LABYRINTH SOLVING PROBLEM

Yüksel Karadeniz
Bilgisayar Mühendisliği
TOBB Ekonomi ve Teknoloji Üniversitesi
Ankara, Türkiye
ykaradeniz@etu.edu.tr
141101040

Efe Karaman
Bilgisayar Mühendisliği
TOBB Ekonomi ve Teknoloji Üniversitesi
Ankara, Türkiye
e.karaman@etu.edu.tr
181101056

Abstract— Java tabanlı olarak labyrinth solving problemini farklı yaklaşımlar ile çözülmesi ve incelenmesi

Keywords— Dijkstra, A*,Best First Search, Algorithm Analysis, Java, Labyrinth Solver

I. GIRIŞ

Bu rapor siber güvenlik dersi kapsamında gerçekleştirilen Labyrinth Solver projesinin yazılım aşamalarını, amacını, çalışma şeklini, kullanılan yapay zeka algoritmalarını, bu algoritmaların birbirleriyle olan karşılaştırılmalarını ve projenin genel detaylarını özetlemektedir.

II. PROJENIN AMAÇLARI

Proje kapsamında yapılması amaçlanan ve proje sonucunda tamamlanan maddeler şunlardır:

En temel yapay zeka problemlerinden biri olan labyrinth solver problemini bilgisayar ortamında simüle etmek.

Kullanıcı tercihlerine göre farklı haritalar ile farklı algoritmaları test etmek ve algoritmaların birbirlerine olan üstünlüklerini test etmektir.

Elde ettiğimiz sonuçlar ışığında hangi algoritmaların daha verimli olduğunu veya kullanılan algoritmaların birbirlerine göre avantajlarını ve dezavantajlarını incelemektir.

III. KULLANILAN ALGORİTMALAR

Proje kapsamında kullanılan temel arama algoritmaları şunlardır:

- Dijkstra Algoritması
- Best First Search(BFS) Algoritması
- A* Algoritması

IV. DIJKSTRA ALGORİTMASI

1977 yılında Donald B. Johnson tarafından geliştirilmiştir. Dijkstra algoritması,ağırlıklı bir graph üzerinde yani kenarları(edge) belli bir metrik değere göre değerleri olan herhangi iki düğüm arası en kısa mesafeyi bulmamızı sağlayan bir algoritmadır.

Algoritma,Hollandalı matematikçi ve bilgisayar bilimci Edsger Wybe Dijkstra tarafından bulunmuştur.Bu algoritma routing başta olmak üzere birçok alanda kullanılmaktadır. Hatta birçoğumuzun telefonlarında veya diğer teknolojik cihazlarında bulunan navigasyonun temeli bu algoritmaya dayanır. Dijkstra algoritmasını zaman karmaşıklığı yani büyük o notasyonu O(MlogN)‘dir.

Dijkstra algoritması Açgözlü yani Greedy bir algoritmadır. Yani dijkstra algoritması,bir düğümden diğer bir düğüme geçerken mevcut durumun ‘en iyi’

çözümünü seçer. Bu algoritmanın amacı bir düğümden başlayarak o düğümün tüm graph üzerinde bulunan düğümlerine en kısa mesafede ulaşmasını garanti etmesidir. Ayrıca bundan daha kısa bir yol bulunmayacağını iddia etmektedir.

Bu algoritmanın dezavantajı ise negatif değer taşıyan kenarlar üzerinde sağlıklı bir şekilde çalışamamasıdır. Çünkü sürekli olarak mevcut durumdan daha iyi bir sonuç üretmeye çalıştığı için algoritma sonsuz döngüye girebilmektedir.

V. A* SEARCH ALGORİTMASI

Bilgisayar bilimlerinde en kısa yol bulmak için kullanılan algoritmalardan birisidir. Örneğin seyyar tüccar problemi (travelling salesman problem, TSP) gibi bir problemin çözümünde kullanılabilir. Benzer şekilde oyun programlamada, oyunda bulunan oyuncuların en kısa yolu bularak hedefe gitmeleri için de sıklıkla kullanılan algoritmadır.
Kısaca bir düğümden (node) hedef bir düğüme (target node) en kısa hangi düğümler üzerinden gidileceğini bulmaya yarayan "en iyi yerleştirme (best fit)" algoritmasıdır.

A* algoritması yapı olarak muteber sezgisel (admissible heuristic) bir algoritma olarak sınıflandırılabilir. Bunun sebebi algoritmasının mesafe hesaplamada kullandığı fonksiyondur:

$$f(n) = g(n) + h(n)$$

denklemindeki

$f(n)$ = hesaplama yapan sezgisel (heuristic) fonksiyon.

$g(n)$ = Başlangıç düğümünden mevcut düğüme kadar gelmenin maliyeti

$h(n)$ = Mevcut düğümden hedef düğüme varmak için tahmin edilen mesafe.

Dikkat edileceği üzere f(n) fonksiyonunun sezgisel olma sebebi, bu fonksiyon içerisinde bulunan ve tahmine dayalı olan h(n) sezgisel fonksiyonudur.

Algoritmanın çalışması:

Algoritma yukarıdaki toplama işlemini kullanan oldukça basit bir yapıya sahiptir. Veri yapısı olarak bir öncelik sırası (priority queue) kullanan algoritmada en öncelikli olan düğüm f(n) değeri en düşük olan düğümdür.

1. Algoritma her adımda en düşük değeri (Ve dolayısıyla en önemli) düğümü alır (yani bu düğüme gider) ve düğümü sıradan (queue) çıkarır.
2. Gidilen bu düğüme göre komşu olan bütün düğümlerin değerleri güncellenir (artık bu düğüme gelmenin bir maliyeti vardır ve dikkat edilirse f(n) fonksiyonu içerisinde bu değer yer almaktadır.)
3. Algoritma yukarıdaki adımları hedefe varana kadar (yani hedef düğümü öncelik sırasında (priority queue) en öne gelene kadar) veya sırada (queue) düğüm kalmayana kadar tekrarlar.

VI. BEST FIRST SEARCH ALGORİTMASI

Best First Search algoritması temel olarak Breadth First algoritması ile Depth First Algoritmalarının en iyi yönlerini birleştirmeyi hedeflemiştir. Düğümler değerlendirme fonksiyonu F(n)' e göre genişletilir. Burada ki F geleneksel bir maliyet ölçüsüdür. Üretilen düğümler içinden en uygunu seçilir ve bu düğüm genişletilir. Burada seçme işlemi sezgisel olarak H(n) fonksiyonuna göre yapılmaktadır. H(n) bize mevcut düğümden bitiş düğümüne olan tahmini en ucuz maliyeti bildirmektedir. Eğer H(n) = 0 ise hedef düğüme varılmış anlamına gelmektedir. Temel olarak A* Search algoritması Best First Search algoritmasının özelleşmiş halidir.

$f(n) = h(n)$

denklemindeki

$f(n)$ = hesaplama yapan sezgisel (heuristic) fonksiyon.

$h(n)$ = Mevcut düğümden hedef düğüme varmak için tahmin edilen mesafe.

Dikkat edileceği üzere f(n) fonksiyonunun sezgisel olma sebebi, bu fonksiyon içerisinde bulunan ve tahmine dayalı olan h(n) sezgisel fonksiyonudur.

VII. SİMULASYON ORTAMININ OLUŞTURULMASI

```
long dj = 0L, astar = 0L, bfs = 0L;
int count = 0;
for (int i = 0; i < 100000; i++) {
    randomLabyrinth(30);
    int randomI1 = (int)(Math.random() * grid.length);
    int randomJ1 = (int)(Math.random() * grid[0].length);
    int randomI2 = (int)(Math.random() * grid.length);
    int randomJ2 = (int)(Math.random() * grid[0].length);
    grid[randomI1][randomJ1].setType(Cell.START);
    grid[randomI2][randomJ2].setType(Cell.EXIT);

    try {
        Dijkstra dij = new Dijkstra();
        boolean pathFoundD = dij.getPathFound();

        AStar a = new AStar();
        boolean pathFoundA = a.getPathFound();

        BFS bestFirst = new BFS();
        boolean pathFoundB = bestFirst.getPathFound();

        if (pathFoundA && pathFoundD && pathFoundB) {
            dj += dij.timeTaken;
            astar += a.timeTaken;
            bfs += bestFirst.timeTaken;
            count++;
        }
    } catch(Exception e) {
        continue;
    }
}
```

Fig. 1. Test için kullanılan kod parçası

Yukarıdaki kod parçası, 3 algoritmayı da rastgele oluşturulan labirentler üzerinde test etmek için kullanılmıştır. Her algoritmanın labirenti ne kadar sürede çözdüğü ve üçünün birden çözebilmiş olduğu labirent sayısı kayıt edilmiştir. Algoritmaların çözümü olmayan labirentlerde çözüm bulamamasının yanı sıra, 2 tanesinin açgözlü algoritma olması, test edilen labirentlerin hepsinin çözülememesinin sebebidir.

VIII. ALGORİTMALARIN UYGULANMASI

Proje içerisinde analiz edilen ve karşılaştırılan algoritmaların kod içerisinde uygulanması aşağıdaki şekillerde gösterildiği gibi yapılmıştır.

```
public List findPath(Node start, Node exit) {
    Comparator<Node> comparatorF = new Comparator<Node>() {
        public int compare(Node n1, Node n2) {
            return n1.compareTo(n2);
        }
    };

    PriorityQueue open = new PriorityQueue(comparatorF);
    LinkedList closed = new LinkedList();

    start.G = 0;
    start.H = start.findHeuristicDistance(exit);
    start.parent = null;
    open.add(start);

    while(!open.isEmpty()) {
        Node node = (Node)open.remove();
        if (node == exit)
            return getPath(exit);
```

Fig. 2. A* search algoritmasının kodlanması-1

```
        ArrayList<Node> neighbors = node.neighbors;
        for (int i = 0; i < neighbors.size(); i++) {
            Node adj = neighbors.get(i);
            boolean isOpen = open.contains(adj);
            boolean isClosed = closed.contains(adj);
            double G = node.findG(start) + node.findG(adj);

            if ((!isOpen && !isClosed) || G < adj.findG(node)) {
                adj.parent = node;
                adj.G = G;
                adj.H = adj.findHeuristicDistance(exit);
                if (isClosed)
                    closed.remove(adj);
                if (!isOpen)
                    open.add(adj);
            }
        }
        closed.add(node);
    }
    return null;
}
```

Fig. 3. A* search algoritmasının kodlanması-2

```
private Node solve(Node start, Node exit, Comparator<Node> distanceComp) {
    PriorityQueue<Node> pq = new PriorityQueue<Node>(distanceComp);
    pq.add(start);

    Node currentNode = null;
    while(!pq.isEmpty()) {
        currentNode = pq.poll();
        currentNode.visited = true;
        for (Node neighbor : currentNode.getUnvisitedNeighbors()) {
            int minDistance = (int)(Math.min(neighbor.distance, currentNode.distance + 1));
            if (minDistance != neighbor.distance) {
                neighbor.distance = minDistance;
                neighbor.parent = currentNode;

                //this is for making sure neighbors priority is updated in pq
                //it might not be necessary
                if (pq.contains(neighbor)) {
                    pq.remove(neighbor);
                    pq.add(neighbor);
                }
                if (!pq.contains(neighbor))
                    pq.add(neighbor);
            }
        }
    }

    return exit;
}
```

Fig. 4. Dijkstra algoritmasının kodlanması

```
public List findPath(Node start, Node exit) {
    Comparator<Node> comparatorF = new Comparator<Node>() {
        public int compare(Node n1, Node n2) {
            return n1.compareTo(n2);
        }
    };

    PriorityQueue open = new PriorityQueue(comparatorF);
    LinkedList closed = new LinkedList();

    start.heuristic = start.findHeuristicDistance(exit);
    start.parent = null;
    open.add(start);

    while(!open.isEmpty()) {
        Node node = (Node)open.remove();
        if (node == exit)
            return getPath(exit);

        ArrayList<Node> neighbors = node.adj;
        for (int i = 0; i < neighbors.size(); i++) {
            Node adj = neighbors.get(i);
            boolean isOpen = open.contains(adj);
            boolean isClosed = closed.contains(adj);

            if ((!isOpen && !isClosed) || node.heuristic > adj.heuristic) {
                adj.parent = node;
                if (isClosed)
                    closed.remove(adj);
                if (!isOpen)
                    open.add(adj);
            }
        }
        closed.add(node);
    }
    return null;
}
```

Fig. 5. Best First Search algoritmasının kodlanması

IX. TEST AŞAMASI

Çalıştırılan simulasyon ortamının 100000' er defa tekrarlanması sonucunda elde edilen bulgular aşağıda sunulmuştur.

```
each algorithm found a path in 28947/100000 tests
total time taken (millisec): 112575
total time for Dijkstra (millisec): 1014
total time for A* (millisec): 826
total time for BFS (millisec): 150
average time taken for Dijkstra (millisec): 0.0350295367395585O5
average time taken for A*: 0.028534908626109787
average time taken for BFS: 0.005181884133070785
BFS was fastest.
BFS was faster than Dijkstra by: 85%
BFS was faster than A* by: 81%
```

Fig. 6. Algoritmaların Çalışma Zamanları-1

```
each algorithm found a path in 29242/100000 tests
total time taken (millisec): 128409
total time for Dijkstra (millisec): 957
total time for A* (millisec): 953
total time for BFS (millisec): 165
average time taken for Dijkstra (millisec): 0.032726899664865604
average time taken for A*: 0.03259011011558717
average time taken for BFS: 0.005642568907735449
BFS was fastest.
BFS was faster than Dijkstra by: 82%
BFS was faster than A* by: 82%
```

Fig. 7. Algoritmaların Çalışma Zamanları-2

```
each algorithm found a path in 29349/100000 tests
total time taken (millisec): 130993
total time for Dijkstra (millisec): 997
total time for A* (millisec): 945
total time for BFS (millisec): 154
average time taken for Dijkstra (millisec): 0.033970493032130566
average time taken for A*: 0.03219871205151794
average time taken for BFS: 0.005247197519506627
BFS was fastest.
BFS was faster than Dijkstra by: 84%
BFS was faster than A* by: 83%
```

Fig. 8. Algoritmaların Çalışma Zamanları-3

Yukarıda gösterilen analizler sonucunda Best First Search Algoritmasının diğer algoritmalara oranla çok daha hızlı ve verimli çalıştığı görülmüştür. Performans açısından Best First Search algoritmasının arkasından sırasıyla A* search algoritması ve Dijkstra algoritmaları yer almaktadır.

Best First Search algoritması A* search algoritmasından yaklaşık olarak %81-83 oranında daha hızlı çalıştığı tespit edilmiştir.

Best First Search algoritması Dijkstra algoritmasından yaklaşık olarak %82-85 oranında daha hızlı görülmüştür. Ancak bu sonuçlar üç algoritmanın da birlikte çalışabileceği bir simülasyon ortamından alınmıştır.

Ancak farklı ortamlarda testler yapıldığında hedefe yönelik en pratik ve verimli çalışan algoritma A* search algoritması olduğu görülmüştür.

Dijkstra search algoritması tüm seçeneklere her aşamada bakması gerektiği için diğer algoritmalara göre daha yavaş çalıştığı sonucuna varılmıştır. A* search algoritması en düşük maliyetli yolu keşfederken sezgisel değerlerle birlikte elindeki mevcut değerleri de hesaba kattığı için Dijkstra search algoritmasından daha kısa zamanda sonuca ulaştığı tespit edilmiştir.

Bununla birlikte simule ettiğimiz probleme odaklanırsak Best First Search algoritması önünde olan engelleri dikkate almadan sadece kuş uçuşu sezgisel değerlere göre rota hesapladığı için graph içerisinde yer alan leaf node lara kadar ilerleyip kendini çıkmaz sokaklara sokabildiği görülmüştür. Bu durumda gidebileceği bir yer kalmadığı için ya geri dönmesi ya da uygun path bulunamadı diyerek başarısızlıkla dönmesi kaçınılmaz olduğu tespit edilmiştir.

A* search algoritmasının optimalliğinin Dijkstra search algoritmasından daha iyi olmasının sebebi gidilebilecek komşuları seçerken sezgisel işlemleri kullanması olduğu tespit edilmiştir. A* search algoritması hız odaklı bir algoritma olduğu için daha fazla bellek ve node başına daha fazla işlem maliyeti gerektirse bile Dijkstra search algoritmasından daha hızlı çalışmasının sebebi çok daha az node'u araştırmasıdır.

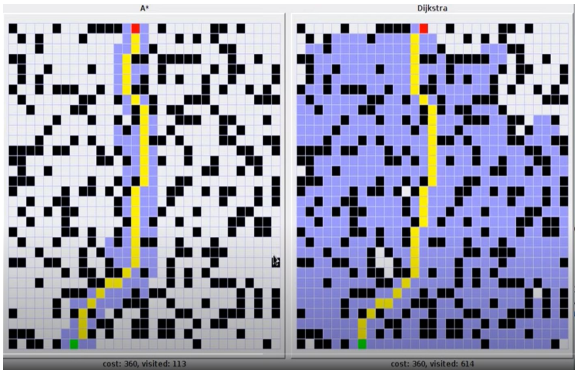

Fig. 9. A* ve Dijkstra Algoritmalarının Karşılaştırılması

Yukarıdaki şekilde A* algoritmasının Dijkstra search algoritması ile karşılaştırılması resmedilmiştir.

Burada görüldüğü üzere A* algoritması hedefe doğru ilerlerken çok daha verimli bir şekilde ilerlemekte ve maliyeti çok düşük seviyelerde tutmuştur. Dijkstra search algoritması ise neredeyse tüm harita içerisinde ki nodeları ziyaret ederek işlem ve zaman yükünü oldukça yüksek seviyelerde tutmuştur.

Özetle A* search algoritması sezgisel yaklaşımı ile işlem,hız ve maliyet karşılaştırmaları açısından Best First Search ve Dijkstra algoritmalarından daha verimli çalışmaktadır.

X. UYGULAMANIN KULLANIMI

Uygulama çalıştırıldığında "Labyrinth Solver" ve "Controls" isimlerini taşıyan iki pencere açılır.

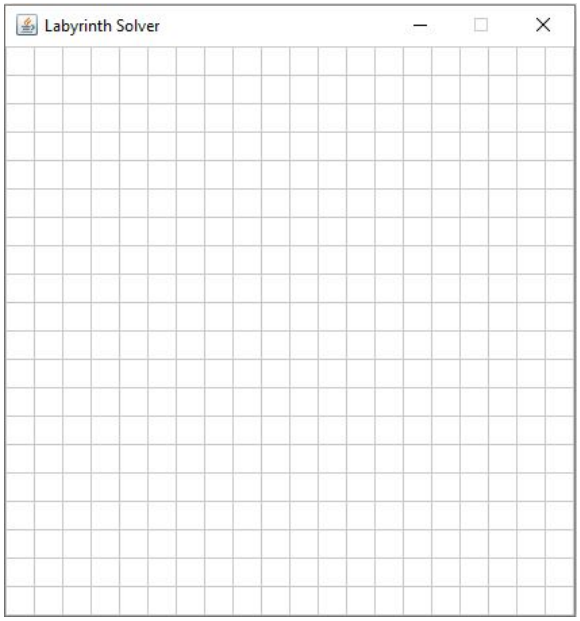

Fig. 10. Labyrinth Solver Penceresi

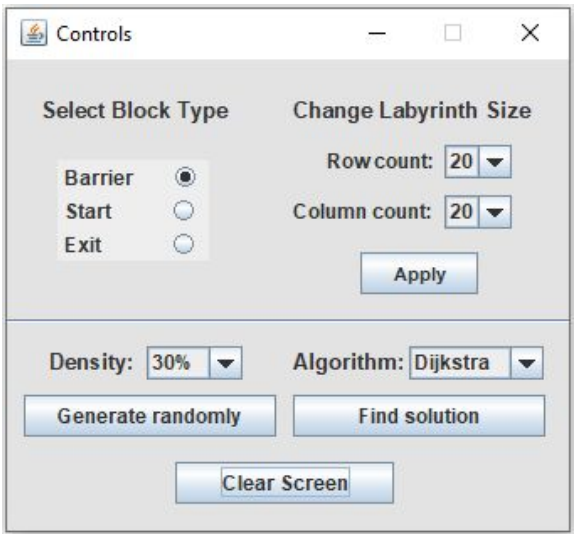

Fig. 11. Controls Penceresi

Uygulamayı kullanmak için gereken bütün seçenekler Controls penceresinde bulunabilir.

"Select Block Type" başlığı altında labirente yerleştirilebilecek blok türleri bulunur. Buradan seçilen blok türünü yerleştirmek için Labyrinth Solver penceresinde herhangi bir kareye farenin sol tuşu ile tıklanabilir. Aynı şekilde bir karenin içini boşaltmak için üstüne farenin sol tuşuyla tıklanması yeterlidir.

Barrier seçeneği seçildiğinde, labirentte üstüne tıkladığınız kareler engele dönüşür, bu engellerin üstünden yürümek mümkün değildir. Barrier bloğu ekranda siyah renkli bir kare olarak gösterilir.

Start seçeneği labirentin başlangıç noktasını yerleştirmek için kullanılır ve bu bloktan en fazla 1 tane yerleştirilebilir. Ekranda içinde 'S' harfi bulunan mor bir kare olarak gösterilir.

Son olarak Exit seçeneği ise labirente bir çıkış noktası koymanızı sağlar, bu bloktan istediğiniz sayıda yerleştirebilirsiniz. Exit bloğu ekranda içinde 'E' harfi yazan yeşil bir kare olarak gösterilir.

"Change Labyrinth Size" başlığının altındaki Row count ve Column count seçenekleri, Labyrinth Solver penceresinde bulunan satır ve sütun sayısını değiştirmenize olanak sağlar. İstenilen sayılar seçildikten sonra Apply tuşuna basarak pencere boyutundaki değişikliklerinizi uygulayabilirsiniz.

Controls penceresinin alt bölümünde bulunan ilk seçenek olan Density, rastgele labirent oluştururken karelerin yüzde kaçının dolu olmasını istediğinizi seçmenize

olanak sağlar. Bu seçimi yaptıktan sonra Generate randomly tuşunu kullanarak, labirenti seçtiğiniz yüzdeye karşılık gelecek sayıda Barrier bloğu ile doldurabilirsiniz.

Algorithm seçeneği, labirenti çözmek için, kayıtlı algoritmalardan birinin seçilmesine olanak sağlar. Bu seçenekler Dijkstra, A* ve BFS (Best First Search) algoritmalarıdır. Find solution tuşuna basıldığında seçili algoritma çalıştırılır. Giriş ve çıkışlar arası bir yol bulunursa labirent üzerinde açık yeşil renkte işaretlenecektir. Yol bulunamadığı takdirde bir uyarı penceresi açılır ve kullanıcı bilgilendirilir.

Fig. 12. Çözüm Bulunamadığında Gösterilen Uyarı Penceresi

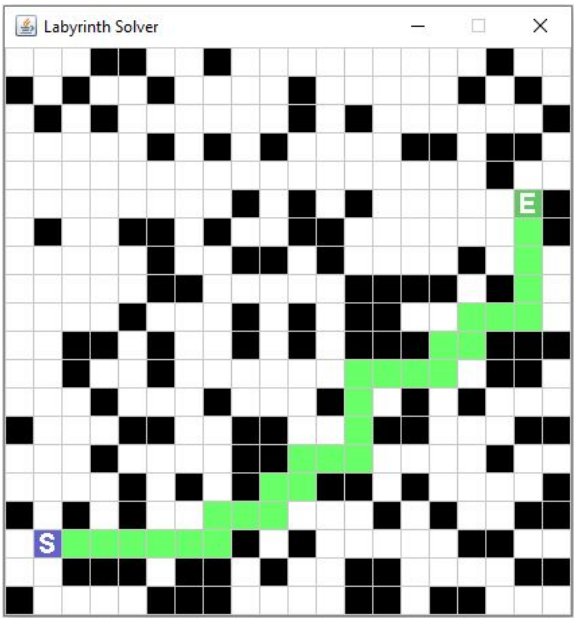

Fig. 13. Başarıyla Çözülmüş Bir Labirent Örneği

Son olarak, Controls penceresinin en altında bulunan Clear screen tuşuna basılarak labirentteki tüm karelerin içi boşaltılabilir. Bu sayede yeni bir senaryo için haritanın sıfırlanması gerçekleştirilmiş olacaktır.

XI. KAYNAKÇA

Dijkstra Algoritması

https://hurna.io/academy/algorithms/maze_pathfinder/dijkstra.html

A Search Algoritması*

https://www.peachpit.com/articles/article.aspx?p=101142&seqNum=2#:~:text=The%20A*%20algorithm%20works%20the,words%2C%20it%27s%20a%20priority%20queue.

Algoritma Performans Karşılaştırmaları

https://www.youtube.com/watch?v=g024lzsknDo

Java Oyun Motoru

https://www.youtube.com/watch?v=1gir2R7G9ws&t=1s&ab_channel=RealTutsGML

Yapay Zeka ile Temel Arama Algoritmaları

http://bilgisayarkavramlari.com/2009/11/23/arama-algoritmalari-search-algorithms/?highlight=search

Yapay Zeka (A) ile Maze Solver Yaklaşımı*

http://www.ijmerr.com/uploadfile/2017/0904/20170904105839434.pdf