Gidiş Grafı

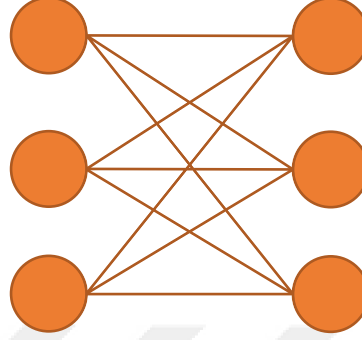
Algoritmanın kaba kodu Şekil 3.10'da gösterilmiştir. Bu kaba koda bağlı kalınarak mobil Android uygulaması geliştirilecektir.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:          // Graftaki bütün noktalar için döngü
        dist[v] ← INFINITY               // Kaynak noktadan hedef noktaya uzaklık bilinmediği için sonsuz yaz
        prev[v] ← UNDEFINED              // Bir önceki noktanın bulunmuş en iyi kısa yoluna sonsuz yaz
        add v to Q                       // Bütün noktaları ziyaret edilmemiş nokta yap

    dist[source] ← 0                     // Kaynak noktadan kaynak noktaya uzaklık 0

    while Q is not empty:                // Ziyaret edilmeyen noktalar bitene kadar döngüyü çalıştır
        u ← vertex in Q with min dist[u] // En düşük uzaklıktaki noktayı bul ve seç

        remove u from Q                  // Bu noktayı ziyaret edilmemiş noktalar listesinden sil

        for each neighbor v of u:        // u'nun bütün komşu noktaları için döngü
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:            // v noktasına en kısa yol mu?
                dist[v] ← alt            //  uzaklık değerini güncelle
                prev[v] ← u              // kendinden önceki noktayı belirle

    return dist[], prev[]                //noktaları ve önceki nokta listesini geri döndür
```

Şekil 3.10. Dijkstra Algoritması Kaba Kodu

3.10.2. Floyd-Warshall

Floyd-Warshall algoritması Şekil 3.11'de gösterildiği gibi birçok noktadan birçok noktaya olan uzaklıkları tek seferde bulabilen bir algoritmadır.



Şekil 3.11. Birçok Noktadan Birçok Noktaya Gidiş Grafı

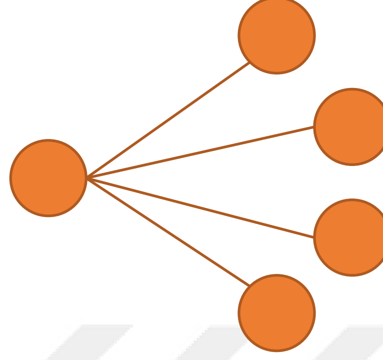
Algoritmanın kaba kodu Şekil 3.12'de gösterilmiştir. Bu kaba koda bağlı kalınarak mobil Android uygulaması geliştirilecektir.

```
for i = 1 to N // Tüm noktalar için döngü başlat
   for j = 1 to N // Tüm noktaları karşılaştırmak için ikinci döngüyü başlat
      if there is an edge from i to j // i noktasından j noktasına bir yol var ise
         dist[0][i][j] = the length of the edge from i to j // bu yolu diziye ekle
      else // yoksa
         dist[0][i][j] = INFINITY // İki nokta arasındaki yol uzunluğunu sonsuz yap

for k = 1 to N  // Tüm noktalar için döngü başlat
   for i = 1 to N // Tüm noktaları karşılaştırmak için ikinci döngüyü başlat
      for j = 1 to N // Tüm noktaları karşılaştırmak için üçüncü döngüyü başlat
         dist[k][i][j] = min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j])
         // Noktalar arasındaki uzaklık ise önceki hesaplanan uzaklığı karşılaştır
         // Eğer bulunan yeni uzaklık daha az ise yol daha kısa olacağından onu seç

return dist // Uzaklık dizisini geri döndür
```

Şekil 3.12. Floyd-Warshall Algoritması Kaba Kodu

3.10.3. Bellman-Ford

Bellman-Ford algoritması Şekil 3.13'de gösterildiği gibi bir noktadan birçok noktaya olan uzaklıkları negatif değerli olsa bile tek seferde bulabilen bir algoritmadır.



Şekil 3.13. Bir Noktadan Birçok Noktaya Gidiş Grafı

Algoritmanın kaba kodu Şekil 3.14'te gösterilmiştir. Bu kaba koda bağlı kalınarak mobil Android uygulaması geliştirilecektir.

```
function bellmanFord(G, S) // G graf, S kaynak nokta
    for each vertex V in G // Graftaki tüm noktalar için döngü başlat
            distance[V] <- infinite // Uzaklık değerlerini başlangıçta sonsuz yap
            previous[V] <- NULL // O noktaya gelinen önceki noktayı tanımsız yap
    distance[S] <- 0 // Kaynak noktasının tablo değerini 0 olarak belirle
    for each vertex V in G // Graftaki tüm noktalar için döngü başlat
        for each edge (U,V) in G // U noktasından V noktasına giden tüm yollar için döngü
            tempDistance <- distance[U] + edge_weight(U, V) // Geçici uzaklığı belirle
            if tempDistance < distance[V] // Eğer yeni hesaplanan uzaklık eskisinden kısa ise
               distance[V] <- tempDistance // Yeni uzaklık değerini kısa olan olarak belirle
               previous[V] <- U // Bu yeni kısa yolun hangi noktadan geldiğini belirle

    for each edge (U,V) in G // U noktasından V noktasına giden tüm yollar için döngü
        If distance[U] + edge_weight(U, V) < distance[V} // Eğer ağırlık toplamı uzaklıktan küçük ise
            Error: Negative Cycle Exists // Negatif döngü vardır demektir ve program sonlanır

    return distance[], previous[]  //noktaları ve önceki nokta listesini geri döndür
```

Şekil 3.14. Bellman-Ford Algoritması Kaba Kodu

3.10.4. A* Algoritması

A* algoritması Şekil 3.15’te gösterildiği gibi bir noktadan bir noktaya olan uzaklığı tek seferde bulabilen bir algoritmadır.

Şekil 3.15. Bir Noktadan Bir Noktaya Gidiş Grafı

Algoritmanın kaba kodu Şekil 3.16’da gösterilmiştir. Bu kaba koda bağlı kalınarak mobil Android uygulaması geliştirilecektir.

```
function A*(start, goal)
    closedSet := {} // Kapalı listeyi tanımla
    openSet := {start} // Açık listeyi tanımla
    cameFrom := an empty map // Noktaya nereden gelindiği dizisini tanımla
    gScore := map with default value of Infinity // G değerlerini tutan diziyi tanımla
    gScore[start] := 0 // Kaynak noktanın G değerini 0 yap
    fScore := map with default value of Infinity // F değerlerini tutan diziyi tanımla
    while openSet is not empty // Açık listede nokta olduğu sürece döngü yap
        current := // F değeri en az olan mevcut nokta
        if current = goal // Bitişe gelindiyse döngüyü bitir
            return reconstruct_path(cameFrom, current) // Dizileri geri gönder
        openSet.Remove(current) // Açık listeden noktayı sil
        closedSet.Add(current) // Noktayı kapalı listeye ekle
        for each neighbor of current // Noktanın tüm komşuları için döngü
            if neighbor in closedSet continue //Nokta kapalı listede pas geç
            if neighbor not in openSet openSet.Add(neighbor)
            tentative_gScore := gScore[current] + dist_between(current, neighbor)// G'yi hesapla
            if tentative_gScore >= gScore[neighbor] continue // Değer daha küçük değilse pas geç
            cameFrom[neighbor] := current //noktayı gelinen nokta olarak belirle
            gScore[neighbor] := tentative_gScore // G ve F değerini kaydet
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal)
    return failure
function reconstruct_path(cameFrom, current) // En kısayolu diziden bul
    total_path := [current] // Diziyi tanımla
    while current in cameFrom.Keys: // Geldiği nokta olduğu sürece döngü yap
        current := cameFrom[current] // Mevcut noktayı güncelle
        total_path.append(current) //Noktayı diziye kaydet
    return total_path // Bulunan En kısa yol noktalar dizisini geri döndür
```

Şekil 3.16. A* Algoritması Kaba Kodu

4. KULLANILAN TEKNOLOJİLER

4.1. Java

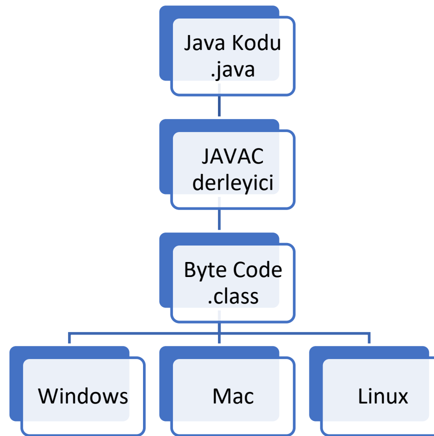
Günümüzdeki en önemli programlama dillerinden birisi Java'dır. Java, Sun Microsystems mühendislerinden James Gosling tarafından geliştirilmeye başlanmış açık kodlu, nesneye yönelik, zeminden bağımsız, çok işlevli, yüksek seviye bir programlama dilidir. Java, GPL lisansı ile açık kodlu hale gelmiştir.
Java uygulamaları bilgisayar mimarisine bağlı olmadan herhangi bir JVM (Java Virtual Machine)'de çalışabilmektedir [22].

4.1.1. Bileşenleri

- **JRE (Java Runtime Environment)** Java programlarını çalıştırmak için gereken ortamdır. Java Platformunun çekirdek sınıflarını ve destekleyici dosyaların paketlerini içerir. Java kodu geliştirmek için gerekli olan derleyici ve debugger gibi paketleri içermez. JRE halihazırda JDK(Java Development Kit)'nın içinde mevcut olarak gelir. Fakat JDK'nin büyüklüğü nedeniyle JRE sadece programın çalıştırılması amacıyla kullanılır. Java sınıflarını herhangi başka bir makinede çalıştırmak için bu pakete ihtiyaç vardır [23].

- **JDK (Java Development Kit)** Java'nın en kapsamlı paketidir. Java tabanlı uygulamaları geliştirmek için kullanılan bir yazılım paketidir. Bu kütüphanelerin genel ismi Yazılım Geliştirme Kiti (SDK - Software Development Kit) adını alır. Bu kütüphane ise Java yazılımını geliştirme kiti olduğu için JDK ismini almıştır. Program yazıldığında bu kodların makine tarafından anlaşılıp derlenebilmesi için yazılan kodları makinanın anlayabileceği kodlara çeviren kütüphanedir [23].

- **Java ME (Micro Edition)** Mobil aygıtlarda çalışan uygulamalar için sağlam, esnek bir ortam sağlar, cep telefonları, uydu alıcıları, Blu-ray Disk oynatıcılar, dijital medya aygıtları, M2M modülleri, yazıcılar vb. aygıtlarda çoğunlukla kullanılır [23].
- **JAVA SE (Standard Edition)** Ağ bağlantılarında, güvenlik ile ilgili firewall makinaları ve sunucularında, veritabanlarında, kullanıcı arayüzleri (GUI - graphical user interface) geliştirmede ve XML parsing kısımlarında çoğunlukla kullanılır [23].
- **JAVA EE (Enterprise Edition)** JAVA SE'ye ek olarak Web ile ilgili özel bileşenler içerir. Web server, EJB, web servisler, Servletler, JSP teknolojilerini içinde barındırır. Java EE, asp .NET veya PHP dillerinden çok daha fazla güvenlik kontrolü içerir. Günümüzde şirketlerin kullandığı sunucuların çoğunda esnek yapısı ve modülerliği nedeniyle daha çok bu teknoloji tercih edilir [23].
- **JVM (Java Virtual Machine)** C++ dilinde yazılmıştır. Bir Java programı «javac.exe» ile derlendikten sonra «byte code» ismi verilen bir ara sürüm oluşur. Byte code, ana işlem biriminin (CPU) anlayacağı cinsten komutlar değildir. Java byte code sadece JVM bünyesinde çalışır. JVM, derlenen Java programı için ana işlemci birimi olma görevini üstlenir. Bu özelliginden dolayı Java programlarını değişik platformlar üzerinde çalıştırmak mümkündür. Her platform için bir JVM sürümü Java programlarını koşturmak için yeterli olacaktır. Bu sebepten dolayı Java "write once, run anywhere (bir kere yaz, her yerde koştur)" ünvanına sahiptir [23].



Şekil 4.1. JVM Diagramı

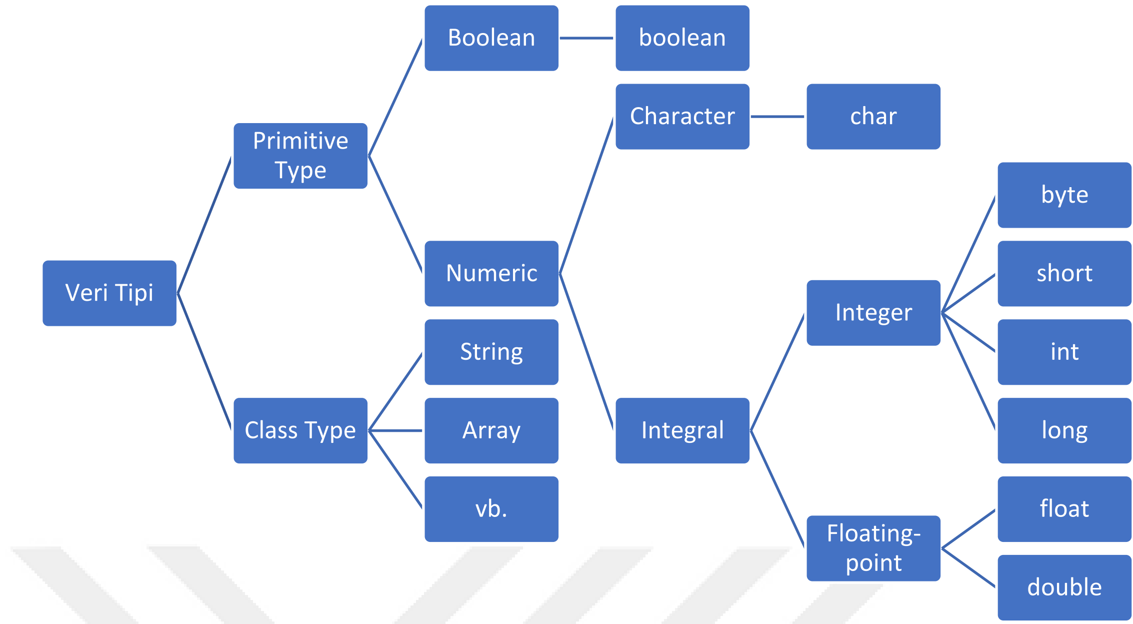
4.1.2. Veri Tipleri

Java'da temel olarak iki tür tip vardır. Primitive type'lar ve class type'lar. Java sayısal, karakter ve durum tiplerini destekler. Kullanılabilen veri tipleri şu şekildedir: ***boolean, char, byte, short, int, long, float, double, String.***

Örneğin int bir primitive type'dır.String ise bir class type'dır. Class type'ları birer OOP(Object Oriented Programming) sınıflarına benzerler. Bu tipler bellekte tuttukları yer kapasitesine bağlı olarak değişkenlikler gösterirler. Algoritmalarda belleği verimli kullanmak ve olabildiğince küçük veri tipleri kullanmak algoritmanın verimliliği ve hızı açısından standart bir yazılıma göre çok daha önemlidir [24].

Veri tiplerinin yazılabilen en yüksek değerleri şu şekildedir:

- **byte** mybyte = 127;
- **short** myshort = 32767;
- **int** myint = 2147483647;
- **long** mylong = 9223372036854775807L;
- **float** myfloat = 3.4028235E38f;
- **double** mydouble = 1.7976931348623157E308;
- **boolean** myboolean = true;//true yada false
- **char** mychar = 'a'; //Sadece 1 harf;
- **String** mystring = "abc";//(En yüksek değeri yoktur)



Şekil 4.2. Java Veri Tipleri

Primitive type değişkenleri "Wrapper class" denen sınıflar şeklinde de tanımlanabilir. Wrapper class'lar, primitive tipindeki değişkenlerin gerekli yerlerde bellekte yer kaplamaması için kullanılan "null" atamasının yapılabilmesini sağlar. Yani primitive tiplere doğrudan "null" ataması yapılırsa "compile error" hatası denilen, daha program derlenmeden ortaya çıkan derleme hatası alınır. Bunun için primitive tipler sadece kendine tahsis edilmiş olan "wrapper class" sına dönüştürülmesi gerekmektedir. Bu dönüştürme işleminden sonra değişke bellekte yer kaplamamasını belirten "null" ataması yapılabilir.

4.1.3. Erişim Belirleyiciler (Access Modifiers)

Java; sınıflar, değişkenler, metotlar ve yapılandırıcılar için kullanılabilen dört farklı erişim belirleyicisine sahiptir [25].

Bunlar erişim seviyesine göre şu şekilde sıralanır:

- **Private:** Sınıflar, metotlar, değişkenler ve yapılandırıcılarda kullanılabilir. Bu belirleyici kullanıldığında sınıf, metot, değişken veya yapılandırıcı sadece tanımlandığı sınıfın içinde erişilebilir olur.
- **Default:** Bu belirleyici, herhangi bir belirleyici tanımlanmadığında varsayılan olarak atanan belirleyicidir. Sınıflar, metotlar, değişkenler ve yapılandırıcılarda kullanılabilir. Bu belirleyici kullanıldığında sınıf, metot, değişken veya yapılandırıcı sadece aynı paket içindeki bir sınıftan erişilebilir olur. Başka bir paketten erişilmeye çalışıldığında derleme hatası olacaktır.
- **Protected:** Kullanıldığı sınıfın alt sınıflarından ve aynı paket içinden erişilebilen bir erişim belirleyicisidir.
- **Public:** Bu belirleyici, aynı paketteki diğer sınıflardan, farklı paketteki diğer sınıflardan ve aynı şekilde tüm objelerden erişilebilen bir belirleyicidir. Erişim kısıtlaması yoktur. Fakat bu belirleyici tutarlılığa karşı dikkatli kullanılmalıdır. Örneğin güvenli olarak tanımladığınız bir kütüphane yazdığınızda üçüncü taraftan bir kişi başka bir paketten gelip yazdığınız değişken veya metotları değiştirebilir.

Bu erişim belirleyicilerinin erişme yetkisi olduğu alanlar Tablo 4.1’de gösterilmiştir.

Tablo 4.1. Java'daki Erişim Belirleyiciler ve Yetkileri

Erişim Belirleyici	private	Default	protected	public
Sınıf içinden	Erişim Var	Erişim Var	Erişim Var	Erişim Var
Aynı paketten	Erişim Yok	Erişim Var	Erişim Var	Erişim Var
Aynı paketteki alt sınıftan	Erişim Yok	Erişim Var	Erişim Var	Erişim Var
Başka paketten	Erişim Yok	Erişim Yok	Erişim Yok	Erişim Var
Başka paketteki alt sınıftan	Erişim Yok	Erişim Yok	Erişim Var	Erişim Var

4.1.4. Değişken Türleri (Variable Types)

Java'da üç çeşit değişken türü bulunmaktadır bunlar, Local değişkenler, Instance değişkenleri ve Class/Static değişkenleridir [26].

- **Local değişkenler**: Lokal değişkenler olarak adlandırılan bu değişken türünde, değişken metot, yapılandırıcı (constructor) veya blok içinde tanımlanır. Bu değişkenler sadece metot veya blok çağrıldığında bellekte oluşturulurlar. Derleme sırası metodun veya bloğun dışına çıktığında derleme işlemi değişkenin artık kullanım alanının dışında olduğundan bellekten silinirler. Bu değişken türünde "Erişim belirleyicisi" kullanılamaz. Lokal değişkenler için ön tanımlı bir değer atama yoktur. Bundan dolayı değişken kullanılmadan önce, değişken oluşturulduktan sonra bir değer ataması yapılıp daha sonra kullanılmalıdır.
- **Instance değişkenler:** Bu değişken türü obje değişkeni olarak adlandırılabilir. Bu değişken türü oluşturulan obje içerisinde oluşturulmuş herhangi bir metot veya sınıf içerisinde kullanılabilir. Değişken tanımı sınıf içerisinde yapıldıktan sonra bu değişkene değer atama işlemi hemen veya daha sonra yapılandırıcı veya herhangi bir metot içinden veya değişkene gerekli erişim belirleyicisi verildiyse objenin dışındaki herhangi bir obje içinden veya sınıftan da değer

ataması yapılabilmektedir. Değişken kendisinin içinde bulunduğu objenin “new” anahtar kelimesi ile oluşturulduğu anda bellekte yer kaplamaya başlar. Kendi objesi hafızadan silindiğinde ise silinir.

- **Class/Static değişkenler:** Sınıf değişkenleri veya statik değişkenler olarak adlandırılırlar. Bu değişken, içine yazıldığı sınıfın objesi ne kadar türetilirse türetilsin bellekte yalnızca bir değişkenlik yer kaplar. Yani obje her oluşturulduğunda ayrı ayrı değişkenler bellekte yer kaplamaz. Bu değişken tüm objeler için aynı değeri taşıyacak olan bir değişken kullanılmak istediğinde kullanılır. Değişken “static” kelimesi kullanılarak oluşturulur. Bu tip değişkenler JVM’de “static memory”(statik bellek) denilen kısımda toplanırlar ve gerektiğinde belleğin bu kısmından çekilirler. Statik değişkenler lokal ve instance değişkenlerinden farklı bir zamanda, program çalıştırıldığı anda bellekte ver kaplamaya başlar ve program sonlandırıldığında bellekten silinirler.

4.1.5. Erişim Dışı Belirleyiciler (Non Access Modifiers)

- **final:** Bu anahtar kelime bir çok biçimde kullanılabilen anahtar kelimedir. Kullanıldığı sınıf’ı, metot’u, metot değişkenini veya sınıf değişkenini bir değer atandıktan sonra başka bir değer ile değiştirilemez yapar. Final olarak belirlenen bir değişkenin ataması sadece bir kez ve “yapılandırıcı” metot’unun içinde yapılabilir. Değişken değeri verildikten sonra hiçbir yerden değişkenin değeri değiştirilemez. Final olarak belirtilen bir metot’un değişkeni güvenlik açısından iyi bir kullanımdır. Bu değişken ilgili metot içerisindeki hiçbir yerde değişikliğe uğratılamaz ve böylelikle hatalardan kaçınılabilir. Final olarak belirtilen bir metot kendisini türeten (extend) sınıfın içinde tekrar ezilemez (overriding) böylelikle korunması istenen metotların tamamı türetilen sınıf içinde korunmuş olur. Final belirleyicisi tanımlanmış bir sınıf ise başka bir sınıf için türetilemez, böylelikle yazılan sınıfların başka kısımlarda türetilip kullanılmasının önüne geçilmiş olur.

4.1.6. Aritmetik Operatörler

Java'da günlük kullanılan matematik işlemlerinin aynı işlemlerini yapan matematiksel aritmetik operatörler mevcuttur. Toplama, çıkarma, çarpma, bölme, mod alma, değişkeni bir artırma ve değişkeni bir azaltma işlemlerini yapan operatörler mevcuttur. Bu operatörler Tablo 4.2'de gösterilmiştir.

Tablo 4.2. Java'daki Operatörler ve Açıklamaları

Operatör	Açıklama	İşlem	Sonuç
+	Toplama işlemi yapar	7+3	10
-	Çıkarma işlemi yapar	7-3	4
*	Çarpma işlemi yapar	7*3	21
/	Bölme işlemi yapar	7/3	2.33
%	Mod alma işlemi yapar	7%3	1
i++	Bir artırma işlemi yapar	7++	8
i--	Bir azaltma işlemi yapar	7--	6

4.1.7. Döngüler

Döngüler algoritmaların vazgeçilmez fonksiyonlarıdır. Her algoritmada muhakkak en az bir algoritma büyük olasılıkla bulunur. Çünkü algoritmaların yapısı gereği tekrar eden, artarak devam eden veya azalarak devam eden işlemler olduğu için döngülerin kullanılması kaçınılmazdır.

Java'da dört çeşit döngü bulunmaktadır. Bunlar kendi içinde parametre alan veya almayan döngülerdir.

- **Break**: Bir döngü denetim sözcüğüdür. Bir döngünün içinde döngüde istenilen işlem yapıldıktan sonra döngüden, döngü işlemi bitmeden çıkabilmek için kullanılan anahtar sözcüktür. Bu sözcüğün kullanıldığı satırdan itibaren program akışı döngüden çıkıp döngü dışındaki satırdan devam eder, döngü işleminin bitmesi beklenmez.

- **Continue**: Bir döngü denetim sözcüğüdür. Bir döngüde belirli bir işlem tamamlandıktan sonra bu sözcük kullanılarak, sözcüğün kullanıldığı satırın altındaki kodlar çalıştırılmadan bir sonraki döngü elemanına atlanıp kod derlemesinin döngünün başından itibaren devam etmesi sağlanır.

- **While Döngüsü**: Bu döngü, parametresine sadece koşul girilebilen bir döngüdür. Parametresine artan veya azalan bir değişken yazılmaz. Program, döngüye geldiğinde döngünün parametresinde yazılan koşul kontrol edilir. Eğer koşul sağlanıyorsa program akışı döngünün içine girer ve döngünün içindeki işlemleri gerçekleştirir. İşlemler bittiğinde döngünün son satırına gelindiğinde program akışı, döngünün başındaki koşulu tekrar kontrol eder ve koşul sağlanıyorsa döngünün içindeki işlemler tekrar gerçekleştirilir. Koşul sağlanmadığı zamana kadar akış bu şekilde devam eder. Koşul sağlanmadığında ise program akışı döngüden çıkar ve bir sonraki satırdan devam eder.

```
int sayi = 20;
while(x < 30) {
        System.out.println("x: " + x);
}
```

- **For Döngüsü:** Bu döngü parametresine döngü her çalıştığında artan veya azalan bir değişken ve bir koşul alır. Program akışı döngüye geldiğinde önce parametresine yazılan değişkene ilk değer ataması yapılır. Daha sonra ikinci parametre olan koşul parametresindeki koşulun sağlanıp sağlanmadığına bakılır. Bu kısımda ikinci parametredeki konuşun mutlaka birinci parametredeki değişkenle bir ilgisinin olması gerekmez. Koşul sağlanıyorsa döngü içine girilir, çalışmıyorsa döngüden çıkılır. Döngüye girildikten sonra ve içerdeki işlemler yapıldıktan sonra döngünün başına gelindiğinde buradaki değişken üçüncü parametredeki değer kadar artırılır veya azaltılır. Bu programcının döngüyü nasıl kullanmak istediğine bağlıdır.

```
for(int x = 1; x < 10; x++) {
        System.out.println("x: " + x);
}
```

- **Do While Döngüsü:** Bu döngü while döngüsünün aynısıdır. Tek farkı "do" bloğunda yazan işlemlerin, "while" parametresindeki koşul kontrol edilmeden en az bir defa çalıştırılmasıdır. Bu döngü genellikle koşul ile bağlantılı durumlarda kullanılır. Örneğin parametrede yazılacak olan koşuldaki değişken ile alakalı bir işlem yapıldıktan sonra koşul kontrol edilir.

```
int x = 1;
do {
    System.out.println("x: " + x);
    x++;
}while( x < 10);
```

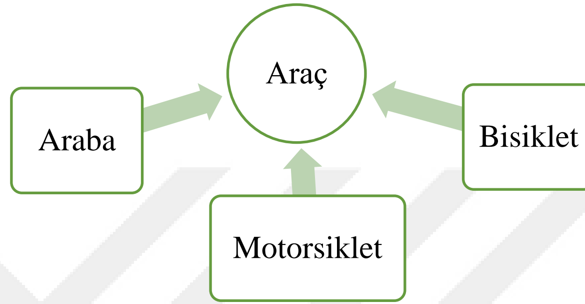
- **Enhanced for Döngüsü:** Bu döngü özel bir döngüdür. Döngünün parametresine "while" veya "for" döngülerindeki gibi artacak veya azalacak olan bir değişken veya koşul yazılmaz. Parametre olarak, içindeki değişkenleri alacak bir dizi yazılır. Bu dizinin türü farklı türlerde olabilir. Tamsayı, kesirli sayı, karakter olabilir veya herhangi bir obje olabilir.

```
int [] numaralar = {100, 101, 102, 103, 104};
for(int n : numaralar) {
    System.out.println(n);
 }
```

4.1.8. Nesne Yönelimli Programlama

Nesne Yönelimli Programlama'nın alt konuları geniş bir yelpaze oluşturur, kullanılan programlama diline ve teknolojiye göre değişiklik göstermektedir. Bu çalışmada sadece kullanılan kavramlar açıklanmıştır [27].

- **Çok Biçimlilik (Polymorfizim): "**Poly", birden fazla anlamına gelmektedir. "Morph" ise şekil veya biçim anlamına gelmektedir. Birden fazla biçimi olan sınıf olarak ifade edilebilir. Örneğin bir "Araç" sınıfı tanımlansın ve bu sınıftan "Araba" ve "Motorsiklet" sınıfları türetilsin. Dizilerde sadece aynı sınıflar bir diziye eklenebildiği için "Araba" ve "Motorsiklet" sınıfları aynı diziye koyulamaz. Fakat polimorfizm'den faydalanarak bu iki sınıf "Araç" sınıf türüne dönüştürülerek Şekil 4.3'te gösterildiği gibi aynı diziye eklenebilir. İşte bu işlem polimorfizm ile oluşabiliyor.



Şekil 4.3. Java Polimorfizm Örneği

- **Encapsulation:** Bazen sınıfta tanımlamış olan değişkenlerin değenlerinin verilmesinde bazı koşullar olması gerektiği durumlar oluşabilir. Bu gibi durumlarda değişkenlere değerler atanırken bazı koşullar sağlandıktan sonra değişkene değer atayan metotlar oluşturulur, bu işleme "Kapsülleme (Encapsulation)" denilmektedir. Örneğin "yıl" adında bir değişken olsun ve bu yıl değişkenine sadece 1900 ve üstü yıllar atanabilsin. Böyle bir durumda bir değişken oluşturup ve bu değişken "public" anahtar kelimesi ile her taraftan değiştirilebilir yapılırsa, kullanıcı bu değişkene dilediği gibi bir yıl yazabilecektir. Hatta negatif sayılar bile ekleyebilir. Bu durumun oluşmaması için "setYil()" isimli bir metot oluşturulur. Bu metodun parametresinden eklenmesi istenen yıl sayısı alınır ve koşul kontrol ile verilen yılın 1900 yılından büyük mü veya küçük mü olduğu kontrol edilir. Eğer yıl 1900'den büyük ise verilen yıl sınıf içindeki değişkene atanır, değil ise hiçbir işlem yapılmaz veya kullanıcıya daha büyük bir yıl girmesi için uyaracak bir mesaj gönderilebilir. Böylelikle sınıfın içindeki değişkenlerin değer atamaları artık metodun kontrolünde olacaktır.

4.2. Android

Google'ın mobil pazara gireceği söylentileri üzerinden çok zaman geçmemiş ve piyasada mobil işletim sistemi arayışlarına giren Google, Android işletim sistemini Temmuz 2015 yılında satın almıştır. Amerika'nın California eyaletinin Palo Alto şehrinde ufak bir şirket olarak mobil pazara giren Google günümüzde yaklaşık 2 milyar telefonda kullanılmaktadır [28]. Şirketin daha önce yaptığı yazılımlar Android kadar bilinmese de şirket Android işletim sistemi ile birlikte büyük oranda tanınmıştır. Android işletim sisteminin logosu Şekil 4.4'te gösterilmiştir.

Şekil 4.4. Android Logosu

Android, günümüzde Google mühendisleri ve Open Handset Alliance (mobil cihazlar için açık kaynaklı standartlar oluşturarak bir araya gelmiş 84 firmanın içinde bulunduğu uluslararası bir oluşum) tarafından geliştirilmektedir [29]. Linux tabanlı bir işletim sistemi olan Android özgür ve ücretsiz bir işletim sistemidir. Açık kaynak kodlu (open source) olan Android'in kodlarını dileyen herkes indirip değiştirebilir veya farklı modlar ekleyerek kendi Android işletim sistemi türevini yapabilir. Android her ne kadar açık kaynak kodlu olsa da kötü niyetli insanlar tarafından kullanıcıların bilgileri vs. çalınmaması için Google işletim sisteminin bazı kısımlarını kapalı hale getirmiştir. Google'ın açık kaynaklı bir yazılım yapmasının sebebi ise açık kaynak yazılımlarının özel yazılımlara göre çok daha hızlı bir şekilde büyümesidir. Bu büyüme hem ulaştığı kullanıcı sayısı hem de kodlama olarak büyümesidir. Öğleki açık kaynaklı yazılımların kodlarının güvenliği ve stabil olması sadece Google tarafından değil, kodu alıp inceleyen herkes tarafından kontrol edilmektedir. Bundan dolayı

Android diğer özel iletim sistemlerine göre daha güvenlidir. Samsung, HTC gibi büyük firmaların da bu ücretsiz işletim sistemini kullanması hem sistemin bilinirliği açısından hem de büyümesi açısından önemlidir. Ayrıca Google Android'in güvenliğini daha da artırmak için işletim sistemindeki açıkları ve hataları bulan yazılımcılara çeşitli para ve ödüller dağıtmaktadır.

Google, Android sistemi üzerine kurmuş olduğu Google Play Marketi üzerinden hem kendi hem de üçüncü şahısların yüklediği oyunlar ve uygulamalar içene eklediği reklamlar sayesinde Android üzerinden para kazanmaktadır. Bu oyun ve uygulamalar ".apk" dosya uzantısıyla yüklenebilen programlardır. Play Markette günümüzde yaklaşık 3.3 milyon uygulama ve oyun bulunmaktadır [30]. Markete geliştirilen uygulama ve oyunlar geniş bir geliştirici ağı ve geliştirici şirketleri tarafından sağlanmaktadır. Android versiyonları ve kullanım oranları Tablo 4.3'te gösterilmiştir.

Tablo 4.3. Android versiyon kullanım oranları (Aralık 2017)

Versiyon	Kodadı	API	Kullanım Oranı
2.3.3-2.3.7	Gingerbread	10	0.4%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.5%
4.1.x	Jelly Bean	16	2.0%
4.2.x		17	3.0%
4.3		18	0.9%
4.4	KitKat	19	13.4%
5.0	Lollipop	21	6.1%
5.1		22	20.2%
6.0	Marshmallow	23	29.7%
7.0	Nougat	24	19.3%
7.1		25	4.0%
8.0	Oreo	26	0.5%

Android, Google'ın java kütüphanesini kullanmaktadır. OHA (Open Handset Alliance), 34 adet yazılım, Telekom ve donanım şirketi tarafından bir tartışma yapmış ve dünya için mobil cihazlarda kullanılabilen açık kaynaklı ve telif hakkı olmayan bir işletim sistemi geliştirilmesinin teknolojinin gelişmesi ve büyümesi için yararlı olacağı konusunda hemfikir olup olumlu yönde karar vermişlerdir [31].

Sistem; kütüphaneler, ara katman yazılımlarıyla birlikte API(Application Programming Interface) C diliyle yazılmış bir işletim sistemidir. Uygulamalar ise "Apache Harmony" teknolojisi üzerine kurulmuş Java kütüphaneleriyle birlikte çalışan bir uygulama iskeleti üzerinde çalıştırılmaktadır. Sistem, yazılan kodun derlenmiş java koduna dönüştürülmüş halini çalıştırmak için Dalvik'in yerini almış olan dinamik çevirmeli Android Runtime (ART) uygulama çalıştırma ortamı kullanır [32].

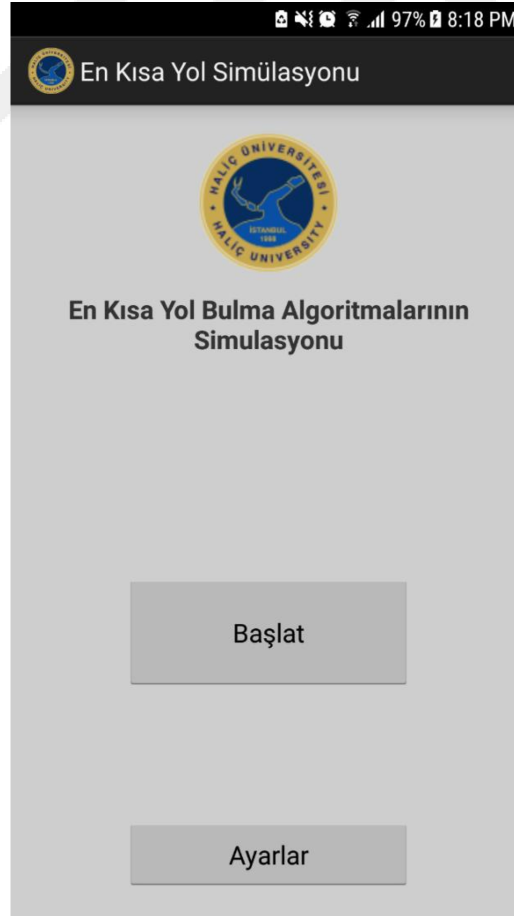
Android işletim sistemi beş ana kısımdan oluşmaktadır [33].

- **Çekirdek:** Linux kerneli üzerine kuruludur. Sürücü modelleri, süreç yönetimleri, ağ yığınları, hazıfa yönetimi ve güvenlik yönetim mekanizmalarını içinde barındırır.
- **Android Runtime:** Android'in sanal makinesidir VM (Virtual Machine) ve Dalvik Sanal Makinesini de içermektedir. Runtime 5.0 ile birlikte Dalvik kaldırılmış ve yenine ART getirilmiştir.
- **Kütüphaneler:** Ara yüz kütüphaneleri, Grafik kütüphaneleri, veri tabanı kütüphaneleri ve web tarayıcı kütüphanelerini içermektedir.
- **Uygulama Çatısı:** Java sınıflarıyla yazılmış yüksek seviyeli servisler sağlayan ve uygulamanın ana iskelet kodlarını oluşturan kısımdır.
- **Uygulama Katmanı:** Bu kısım Android işletim sisteminin en üst kısmıdır. Yazılan uygulama ve oyunlar bu kısıma karşılık gelir. Genellikle uygulama yazarken çekirdek seviyesinde işlemler çok alt seviye kaldığından uygulamaların büyük çoğunluğu bu kısımdadır.

5. MOBİL ANDROİD UYGULAMASININ TASARIMI VE GELİŞTİRİLMESİ

5.1. Giriş Ekranı

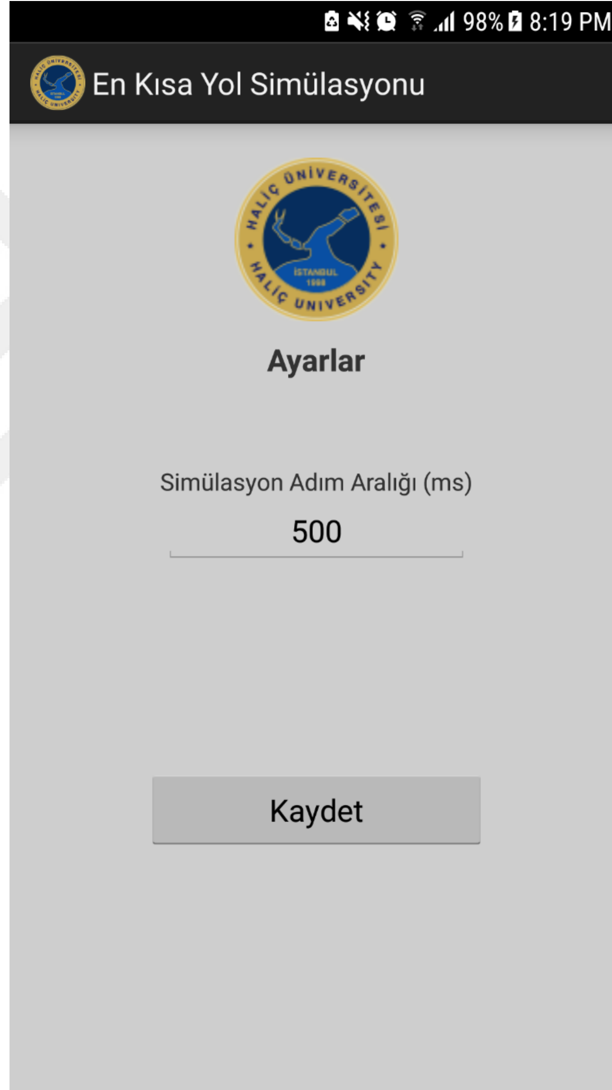
Giriş ekranı uygulama açıldığında ilk açılan ekrandır. Şekil 5.1'de giriş ekranı gösterilmiştir. Bu ekranda "Başlat" ve "Ayarlar" butonları yer almaktadır. "Başlat" butonuna tıklandığında uygulama içinde yüklü olan haritalar ekranı açılmaktadır. "Ayarlar" butonu ise ayarlar sayfasına yönlenmektedir. Giriş ekranı kaynak kodları (EK 1)'de gösterilmiştir.



Şekil 5.1. Giriş Ekranı

5.2. Ayarlar Ekranı

Ayarlar ekranında "Simülasyon Adım Aralığı" denilen milisaniye cinsinden bir değer bulunmaktadır. Bu değer simülasyon ekranında herhangi bir algoritma seçildiğinde ve başlat tuşuna basıldığında simülasyonun adımlarının hangi hızda ilerleyeceğini belirtmektedir. Bu değer milisaniye cinsinden girilmelidir. Ayarlar ekranı Şekil 5.2'de gösterilmiştir. Ayarlar ekranı kaynak kodları (EK 2)'de gösterilmiştir.



Şekil 5.2. Ayarlar Ekranı

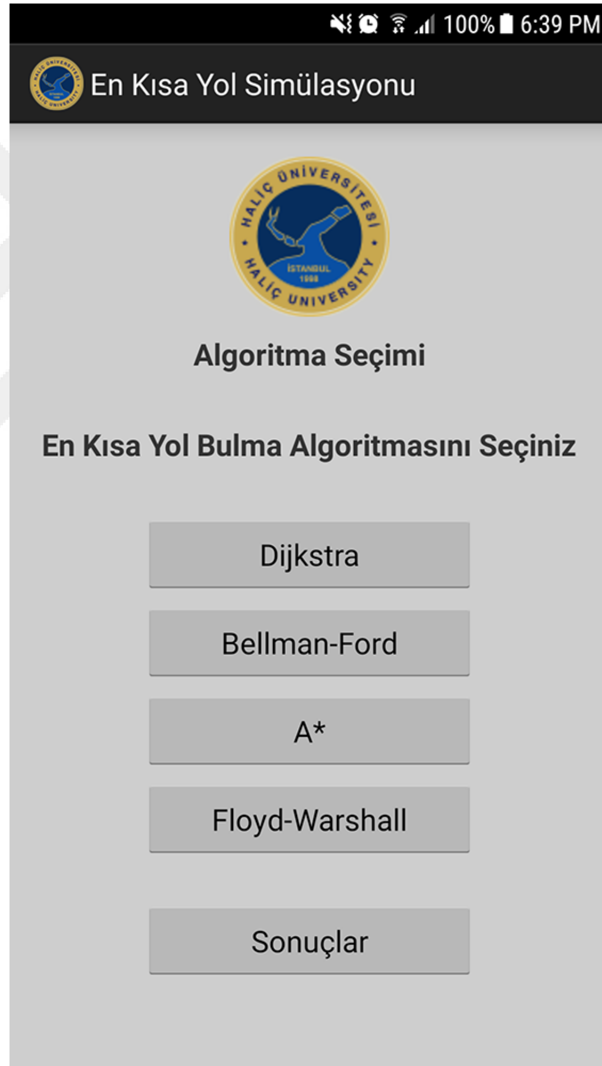
5.3. Harita Seçme Ekranı

Harita seçme ekranında algoritmaların çalışacağı ortak harita seçilebilmektedir. Buradan seçilen önceden hazırlanmış farklı haritalar, içeriğinde başlangıç noktaları, bitiş noktaları ve engeller barındırmaktadır. Uygulama içerisine gömülü 8 harita bulunmaktadır. Harita Seçme ekranı Şekil 5.3'te gösterilmiştir. Harita seçme ekranı kaynak kodları (EK 3)'de gösterilmiştir. Harita üzerindeki "B" harfi "Bir noktadan" anlamındadır. "BÇ" harfleri ise "Birçok" anlamına gelmektedir. "B->BÇ" ile belirtilen harita "Bir noktadan birçok noktaya" anlamına gelmektedir. "Negatif döngülü" yazan haritalar ise haritanın içeriğinde negatif döngü olduğunu belirtmektedir.

Şekil 5.3. Harita Seçme Ekranı

5.4. Algoritma Seçme Ekranı

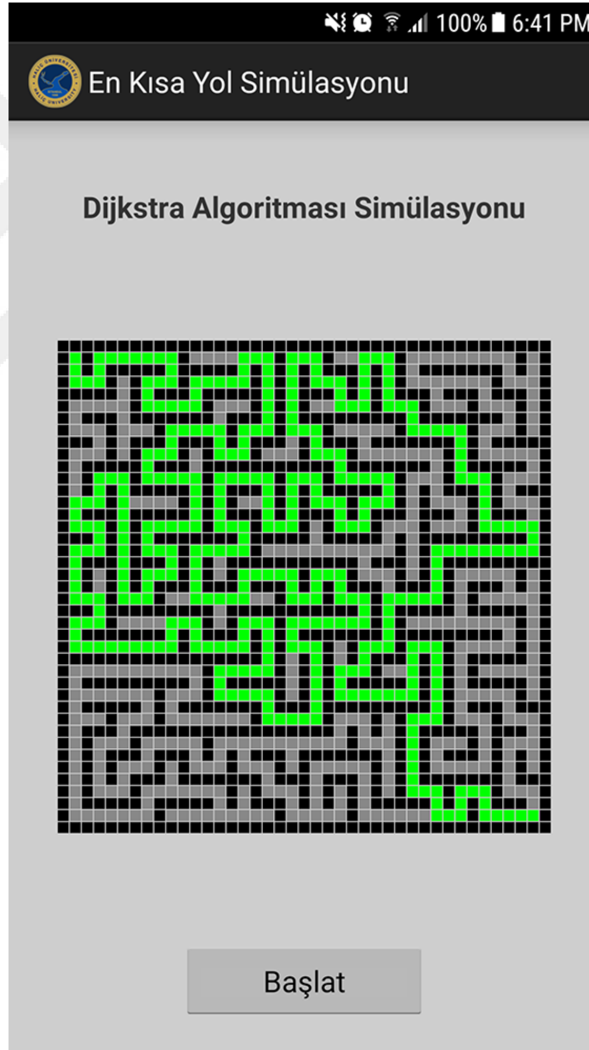
Harita seçildikten sonra uygulama üzerinde yazılmış 4 farklı en kısa yol bulma algoritması karşımıza çıkmaktadır. Bu algoritmalardan birisi sırası önemli olmadan seçilebilir. Bu ekrandan seçilen algoritma, önceki ekrandan seçilen harita ile birlikte simülasyon ekranında kullanılacaktır. "Sonuçlar" butonuna basıldığında ise algoritmalar sırayla çalıştırıldıktan sonra çıkan sonuçların gösterildiği sonuçlar sayfası açılmaktadır. Harita Seçme ekranı Şekil 5.4'te gösterilmiştir. Algoritma seçme ekranı kaynak kodları (EK 4)'te gösterilmiştir.



Şekil 5.4. Algoritma Seçme Ekranı

5.5. Simülasyon Ekranı

Simülasyon ekranı Şekil 5.5'te gösterilmiştir. Ekranın en üstünde hangi algoritmanın seçildiği görünmektedir. Ekranın ortasında Harita Seçme ekranından seçilen haritanın ızgaralar şeklinde karelerden oluşan bir görüntüsü bulunmaktadır. Bu kısımda seçilen algoritma çalıştırıldığında algoritmanın bulduğu en kısa yol, ekranın ortasındaki harita üzerinde yeşil kareler ile adım adım gösterilecektir. Ekranın altında bulunan "başlat" butonuna basarak algoritmanın çalıştırılması sağlanmaktadır. En kısa yol başlangıç noktası en sol üst ve bitiş noktası en sağ alt olmak üzere çalışacaktır. Simülasyon ekranı kaynak kodları (EK 5)'te gösterilmiştir.



Şekil 5.5. Simülasyon Ekranı

5.6. Sonuçlar Ekranı

Haritalar sayfasındaki “Harita1”, bir noktadan bir noktaya en kısa yolu bulan bir haritadır. Bu harita seçildiğinde 4 algoritma da kendi yeteneklerini kullanarak sadece bir başlangıç noktasından sadece bir bitiş noktasına gitmeye çalışır. Burada bazı algoritmalar diğerlerine göre yavaş veya hızlı çalışır. Algoritmanın tasarımına bağlı olarak değişecek olan bu hızlar, algoritmayı kullanacak kişiye bir karşılaştırma sonuçlarını hesaplar ve çıkan bu sonuçlar, sonuçlar sayfasında gösterilir.

Şekil 5.3’te görülen harita seçimi sayfasında “Harita1” seçildikten sonra Şekil 5.4’te görülen algoritma seçimi sayfasında Dijkstra, Bellman-Ford, A* ve Floyd-Warshall algoritmalarından birisi seçilerek simülasyon sayfasının açılması sağlanır. Şekil 5.5’te görülen başlat butonuna basılarak algoritmanın çalışması sağlanır. Seçilen algoritma en kısa yolu bulmayı tamamladığında bulunan bu yol yeşil kutucuklar ile Şekil 5.5’deki gibi gösterilir. Tüm algoritmalar için bu işlem uygulandığında bir noktadan bir noktaya hesaplama yapan “Harita1” haritasının sonuçları Şekil 5.6’da gösterilmiştir.

Harita 1 Sonuçlar
(Bir noktadan bir noktaya)

	Süre	Uzunluk	Etkileşim
Dijkstra	154ms	377	1596
Bellman-Ford	10180ms	377	2681280
A Star	78ms	377	521
Floyd-Warshall	61518ms	377	636805

Şekil 5.6. Harita1’in Hesaplanan Sonuçları

Şekil 5.6’da görülen “Süre” sütunu, algoritmanın çalışması süresini göstermektedir. “Uzunluk” sütunu algoritmanın bulabildiği en kısa yolun uzunluğunu göstermektedir. “Etkileşim” sütunu ise algoritmanın çalışma süresince haritadaki noktalar ile kaç kez işlem yaptığını göstermektedir.

Bir noktadan bir noktaya hesap yapan haritada algoritmalar tek tek çalıştırıldığında Dijkstra algoritması 154ms, Bellman-Ford algoritması 10180ms, A* algoritması 78ms ve Floyd-Warshall algoritması 61518ms de en kısa yolu bulabilmişlerdir. Sonuçlardan görüldüğü gibi, bir noktadan bir noktaya hesap gerektiği durumlarda en iyi performans gösteren algoritma A* algoritması olduğu görüşmüştür. A* algoritması, sadece bir hedefe odaklanan bir algoritma olduğu için diğer algoritmaların aksine sezgisel olarak ilerleyerek en hızlı sonucu bulmuştur.

A* algoritması hız anlamında özellikle strateji oyunlarında çok kullanılmaktadır. Bunun nedeni bu tür oyunlarda genellikler bir noktadan bir noktaya hesaplamalar gerektirmesidir. A* algoritması sezgisel olarak tahmini sonuçlar ürettiğinden harita üzerindeki en kısa yolu diğer algoritmalara göre birkaç adım daha fazla bulabilmektedir. Fakat bu fazlalık algoritmada kullanılan ağırlıklar değiştirilerek oyunlarda fark edilmeyecek kadar küçültülmektedir.

Şekil 5.3'ten "Harita2" seçildiğinde bu harita bir noktadan birçok noktaya en kısa yolu aynı anda bulmaya çalışacaktır. Tüm algoritmalar Şekil 5.4'teki ekrandan seçilip tek tek çalıştırıldığında tüm sonuçlar yine "Sonuçlar" sayfasına aktarılacaktır. "Harita 2" üzerinde algoritmalar çalıştırıldığında çıkan sonuçlar Şekil 5.7'de gösterilmiştir.

Harita 2 Sonuçlar
(Bir noktadan birçok noktaya)

	Süre	Uzunluk	Etkileşim
Dijkstra	127ms	-	-
Bellman-Ford	14144ms	-	-
A Star	18703ms	-	-
Floyd-Warshall	61105ms	-	-

Şekil 5.7. Harita2'nin Hesaplanan Sonuçları

Şekil 5.7’de görüldüğü gibi bir noktadan birçok noktaya hesaplamalar yapıldığında A* algoritmasının gözle görülür bir performans kaybı yaşadığı saptanmıştır. Bunun nedeni bu algoritmanın tasarımı gereği her noktan her noktaya hesaplama yapması gerektiğinden kaynaklanmaktadır. Dijkstra algoritması ise gözle görülür bir biçimde hızlı sonuç vermiştir ve bir noktadan birçok noktaya hesaplama yapılmak istendiğinde en iyi sonucun Dijkstra algoritması olduğu görülmüştür. Bu algoritma kendi tasarımı gereği bir hesaplama yaparken bir noktadan diğer tüm noktalara en kısa yolları bulmaktadır. A* algoritması gibi tüm noktalar için ayrı ayrı hesap yapmadığından hız olarak çok daha hızlı olduğu görülmüştür. Dijkstra algoritması bu özelliğinden dolayı genellikle navigasyon uygulamalarında ve negatif değer içermeyen ağ yönlendirme protokollerinde kullanılmaktadır.

Şekil 5.3’ten “Harita 3” seçildiğinde bu harita birçok noktadan birçok noktaya aynı anda hesaplama yapmaktadır. Tüm algoritmalar Şekil 5.4’teki ekrandan seçilip tek tek çalıştırıldığında çıkan sonuçlar Şekil 5.8’de gösterilmiştir. Şekilde Dijkstra ve A* algoritmalarının bu hesaplamada Floyd-Warshall algoritmasından geri kaldıkları saptanmıştır. Floyd-Warshall algoritması yapısı gereği bir noktadan bir noktaya hesaplama yapıldığında algoritmaya verilen tüm noktalar için en kısa yolu bulmaya çalıştığından diğer algoritmalara göre daha yavaş kalmaktadır. Fakat birçok noktadan birçok noktaya hesaplama yapılmak istendiğinde Şekil 5.8’de görüldüğü gibi diğer algoritmalardan çok daha hızlı tüm noktalardan tüm noktalara en kısa yolları bulduğu görülmüştür. “Tower Defence” türündeki yani tüm objelerin aynı anda diğer objelerin yerini bilmesi gerektiği durumlardaki video oyunlarında kullanılmaktadır. Floyd-Warshall algoritması ayrıca Dijkstra gibi daha karmaşık olan ağ yönlendirme protokollerinde de kullanılmaktadır.

Harita 3 Sonuçlar (Birçok noktadan birçok noktaya)			
	Süre	Uzunluk	Etkileşim
Dijkstra	9835ms	-	-
Bellman-Ford	24887ms	-	-
A Star	37651ms	-	-
Floyd-Warshall	377ms	-	-

Şekil 5.8. Harita3'ün Hesaplanan Sonuçları

Şekil 5.3'teki "Harita 4" maliyetlerinde negatif değerler barındırmaktadır. Diğer haritalarda negatif ağırlıklar olmadığı için Dijkstra, Floyd-Warshall ve A* algoritmaları verimli sonuçlar vermiştir. "Harita 4" seçildiğinde ve tüm algoritmalar bu haritada bir noktadan bir noktaya hesaplama yapılması istendiğinde çıkan sonuçlar Şekil 5.7'de gösterilmiştir. Sonuçlara bakıldığında yalnızca Bellman-Ford algoritmasının "971ms" süre ile bir sonuç bulduğu görülmüştür. Bu diğer algoritmaların negatif maliyetler desteklememesinden kaynaklanmaktadır. Bellman-Ford algoritması negatif değer içeren graflarda ve Yönlendirici Bilgi Protokolü uygulamalarında sıkça kullanılmaktadır.

Harita 4 Sonuçlar (Negatif döngülü harita)			
	Süre	Uzunluk	Etkileşim
Dijkstra	-	-	-
Bellman-Ford	971ms	93	174240
A Star	-	-	-
Floyd-Warshall	-	-	-

Şekil 5.7. Harita4'ün Hesaplanan Sonuçları

6. SONUÇLAR

Teknolojinin hızla geliştiği bu dönemde insanların hıza ihtiyacı gittikçe artmaktadır. Kullanılan algoritmaların ne kadar karmaşık ve hızlı olduğuna bağlı olarak insanların bu algoritmalardan alabilecekleri verimler değişkenlik göstermektedir. Bu çalışma ile özellikle video oyunların, harita hesaplamalarında ve navigasyon uygulamalarında kullanılan en kısa yol bulma algoritmaları incelenmiştir.

En popüler en kısa yol algoritmaları olan Dijkstra algoritması, Floyd-Warshall algoritması, Bellman-Ford algoritması ve A* algoritması detaylı bir biçimde açıklanmış ve bu algoritmaların çalışma mantığı örnekler üzerinde detaylı bir şekilde adım adım anlatılmıştır.

Bu çalışmanın içinde yer alan ve Android üzerinde geliştirilen uygulama kullanılarak farklı haritalar tasarlanıp bu farklı haritalarda yukarıda listelenen algoritmaların nasıl bir performansının gösterdiği incelenmiştir ve bu performanslara dayanarak hangi tür algoritmanın hangi tür oyunlarda kullanılabilmesi daha uygundur önerileri sunulmuştur.

Dijkstra algoritması açgözlü bir yaklaşım ile bir noktadan tüm noktalara hesaplamalar yapmaya çalıştığı için verimlilik konusunda A* algoritmasına göre daha fazla işlem yaptığı görülmüştür. Algoritma her notayı A* algoritmasına göre daha fazla kontrol etmesinden dolayı iterasyon sayısının arttığı görülmüştür. Ne kadar fazla iterasyon yapılırsa bu o kadar çalışma zamanı demektir. A* algoritması ise bitiş noktasına odaklandığından her bir noktaya zaman harcamadan bitiş noktasına en yakın noktalar üzerinde işlemler yapığından dolayı daha performanslı olduğu saptanmıştır.

Dijkstra algoritması A* algoritmasına göre daha kontrolcü bir algoritmadır. A* algoritmasının sezgisel yaklaşımından dolayı A* algoritmasının her zaman en optimum yolu bulmadığı görülmüştür. Bunun nedeni A* algoritmasının en son bulunduğu noktaya göre sezgisel hesaplamalar yapmasıdır. Yani haritada engel sayısı arttığında A* algoritması hesaplamalar yaparken sezgisel yaklaşım kullanması nedeniyle engelleri görmezden geldiği için bitiş noktasına yaklaştıkça algoritmanın

sonlanacağını zannettiği görülmüştür ve buna bağlı olarak performansında düşüş olduğu saptanmıştır.

Bellman-Ford algoritmasını ise Dijkstra algoritmasından ayıran en önemli faktör yollara verilebilen negatif ağırlıklardır. Dijkstra algoritmasına negatif ağırlık verildiğinde hesaplamalar yapılırken sonsuz döngüye girme ihtimali veya yanlış sonuç çıkartma ihtimali bulunmaktadır. Bu algoritma Dijkstra algoritmasında olduğu gibi sadece bir noktadan diğer tüm noktalara hesaplamalar yapmaktadır. Fakat kullanılan grafta negatif ağırlık yok ise Dijkstra algoritması daha performanslı çalıştığı saptanmıştır.

Floyd Warshall algoritmasını ise diğer algoritmalardan ayıran özellik herhangi bir noktadan diğer tüm noktalara olan en kısa yolu bulabilmesidir. Aynı işlem diğer algoritmalarda yapıldığında Floyd Warshall kadar performanslı çalışmadığı görülmüştür. Nitekim Floyd Warshall algoritması "Tower Defence" türündeki, yani her bir objenin birbiri ile arasındaki kısa yolu bilmesini gerektiren oyunlarda daha çok performans gösterecektir.

Bu bilgiler doğrultusunda en kısa yolu seçecek kullanıcının algoritmayı hangi amaçla kullanacağını belirledikten sonra kendisi için en uygun algoritmayı anlatılan avantaj ve dezavantajları göz önünde bulundurarak seçmesi gerekmektedir. Sonuçlardan da görüldüğü gibi hiçbir algoritma en iyi algoritma olarak belirlenmemiştir. Algoritmanın kullanım amacına bağlı olarak algoritmaların farklı haritalarda daha fazla performans gösterdiği görülmüş ve üstünlükleri saptanmıştır. Bu sonuçlar dikkate alınıp en kısa yol algoritması seçimi yapılırsa kullanıcılar için hem algoritmanın kullanılacağı sistem üzerindeki bellek ihtiyacı en aza indirilebilir hem de algoritma daha verimli çalıştırılarak zamandan tasarruf sağlanabilir.

7. KAYNAKLAR

[1] Google Search Statistics. Erişim Tarihi: 17.08.2017,
http://www.internetlivestats.com/google-search-statistics/

[2] Clash of Clans grows. Erişim Tarihi: 20.08.2017,
http://www.businessinsider.com/r-clash-of-clans-maker-supercells-profit-grows-despite-pokemon-challenge-2017-2

[3] Dune 2000 Erişim Tarihi: 25.08.2017,
http://web.archive.org/web/20000930035519/http://www.westwood.com/games/dune2000/press.html

[4] Age_of_Empires 2. Erişim Tarihi: 25.08.2017,
https://en.wikipedia.org/wiki/Age_of_Empires_II

[5] Bodlaender, H.: Complexity of path-forming games. Theoretical Computer Science 110(1), 215–245 (1993)

[6] GPS phones generate traffic information. Erişim Tarihi: 02.09.2017,
https://www.gpsbusinessnews.com/GPS-phones-generate-traffic-information-in-Russia_a1206.html

[7] El-Harezmi Biyografi. Erişim Tarihi: 11.09.2017,
http://matematik.dpu.edu.tr/index/sayfa/3119/el-harezmi

[8] Problem-Solving Strategies. Erişim Tarihi: 13.09.2017,
https://www.education.com/reference/article/problem-solving-strategies-algorithms/

[9] GOTTSCHALK v. BENSON. Erişim Tarihi: 15.09.2017,
http://caselaw.findlaw.com/us-supreme-court/409/63.html

[10] Sidney A. DIAMOND. Erişim Tarihi: 15.09.2017, Commissioner of Patents and Trademarks. https://www.law.cornell.edu/supremecourt/text/450/175#

[11] LZW Patent. Erişim Tarihi: 15.09.2017,
http://www.doc.ic.ac.uk/~nd/surprise_95/journal/vol2/hml/article2.imag4.html

[12] Google Algorithms. Erişim Tarihi: 24.09.2017,
https://www.portent.com/blog/seo/3-google-algorithms-we-know-about-200-we-dont.htm

[13] EDSGER WYBE DIJKSTRA. Erişim Tarihi: 30.09.2017, http://amturing.acm.org/award_winners/dijkstra_1053701.cfm
[14] Big O Reference. Erişim Tarihi: 05.10.2017, http://bigoref.com/
[15] Dijkstra's Algorithm - Shortest Path. Erişim Tarihi: 06.10.2017, http://vasir.net/blog/game_development/dijkstras_algorithm_shortest_path
[16] Shortest path. Erişim Tarihi: 06.10.2017, https://dl.acm.org/citation.cfm?doid=367766.368168
[17] Algebraic structures for transitive closure. Erişim Tarihi: 10.10.2017, https://ac.els-cdn.com/0304397577900561/1-s2.0-0304397577900561-main.pdf?_tid=5b256ffc-ea85-11e7-a663-00000aab0f6b&acdnat=1514324594_b10d3dbb67cedb51cf52ae1672b63481
[18] Algorithms. Erişim Tarihi: 11.10.2017, http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lectures/lecture21.html
[19] Bellman–Ford Algorithm. Erişim Tarihi: 11.10.2017, https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/
[20] Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, 269–271 (1959)
[21] Time complexity, space complexity, and the O-notation. Erişim Tarihi: 15.10.2017, http://www.leda-tutorial.org/en/official/ch02s02s03.html
[22] What Is Java?. Erişim Tarihi: 03.11.2017, https://www.thoughtco.com/what-is-java-2034117
[23] JDK and the JRE. Erişim Tarihi: 07.11.2017, https://docs.oracle.com/javase/9/install/installation-jdk-and-jre-microsoft-windows-platforms.htm#JSJIG-GUID-A7E27B90-A28D-4237-9383-A58B416071CA
[24] Types, Values, and Variables. Erişim Tarihi: 07.11.2017, https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html
[25] Access Modifiers in java. Erişim Tarihi: 08.11.2017, https://www.javatpoint.com/access-modifiers
[26] Declaring Member Variables. Erişim Tarihi: 08.11.2017, https://docs.oracle.com/javase/tutorial/java/javaOO/variables.html
[27] Object-oriented Programming (OOP) Basics. Erişim Tarihi: 12.11.2017, https://www.ntu.edu.sg/home/ehchua/programming/java/J3a_OOPBasics.html

[28] Google announces on Android. Erişim Tarihi: 12.11.2017, https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users
[29] Open Handset Alliance. Erişim Tarihi: 21.11.2017, https://www.openhandsetalliance.com/
[30] Number of available applications in the Google Play Store. Erişim Tarihi: 22.11.2017, https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/
[31] Industry Leaders Announce Open Platform for Mobile Devices. Erişim Tarihi: 01.12.2017, http://www.openhandsetalliance.com/press_110507.html
[32] ART and Dalvik. Erişim Tarihi: 01.12.2017, https://source.android.com/devices/tech/dalvik/
[33] Platform Architecture. Erişim Tarihi: 05.12.2017, https://developer.android.com/guide/platform/index.html

8. EKLER

EK 1: Giriş Ekranı Kaynak Kodları

```
package tr.edu.halic.gbenkisayol.framents;

import ...

public class FragmentGiris extends FragmentBase {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        super.onCreateView(inflater, container, savedInstanceState);

        View view = inflater.inflate(R.layout.fragment_giris, container, attachToRoot: false);

        Button btnStart = view.findViewById(R.id.btn_start);
        Button btnSettings = view.findViewById(R.id.btn_settings);

        btnStart.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                getMA().myFragmentManager.changeFragment(new FragmentHaritaSec());
            }
        });
        btnSettings.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                getMA().myFragmentManager.changeFragment(new FragmentAyarlar());
            }
        });

        return view;
    }

    @Override
    public void onBack() {
        super.onBack();
        Utils.showAlertDialog(getMA(),
                "En Kısa Yol Simülasyonu", message: "Çıkmak istediğinizden emin misiniz?",
                positiveButtonText: "Çıkış", negativeButtonText: "İptal",
                new Runnable() {
                    @Override
                    public void run() {
                        getMA().finish();
                        System.exit( status: 0);
                    }
                }, negativeButtonCallback: null,
                setCancelable: true);
    }

}
```

EK 2: Ayarlar Ekranı Kaynak Kodları

```
package tr.edu.halic.gbenkisayol.framents;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.EditText;

import tr.edu.halic.gbenkisayol.R;

public class FragmentAyarlar extends FragmentBase {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        super.onCreateView(inflater, container, savedInstanceState);

        View view = inflater.inflate(R.layout.fragment_ayarlar, container,  attachToRoot: false);
//
        Button btnSave = view.findViewById(R.id.btn_save);
        EditText etStep= view.findViewById(R.id.et_step);
//
        btnSave.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                getMA().stepSize = Integer.parseInt(etStep.getText().toString());
            }
        });

        return view;
    }

    @Override
    public void onBack() {
        getMA().myFragmentManager.changeFragment(new FragmentGiris());
    }
}
```

EK 3: Harita Seçimi Ekranı Kaynak Kodları

```
package tr.edu.halic.gbenkisayol.framents;

import ...

public class FragmentHaritaSec extends FragmentBase {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        super.onCreateView(inflater, container, savedInstanceState);

        View view = inflater.inflate(R.layout.fragment_harita_sec, container,  attachToRoot: false);

        Button btn1 = view.findViewById(R.id.btn_1);
        Button btn2 = view.findViewById(R.id.btn_2);
        Button btn3 = view.findViewById(R.id.btn_3);
        Button btn4 = view.findViewById(R.id.btn_4);
        Button btn5 = view.findViewById(R.id.btn_5);
        Button btn6 = view.findViewById(R.id.btn_6);
        Button btn7 = view.findViewById(R.id.btn_7);
        Button btn8 = view.findViewById(R.id.btn_8);

        initButtons(btn1,  mapId: 1);
        initButtons(btn2,  mapId: 2);
        initButtons(btn3,  mapId: 3);
        initButtons(btn4,  mapId: 4);
        initButtons(btn5,  mapId: 5);
        initButtons(btn6,  mapId: 6);
        initButtons(btn7,  mapId: 7);
        initButtons(btn8,  mapId: 8);

        return view;
    }

    private void initButtons(Button btn, final int mapId) {
        btn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                getMA().selectedMapId = mapId;
                getMA().myFragmentManager.changeFragment(new FragmentAlgoritmaSec());
            }
        });
    }

    @Override
    public void onBack() {
        getMA().myFragmentManager.changeFragment(new FragmentGiris());
    }
}
```

EK 4: Algoritma Seçimi Ekranı Kaynak Kodları

```java
package tr.edu.halic.gbenkisayol.framents;

import ...

public class FragmentAlgoritmaSec extends FragmentBase {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        super.onCreateView(inflater, container, savedInstanceState);

        View view = inflater.inflate(R.layout.fragment_algoritma_sec, container, attachToRoot: false);

        Button btnDijkstra = view.findViewById(R.id.btn_dijkstra);
        Button btnBellmanFord = view.findViewById(R.id.btn_bellman_ford);
        Button btnAStar = view.findViewById(R.id.btn_a_star);
        Button btnFloydWarshall = view.findViewById(R.id.btn_floyd_warshall);

        Button btnResult = view.findViewById(R.id.btn_results);
        btnResult.setEnabled(false);

        initButtons(btnDijkstra, MainActivity.ALGORITHM.DIJKSTRA);
        initButtons(btnBellmanFord, MainActivity.ALGORITHM.BELLMAN_FORD);
        initButtons(btnAStar, MainActivity.ALGORITHM.A_STAR);
        initButtons(btnFloydWarshall, MainActivity.ALGORITHM.FLOYD_WARSHALL);

        return view;
    }

    private void initButtons(Button btn, final MainActivity.ALGORITHM algorithm) {
        btn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                getMA().selectedAlgorithm = algorithm;
                getMA().myFragmentManager.changeFragment(new FragmentSimulasyon());
            }
        });
    }

    @Override
    public void onBack() {
        getMA().myFragmentManager.changeFragment(new FragmentHaritaSec());
    }
}
```

EK 5: Simülasyon Ekranı Kaynak Kodları

```
package tr.edu.halic.gbenkisayol.framents;

import android.graphics.Color;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.LinearLayout;
import android.widget.TextView;

import tr.edu.halic.gbenkisayol.R;

public class FragmentSimulasyon extends FragmentBase {

    private View[][] views;

    public int[][] map;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        super.onCreateView(inflater, container, savedInstanceState);

        View view = inflater.inflate(R.layout.fragment_simulasyon, container,  attachToRoot: false);

        TextView tvTitle = view.findViewById(R.id.tv_title);

        tvTitle.setText(getMA().selectedAlgorithm.title + " Algoritması Simülasyonu");

        LinearLayout llRoot = view.findViewById(R.id.ll);

        int rows = 18;
        int cols = 18;

        views = new View[rows][cols];

        for (int i = 0; i < rows; i++) {
            LinearLayout.LayoutParams llp = new LinearLayout.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT, ViewGroup.LayoutParams.WRAP_CONTENT);
            LinearLayout llnew = new LinearLayout(getMA());
            llnew.setLayoutParams(llp);

            for (int j = 0; j < cols; j++) {
                View v = new View(getMA());

                int margin = 2;
                LinearLayout.LayoutParams buttonLayoutParams = new LinearLayout.LayoutParams( width: 50,  height: 50);
                buttonLayoutParams.setMargins(margin, margin, margin, margin);

                v.setLayoutParams(buttonLayoutParams);
                v.setBackgroundColor(Color.GRAY);

                llnew.addView(v);

                views[i][j] = v;
            }

            llRoot.addView(llnew);
        }

        for (int i = 0; i < map.length; i++) {
            for (int j = 0; j < map[0].length; j++) {
                if (map[i][j] == 1) {
                    views[i][j].setBackgroundColor(Color.BLACK);
                }
            }
        }

        return view;
    }

    @Override
    public void onBack() {
        getMA().myFragmentManager.changeFragment(new FragmentAlgoritmaSec());
    }

}
```

9. ÖZGEÇMİŞ

1990 yılında İstanbul'un Fatih ilçesinde doğdu. İlköğretimini Yıldız Koleji ve Özkan Koleji'nde tamamladı. Ortaokulunu Dr. Refik Saydam ortaokulunda, lise öğrenimini ise Kemal Hasoğlu Lisesinde tamamladı. Lisans eğitimini T. C. Haliç Üniversitesinde Bilgisayar Mühendisliği bölümünde aldı. Halen T. C. Haliç, Üniversitesi Fen bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim dalı Tezli Yüksek Lisans öğrencisi olarak devam etmektedir. 2008 yılında başladığı serbest yazılım geliştirme sürecine daha sonra Türk Telekom şirketinde staj yaptıktan sonra Argela Teknoloji, İnvio yazılım şirketlerinde yazılım geliştirici olarak çalıştı ve bugün Miops Teknoloji şirketinde Senior Android ve Fullstack Web Developer olarak iş hayatına devam etmektedir.