



T.C.  
İSTANBUL ÜNİVERSİTESİ-CERRAHPAŞA  
MÜHENDİSLİK FAKÜLTESİ



## BİTİRME PROJESİ

|UNITY ÜZERİNDE C# İLE OYUN GELİŞTİRME|

|ELEKTRONİK ANABİLİM DALI|

YASİN ENES ÖZDEMİR

1316130047

DANIŞMAN  
|Yrd. Doç. Dr. Abdurrahim Akgündoğdu |

|Haziran, 2020|

İSTANBUL

## ÖNSÖZ

| Yapay zekâ neredeyse ilkel bilgisayarların çıkışından beri insanlığın yaratmak istediği unsurlar arasındadır. Geçmiş tarihe bakıldığından Yapay zekâ kavramını ilk olarak 1308 yılında Katalan şair ve teolog Ramon Llull Ars, Generalis Ultima (The Ultimate General Art) adlı kitabında kullanmıştır. Burada kavramların kombinasyonlarından oluşan yeni bir bilgi türünden ilk kez bahsedilmiştir. Yapay zekanın ilk olarak kavramlaştırılması Ramon Llull tarafından yapılmıştır.

Hesap makinelerinin babası sayılan matematikçi Blaise Pascal 1642 yılında, vergi memuru olan babasına yardımcı olması için ilk mekanik hesap makinesini geliştirmiştir. Alan Mathison Turing “Makineler düşünebilir mi?” sorunsalını ortaya atarak makine zekâsını tartışmaya açan kişi olmuştur. 1943’te II. Dünya Savaşı sırasında Cripto analizi gereksinimleri ile üretilen elektromekanik cihazlar sayesinde bilgisayar bilimi ve yapay zekâ kavramları oluşmuştur.

İspanyol mühendis Leonardo Torres y Quevedo, 1914 yılına gelindiğinde ilk satranç oynayabilen makineyi tanıtmıştır. Makine hiçbir insan etkisi olmadan oyunu kendi devam ettirebilmektedir. 1951 yılında Nim isimli bir strateji oyunu sanal ortama uyarlanmıştır. Oyun için yaratılan yapay zekâ, insanları yenebilecek düzeyde olmuştur. 1997 yılında yapılan Deep Blue isimli bilgisayar yapay zekâsı sayesinde başlangıçta orta düzey satranç oyuncularını yenmekteydi. İlerleyen zamanlarda yapılan geliştirmeler sonucunda dünyanın en ünlü satranç oyuncusu olan Garry Kasparov'u yenmiştir.

Günümüze bakıldığından ise yapay zekâ konusunda birçok ilerleme kaydedilmiştir. Kullanım alanları arasında sağlık sektörü, sesli asistanlar, e-ticaret, otonom araçlar ve iletişim ön plana çıkışmış durumdadır. Çağımızda yapay zekanın hemen hemen kullanılmadığı yer yoktur. Dijital oyunlarda yapay zekâ tekniklerinden olan derin öğrenme modelli yapay zekâ ve kural tabanlı yapay zekâ kullanımı yaygınlaşmıştır. |

Haziran 2020

|Yasin Enes Özdemir|

# **İÇİNDEKİLER**

	Sayfa No
ÖNSÖZ	i
İÇİNDEKİLER	ii
1. GİRİŞ	1
1.1. Tanım	2
1.2. Tarihçe	2
1.3. Gelişim süreci	3
1.3.1. Sembolik Yapay Zekâ	4
1.3.2. Sibernetik Yapay Zekâ	5
1.3.3. Uzman Sistemler	5
2. YAPAY ZEKA	7
2.1 Ajanlar	7
2.2 Arama Yöntemleri	8
2.2.1. Bilgisiz Arama Yöntemleri	9
2.2.1.1. Enine/Yayılımlı Arama	9
2.2.1.2. Eşit maliyetli Arama	10
2.2.1.3. Derinlik öncelikli Arama	10
2.2.1.4. Derinlik sınırlı Arama	12
2.2.1.5. Yinelemeli Derinleşen Arama	13
2.2.1.6. İki Yönlü Arama	13
2.2.2. Bilgili Arama Yöntemleri	14
2.2.2.1. 1. Aç Gözülü Arama	14
2.2.2.2. A* Arama	16

<b>3. DAVRANIŞ AĞAÇLARI</b>	<b>18</b>
<b>3.1. Operatörler</b>	<b>18</b>
<b>3.1.1. Ve Operatörü - Sıra Görevi</b>	<b>18</b>
<b>3.1.2. Veya Operatörü - Seçim Görevi</b>	<b>20</b>
<b>3.1.3. Paralel Görev</b>	<b>21</b>
<b>3.2. Dekaratörler</b>	<b>23</b>
<b>3.2.1 Tekrarlayıcı</b>	<b>23</b>
<b>3.2.2 Çevirici</b>	<b>23</b>
<b>3.2.3 Hata Döndür</b>	<b>24</b>
<b>3.2.4 Başarı Döndür</b>	<b>25</b>
<b>3.2.5 Başarısızlığa Kadar</b>	<b>26</b>
<b>3.2.6 Başarıya Kadar</b>	<b>27</b>
<b>4. Uygulamalar</b>	<b>29</b>
<b>4.1. Nesneyi Gör Görevi</b>	<b>29</b>
<b>4.2. Nesneyi Duy Görevi</b>	<b>30</b>
<b>4.3. Nesneyi Yakala Görevi</b>	<b>31</b>
<b>4.4. Nesneyi Takip Et Görevi</b>	<b>33</b>
<b>4.5. Nesneyi Ara Görevi</b>	<b>34</b>
<b>4.6. Devriye Gez Görevi</b>	<b>36</b>
<b>4.7. Beklemede Kal Görevi</b>	<b>38</b>
<b>4.8. Nesneye Saldır Görevi</b>	<b>40</b>
<b>4.9. Nesneyi Savun Görevi</b>	<b>40</b>
<b>5. ÖRNEK SAHNE</b>	<b>42</b>
<b>5.1.Sahne Yapısı</b>	<b>42</b>
<b>5.2.Ajan Çeşitleri</b>	<b>43</b>
<b>5.2.1. Savunan Ajanlar</b>	<b>43</b>
<b>5.2.2. Saldıran Ajanlar</b>	<b>44</b>
<b>5.2.3. Devriye Gezen Ajanlar</b>	<b>46</b>
<b>KAYNAKLAR</b>	<b>47</b>

## 1. GİRİŞ

Bilgisayar ve Video Oyunları pazarı son yıllarda giderek ivmelenen bir gelişme göstermeye başlamıştır. Bunun temel nedenlerinden birisi gelişmiş oyunların artık sadece belirli oyun salonlarından çıkip masaüstü sistemlerin ve kullanılan özelleşmiş grafik hızlandırıcıları ile hemen her alanda karşımıza çıkmaya başlamasıdır. Hemen her yıl ikiye katlanan işlemci ve grafik işleyici yongaların hızı oyun konusundaki rekabeti körüklemiştir. Özellikle son yıllarda yeni pazarlar arayışına giren dev şirketlerin oyun programlama ve özelleşmiş oyun konsolu tasarımlına ağırlık vermeleri de pazardaki rekabetin ve potansiyelin açık bir göstergesidir.

Yıllar geçtikçe işlem gücünün artması sadece oyunların daha hızlı çalışmasını değil oyunların biçimlerini de değiştirmeye başlamıştır. Önceleri sadece hamle tabanlı (satranç, dama ya da basit zeka oyunları) ve düşük kaliteli iki boyutlu karakterlerin tekdone bir mantığa göre hareketlerine dayanan oyunlar mevcutken günümüz bilgisayarlarında gerçek zamanlı ve neredeyse gerçeğe yakın görünümde ortamlarda onlarca karakterin birbiri ile insan davranışına yakın hareketlerle etkileşimiini sergilemektedir. Oyunların ve oyun şirketlerinin her geçen gün artması da kullanıcıların daha gerçekçi grafik ve daha akıllı oyun karakterlerine olan isteğini arttırmıştır. Bu noktada ise oyun programının en zor ve en etkileyici yönlerinden yapay zeka devreye girer.

Oyun programlamada kullanılan yapay zekanın amacı oyuncu ile etkileşimde bulunan karakterlerin ya da oyunun geçtiği ortamın mümkün olduğunda gerçek insan, topluluk ve dünya (ya da kimyasal- fiziksel – biyolojik olarak anlamlı ortam) yaşam ortamına benzetilmeye çalışılmasıdır. Başka bir anlamda da bilgisayarın yetenekli bir oyuncuya aynı derecede yetenekli karşılık vermesi, oyuncunun kurduğu planlara benzeyen planlar ya da tuzaklar kurması, gerektiğinde oyuncuyu zor duruma düşürüp onu yenebilmesi oyun programındaki yapay zeka kalitesini belirler.

Ancak burada gözden kaçırılmaması gereken bir nokta oyunlardaki yapay zeka dozunun iyi ayarlanmasıdır. Eğer bilgisayarı oyuncuya karşı çok hızlı reaksiyon verecek şekilde ayarlarsanız ya da oyuncuyu bunaltacak şekilde iyi taktikler gerçekleştirirse oyunun “oynanabilirlik” seviyesi düşer ve kaçınılmaz bir pazar başarısızlığı ortaya çıkabilir. Bunun

önüne geçmek için oyunlarda çeşitli yöntemler izlenir. Bunlardan bir tanesi oyuna “seviye” özelliği eklenir ve oyuncu bilgisayarın ne kadar “zeki” davranışacağını kendisi belirler. Yukarıda bahsedilen gerekçeden dolayı modern oyunlarda günümüz akademik yapay zeka araştırmalarının popüler konularından olan Yapay sinir ağları (Artificial Neural Networks) ve Genetik Algoritmalar gibi konular birkaç oyun türü hariç halen yeni oyunlarda tercih edilmemektedir.

### **1.1. Tanım**

İdealize edilmiş bir yaklaşımı göre yapay zekâ, insan zekâsına özgü olan, algılama, öğrenme, çoğul kavramları bağlama, düşünme, fikir yürütme, sorun çözme, iletişim kurma, çıkarımsama yapma ve karar verme gibi yüksek bilişsel fonksiyonları veya otonom davranışları sergilemesi beklenen yapay bir işletim sistemidir. Bu sistem aynı zamanda düşüncelerinden tepkiler üretebilmeli (eyleyici yapay zekâ) ve bu tepkileri fiziksel olarak dışa vurabilmelidir.

### **1.2. Tarihçe**

"Yapay zekâ" kavramının geçmişi modern bilgisayar bilimi kadar eskidir. Fikir babası, "Makineler düşünübilir mi?" sorunsalını ortaya atarak makine zekâsını tartışmaya açan Alan Mathison Turing'dir. 1943'te II. Dünya Savaşı sırasında Kripto analizi gereksinimleri ile üretilen elektromekanik cihazlar sayesinde bilgisayar bilimi ve yapay zekâ kavramları doğmuştur.

Alan Turing, Nazilerin Enigma makinesinin şifre algoritmasını çözmeye çalışan matematikçilerin en ünlü olanlarından biriydi. İngiltere, Bletchley Park'ta şifre çözme amacıyla başlatılan çalışmalar, Turing'in prensiplerini oluşturduğu bilgisayar prototipleri olan Heath Robinson, Bombe Bilgisayarı ve Colossus Bilgisayarları, Boole cebirine dayanan veri işleme mantığı ile Makine Zekâsı kavramının oluşmasına sebep olmuştur.

Modern bilgisayarın atası olan bu makineler ve programlama mantıkları aslında insan zekâından ilham almışlardı. Ancak sonraları, modern bilgisayarlarımız daha çok uzman sistemler diyebileceğimiz programlar ile gündelik hayatımızın sorunlarını çözmeye yönelik kullanım alanlarında daha çok yaygınlaştılar. 1970'li yıllarda büyük bilgisayar üreticileri olan Microsoft, Apple, Xerox, IBM gibi şirketler kişisel bilgisayar (PC Personal Computer) modeli

ile bilgisayarı popüler hale getirdiler ve yaygınlaştırıldılar. Yapay zekâ çalışmaları ise daha dar bir araştırma çevresi tarafından geliştirilmeye devam etti.

Bugün, bu çalışmaları teşvik etmek amacıyla Turing'in adıyla anılan Turing Testi ABD'de Loebner ödülleri adı altında makine zekâsına sahip yazılımların üzerinde uygulanarak başarılı olan yazılımlara ödüller dağıtılmaktadır.

Turing Testinin içeriği kısaca şöyledir: birbirini tanımayan birkaç insandan oluşan bir denek grubu birbirleri ile ve bir yapay zekâ diyalog sistemi ile geçerli bir süre sohbet etmektedirler. Birbirlerini yüz yüze görmeden yazışma yolu ile yapılan bu sohbet sonunda deneklere sorulan sorular ile hangi deneğin insan hangisinin makine zekâsı olduğunu saptamaları istenir. İlginçtir ki, şimdije kadar yapılan testlerin bir kısmında makine zekâsı insan zannedilirken gerçek insanlar makine zannedilmiştir.

Loebner Ödülü kazanan yapay zekâ diyalog sistemlerinin yeryüzündeki en bilinen örneklerinden biri A.L.I.C.E'dir. Carnegie üniversitesinden Dr.Richard Wallace tarafından yazılmıştır. Bu ve benzeri yazılımlarının eleştiri toplamalarının nedeni, testin ölçümlendiği kriterlerin konuşmaya dayalı olmasından dolayı programların ağırlıklı olarak diyalog sistemi (chatbot) olmalarıdır.

Türkiye'de de makine zekâsı çalışmaları yapılmaktadır. Bu çalışmalar doğal dil işleme, uzman sistemler ve yapay sinir ağları alanlarında Üniversiteler bünyesinde ve bağımsız olarak sürdürülmektedir. Bunlardan biri, D.U.Y.G.U. - Dil Uzam Yapay Gerçek Uslamlayıcı 'dır.

### **1.3. Gelişim süreci**

İlk araştırmalar ve yapay sinir ağları. İdealize edilmiş tanımlı yapay zekâ konusundaki ilk çalışmaların biri McCulloch ve Pitts tarafından yapılmıştır. Bu araştırmacıların önerdiği, yapay sinir hücrelerini kullanan hesaplama modeli, önermeler mantığı, fizyoloji ve Turing'in hesaplama kuramına dayanıyordu. Herhangi bir hesaplanabilir fonksiyonun sinir hücrelerinden oluşan ağlarla hesaplanabileceğini ve mantıksal ve ve veya işlemlerinin gerçekleştirilebileceğini gösterdiler. Bu ağ yapılarının uygun şekilde tanımlanması hâlinde öğrenme becerisi kazanabileceğini de ileri sürdüler. Hebb, sinir hücreleri arasındaki bağlantıların şiddetlerini değiştirmek için basit bir kural önerince, öğrenebilen yapay sinir ağlarını gerçekleştirmek de olası hale gelmiştir.

1950'lerde Shannon ve Turing bilgisayarlar için satranç programları yazıyorlardı. İlk yapay sinir ağlı temelli bilgisayar SNARC, MIT'de Minsky ve Edmonds tarafından 1951'de yapıldı. Çalışmalarını Princeton Üniversitesi'nde sürdürən Mc Carthy, Minsky, Shannon ve Rochester'le birlikte 1956 yılında Dartmouth'da iki aylık bir açık çalışma düzenledi. Bu toplantıda birçok çalışmanın temelleri atılmakla birlikte, toplantının en önemli özelliği Mc Carthy tarafından önerilen yapay zekâ adının konmasıdır. İlk kuram ispatlayan programlardan Logic Theorist (Mantık kuramcısı) burada Newell ve Simon tarafından tanıtılmıştır.

Yeni yaklaşımlar. Daha sonra Newell ve Simon, insan gibi düşünme yaklaşımına göre üretilmiş ilk program olan Genel Sorun Çözücü (General Problem Solver)'ı geliştirmiştirlerdir. Simon, daha sonra fiziksel simge varsayımini ortaya atmış ve bu kuram, insandan bağımsız zeki sistemler yapma çalışmalarıyla uğraşanların hareket noktasını oluşturmuştur. Simon'ın bu tanımlaması bilim adamlarının yapay zekâya yaklaşımlarında iki farklı akımın ortaya çıktığını belirginleştirmesi açısından önemlidir: Sembolik Yapay Zekâ ve Sibernetik Yapay Zekâ.

### **1.3.1. Sembolik yapay zekâ**

Simon'ın sembolik yaklaşımından sonraki yıllarda mantık temelli çalışmalar egemen olmuş ve programların başarımlarını göstermek için bir takım yapay sorunlar ve dünyalar kullanılmıştır. Daha sonraları bu sorunlar gerçek yaşamı hiçbir şekilde temsil etmeyen oyuncak dünyalar olmakla suçlanmış ve yapay zekânın yalnızca bu alanlarda başarılı olabileceği ve gerçek yaşamdaki sorunların çözümüne ölçeklenemeyeceği ileri sürülmüştür.

Geliştirilen programların gerçek sorunlarla karşılaşıldığında çok kötü bir başarım göstermesinin ardındaki temel neden, bu programların yalnızca sentaktik süreçleri benzeşimevdirerek anlam çıkarma, bağlantı kurma ve fikir yürütme gibi süreçler konusunda başarısız olmasıydı. Bu dönemin en ünlü programlarından Weizenbaum tarafından geliştirilen Eliza, karşısındaki ile sohbet edebiliyor gibi görünmesine karşın, yalnızca karşısındaki insanın cümleleri üzerinde bazı işlemler yapıyordu. İlk makine çevirisi çalışmaları sırasında benzeri yaklaşımlar kullanılıp çok gülünç çevirilerle karşılaşılınca bu çalışmaların desteklenmesi durdurulmuştu. Bu yetersizlikler aslında insan beynindeki semantik süreçlerin yeterince incelenmemesinden kaynaklanmaktadır.

### **1.3.2. Sibernetik yapay zekâ**

Yapay sinir ağları çalışmalarının dahil olduğu sibernetik cephede de durum aynıydı. Zeki davranışları benzeşime lendirmek için bu çalışmalarında kullanılan temel yapılardaki bazı önemli yetersizliklerin ortaya konmasıyla birçok araştırmacılar çalışmalarını durdurdu. Buna en temel örnek, Yapay sinir ağları konusundaki çalışmaların Minsky ve Papert'in 1969'da yayınlanan Perceptrons adlı kitaplarında tek katmanlı algaçların bazı basit problemleri çözemeyeceğini gösterip aynı kısırlığın çok katmanlı algaçlarda da beklenilmesi gerektiğini söylemeleri ile bıçakla kesilmiş gibi durmasıdır.

Sibernetik akımın uğradığı başarısızlığın temel sebebi de benzer şekilde Yapay Sinir Ağının tek katmanlı görevi başarması fakat bu görevle ilgili yargıların veya sonuçların bir yargıya dönüşerek diğer kavramlar ile bir ilişki kurulamamasından kaynaklanmaktadır. Bu durum aynı zamanda semantik süreçlerin de benzeşime lendirilememesi gerektiğini doğurdu.

### **1.3.3. Uzman sistemler**

Her iki akımın da uğradığı başarısızlıklar, her sorunu çözecek genel amaçlı sistemler yerine belirli bir uzmanlık alanındaki bilgiyle donatılmış programları kullanma fikrinin gelişmesine sebep oldu ve bu durum yapay zekâ alanında yeniden bir canlanmaya yol açtı. Kısa sürede Uzman sistemler adı verilen bir metodoloji gelişti.

Uzman sistemler bir konuda belli ön koşullar aynı anda var olduğunda konunun bir uzmanın (bazen ne olasılıkla) ne karar alacağını belirleyen kuralların tümünü içeren bir programı gelen problemlere uygulamak temellidir. Bunun bir avantajı her verilen kararın hangi kurallar uygulanarak verildiğinin kolayca bilinmesi idi. Bu birçok kuralcı bürokratik karar örgütleri için kolayca uygulamalar geliştirilebilmesi demekti. Bu doğal olarak bir otomobilin tamiri için önerilerde bulunan uzman sistem programının otomobilin ne işe yaradığından haberini olmaması da demekti. Buna rağmen uzman sistemlerin başarıları beraberinde ilk ticari uygulamaları da getirdi.

Yapay zekâ yavaş yavaş bir endüstri hâline geliyordu. DEC tarafından kullanılan ve müşteri siparişlerine göre donanım seçimi yapan R1 adlı uzman sistem şirkete bir yılda 40 milyon dolarlık tasarruf sağlamıştı. Birden diğer ülkeler de yapay zekâyı yeniden keşfettiler ve

arastırmalara büyük kaynaklar ayrılmaya başlandı. 1988'de yapay zekâ endüstrisinin cirosu 2 milyar dolara ulaşmıştı.

## 2. YAPAY ZEKA

Bu bölümde oyun geliştirme aşamalarında bilinmesi gereken yapay zeka teorik bilgileri incelenecaktır.

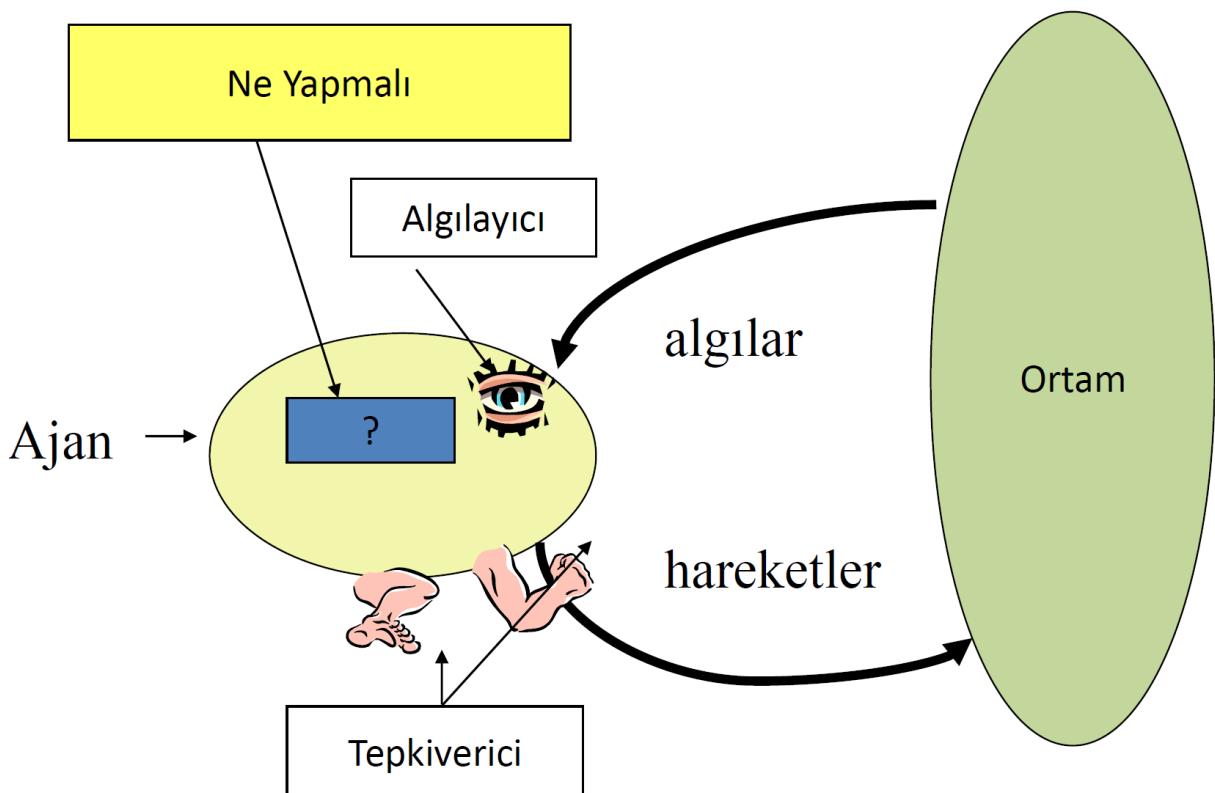
### 2.1.Ajanlar

Algıladıklarını davranışlara haritalayan programlardır. Ortamdan bilgiyi alarak davranışları hakkında karar verirler ve bu kararları uygularlar. Zeki etmenlerin yazılım etmenleri, özerk etmenler, uyarlanabilir ara yüzeyler, kişisel ara yüzeyler, ağ etmenleri gibi farklı isimleri vardır. Tüm bu kavamlar arasında küçük farklılıklar olmasına rağmen, bu kavamların tamamı zeki etmenleri ifade etmede kullanılırlar. Bir zeki etmen, muhakeme yapma ve öğrenme davranışlarının derecelenmesi olarak ifade edilen zekâ ve sistem içinde etmenler ve diğer varlıklar arasındaki etkileşimin doğası ile nitelenebilen özerklik boyutları ile tanımlanır.

En az düzeyde ortamı, durumu algılama ve yorumlama, kararlar verme ve davranışları kontrol etme yeteneği olan zekâ kavramı; ‘Algılayıcılar aracılığı ile çevresini fark edebilen ve çevresini etkileyebilen herhangi bir şey’ olarak tanımlanan akıllı ajanların temelini oluşturmaktadır.

Temel amaç ortamdan algılanan olaylara otomatik ve uygun cevaplar veren özerk sistemler oluşturabilmektir.

Ajanlar algıları tepkilere dönüştüren fonksiyondur ve yeni algılar geldikçe güncellenecek olan veri yapılarını içermektedir. Bu veri yapıları ajanın karar verme yöntemlerine göre işlenerek verilecek tepki üretilir ve algılayıcılardan elde edilen algıları programa aktararak programı çalıştırıp, programın tepki seçenekleri etkileyicilere iletilir.



Şekil 1 Ajanın Yapısı

## 2.2. Arama Yöntemleri

Arama algoritmaları yapay zeka uygulamalarında önemli yere sahiptirler. Birçok problemin çözümünde arama algoritmaları hızlı ve etkili çözümler sunmaktadır. Yapay zeka problemlerinin çözümünde arama evrensel bir tekniktir.

Bir problem için doğru arama stratejisini seçmek gereklidir. Burada arama stratejilerinden hangisini seçeceğini belirlerken aşağıdaki kriterlere bakmak gereklidir:

**Tamlık-Bütünlük (Completeness)** : Problemin bir çözümü olduğunda seçilecek olan strateji bu çözümü bulabilecek mi?

**Zaman Karmaşıklığı (Time Complexity)** : Bir çözüm bulmak ne kadar zaman alacak?

**Uzay Karmaşıklığı (Space Complexity)** : Aramayı yapmak için ne kadarlık bir hafıza(memory) gereklidir? Rekürsif olan algoritmalarla bir önceki durum saklanacağından dolayı

problemin çözümü için gerekli hafıza alanı rekürsif olmayan algoritmala göre daha fazla olacaktır.

**Optimalite-Optimallık-Eniyileme (Optimality) :** Farklı çözümler mevcut iken seçilen stratejinin çözümü optimal olan mı?

**Körüne Arama-Bilgisiz Arama (Blind Search) :** Burada sonuca varmak için içinde bulunan durumun sonuç ile arasındaki adım sayısı veya mesafesinin bilinmemesidir. Yani çözüme ne zaman varılacağı bilinmemektedir. Burada sadece bilinen şey içinde bulunan durumun hedef durum olup olmadığıdır.

### **2.2.1. Bilgisiz Arama Yöntemleri**

Arama işleminin bilmeyerek yapılması demek, arama algoritmasının, probleme özgü kolaylıklarını barındırmaması demektir. Yani her durumda aynı şekilde çalışan algoritmala uninformed search (bilmeden arama) ismi verilir.

#### **2.2.1.1. Enine/Yayılımlı Arama**

Yaygın olan bir arama stratejisidir. Bu stratejide bulunan durum yani düğüm başlangıçta kök düğüm olduğundan ilk önce kök düğüm genişletilir. Genişletmekten kasıt kök düğümden giden bütün yolların ziyaret edilmesidir. Böylece kök düğümün bütün komşuları keşfedilmiş olur. d derinliğinde bulunan tüm düğümler arama ağacında  $d+1$  derinlikte bulunan düğümlerden önce açılırlar(genisletilirler). Bir problemin çözümü varsa enine arama algoritması çözümü garanti eder. Eğer birkaç çözüm var ise en üstte olan yani ilk bulunan çözümü çözüm olarak kabul eder. Yukarıdaki kriterlere göre enine arama tamdır, çünkü çözüm var ise bulmayı garanti eder.

Enine arama algoritması, problemin çözümü daha derinlerde ise verimsiz bir algoritmadır. Çünkü açılan düğüm sayısı artacaktır. Çözümün derinde olması dallanma etmeni(branching factor) olarak bilinmektedir. Başlangıç düğümü de dahil olmak üzere her düğümün  $b$  sayıda çocuğu olduğunu varsayıyalım. Kök ilk genişletildiğinde  $b$  sayıda dal olacaktır bir sonraki derinlikte  $b^2$  dallanma olacaktır. Bu şekilde  $d$ . derinlikte  $b$  üzeri  $k$  olacaktır.

Toplamda değerlendirilmesi gereken düğüm sayısı= $1+b+b^2+b^3\dots$  olacaktır. Bu da zaman karmaşıklığı bakımından önemlidir.

Örnek olarak eğer her düğümün dallanma sayısı  $b=10$  alırsak ve derinliği de  $d=10$  alırsak,

toplam düğüm sayısı= $10\ 000\ 000\ 000$

gereken zaman= $128$  gün

gerekli bellek= $1$  terabyte

Yukarıdaki örnekte de görüldüğü gibi enine arama algoritması dallanma faktörüne göre verimli olup olmadığı belli olmaktadır.

#### Algoritmanın adımları:

- Öncelikle bir başlangıç düğümü seçilir ve bu düğüm işaretlenir.
- Bu düğümün komşuları sırasıyla bir K listesine yazılır. Bu K listesi komşulukları tutan bir listedir.
- Bu komşu düğümler teker teker ziyaret edilir ve işaretlenerek Z listesine eklenir. Z listesi ziyaret edilen düğümleri tutan listedir.
- K listesindeki ilk düğüm alınarak işaretlenir ve K listesinden silinir.
- Silinen düğümün komşuları K listesine eklenir. Bu ekleme yapılırken eğer eklenecek düğüm Z listesinde varsa yani daha önce ziyaret edilmişse eklenmeyecektir.
- Silinen düğümün ziyaret edilen düğümleri Z listesine eklenir (var olanlar eklenmeyecek).
- Bu şekilde K listesindeki tüm düğümler silininceye kadar devam edilir.

#### **2.2.1.2.Eşit maliyetli Arama**

Bir düğüme olan yolları maliyetlerine göre artacak şekilde sıralar. Genişlemeyi yani komşuları ziyaret etmeyi maliyeti en az olan, en yakın olanı seçer ve genişlemeyi bu düğümle yapar.

Eğer tüm yolların maliyeti aynı ise enine arama gibidir. Yolları, maliyeti artan şeklinde genişletir(ziyaret eder)

#### **2.2.1.3.Derinlik öncelikli Arama**

Derinlik öncelikli aramada, arama ağaçları fazla dallanmadan kökten gidilebilecek en uzak düzüme kadar ilerlenir. Burada kök olarak herhangi bir düğüm alınabilir. Kök düğümden

yaprak düğüme giden, tek yol gibi düğümler her iterasyonda saklanır ve düğümler doğrusal bir yapıda depolanırlar. Bundan dolayı kullanılan veri yapısı LIFO Last Input First Out(son giren ilk çıkar)'dur. Bu şekilde kökten yaprağa tek yol olarak tutulmasından dolayı bellek gereksinimi oldukça makulddür.

Dallanma faktörüne baktığımızda ise  $b$  dallanma sayısı ve  $d$  derinliği için gerekli bellek alanı  $b^*d$  olacaktır. Enine aramaya göre ne kadar verimli olduğu görülmektedir.

Derinine aramada olası sorunlar problemin çözümünde gidilen yanlış yolda sıkışıp kalmaktır. Birçok problemin sonsuz veya çok derin ağaçları olabilmektedir böyle durumlarda derinine arama uygun bir çözüm bulamayacaktır. Bu şekilde arama bir önceki duruma dönmeden sürekli devam edecektir, hatta bir önceki seviyede bir çözüm bulunmuşsa bile yine devam edecektir. Böylece derinine arama algoritması bu tür sonsuz veya çok derin ağaçlarda sonsuz döngüye girer ve asla bir çözüme ulaşamaz çözüm bir önceki seviyede olsa bile. Bir diğer durum ise algoritma bir çözüm bulabilir ama bu optimal çözüm olmayabilir. Yani bir üst seviyedeki çözüm ile en son bulunan çözüm arasında zaman, bellek bakımından farklar olacaktır. Derine arama yukarıdaki kriterlere göre tam değildir. Reküratif çalışan bir fonksiyon olarak yazılabilir.

#### Algoritmanın adımları:

- Başlangıç düğümü seçilir ve ziyaret edilir.
- Başlangıç düğümünün ziyaret edilmemiş komşularından bir tanesi seçilir ve ziyaret edilir.
- Bu ziyaret edilen komşunun da komşularında gidilir ve ve ziyaret edilmeyen bir komşusu seçilerek ziyaret edilir.
- Bu şekilde sürekli derine inilir, en son derinlikte ziyaret edilmeyen düğüm kalmayınca tekrar yukarı çıkarılır ve işlemler aynı şekilde devam edilir.

Özet olarak; verilen tüm düğümler gezilecek şekilde, sırayla ziyaret edilir, gezilmemiş bir komşusu var ise ona gidilir ve daha sonra sıradaki kendi komşusundan önce komşusunun komşusuna gidilir böyle devam eder.

#### 2.2.1.4.Derinlik sınırlı Arama

Bu algoritma esas olarak derin öncelikli arama (depth first search DFS) ile aynı çalışmaktadır ancak tek farkı arama işlemi sırasında özellikle dairelere (cycles) takılma ihtimaline karşı sınır önlemi alınmış olmasıdır.

Kod yapısı şu şekildedir.

```

1  DLS(bakilan,hedef,azamiDerinlik)
2      {
3          if(bakilan==hedef)
4              {
5                  return bakilan
6              }
7          yigin_koy(bakilan);
8
9          while(yigin_doluyken)
10         {
11             gecici = yigin_al();
12
13             if(gecici.derinlik < azamiDerinlik)
14                 {
15                     DLS(gecici,hedef,azamiDerinlik);
16                 }
17         }
18     }
```

Şekil 2 Derinlik sınırlı Arama Kod Yapısı

Özyineli fonksiyonda (recursive function) bakılan düğüm hedef olana kadar dolaşma işlemi devam etmektedir. Dolaşma işlemi sırasında klasik derin öncelikli aramalarda kullanılan yığın (stack) kullanılmış ve geçen düğümler geri dönülüp aranmak üzere yığında tutulmuştur.

Şayet aranan düğüm verilen derinlikten daha derin değilse arama işlemi devam etmektedir ancak verilen derinlik geçildiği zaman arama işlemi daha derine gitmemekte ve artık o ana kadar aranmak üzere yığınladığı düğümleri işlemektedir.

### **2.2.1.5.Yinelemeli Derinleşen Arama**

Derinlik öncelikli drama (depth first search) üzerine kurulu olduğu için, literatürde “iterative deepening depth first search (yinelemeli derinleşen, derin öncelikli arama)” olarak da geçmektedir.

Derinlik değerini bir değişkende tutmakta ve bu değeri her adımda artırmaktadır.

Yineleme yapısı (iteration) basit bir döngü (loop) olarak düşünülebilir ve her adımda derinliğin, bir döngü değişkeni (loop variable) gibi düşünülerek derinleştiği kabul edilebilir.

### **2.2.1.6.İki Yönlü Arama**

bir bilgi kaynağı veya veri yapısı üzerinde problemi her adımda iki parçaya bölgerek yapılan arama algoritmasının ismidir. Bu anlamda bazı kaynaklarda bölgerek arama olarak da geçmektedir.

Aramaya başlangıç durumundan ileri doğru ve sondan geriye doğru arama yapar belirlenen bir durumda ikisi karşılaşana kadar devam eder. Baştaki arama ve sondaki arama her birisi toplam yolun yarısını almış olur.

Algoritmanın şu şekildedir:

- Problemde aranacak uzayın tam orta noktasına bak
- Şayet aranan değer bulunduysa bit
- Şayet bakılan değer aranan değerden büyükse arama işlemini problem uzayının küçük elemanlarında devam ettir.
- Şayet bakılan değer arana değerden küçükse arama işlemini problem uzayının büyük elemanlarında devam ettir.
- Şayet bakılan aralık 1 veya daha küçükse aranan değer bulunamadı olarak bitir.

Problem her seferinde aranan sayıya göre ortadan ikiye bölünmektedir. Bu anlamda problemin  $\log_2(n)$  adımda çözülmesi beklenir. Burada logaritmik fonksiyonların üssel fonksiyonların tersi olduğu ve her adımda problemi iki parçaya böldüğümüzü hatırlayınız.

Arama algoritmasının bir dizi üzerinde başarılı çalışması için dizinin sıralı olması gereklidir. Yukarıdaki koddan anlaşılacağı üzere ortadaki elemana bakıldığında, aranan sayı ya bakılan sayıdan büyük ya da küçüktür. Dolayısıyla algoritmamız, ya bakılan sayıdan küçük olan sayılar arasında arama yapacak ya da büyük olan sayılar arasında arama yapacaktır.

### **2.2.2. Bilgili Arama Yöntemleri**

Arama işleminin bilerek yapılması, algoritmanın probleme ait bazı özellikleri bünyesinde barındırması ve dolayısıyla arama algoritmasının problem bazlı değişiklik göstermesi demektir.

#### **2.2.2.1.Aç Gözlu Arama**

Algoritma üretme yöntemlerinden birisi olan açgözlu yaklaşımına göre mümkün olan ve sonuca en yakın olan seçim yapılır. Yani basitçe bir seçim yapılması gerekiğinde sonuca en çok yaklaştırılacak olan seçimin yapılmasını önerir. Ancak mâmum olduğu üzere bu seçim her zaman için en iyi seçim değildir.

Örneğin para üzeri verilmesi (coin exchange problem) için açgözlu yaklaşımının kullanılmasını düşünelim. Bu problemde bir satıcı kendisinden alışveriş yapan kişiye para üzeri vermektedir. Ödenmesi gereken miktar bu örnekte 24 olsun ve para birimlerimiz 20, 19, 5, 1 olsun. (yani para birimi olarak bu para birimleri bulunuyor)

açgözlu yaklaşımına göre satıcı 24'ü tamamlamak için elindeki para birimlerinden sonuca en çok yaklaştırın 20lik birimi seçecektir. Daha sonra geriye kalan boşluğu ( $24-20=4$ ) doldurmak için elindeki tek imkan olan 4 tane 1lik birimle dolduracaktır ve toplamda 5 adet bozuk parayı müşteriye geri verecektir. Oysaki aynı problem bir 19luk bir de 5lik bozuk paralar ile çözülerek 2 bozuk para vermek mümkün olabilirdi.

Hedef düğüme en yakın olduğu tahmin edilen düğümü genişletir. Düğümleri  $f(x)=h(x)$  baz alarak açar. Yani toplam maliyet sezgisel(tahmini)'dir. Öncelikli kuyruk veri yapısı uygulanır.

Kod yapısı şu şekildedir:

```

1 def _greedy(self):
2     open_list = [ [self.startState] ] # bu yollardan olusan bir array (array arrayi)
3     # her bir yol icinde koordinat barindiran bir array
4     closed_list = [] # gezilmis yollar
5
6     while len(open_list) > 0: # open list icinde yol bulunmamasi
7         # cozumun olmadigina isaret
8         ##### secilecek yollar listesi icinde en iyi yolu sec
9         curr_path = None
10        curr_path_heuristic = MAXVAL
11        for path in open_list:
12            # pathlerin arasinda son state i en iyi olani al
13            h_n = Maze.manhattanHeuristic(path[-1], self.goalState)
14            # yukarda tanimladigimiz manhattan heuristic
15            if h_n < curr_path_heuristic:
16                curr_path_heuristic = h_n
17                curr_path = path
18        ##### secilecek yollar icinde en iyisi curr_path
19
20        ##### o anki yolin sonundaki koordinat bizim hedefimiz mi?
21        if self.isGoal(curr_path[-1]):
22            return curr_path
23        #####
24
25        # curr_path'in son koordinatindan gidilebilecek yollarin listesi
26        succ = self.getSuccessors(curr_path[-1])
27        ##### Bu kisimi anlamak çok guç fakat onemli. Asama asama ilerleyelim
28        n = [] # aday yollar icin bir bos degisken olusturalim
29        for s in succ:
30            if not s in closed_list: # eger bu koordinata gelmemissek
31                n[:] = curr_path # mevcut yolu bos degiskene ata
32                n.append(s) # bos degiskene secilmis koordinati ekle
33                if n in open_list:
34                    pass # zaten mevcut
35                else:
36                    open_list.append(n) # secilebilir yollara ekle
37                    n = [] # bos degiskeni sifirla
38        ##### secilecek yollar listesine yeni yollar eklendi.
39
40        # gittigimiz bir yeri tekrar kontrol etmemek icin ziyaret ettigimiz koordinatlari
41        # gezilmis yollar listesine ekleyelim
42        closed_list.append(curr_path[-1])
43        # kontrol ettigimiz bir yolu tekrar kontrol etmemek icin secilecek yollandan kaldiralim
44        open_list.remove(curr_path)
45    else: # while else yapisi secilecek yollar listesinde eleman kalmasa
46        return None # cozum mevcut degil
47
48 from pprint import pprint
49 sol = m.solve('greedy')
50 print "Yol: ",sol," Yol uzunlugu: ", len(sol) -1
51 pprint(m.visualizeSolution(sol))
52
53 """
54 Goal state: [2, 6]
55 Yol: [[3, 0], [3, 1], [4, 1], [4, 2], [4, 3], [4, 4], [4, 5], [4, 6], [4, 7], [4, 8], [4, 9], [4, 10], [3, 10], [2, 1
0], [1, 10], [0, 10], [0, 9], [0, 8], [0, 7], [0, 6], [0, 5],[0, 4], [1, 4], [2, 4], [2, 5], [2, 6]] Yol uzunlugu: 25
56 [[0, 0, 0, 0, 21, 20, 19, 18, 17, 16, 15],
57 [0, 1, 0, 0, 22, 1, 1, 1, 1, 1, 14],
58 [0, 1, 1, 0, 23, 24, 25, 0, 0, 1, 13],
59 [0, 1, 1, 1, 1, 1, 1, 1, 1, 12],
60 [0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]]
61 """

```

Şekil 3 Açı Gözülü Arama Kod Yapısı

### 2.2.2.2.A\* Arama

Maliyeti fazla olan yolların genişletilmesini engeller, sadece en faydalı olabilecek yolları genişletir.

Başlangıç s düğümünden x düğümüne kadar olan yolu  $g(x)$  ile x düğümünden ise f hedef düğümüne olan yol değerini  $h(x)$  ile ifade edersek bu algoritma için aşağıdaki bağıntı elde edilir.

$$f(x) = g(x) + h(x),$$

$g(x)$  x durumunun gerçek olan o anki değeri,

$h(x)$  ise x düğümünden çözüme olan gidişlerin sezgisel değeridir.

$f(x)$  ise toplam n yolun içinden hedefe olan toplam tahmini maliyettir. Artan sırada öncelikli kuyruk veri yapısı uygulanır.

Kod yapısı şu şekildedir:

```

1 def _astar(self):
2     open_list = [ [self.startState] ] # bu yollardan olusan bir array
3     # her bir yol icinde koordinat barindiran bir array
4     # fizibil yollar
5     closed_list = [] # gezilmis yollar
6
7     while len(open_list) > 0: # open list icinde bir path var
8         # get best in open list
9         curr_path = None
10        curr_path_f_n = MAXVAL
11        for path in open_list:
12            # pathlerin arasinda son state i en iyi olani al
13            g_n = len(path) # bu problemde guzel olan grid uzerinde oldugumuz icin
14            # butun hareketlerin bize cikardigi g degeri o yolin uzunluguna esit
15            # graph uzerinde bu hareketi yapamazdik
16            h_n = Maze.manhattanHeuristic(path[-1], self.goalState)
17            f_n = g_n + h_n
18            if f_n < curr_path_f_n:
19                curr_path_f_n = f_n
20                curr_path = path
21        if self.isGoal(curr_path[-1]):
22            return curr_path
23
24        # current path in son stateinden gidilebilecek
25        succ = self.getSuccessors(curr_path[-1])
26        # gelen durumlari mevcut duruma ekleyip mumkun olan gidilebilecek yer sayisi kadar yol olustur
27        n = []
28        for s in succ:
29            if not s in closed_list:
30                n[:] = curr_path
31                n.append(s)
32                if s in open_list:
33                    pass # zaten mevcut
34                else:
35                    open_list.append(n)
36                    n= []
37                closed_list.append(curr_path[-1])
38                open_list.remove(curr_path)
39            else:
40                return None
41
42 from pprint import pprint
43 sol = m.solve('astar')
44 print "Yol: ",sol," Yol uzunlugu: ", len(sol) -1
45 pprint(m.visualizeSolution(sol))
46
47 """
48 Goal state: [2, 6]
49 Yol: [[[3, 0], [2, 0], [1, 0], [0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [1, 4], [2, 4], [2, 5], [2, 6]] Yol uzunlugu: 11
50 [[3, 4, 5, 6, 7, 0, 0, 0, 0, 0, 0, 0],
51 [2, 1, 0, 0, 8, 1, 1, 1, 1, 1, 0],
52 [1, 1, 1, 0, 9, 10, 11, 0, 0, 1, 0],
53 [0, 0, 1, 1, 1, 1, 1, 1, 1, 0],
54 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
55 """

```

Şekil 4 A Star Kod Yapısı

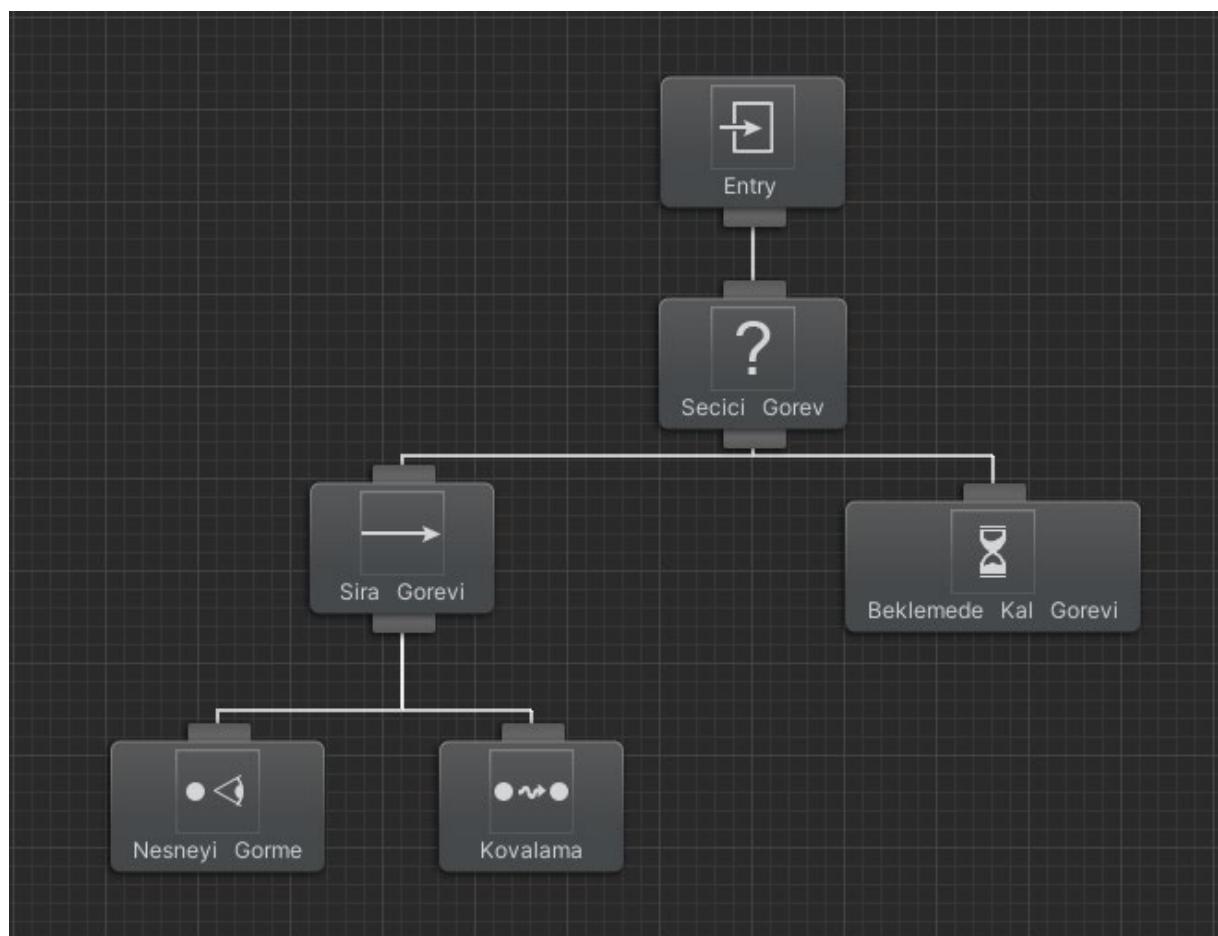
### **3. DAVRANIŞ AĞAÇLARI UYGULAMALARI**

Davranış ağaçları yukarıdan aşağı ve soldan sağa doğru öncelikli çalışır.

#### **3.1. Operatörler**

##### **3.1.1. Ve operatörü - Sıra görevi**

Sıra görevi \ve\ işlemine benzer. Eğer alt görevlerinden biri hata döndürür ise Sıra görevi de hata döndürür ve karar ağacında bulunan bir sonraki görevde geçilir. Bir alt görev başarılı döndürürse sırasıyla ve varsa sonraki alt görevler hata döndürenee kadar çalışmaya devam eder. Tüm alt görevler başarılı döndürürse Sıra görevi de başarılı döndürür. Örnek üzerinden inceleyelim.



Şekil 5 Sıra Görevi Davranış Ağacı Örnek Şeması

Sıra görevi aktif olduğunda bir alt görevleri çalıştırılmaya başlar. Öncelik soldaki görevin olduğu için Nesneyi Görme çalışır. Eğer görev başarısız olursa bir sonraki görev olan Kovalama çalışmaz ve Sıra Görevi hata döndürür. Aynı katmandaki Beklemede Kal Görevi çalışmaya başlar. Eğer Nesneyi Görme başarılı olursa aynı katmanda bulunan Kovalama adlı görev çalışır. Eğer Kovalama başarısız olursa Sıra Görevi hata döndürür ve aynı katmandaki Beklemede Kal Görevi çalışmaya başlar. Eğer Kovalama başarılı olursa Sıra Görevi de başarı döndürür ve aynı katmandaki Beklemede Kal Görevi çalışmaya başlar.

Sıra görevinin kod yapısı şu şekildedir.

```

1  public class Sira_Gorevi : Composite
2  {
3      // Çalışan veya çalıştırılmak üzere olan alt nesnenin (child) indeksi.
4      private int currentChildIndex = 0;
5      // Son çalışan alt nesnenin (child) görev durumu.
6      private TaskStatus executionStatus = TaskStatus.Inactive;
7
8      public override int CurrentChildIndex()
9      {
10         return currentChildIndex;
11     }
12
13     public override bool CanExecute()
14     {
15         // Alt nesneler başarısız olmadıkça ve henüz çalıştırılmamış alt nesneler varsa her karede çalıştırıma devam edebiliriz.
16         return currentChildIndex < children.Count && executionStatus != TaskStatus.Failure;
17     }
18
19     public override void OnChildExecuted(TaskStatus childStatus)
20     {
21         // Bir alt öğe çalıştırıldıktan sonra alt indeksi artırırız ve yürütme durumunu güncelleriz.
22         currentChildIndex++;
23         executionStatus = childStatus;
24     }
25
26     public override void OnConditionalAbort(int childIndex)
27     {
28         // Geçerli alt indeksi iptal etmeye neden olan indekse atama yaparız.
29         currentChildIndex = childIndex;
30         executionStatus = TaskStatus.Inactive;
31     }
32
33     public override void OnEnd()
34     {
35         //Bütün alt nesneler çalıştı dolayısıyla değişkenleri başlangıç değerlerine döndürmeliyiz .
36         executionStatus = TaskStatus.Inactive;
37         currentChildIndex = 0;
38     }
39 }

```

Şekil 6 Sıra Görevi Kod Yapısı

### 3.1.2. Veya operatörü - Seçim görevi

Seçim görevi \veya\ işlemine benzer. Alt görevlerinden biri başarıya döndüğünde başarı döndürür. Bir alt görev başarısızlık döndürürse, sıralı olarak bir sonraki görevi çalıştırır. Hiçbir alt görev başarı döndürmezse başarısızlık döndürür.

Seçim görevinin kod yapısı şu şekildedir.

```

1  public class Secim_Gorevi : Composite
2  {
3      // Çalışan veya çalıştırılmak üzere olan alt nesnenin (child) indeksi.
4      private int currentChildIndex = 0;
5      // Son çalışan alt nesnenin görev durumu.
6      private TaskStatus executionStatus = TaskStatus.Inactive;
7
8      public override int CurrentChildIndex()
9      {
10         return currentChildIndex;
11     }
12
13     public override bool CanExecute()
14     {
15         // Alt nesneler başarısız olmadıkça ve henüz çalıştırılmamış alt nesneler varsa her karede çalıştırıma devam edebiliriz.
16         return currentChildIndex < children.Count && executionStatus != TaskStatus.Success;
17     }
18
19     public override void OnChildExecuted(TaskStatus childStatus)
20     {
21         // Bir alt öğe çalıştırıldıktan sonra alt indeksi artırırız ve yürütme durumunu güncelleriz.
22         currentChildIndex++;
23         executionStatus = childStatus;
24     }
25
26     public override void OnConditionalAbort(int childIndex)
27     {
28         // Geçerli alt indeksi iptal etmeye neden olan indekse atama yaparız.
29         currentChildIndex = childIndex;
30         executionStatus = TaskStatus.Inactive;
31     }
32
33     public override void OnEnd()
34     {
35         // Bütün alt nesneler çalıştı dolayısıyla değişkenleri başlangıç değerlerine döndürmeliyiz .
36         executionStatus = TaskStatus.Inactive;
37         currentChildIndex = 0;
38     }
39 }

```

Şekil 7 Seçim Görevi Kod Yapısı

### 3.1.3. Paralel görev

Sıra görevine benzer şekilde, paralel görev, alt görev başarısız olana kadar her alt görevi çalıştırır. Aradaki fark, paralel görevin tüm alt görevlerini aynı anda yürütmesi ile her görevi aynı anda yürütmesi. Dizi sınıfı gibi, paralel görev de tüm alt görevleri geri döndüğünde başarı döndürür. Bir görev hata döndürürse, paralel görev tüm alt görevleri sonlandırır ve hata döndürür.

```

1  public class Parallel_Gorev : Composite
2  {
3      // Çalışan veya çalıştırılmak üzere olan alt nesnenin (child) indeksi.
4      private int currentChildIndex;
5      // Her alt nesnenin (child) görev durumu.
6      private TaskStatus[] executionStatus;
7
8      public override void OnAwake()
9      {
10         // Tüm alt görevlerin yürütme durumunu tutacak yeni bir görev durumu dizisi oluşturma
11         executionStatus = new TaskStatus[children.Count];
12     }
13
14     public override void OnChildStarted(int childIndex)
15     {
16         // Alt nesnelerden biri çalışmaya başladığında için alt indeksi artırmalı ve
17         // o alt nesnenin geçerli görev durumunu çalışacak şekilde ayarlamalıyız.
18         currentChildIndex++;
19         executionStatus[childIndex] = TaskStatus.Running;
20     }
21
22     public override bool CanRunParallelChildren()
23     {
24         // Bu görevi paralel nesneler yürütülebilir.
25         return true;
26     }
27
28     public override int CurrentChildIndex()
29     {
30         return currentChildIndex;
31     }
32
33     public override bool CanExecute()
34     {
35         // Henüz çalışmaya başlamamış alt nesne varsa yürütülebilir.
36         return currentChildIndex < children.Count;
37     }
38
39     public override void OnChildExecuted(int childIndex, TaskStatus childStatus)
40     {
41         // Alt nesnelerden birinin yürütmesi durduğu için görev durumunu ayarlamalıyız.
42         executionStatus[childIndex] = childStatus;
43     }
44
45     public override TaskStatus OverrideStatus(TaskStatus status)
46     {
47         // Tüm alt nesnelerin yürütmesini bitirdiğini varsayıp her alt nesnenin
48         // yürütme durumunu gözden geçirmeli ve şu anda herhangi bir görevin çalışıp
49         // çalışmadığını kontrol etmeliyiz. Bir görev hala çalışıysa, tüm alt nesnelerin
50         // yürütülmesi tamamlanmaz ve paralel görev çalışanın görev durumunu döndürmeye devam etmelidir.
51         // Karar ağacı tüm alt nesnelerin görevlerini durduracaktır. Hiçbir alt görev çalışmıyorsa veya
52         // başarısız olduysa, paralel görev başarılı olur.
53         bool childrenComplete = true;
54         for (int i = 0; i < executionStatus.Length; ++i) {
55             if (executionStatus[i] == TaskStatus.Running) {
56                 childrenComplete = false;
57             } else if (executionStatus[i] == TaskStatus.Failure) {
58                 return TaskStatus.Failure;
59             }
60         }
61         return (childrenComplete ? TaskStatus.Success : TaskStatus.Running);
62     }
63
64     public override void OnConditionalAbort(int childIndex)
65     {
66         // Baştan başla
67         currentChildIndex = 0;
68         for (int i = 0; i < executionStatus.Length; ++i) {
69             executionStatus[i] = TaskStatus.Inactive;
70         }
71     }
72
73     public override void OnEnd()
74     {
75         // Çalışma durumunu resetle ve başlangıç değerine geri dön
76         for (int i = 0; i < executionStatus.Length; ++i) {
77             executionStatus[i] = TaskStatus.Inactive;
78         }
79         currentChildIndex = 0;
80     }
81 }

```

Şekil 8 Paralel Görev Kod Yapısı

### 3.2.Dekaratorler

#### 3.2.1. Tekrarlayıcı

Tekrarlayıcı , alt görev belirtilen sayıda çalıştırılana kadar alt görevinin yürütülmesini tekrarlar.

Alt görev başarısız olsa bile alt görevi sürdürmeye devam etme seçeneği var.

Kod yapısı şu şekildedir.

```

1  public class Tekrarlayıcı : Decorator
2  {
3      public SharedInt count = 1;
4      public SharedBool repeatForever;
5      public SharedBool endOnFailure;
6
7      // Alt görevin çalıştırılma sayısı.
8      private int executionCount = 0;
9      // Alt görevin çalışması bittikten sonraki durumu
10     private TaskStatus executionStatus = TaskStatus.Inactive;
11
12    public override bool CanExecute()
13    {
14        // Sayıma ulaşana veya alt görev başarısızlıkla sonuclanınca kadar yürütülmeye devam edip bir hata allığımızda durdurmalıyız.
15        return (repeatForever.Value || executionCount < count.Value) && (!endOnFailure.Value || (endOnFailure.Value && executionStatus != TaskStatus.Failure));
16    }
17
18    public override void OnChildExecuted(TaskStatus childStatus)
19    {
20        // Alt görev yürütülmeye tamamlandı. Yürütme sayısını artırıp ve yürütme durumunu güncellememeliz.
21        executionCount++;
22        executionStatus = childStatus;
23    }
24
25    public override void OnEnd()
26    {
27        // Değişkenleri başlangıç değerlerine sıfırlarız.
28        executionCount = 0;
29        executionStatus = TaskStatus.Inactive;
30    }
31
32    public override void OnReset()
33    {
34        // Public tanımlanmış fonksiyonları orijinal değerlerine sıfırlarız.
35        count = 0;
36        endOnFailure = true;
37    }
38 }
```

Şekil 9 Tekrarlayıcı Kod Yapısı

#### 3.2.2. Çevirici

Çevirici görevi, yürütme işlemi tamamlandıktan sonra alt görevin dönüş değerini ters çevirir.

Alt görev başarılı döndürürse sürücü görevi başarısız olur. Alt görev başarısızlık döndürürse sürücü görevi başarıya döner.

Kod yapısı şu şekildedir.

```

1  public class Çevirici : Decorator
2  {
3      // Alt nesnenin çalışmasının bittikten sonraki durumu.
4      private TaskStatus executionStatus = TaskStatus.Inactive;
5
6      public override bool CanExecute()
7      {
8          // Alt görev başarılı veya başarısız olana kadar yürütmeye devam ederiz.
9          return executionStatus == TaskStatus.Inactive || executionStatus == TaskStatus.Running;
10     }
11
12    public override void OnChildExecuted(TaskStatus childStatus)
13    {
14        // Bir alt görev çalışmasını bitirdikten sonra yürütme durumunu güncelliyoruz.
15        executionStatus = childStatus;
16    }
17
18    public override TaskStatus Decorate(TaskStatus status)
19    {
20        // Invert the task status.
21        if (status == TaskStatus.Success) {
22            return TaskStatus.Failure;
23        } else if (status == TaskStatus.Failure) {
24            return TaskStatus.Success;
25        }
26        return status;
27    }
28
29    public override void OnEnd()
30    {
31        // Yürütme durumunu başlangıç değerlerine sıfırlarız.
32        executionStatus = TaskStatus.Inactive;
33    }
34}

```

Şekil 10 Çevirici Kod Yapısı

### 3.2.3. Hata Döndür

Hata döndür görevi, alt görevin çalıştığı durumlar dışında her zaman başarısızlık döndürür.

Kod yapısı şu şekildedir.

```

1  public class Hata_Dondur : Decorator
2  {
3      // Alt nesnenin çalışmasının bittikten sonraki durumu.
4      private TaskStatus executionStatus = TaskStatus.Inactive;
5
6      public override bool CanExecute()
7      {
8          // Alt görev başarılı veya başarısız olana kadar yürütmeye devam ederiz.
9
10         return executionStatus == TaskStatus.Inactive || executionStatus == TaskStatus.Running;
11     }
12
13     public override void OnChildExecuted(TaskStatus childStatus)
14     {
15         // Bir alt görev çalışmasını bitirdikten sonra yürütme durumunu güncellemeliyiz.
16         executionStatus = childStatus;
17     }
18
19     public override TaskStatus Decorate(TaskStatus status)
20     {
21         // Alt görev başarılı olsa bile hata döndür.
22         if (status == TaskStatus.Success) {
23             return TaskStatus.Failure;
24         }
25         return status;
26     }
27
28     public override void OnEnd()
29     {
30         // Yürütme durumunu başlangıç değerlerine sıfırlarız.
31         executionStatus = TaskStatus.Inactive;
32     }
33 }

```

Şekil 11 Hata Döndür Kod Yapısı

### 3.2.4. Başarı Döndür

Başarı döndür görevi, alt görevin çalıştığı durumlar dışında her zaman başarı döndürür.

Kod yapısı şu şekildedir.

```

1  public class Basari_Dondur : Decorator
2  {
3      // Alt nesnenin çalışmasının bittikten sonraki durumu.
4      private TaskStatus executionStatus = TaskStatus.Inactive;
5
6      public override bool CanExecute()
7      {
8          // Alt görev başarılı veya başarısız olana kadar yürütmeye devam ederiz.
9          return executionStatus == TaskStatus.Inactive || executionStatus == TaskStatus.Running;
10     }
11
12     public override void OnChildExecuted(TaskStatus childStatus)
13     {
14         // Bir alt görev çalışmasını bitirdikten sonra yürütme durumunu güncellememeliyiz.
15         executionStatus = childStatus;
16     }
17
18     public override TaskStatus Decorate(TaskStatus status)
19     {
20         // Alt görev başarısız olsa bile başarı döndür.
21         if (status == TaskStatus.Failure) {
22             return TaskStatus.Success;
23         }
24         return status;
25     }
26
27     public override void OnEnd()
28     {
29         // Yürütme durumunu başlangıç değerlerine sıfırlarız.
30         executionStatus = TaskStatus.Inactive;
31     }
32 }

```

Şekil 12 Başarı Döndür Kod Yapısı

### 3.2.5. Başarısızlığa Kadar

Başarısızlığa kadar görevi, alt görev başarısızlık döndürenee kadar alt görevini yürütmeye devam eder.

Kod yapısı şu şekildedir.

```
1 public class Basarisizliga_Kadar : Decorator
2 {
3     // Alt nesnenin çalışmasının bittikten sonraki durumu.
4     private TaskStatus executionStatus = TaskStatus.Inactive;
5
6     public override bool CanExecute()
7     {
8         // Alt görev hata döndürenee kadar çalışmaya devam eder.
9         return executionStatus == TaskStatus.Success || executionStatus == TaskStatus.Inactive;
10    }
11
12    public override void OnChildExecuted(TaskStatus childStatus)
13    {
14        // Bir alt görev çalışmasını bitirdikten sonra yürütme durumunu güncellememeliyiz.
15        executionStatus = childStatus;
16    }
17
18    public override void OnEnd()
19    {
20        // Yürütme durumunu başlangıç değerlerine sıfırlarız.
21        executionStatus = TaskStatus.Inactive;
22    }
23}
```

Şekil 13 Başarısızlığa Kadar Kod Yapısı

### 3.2.6. Başarıya Kadar

Başarıya kadar görevi, alt görev başarıya dönene kadar alt görevini yürütmeye devam eder.

Kod yapısı şu şekildedir.

```
1 public class Basariya_Kadar : Decorator
2 {
3     // Alt nesnenin çalışmasının bittikten sonraki durumu.
4     private TaskStatus executionStatus = TaskStatus.Inactive;
5
6     public override bool CanExecute()
7     {
8         // Alt görev başarı döndürene kadar çalışmaya devam eder.
9         return executionStatus == TaskStatus.Failure || executionStatus == TaskStatus.Inactive;
10    }
11
12    public override void OnChildExecuted(TaskStatus childStatus)
13    {
14        // Bir alt görev çalışmasını bitirdikten sonra yürütme durumunu güncelleliyoruz.
15        executionStatus = childStatus;
16    }
17
18    public override void OnEnd()
19    {
20        // Yürütme durumunu başlangıç değerlerine sıfırlarız.
21        executionStatus = TaskStatus.Inactive;
22    }
23}
```

Şekil 14 Başarıya Kadar Kod Yapısı

## 4. UYGULAMALAR

Visual studio üzerinde C# programlama dili kullanılarak geliştirilmiştir. Unity NavMesh’i kullanarak oyumlara uyarlanmış görevler geliştirdim. Bu geliştirme sürecinde önceden test edilmiş ve kütüphaneye eklenmiş olan bazı fonksiyon ve sınıfları ve bu sınıflara ait değişken ve fonksiyonları kullandım. Bu görevleri karar ağaçlarında mantıklı kombinasyonlar ile bir araya getirip oyun içinde yapay zeka yada bilgisayar diye adlandırılan botları oluşturmayı planlıyorum.

### 4.1. Nesneyi Gör Görevi

Ajanın açısı ve alanı daha önce belirlenmiş olan menzilde bir nesne varsa görür.

Kod yapısı şu şekildedir.

```

1  public class Nesneyi_Gor : Conditional
2  {
3      // Hedef nesne
4      public SharedGameObject targetObject;
5      // Ajanın görüş açısı (Derece Cinsinden)
6      public SharedFloat fieldOfViewAngle = 90;
7      // Ajanın görüş mesafesi
8      public SharedFloat viewDistance = 1000;
9      // Görüş alanına giren nesne
10     public SharedGameObject returnedObject;
11
12    public override TaskStatus OnUpdate()
13    {
14        returnedObject.Value = WithinSight(targetObject.Value, fieldOfViewAngle.Value, viewDistance.Value);
15        if (returnedObject.Value != null) {
16            // Bir nesne bulunursa başarı döndür
17            return TaskStatus.Success;
18        }
19        // Görüş alanında bir nesne yok dolayısıyla hata döndür
20        return TaskStatus.Failure;
21    }
22
23    private GameObject WithinSight(GameObject targetObject, float fieldOfViewAngle, float viewDistance)
24    {
25        if (targetObject == null) {
26            return null;
27        }
28
29        var direction = targetObject.transform.position - transform.position;
30        direction.y = 0;
31        var angle = Vector3.Angle(direction, transform.forward);
32        if (direction.magnitude < viewDistance && angle < fieldOfViewAngle * 0.5f) {
33            // İabet ajanının mevcut ajanın görüşü dahilinde olması gerekiyor
34            if (LineOfSight(targetObject))
35                return targetObject; // görüş alanında olan nesneyi döndür
36        }
37    }
38    return null;
39 }
40
41    private bool LineOfSight(GameObject targetObject)
42    {
43        RaycastHit hit;
44        if (Physics.Linecast(transform.position, targetObject.transform.position, out hit)) {
45            if (hit.transform.IsChildOf(targetObject.transform) || targetObject.transform.IsChildOf(hit.transform)) {
46                return true;
47            }
48        }
49        return false;
50    }
51 }
```

Şekil 15 Nesneyi Gör Kod Yapısı

## 4.2. Nesneyi Duy Görevi

Mevcut nesnelerin işitme menzilinde herhangi bir nesne olup olmadığını kontrol eder.

Kod yapısı şu şekildedir.

```

1  public class Nesneyi_Duy : Conditional
2  {
3      // Fizik 2D motoru kullanılmalı mı
4      public bool usePhysics2D;
5      // Aradığımız nesne
6      public SharedGameObject targetObject;
7      // Aradığımız nesneler
8      public SharedGameObjectList targetObjects;
9      // Aradığımız nesnenin etiketi
10     public SharedString targetTag;
11     // Aradığımız nesnelerin maskelemesi
12     public LayerMask objectLayerMask;
13     // Duyma mesafesi
14     public SharedFloat hearingRadius = 50;
15     // Ses kaynağı ne kadar uzaksa o kadar az ihtimalle işitilir
16     // Minimum mesafe için atama yapmalıyız
17     public SharedFloat audibilityThreshold = 0.05f;
18     // Pivot konumuna göre
19     public SharedVector3 offset;
20     // Döndürülen objenin duyulması için
21     public SharedGameObject returnedObject;
22
23     // Nesne bulunursa başarı döndür
24     public override TaskStatus OnUpdate()
25     {
26         if (targetObjects.Value != null && targetObjects.Value.Count > 0) { // Grup listesinde nesneler varsa, o listedeki nesneyi arayın
27             GameObject objectFound = null;
28             for (int i = 0; i < targetObjects.Value.Count; ++i) {
29                 float audibility = 0;
30                 GameObject obj;
31                 if (Vector3.Distance(targetObjects.Value[i].transform.position, transform.position) < hearingRadius.Value) {
32                     if ((obj = MovementUtility.WithinHearingRange(transform, offset.Value, audibilityThreshold.Value, targetObjects.Value[i], ref audibility)) != null) {
33                         objectFound = obj;
34                     }
35                 }
36             }
37             returnedObject.Value = objectFound;
38         } else if (targetObject.Value == null) { // Hedef nesne yoksa işitme mesafesinde bir nesne olup olmadığını kontrol et
39             if (usePhysics2D) {
40                 returnedObject.Value = MovementUtility.WithinHearingRange2D(transform, offset.Value, audibilityThreshold.Value, hearingRadius.Value, objectLayerMask);
41             } else {
42                 returnedObject.Value = MovementUtility.WithinHearingRange(transform, offset.Value, audibilityThreshold.Value, hearingRadius.Value, objectLayerMask);
43             }
44         } else {
45             GameObject target;
46             if (!string.IsNullOrEmpty(targetTag.Value)) {
47                 target = GameObject.FindGameObjectWithTag(targetTag.Value);
48             } else {
49                 target = targetObject.Value;
50             }
51             if (Vector3.Distance(target.transform.position, transform.position) < hearingRadius.Value) {
52                 returnedObject.Value = MovementUtility.WithinHearingRange(transform, offset.Value, audibilityThreshold.Value, targetObject.Value);
53             }
54         }
55
56         if (returnedObject.Value != null) {
57             // Bir nesne duyulursa başarı döndür
58             return TaskStatus.Success;
59         }
56         // Bir nesne duyulmazsa hata döndür
60         return TaskStatus.Failure;
61     }
62
63     // Public tanımlı değişkenleri resetle
64     public override void OnReset()
65     {
66         hearingRadius = 50;
67         audibilityThreshold = 0.05f;
68     }
69 }

```

Şekil 16 Nesneyi Duy Kod Yapısı

### 4.3.Nesneyi Yakala Görevi

Unity NavMesh'i kullanarak belirtilen hedefi arar.

Kod yapısı şu şekildedir.

```
1 public class Nesneyi_Yakala : NavMeshMovement
2 {
3     // Aranan nesne
4     public SharedGameObject target;
5     // Hedef yoksa hedefin konumunu kullan
6     public SharedVector3 targetPosition;
7
8     public override void OnStart()
9     {
10         base.OnStart();
11
12         SetDestination(Target());
13     }
14
15     // Hedefi kovalarız ve yakalayınca başarı döndürürüz.
16     // Hedefe ulaşmadığımız sürece yürütmeye devam ederiz.
17     public override TaskStatus OnUpdate()
18     {
19         if (HasArrived())
20             return TaskStatus.Success;
21     }
22
23     SetDestination(Target());
24
25     return TaskStatus.Running;
26 }
27
28 // Hedef yok ise targetPosition nesnesini döndürürüz.
29 private Vector3 Target()
30 {
31     if (target.Value != null)
32         return target.Value.transform.position;
33     }
34     return targetPosition.Value;
35 }
36
37     public override void OnReset()
38     {
39         base.OnReset();
40         target = null;
41         targetPosition = Vector3.zero;
42     }
43 }
```

Şekil 17 Nesneyi Yakala Kod Yapısı

#### 4.4.Nesneyi Takip Et Görevi

Unity NavMesh'i kullanarak belirtilen hedefi takip eder.

Kod yapısı şu şekildedir.

```

1  public class Nesneyi_Takip_Et : NavMeshMovement
2  {
3      // Takip edilen nesne
4      public SharedGameObject target;
5      //Hedef belirtilen mesafeden fazla ise hedefe doğru ilerlemeye başla
6      public SharedFloat moveDistance = 2;
7
8      private Vector3 lastTargetPosition;
9      private bool hasMoved;
10
11     public override void OnStart()
12     {
13         base.OnStart();
14
15         lastTargetPosition = target.Value.transform.position + Vector3.one * (moveDistance.Value + 1);
16         hasMoved = false;
17     }
18
19     // Hedefi takip et. Belirlenen mesafeye ulaş.
20     // Ancak hiçbir zaman hedefe ulaşamayacağı için görev başarıya hiçbir zaman döndürmez.
21     public override TaskStatus OnUpdate()
22     {
23         if (target.Value == null) {
24             return TaskStatus.Failure;
25         }
26
27         // Hedef moveDistance değişkeninden fazla hareket etmişse harekete devam et.
28         var targetPosition = target.Value.transform.position;
29         if ((targetPosition - lastTargetPosition).magnitude >= moveDistance.Value) {
30             SetDestination(targetPosition);
31             lastTargetPosition = targetPosition;
32             hasMoved = true;
33         } else {
34             // Hedef moveDistance değişkeninden az hareket etmişse hareketi bırak.
35             if (hasMoved && (targetPosition - transform.position).magnitude < moveDistance.Value) {
36                 Stop();
37                 hasMoved = false;
38                 lastTargetPosition = targetPosition;
39             }
40         }
41
42         return TaskStatus.Running;
43     }
44
45     public override void OnReset()
46     {
47         base.OnReset();
48         target = null;
49         moveDistance = 2;
50     }
51 }
```

## Şekil 18 Nesneyi Takip Et Kod Yapısı

### 4.5.Nesneyi Ara Görevi

Unity NavMesh'i kullanarak dolaşmayı, işitme menzili içinde ve menzil içindeki görevleri birleştirerek bir hedef aramayı gerçekleştirir.

Kod yapısı şu şekildedir.

```

1  public class Nesneyi_Ara : NavMeshMovement
2  {
3      // Varış noktası ile mevcut konum arasındaki minimum mesafe
4      public SharedFloat minWanderDistance = 20;
5      //Varış noktası ile mevcut konum arasındaki maksimum mesafe
6      public SharedFloat maxWanderDistance = 20;
7      // Ajancın dönen miktarı
8      public SharedFloat wanderRate = 1;
9      // Ajancın her bir hedefte duraklaması gereken minimum süre
10     public SharedFloat minPauseDuration = 0;
11     // Aracının her bir hedefte duraklaması gereken maksimum süre (devre dışı bırakmak için sıfır yaz)
12     public SharedFloat maxPauseDuration = 0;
13     // Maksimum deneme sayısı
14     public SharedInt targetRetries = 1;
15     // Ajancın görüş açısı (derece olarak)
16     public SharedFloat fieldOfViewAngle = 90;
17     // Ajancın görülebileceği mesafe
18     public SharedFloat viewDistance = 30;
19     // Görüntü çizgisi kontrolü yapılırken göz ardı edilecek nesnelerin maskelemesi
20     public LayerMask ignoreLayerMask = 1 << LayerMask.NameToLayer("Ignore Raycast");
21     // Ses duyulursa arama sona ermeli mi kontrolü
22     public SharedBool senseAudio = true;
23     // Ajancın ne kadar uzaya isitebileceğini belirle
24     public SharedFloat hearingRadius = 30;
25     // Pivot konumuna göre raycast uzaklığı
26     public SharedVector3 offset;
27     // Pivot konumuna göre hedef raycast uzaklığı
28     public SharedVector3 targetOffset;
29     // Aradığımız nesnelerin maskelemesi
30     public LayerMask objectLayerMask;
31     // Kontrol
32     public SharedBool useTargetBone;
33     // Kimlik saptama
34     public HumanBodyBones targetBone;
35     // Ses kaynağı ne kadar uzaksa, ajancın onu duyma olasılığı o kadar düşük olur.
36     // Ajancın duyabileceğii minimum duyluallılık seviyesi için atama yap
37     public SharedFloat audibilityThreshold = 0.05f;
38     // Bulunan nesne
39     public SharedGameObject returnedObject;
40
41     private float pauseTime;
42     private float destinationReachTime;
43
44     // Bir nesne görünenye veya duyulana kadar aramaya devam et (senseAudio etkin olmalı)
45     public override TaskStatus OnUpdate()
46     {
47         if (HasArrived())
48         {
49             // Temsilci, yalnızca maksimum duraklatma süresi 0'dan büyükse hedefte duraklamalı
50             if (maxPauseDuration.Value > 0)
51             {
52                 if (destinationReachTime == -1)
53                 {
54                     destinationReachTime = Time.time;
55                     pauseTime = Random.Range(minPauseDuration.Value, maxPauseDuration.Value);
56                 }
57                 if (destinationReachTime + pauseTime <= Time.time)
58                 {
59                     // Yalnızca bir hedef ayarlanmışsa zamanı sıfırla.
60                     if (TrySetTarget())
61                         destinationReachTime = -1;
62                 }
63             }
64
65             // Görüntünde herhangi bir nesne olup olmadığını tespit et
66             returnedObject.Value = MovementUtility.WithinSight(transform, offset.Value, fieldOfViewAngle.Value, viewDistance.Value, objectLayerMask,
67             targetOffset.Value, ignoreLayerMask, useTargetBone.Value, targetBone);
68             // Bir nesne görüldüğse başarı döndür
69             if (returnedObject.Value != null)
70             {
71                 return TaskStatus.Success;
72             }
73             // Herhangi bir nesnenin ses aralığında olup olmadığını algılama (etkinleştirilmişse)
74             if (senseAudio.Value)
75             {
76                 returnedObject.Value = MovementUtility.WithinHearingRange(transform, offset.Value, audibilityThreshold.Value, hearingRadius.Value,
77                 objectLayerMask);
78                 // Bir nesne duyulduysa başarı döndür
79                 if (returnedObject.Value != null)
80                 {
81                     return TaskStatus.Success;
82                 }
83
84             private bool TrySetTarget()
85             {
86                 var direction = transform.forward;
87                 var validDestination = false;
88                 var attempts = targetRetries.Value;
89                 var destination = transform.position;
90                 while (!validDestination && attempts > 0)
91                 {
92                     direction = direction + Random.insideUnitSphere * wanderRate.Value;
93                     destination = transform.position + direction.normalized * Random.Range(minWanderDistance.Value, maxWanderDistance.Value);
94                     validDestination = SamplePosition(destination);
95                     attempts--;
96                 }
97                 if (validDestination)
98                     SetDestination(destination);
99             }
100             return validDestination;
101
102         // Genel değişkenleri sıfırla
103         public override void OnReset()
104         {
105             base.OnReset();
106
107             minWanderDistance = 20;
108             maxWanderDistance = 20;
109             wanderRate = 2;
110             minPauseDuration = 0;
111             maxPauseDuration = 0;
112             targetRetries = 1;
113             fieldOfViewAngle = 90;
114             viewDistance = 30;
115             senseAudio = true;
116             hearingRadius = 30;
117             audibilityThreshold = 0.05f;
118         }
119     }

```

Şekil 19 Nesneyi Ara Kod Yapısı

#### 4.6. Devriye Gez Görevi

Unity NavMesh'i kullanarak belirtilen geçiş noktalarında devriye gezer.

Kod yapısı şu şekildedir.

```

1  public class Devriye_Gez : NavMeshMovement
2  {
3      // Ajan yol noktalarını rastgele devriye gezmeli mi kontrolü
4      public SharedBool randomPatrol = false;
5      // Ajanın bir ara noktaya vardığında duraklaması gereken süre
6      public SharedFloat waypointPauseDuration = 0;
7      // Hareket edilecek olan yol noktaları
8      public SharedGameObjectList waypoints;
9
10     // Geçiş noktaları dizisi içinde ilerlediğimiz geçerli indeks
11     private int waypointIndex;
12     private float waypointReachedTime;
13
14     public override void OnStart()
15     {
16         base.OnStart();
17
18         // Başlangıçta en yakın ara noktaya doğru hareket et.
19         float distance = Mathf.Infinity;
20         float localDistance;
21         for (int i = 0; i < waypoints.Value.Count; ++i) {
22             if ((localDistance = Vector3.Magnitude(transform.position - waypoints.Value[i].transform.position)) < distance) {
23                 distance = localDistance;
24                 waypointIndex = i;
25             }
26         }
27         waypointReachedTime = -1;
28         SetDestination(Target());
29     }
30
31     // Ara nokta dizisinde belirtilen farklı ara noktaların etrafında devriye gez.
32     // Her zaman çalışan bir görev durumu döndürmeliyiz. Yoksa sonsuz çevrime giriyor.
33     public override TaskStatus OnUpdate()
34     {
35         if (waypoints.Value.Count == 0) {
36             return TaskStatus.Failure;
37         }
38         if (HasArrived()) {
39             if (waypointReachedTime == -1) {
40                 waypointReachedTime = Time.time;
41             }
42             // Ara noktaları değiştirmeden önce gerekli süreyi bekle.
43             if (waypointReachedTime + waypointPauseDuration.Value <= Time.time) {
44                 if (randomPatrol.Value) {
45                     if (waypoints.Value.Count == 1) {
46                         waypointIndex = 0;
47                     } else {
48                         // Aynı yol noktasının seçilmesini önde.
49                         var newWaypointIndex = waypointIndex;
50                         while (newWaypointIndex == waypointIndex) {
51                             newWaypointIndex = Random.Range(0, waypoints.Value.Count);
52                         }
53                         waypointIndex = newWaypointIndex;
54                     }
55                 } else {
56                     waypointIndex = (waypointIndex + 1) % waypoints.Value.Count;
57                 }
58                 SetDestination(Target());
59                 waypointReachedTime = -1;
60             }
61         }
62         return TaskStatus.Running;
63     }
64
65     // Geçerli ara nokta indeks konumunu döndür
66     private Vector3 Target()
67     {
68         if (waypointIndex >= waypoints.Value.Count) {
69             return transform.position;
70         }
71         return waypoints.Value[waypointIndex].transform.position;
72     }
73
74     // Genel değişkenleri sıfırla
75     public override void OnReset()
76     {
77         base.OnReset();
78
79         randomPatrol = false;
80         waypointPauseDuration = 0;
81         waypoints = null;
82     }
83 }
```

## Şekil 20 Devriye Gez Kod Yapısı

### 4.7. Beklemede Kal Görevi

Belirli bir süre bekler. Görev, bekleyen iş bitene kadar çalışmaya devam eder. Bekleme süresi geçtikten sonra başarıya döndürür.

Kod yapısı şu şekildedir.

```

1  public class Beklemede_Kal : Action
2  {
3      // Bekleme süresi
4      public SharedFloat waitTime = 1;
5      // Bekleme rastgele olmalı mı kontrolü
6      public SharedBool randomWait = false;
7      //Rastgele bekleme etkinse minimum bekleme süresi
8      public SharedFloat randomWaitMin = 1;
9      // Rastgele bekleme etkinse maksimum bekleme süresi
10     public SharedFloat randomWaitMax = 1;
11
12     // Bekleme süresi
13     private float waitDuration;
14     // Görevin beklemeye başladığı zaman.
15     private float startTime;
16     // Görevin duraklatıldığı zamanı ata.
17     // Böylece duraklatılan zaman bekleme süresine katkıda bulunmaz.
18     private float pauseTime;
19
20     public override void OnStart()
21     {
22         // Remember the start time.
23         startTime = Time.time;
24         if (randomWait.Value) {
25             waitDuration = Random.Range(randomWaitMin.Value, randomWaitMax.Value);
26         } else {
27             waitDuration = waitTime.Value;
28         }
29     }
30
31     public override TaskStatus OnUpdate()
32     {
33         // Görev, başlatıldığından beri waitDuration süresi geçtiyse beklet.
34         if (startTime + waitDuration < Time.time) {
35             return TaskStatus.Success;
36         }
37         // Aksi takdirde beklemeye devam et.
38         return TaskStatus.Running;
39     }
40
41     public override void OnPause(bool paused)
42     {
43         if (paused) {
44             // Davranışın duraklatıldığı zamanı ata
45             pauseTime = Time.time;
46         } else {
47             // Yeni bir başlangıç zamanı bulmak için Time.time ve pauseTime arasındaki farkı ekle.
48             startTime += (Time.time - pauseTime);
49         }
50     }
51
52     public override void OnReset()
53     {
54         // Genel fonksiyonları orijinal değerlerine sıfırla.
55         waitTime = 1;
56         randomWait = false;
57         randomWaitMin = 1;
58         randomWaitMax = 1;
59     }
60 }

```

Şekil 21 Beklemede Kal Kod Yapısı

#### 4.8. Nesneye Saldır Görevi

En yakın hedefe gider ve ajan belirlenen mesafe içinde olur olmaz saldırımıaya başlar

Kod yapısı şu şekildedir.

```
1  public class Nesneye_Saldır : NavMeshTacticalGroup
2      {
3          public override TaskStatus OnUpdate()
4          {
5              var baseStatus = base.OnUpdate();
6              if (baseStatus != TaskStatus.Running || !started) {
7                  return baseStatus;
8              }
9
10             if (MoveToAttackPosition()) {
11                 tacticalAgent.TryAttack();
12             }
13
14             return TaskStatus.Running;
15         }
16     }
```

Şekil 22 Nesneye Saldır Kod Yapısı

#### 4.9.Nesneyi Savun Görevi

Nesneyi savunma yarıçapı içinde korur. Belirli bir yarıçap içindeki bir hedefi arar ve saldırır.

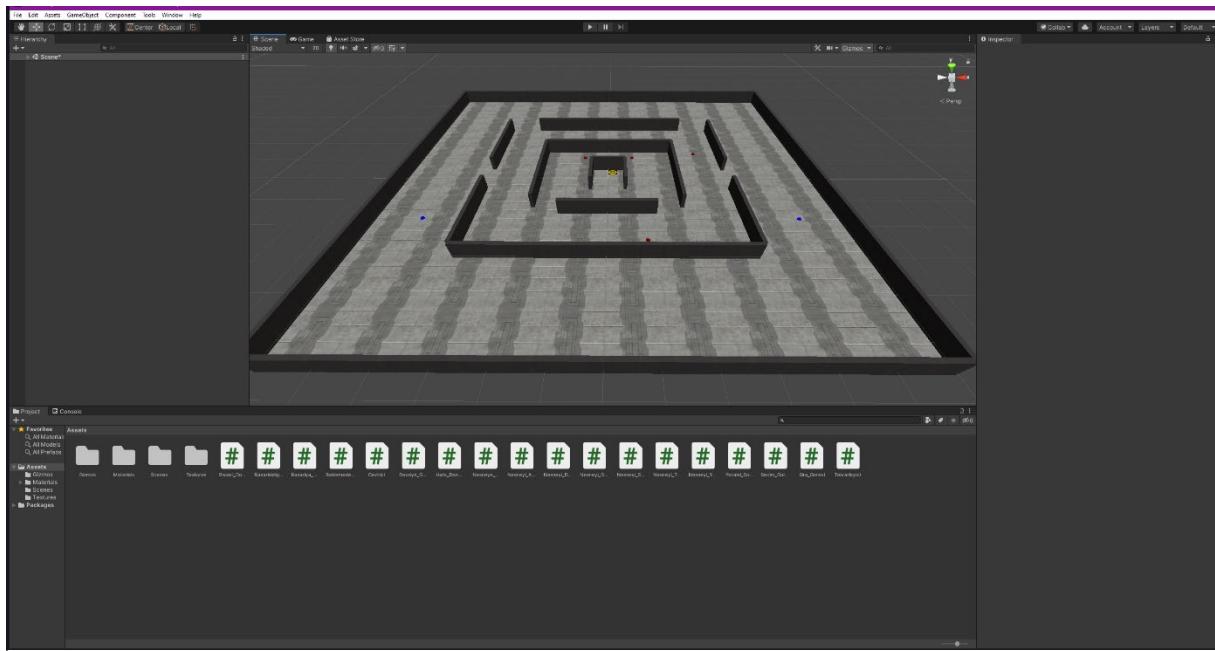
Kod yapısı şu şekildedir.

```

1 public class Nesneyi_Savun : NavMeshTacticalGroup
2 {
3     //Savunacak nesne
4     public SharedGameObject defendObject;
5     // Ajanları konumlandırmak için savunma nesnesinin çevresindeki yarıçap
6     public SharedFloat radius = 3;
7     // Savunma için savunma nesnesinin çevresindeki yarıçap
8     public SharedFloat defendRadius = 10;
9     // Ajanların savunma nesnesinden koruyabileceği maksimum mesafe
10    public SharedFloat maxDistance = 15;
11
12    private float theta;
13
14    protected override void AddAgentToGroup(Behavior agent, int index)
15    {
16        base.AddAgentToGroup(agent, index);
17
18        // 2 * PI = 360 derece
19        theta = 2 * Mathf.PI / agents.Count;
20    }
21
22    protected override int RemoveAgentFromGroup(Behavior agent)
23    {
24        var index = base.RemoveAgentFromGroup(agent);
25
26        // 2 * PI = 360 derece
27        theta = 2 * Mathf.PI / agents.Count;
28
29        return index;
30    }
31
32    public override TaskStatus OnUpdate()
33    {
34        var baseStatus = base.OnUpdate();
35        if (baseStatus != TaskStatus.Running || !started) {
36            return baseStatus;
37        }
38
39        // Ajanın bir hedefi varsa hedefe saldır
40        if (tacticalAgent.TargetTransform != null) {
41            // Hedef savunma nesnesinden çok uzaklaşırsa saldırmayı bırak.
42            if ((transform.position - defendObject.Value.transform.position).magnitude > maxDistance.Value || !tacticalAgent.
43 TargetDamagable.IsAlive()) {
44                tacticalAgent.TargetTransform = null;
45                tacticalAgent.TargetDamagable = null;
46                tacticalAgent.AttackPosition = false;
47            } else {
48                // Hedef mesafe içinde. Ona doğru ilerle.
49                tacticalAgent.AttackPosition = true;
50                if (MoveToAttackPosition()) {
51                    tacticalAgent.TryAttack();
52                }
53            }
54        } else {
55            // Olası hedef dönüşümleri arasında döngü yapıp her bir ajana en yakın dönüşümüngin hangisi olduğunu belirle.
56            for (int i = targetTransforms.Count - 1; i > -1; --i) {
57                // Hedef hayatı olmalı.
58                if (targets[i].IsAlive()) {
59                    // Hedef çok yaklaşırsa saldırmaya başla.
60                    if ((transform.position - targetTransforms[i].position).magnitude < defendRadius.Value) {
61                        tacticalAgent.TargetDamagable = targets[i];
62                        tacticalAgent.TargetTransform = targetTransforms[i];
63                    }
64                } else {
65                    // Hedef artık hayatı değil - listeden kaldır.
66                    targets.RemoveAt(i);
67                    targetTransforms.RemoveAt(i);
68                }
69            }
70
71            // Ajan saldırıyor. Savunma nesnesinin yanına git.
72            if (!tacticalAgent.AttackPosition) {
73                var targetPosition = defendObject.Value.transform.TransformPoint(radius.Value * Mathf.Sin(theta * formationIndex), 0,
radius.Value * Mathf.Cos(theta * formationIndex));
74                tacticalAgent.UpdateRotation(true);
75                tacticalAgent.SetDestination(targetPosition);
76                if (tacticalAgent.HasArrived()) {
77                    // Savunan nesneden uzak dur.
78                    var direction = targetPosition - defendObject.Value.transform.position;
79                    direction.y = 0;
80                    tacticalAgent.RotateTowards(Quaternion.LookRotation(direction));
81                }
82            }
83
84            return TaskStatus.Running;
85        }
86
87        public override void OnReset()
88        {
89            base.OnReset();
90
91            defendObject = null;
92            radius = 5;
93            defendRadius = 10;
94        }
95    }

```

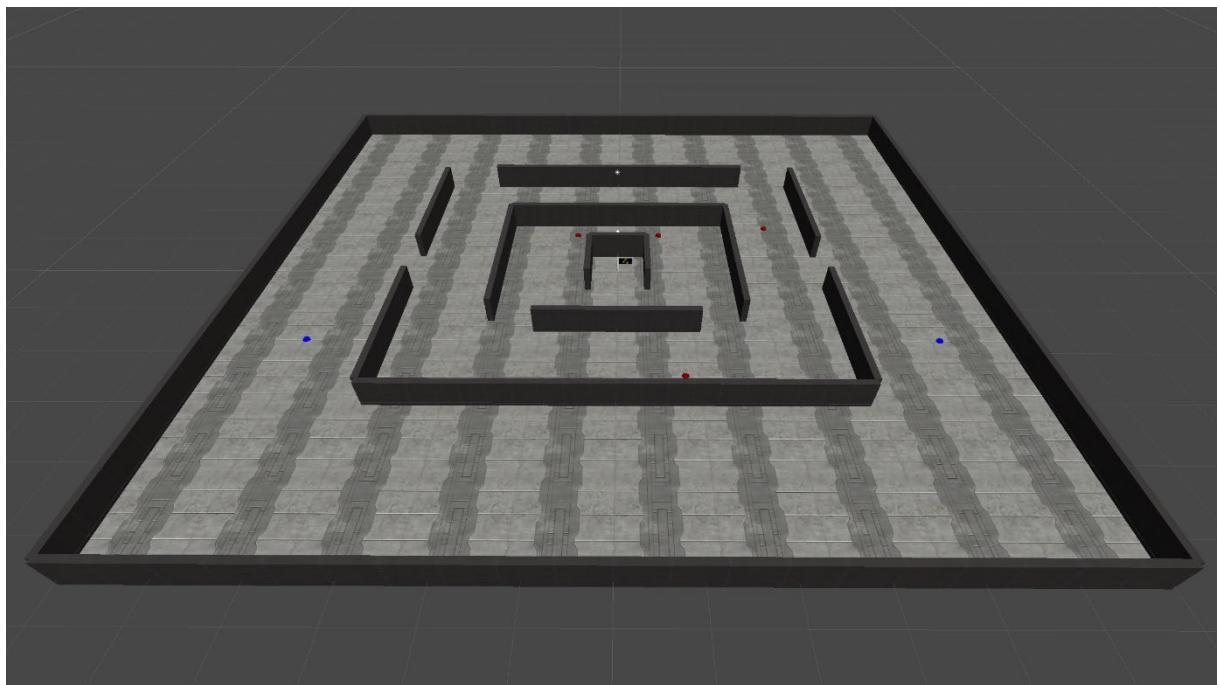
## 5. ÖRNEK SAHNE



Şekil 23 Unity Arayüzü

### 5.1.Sahne Yapısı

Oluşturduğum sahnede bir labirent ve çeşitli görevlere sahip ajanlar bulunmaktadır. Ajanlar devriye gezen, savunan ve saldırın olarak gruplanmaktadır ve sayıları çeşitlilik göstermektedir. Sahnedeki görev ise mavi renkli takımın bayrağa ulaşması ve belirlenen noktaya taşımasıdır. Kırmızı takımın görevi ise bayrağı korumak ve belirlenen alanı düşman takımına karşı savunmaktadır.



Şekil 24 Labirent

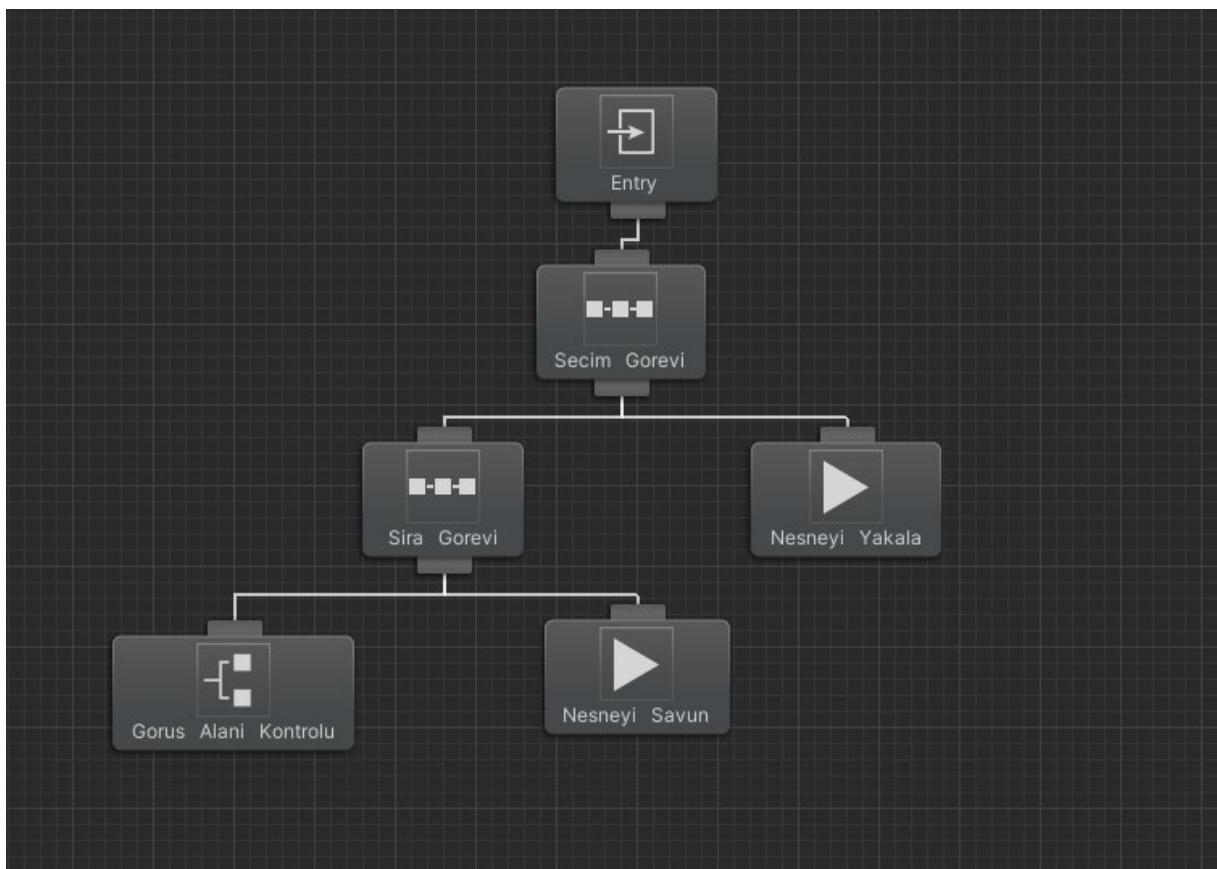
## 5.2.Ajan Çeşitleri

Sahnede bulunan ajanlar 2 takıma ve 3 farklı görevde ayrılmışlardır.

### 5.2.1. Savunan Ajanlar

Kırmızı renkli olan bu ajanlar 3 tanedir. Labirentin iç bölgesinde konumlandırılmışlardır. Görevleri görüş alanına giren mavi renkli düşman birimlerine karşı bölgeyi savunmak ve sonrasında düşmanı takip edip yakalamaktır.

Davranış ağacı şeması şekildeki gibidir.



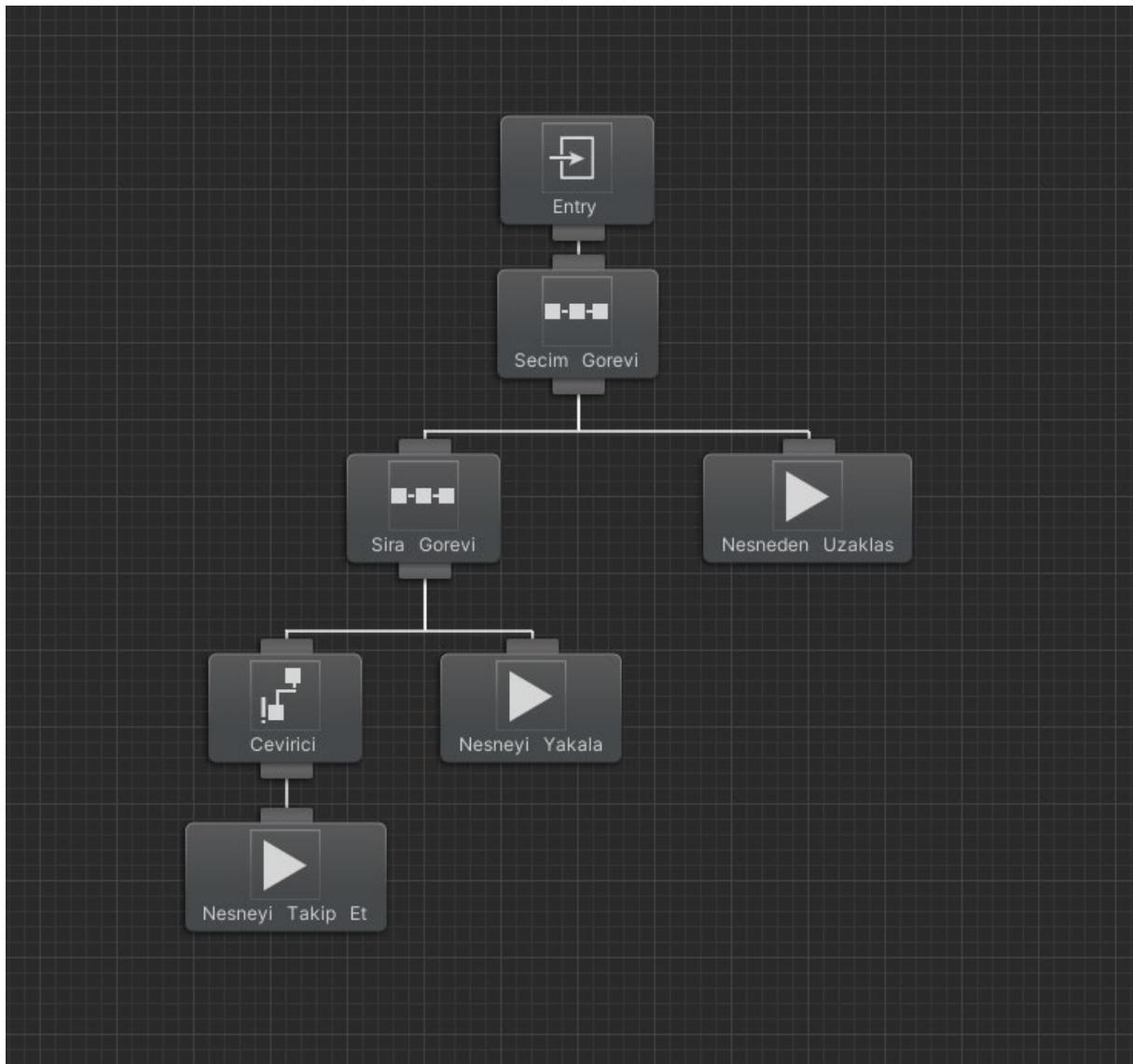
Şekil 25 Savunan Ajan Davranış Ağacı Şeması

Şema üzerinden anlaşılacağı gibi seçim görevi ile sol ve sağ kollar arasında seçim yapılmaktadır. Sol kolda ise görüş alanı kontrolü yapılır ve görüş alanına önceden belirlenen tipteki ajanlar girerse başarı döndürür ve Nesneyi Savun görevi çalışmaktadır. Burada önceden belirlenen ajan tipleri ister kod ister Unity ara yüzü ile değiştirilebilmektedir. Ben bu durumda mavi renkli ajanları seçtim. Bölge savunmasından sonra da bayrak düşman birimler tarafından ele geçirilirse Nesneyi Yakala görevi aktifleşmekte ve davranış ağacı sonlanmaktadır.

### 5.2.2. Saldırıran Ajanlar

Mavi renkli olan bu ajanlar 2 tanedir. Labirentin dış bölgesinde konumlandırılmışlardır. Görevleri labirentin merkezinde bulunan bayrağı almak ve belirlenen bitiş noktasına düşman birimlerine yakalanmadan ulaşmaktır.

Davranış ağacı şeması şekildeki gibidir.



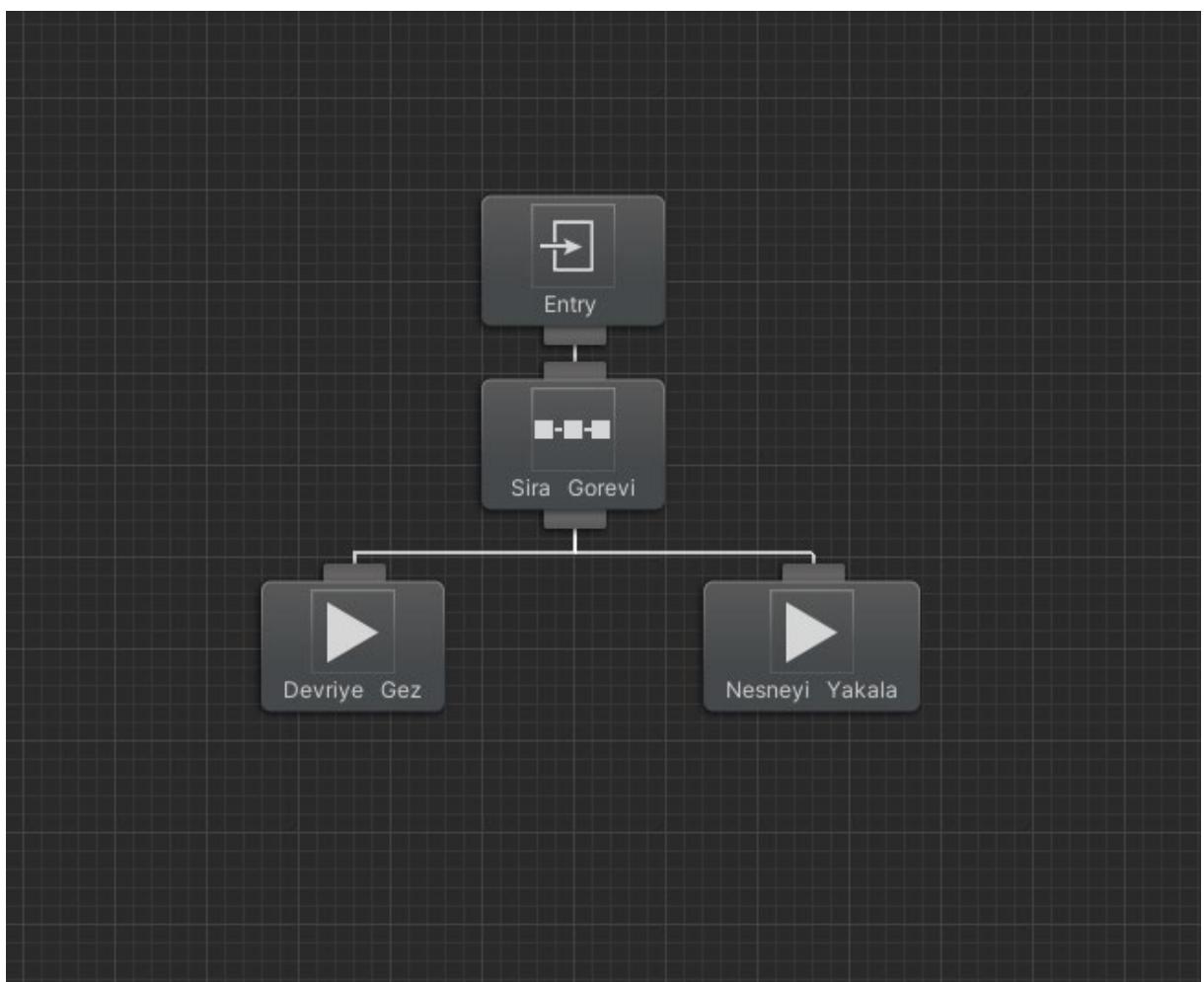
Şekil 26 Saldırı Ajan Davranış Şeması

Şemadan da anlaşılacağı gibi seçim görevi ile sol ve sağ kollar arasında seçim yapılmaktadır. Sol kolda sıra görevi Nesneyi Takip Et hata döndürdüğünde başarı döndürür ve Nesneyi Yakala görevi aktifleşir. Sağ kolda ise Nesneden Uzaklas görevi ancak Nesneyi Takip Et hata döndürdüğü zaman çalışır.

### 5.2.3. Devriye Gezen Ajanlar

Kırmızı renkli olan bu ajanlar 3 tanedir. Labirentin orta bölgesinde konumlandırılmışlardır. Görevleri önceden belirlenmiş olan güzergah üzerindeki kontrol noktaları arasında hareket etmek ve düşman birimlerinin görüş alanına girip girmedğini kontrol etmektir. Yine aynı şekilde düşman birimlerinin tanımlanması kod üzerinden yada Unity ara yüzü üzerinden yapılabilmektedir.

Davranış ağacı şeması şekildeki gibidir.



Şekil 27 Devriye Gezen Ajan Davranış Şeması

Şemadan da anlaşılacağı gibi sol kolda Devriye Gez görevi gerçekleştirken sağ kolda ise Nesneyi Yakala görevi yerine getirilir.

## KAYNAKLAR

|Yapay Zeka Ders notları – Dr. Abdurrahim Akgündoğdu, 2019

[https://tr.wikipedia.org/wiki/Yapay\\_zekâ](https://tr.wikipedia.org/wiki/Yapay_zekâ)

[https://www.gamasutra.com/view/feature/130663/gdc\\_2005\\_proceeding\\_handling\\_.php](https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php)

<http://bilgisayarkavramlari.sadievrenseker.com/2009/11/23/arama-algoritmaları-search-algorithms/>

<https://medium.com>

<https://www.sciencedirect.com/science/article/pii/S2452315118301516>

<https://docs.unity3d.com/Manual/bolt-connections.html>

<https://docs.microsoft.com/tr-tr/dotnet/csharp/programming-guide/classes-and-structs/inheritance>

<https://app.diagrams.net>

[https://en.wikipedia.org/wiki/Artificial\\_intelligence\\_in\\_video\\_games](https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games)

<https://www.aithority.com/computer-games/understanding-the-role-of-ai-in-gaming/>

<https://www.cc.gatech.edu/~riedl/pubs/cig13.pdf>

|

**EKLER**

[ |