TOBB EKONOMİ VE TEKNOLOJİ ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

2020-2021 GÜZ DÖNEMİ BİL 441 DERSİ PROJE RAPORU

Ayça Gürleyik	151101032
Tuna Akyürek	151101061

İçindekiler

PROJENİN ÖZETİ

Bu dönem almış olduğumuz BİL 441 kodlu Yapay Us dersinde, derste öğrendiğimiz algoritma ve yöntemlerini de kullanarak maze problemini çözen yapay zekâ algoritmaları üretmemiz istenmektedir.

PROBLEM HAKKINDA

Bu problem özet olarak 2 boyutlu grid yapısı ile oluşturulmuş bir maze içerisinde, başlangıç noktasından bitiş noktasına engellerden kaçınarak en kısa pathi bulmayı gerektiriyor. Maze, cell'lerden oluşuyor, bu cell'lerde çeşitli engeller bulunabiliyor. Başlangıç noktası, maze içerisinde rastgele veya kullanıcı tarafından seçilen bir yere yerleştiriliyor. Üretilen yol her yöne gidebiliyor ama yalnızca engel olmayan boş cell'lerde ilerleyebiliyor.

Maze'in nasıl olduğu ve karakteristiği yapay zekanın öğrenme complexitysini etkilemektedir. Problem özet olarak bu şekilde olsa da çok sayıda farklı varyasyonları bulunmaktadır ve bu varyasyonlara literatür incelemesi kısmında değineceğiz.

AMAÇ

Projemizde amacımız, maze problemini sezgisel bir şekilde çözen bir yapay zeka oluşturmaktır. Maze oluşturma algoritmasını, path finder algoritmasını ve programımızın ara yüzünü python dili ile gerçekledik. Program ilk çalıştırıldığında, kullanıcıya yeni bir maze oluşturma veya var olan mazelerden seçim yapma imkânı sunulmaktadır. Maze belirlendikten sonra kullanıcıdan başlangıç ve bitiş noktaları belirlemesi istenmektedir. Bu aşamada kullanıcı yeni engeller ekleyebilir, var olan engelleri kaldırabilir veya sonraki aşamaya geçebilir. Kullanıcının bu maze üzerinde test edebileceği iki farklı pathfinder algoritması sunulması planlanmıştır. İki algoritmanın da her çalışmada başlangıç noktasından bitiş noktasına engellerden kaçınarak en kısa yolu bulması ve costu hesaplaması gerekmektedir. Bu iki algoritma için performansları kıyaslanacaktır ve çıkarım yapılacaktır. Kullanmış olduğumuz sezgisel algoritmanın başarısı değerlendirilecektir.

LİTERATÜR İNCELEMESİ

Maze Problemi Çeşitleri (1)

1. Fare-Labirent Problemi: En bilindik maze problemidir. Bu problemde, bitiş noktasına kadar olan bütün olası pathleri bulmak amaçlanır. Genellikle başlangıç noktası ilk cell (örneğin M [1,1]) ve bitiş noktası son cell (örneğin M [n,n]) olur.

2. Costları Farklı olan Pathler Problemi: Bu problemde her path (başlangıç noktası, bitiş noktası veya engel olmayan her cell) farklı bir costa veya weighte sahip olur. Burada amaç en az maliyet ile bitiş noktasına varmaktır.

3. Agent Hareketleri Özelleştirilmiş Problemler: Bu problem tipinde agentın hareketleri sadece sağ, sol, yukarı, aşağı ile kısıtlı değildir. Diyagonal olabilir, zıplayabilir problemi yaratan kişilerin fikirlerine bağlıdır.

Problem özünde varış noktasından bitiş noktasına varmak olsa da bahsettiğimiz farklılıklara göre hesaplamalar değişebilmektedir.

Maze problemini matematiksel olarak ilk inceleyen kişi Leonhard Euler olmuştur (2). Euler'in uğraştığı maze probleminin adı Köningsberg problemidir (3). Bu sayede matematiğe topology kavramını katmıştır. Loop içermeyen mazelere "standart" maze denmektedir ve bu mazeler graph theory için tree kavramına eşdeğerdir. Bu yüzden de maze problemi çözen algoritmaların çoğu graph theory ile ilgilidir.

KARŞILAŞILAN ZORLUKLAR

Search esnasında karşılaşılan zorluklar :
A* search ile birden fazla çıkış noktası bulunduğunda algoritmanın bu çıkış noktalarını tek tek hesaplayıp sonra içlerinden minimum olanı dönmesi gerekmektedir. Tek çıkış için çalışmasını ayarladıktan sonra birden fazla çıkış için de doğru mantıkla çalışabilmesi için bir array içinde tüm çıkışların olası çözümlerini tutmamız gerekmektedir. Ayrıca çözümü olmayan çıkışlarında bu aşamada fark edilmesi ve o çıkışların çözüm arrayine eklenmemesine de dikkat edilmesi gerekmektedir. Eğer tüm çıkışlar ulaşılmaz ise algoritma çalıştıktan sonra bir çözüm yolu bulamayacaktır ve çözüm arrayi boş olması gerekmektedir.Aynı şekilde DFS için de complete bir algoritma olmamasından dolayı ulaşılmayan çıkışlarda sonsuz döngüye girmesini engellemek gerekmektedir. Çözümü olan çıkış noktasını bulmak ve diğer ulaşılmaz olan çıkış noktalarını aramamak gerekmektedir. Eğer tüm çıkış noktaları ulaşılmaz ise algoritmanın sonsuz döngüye girmemesi ve çıkış yolu olmadığını fark etmesini sağlamak gerekmektedir.
Doğrulama ve değerlendirme:
A* search algoritmasının farklı durumlarda ve mazelerde çalıştırılarak hepsi içinde optimal çözümü dönme mantığının doğru bir şekilde çalıştırılması gerekmektedir. Heuristic seçiminin de algoritmanın performansını büyük ölçüde etkilediği gerçeğinden dolayı bu projede en uygun heuristicin bulunması gerekmektedir. Projemizde en uygun heuristice büyük ölçüde zaman harcanmıştır ve bu heuristic sonucu büyük ölçüde etkilemiştir.

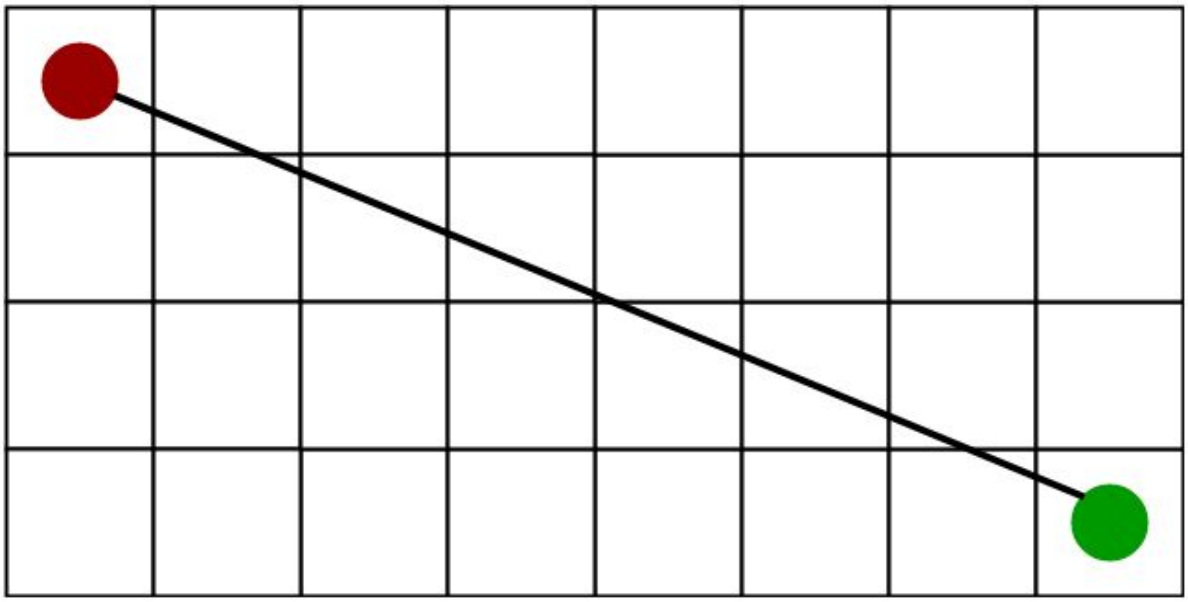
KULLANILAN METODLAR

Sezgisel fonksiyon nasıl geliştirildi?
Oyunda A* search içinde Euclidean heurictic kullanılmıştır. Bu sezgisel fonksiyonda nodelarımızın farklı costları vardır. Bu costların hesaplanması ise örneğin şu anda bulunduğumuz node, current node diye adlandırıldığı varsayalım ve x,y koordinatlarında bulunmaktadır. Start

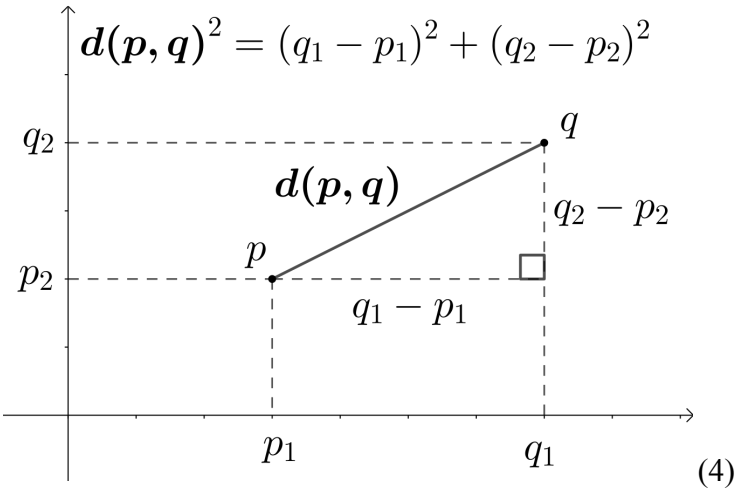
node x0,y0 koordinatlarında ve goal node xn,yn koordinatlarında bulunmaktadır. f(n) = g(n) + h(n) şeklinde hesaplama yapılacağından f(n) = (x-x0)^2+(y-y0)^2 + (x-xn)^2+(y-yn)^2 diye bulunmaktadır ve bu current node için toplam costtu ifade etmektedir.

Sezgisel fonksiyonumuz:

g(n) : Başlangıç noktasından şu anki noktaya kadarki cost
h(n) : Şu anki noktadan goal noktasına kadarki cost (heuristic)
f(n) : Toplam cost f(n) = g(n) + h(n)



Euclidean Distance



(4)

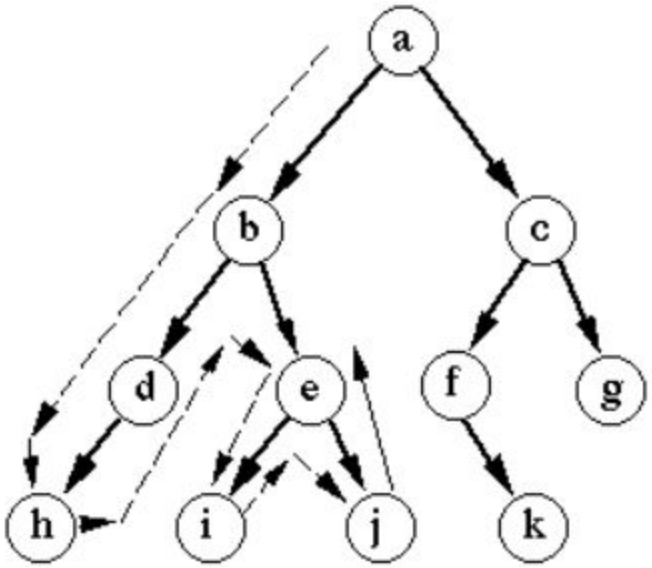
Search adımları nasıl yapılıyor ?

A* search için toplam costu heuristic fonskiyonu da kullarak hesaplanmaktadır ve her seferinde minimum costa sahip node seçilmektedir. Optimal çözümü bulana kadar algoritma çalışmaya devam etmektedir.

DFS için öncelikli derinliğe doğru arama yapılmaktadır ve bu yoldaki nodelar seçilmektedir.

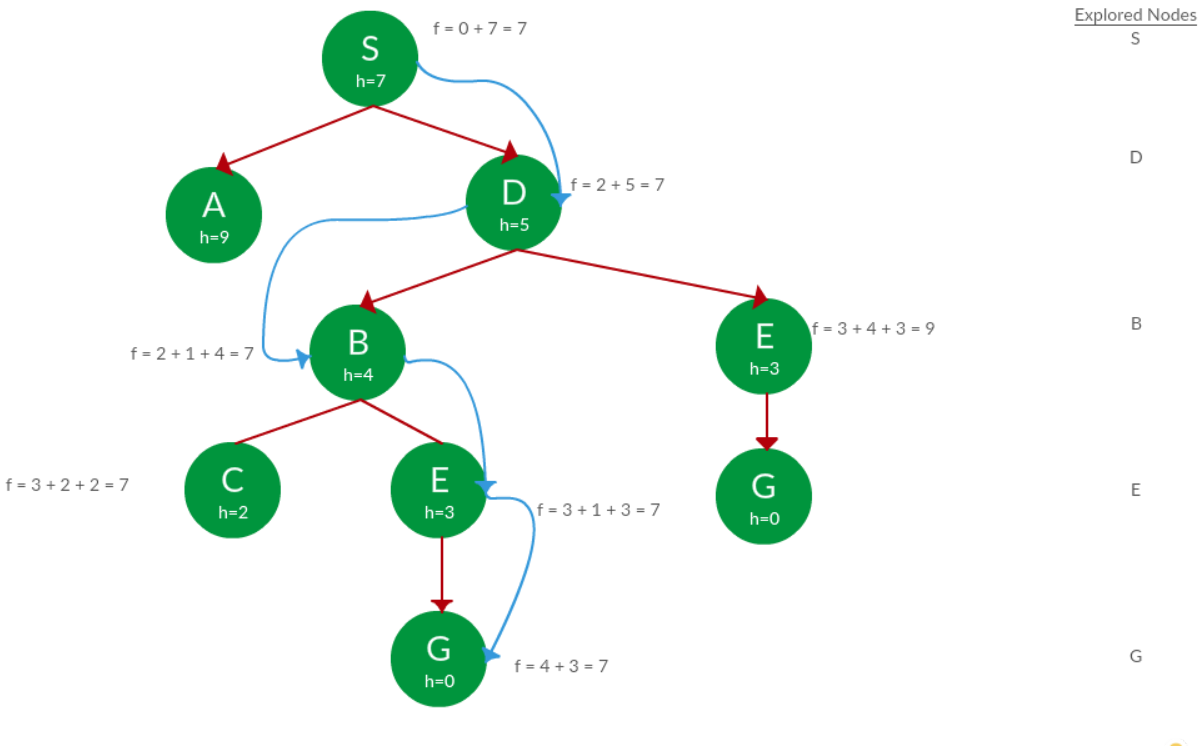
A* Search ve DFS algoritmalarına getirilen kısıtlar ve geliştirmeler nelerdir?

İki algoritma için maze içinde manuel olarak eklenebilen engeller vardır ve çözüm aranırken bu engellerden kaçınılması gerekmektedir, birden fazla çıkış noktası bulunabilmektedir A* bunlardan minimum olanı yani optimal çözümü bulmaktadır ve dfs bu noktalardan birindeki çözümü dönmektedir, ulaşılmayan çıkışlarda ise çözüm yolu olmadığı belirtilmektedir..



Depth-first search

(DFS Şekil 1)



(A* Şekil 2)

SEZGİSEL YÖNETİMİ

Bu proje kapsamında Maze Solver oyununu sezgisel olarak çözebilmek için, literatür incelemesi bölümünde bahsettiğimiz A* search algoritması ile birlikte Euclidean Distance kullanılmıştır. Kullanılan sezgisel yöntemi ile A* search ve DFS karşılaştırılacaktır.

ÇALIŞMA PERFORMANSI VE KARŞILAŞTIRMALAR

Oyunumuzda kullanılan iki farklı search algoritmasının performansları karşılaştırılmaktadır. A* search heuristic olarak Euclidean Distance yöntemini kullanırken , DFS derinliğe göre search yapmaktadır. Çözüm yoluna giden path üzerindeki node sayısı, toplamda expand edilen node sayısı ve çalışma süreleri karşılaştırılmaktadır. A* algoritması optimal çözümü garanti etmektedir.

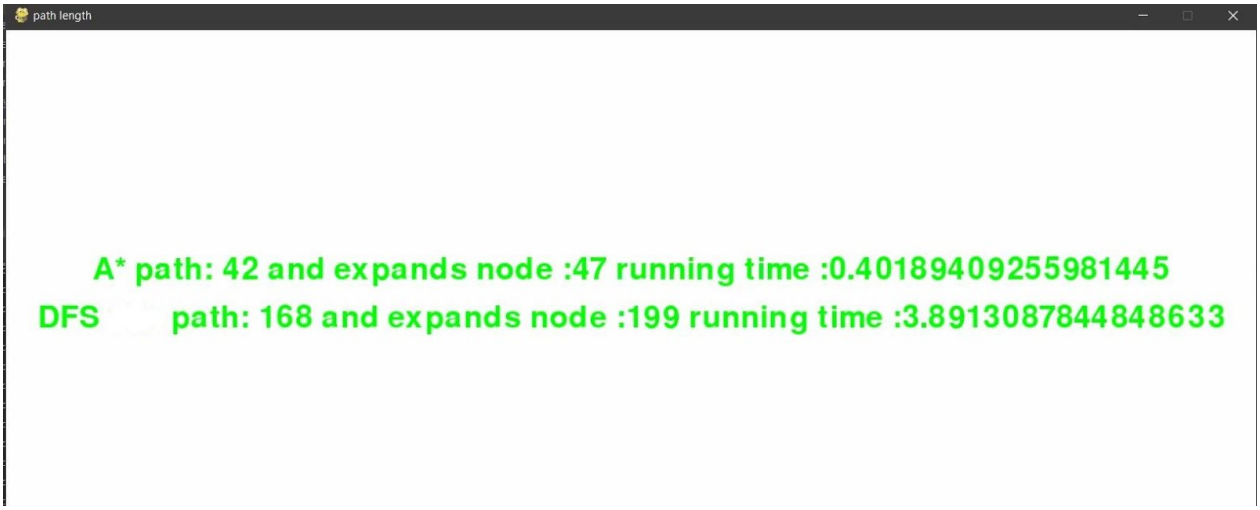
MAZE

Green tile= start point, Red tile = goal points, Blue tile= barrier, Black tile= wall, White tile= available, Purple tile= solution

A* SOLUTION

DFS SOLUTION

PERFORMANCE



Aynı maze üzerinde iki farklı search algoritmasının çalışması sonucunda A*optimal çözümü verirken, DFS bulunan herhangi bir çözümü vermektedir.

ALGORİTMA

aStarSearch.py

aStar():

- Verilen bir maze'i A* algoritmasi ile çözüyor.
- Grid'i ve çözüm yolundaki tile sayısını,running timeı ve expand edilen node sayısını döndürüyor.

aStarSearchFunctions.py

convertMaze(mazeFile):

- Grid bilgilerini içeren csv dosyası için adres alıyor.
- Dosyadaki maze'i import ediyor.
- Grid return ediyor.

deleteNodeFrontier(node, frontier):

- frontier listesinden silinmesi gereken nodeu alıyor.
- frontier, nodelar listesi.
- node silinmiş olarak yeni frontier listesi döndürüyor.

nodeCost(position, goal, start):

- position, x ve y pozisyonu için iki integer bulunduruyor. Bu position, costunu hesaplamak istediğimiz node'a ait.
- goal, goal pozisyonu için iki integer bulunduruyor.
- start, start pozisyonu için iki integer bulunduruyor.
- cost return ediyor.

- Bu cost position-goal+ position-start arası euclidean distance ile bulunuyor.

inMaze(step, dimension):
- step, o anki node'un pozisyonu(current position).
- dimension, kullanılan Grid'in büyüklüğü
- Step pozisyonunun Grid'de olup olmadığına bakıyor öyleyse true değilse false dönüyor.

findSuccessors(currentNode, maze, seen, curGoal, start):
- Current node'un successorunu hesaplıyor.
- maze, mazein gridi.
- seen, önceden bakılmış nodeların listesi.
- curGoal, goal listesindeki o an aramada yer alan current goal.
- start, başlangıç noktası.
- Successor nodeların listesini döndürüyor.

selectNode(node_list, start):
- Node listesinden costu en küçük olan node paylaşıyor.

computePath(maze, frontier, seen, current_node,curGoal,detect,start):
- Current node için bir step oluşturuyor.
- Grid'i, Frontier listesini ve seeni güncelliyor.
- Çözüm bulamazsa detecti güncelliyor.

dfsSearch.py

dfs():
- Verilen maze'i DFS algoritması ile çözüyor.
- A* algoritmasına göre efficent değil.
- Çözüm yolundaki tile sayısını, expand edilen node sayısını, mazei ve running timeı döndürüyor.

dfsSearchFunctions.py

convertMaze(mazeFile):
- grid bilgilerini içeren csv dosyası için adres alıyor.
- Dosyadaki maze'i import ediyor.
- grid return ediyor.

inMaze(step, dimension):
- step, o anki node'un pozisyonu(current position).
- dimension, kullanılan Grid'in büyüklüğü.
- step pozisyonunun Grid'de olup olmadığına bakıyor öyleyse true değilse false dönüyor.

allNextSteps(dimension, currentNode,maze):
- dimension: grid'in boyutu.
- currentNode: o anki node'un pozisyonu (xy kordinatları).
- Her yönde olası xy pozisyonları listesi dönüyor.

computePath(maze, currentNode, seen, backward,arrGoal):

- seen: gezilen nodeların listesi, 1 kere gördükten sonra bu listeye ekliyor.
- maze[x][y]'yi değiştiriyor.
- backwardı gerekli durumlarda güncelliyor.

mazeCreator.py

mazeCreator():

- DFS ile Maze oluşturuyor, pygame ile bu maze'i görselleştiriyor.

mazeCreatorFunctions.py

inMaze(step, dimension):

- step, o anki node'un pozisyonu(current position).
- dimension, kullanılan Grid'in büyüklüğü
- Step pozisyonunun Grid'de olup olmadığına bakıyor öyleyse true değilse false dönüyor.

allNextSteps(dimension, currentNode):

- dimension: gridin boyutu
- currentNode: current positionın xy kordinatları
- her yönde olası xy pozisyonları listesi dönüyor

createMaze(maze, currentNode, seen, backward):

- seen: gezilen nodeların listesi, 1 kere gördükten sonra bu listeye ekliyor.
- maze[x][y]yi değiştiriyor.

DEMO

Kullanıcı öncelikle main.py dosyasını başlatıyor. Ardından kullanıcıya eski mazelerden birini mi kullanacağı yoksa yeni bir maze mi oluşturulması istendiği soruluyor. Kullanıcının seçimine göre yeni bir maze oluşturuluyor ya da eski mazelerden birini seçmesi isteniyor. Kullanıcı maze seçimini yaptıktan sonra kaç goal noktası koyacağı bilgisi isteniyor, kullanıcı bir sayı değer verdikten sonra pygame guisi açılıyor. Bu ekranda görselleştirilmiş mazede kullanıcı öncelikle start noktasını yerleştiriyor ardından goal noktalarını yerleştiriyor ve yeni engeller koyabiliyor, engelleri kaldırabiliyor. Maze üzerinde değişiklikler tamamlandıktan sonra kullanıcının entera basması ile A* algoritması start noktasından goal noktalarına path bulma işlemine başlıyor. Goallardan ulaşılabilir en yakın olanını buluyor ve path çiziyor. Bulamaması halinde konsoldan bulunamadı uyarısı geliyor. Tekrar enter'a basıldığında pygame ekranı kapanıyor ve bu defa DFS algoritmasının çalıştırılacağı maze için yeniden maze seçim işlemleri soruluyor. Kaç goal istediği bilgisi de girildikten sonra tekrar maze ekranı geliyor. Kullanıcı start noktasını ve goal noktalarını yerleştiriyor ve entera bastıktan sonra dfs algoritması bu maze üzerinde çalışırılıyor. DFS algoritması sonlandıktan sonra bir pygame ekranı daha çıkıyor ve bu ekranda her iki algoritma için pathin kaç nodedan oluştuğu ve yürütülme süresi gösteriliyor.

PROJE SONUCU

“Maze Solver oyununa yapay zeka yaklaşımı” projemizin sonucunda belirlediğimiz probleme başarılı bir şekilde çözüm getiren farklı algoritmaları gerçekleyerek sonuçlarımız raporlanmıştır. Oluşturduğumuz 2 farklı algoritma ve bu algoritmalarında birinde kullanılan sezgisel fonksiyonun başarımları hesaplanıp iki farklı algoritmanın sonuçları kıyaslanmıştır. Programımız içerisinde iki farklı ana bölüm bulunmaktadır.

1) DFS ile maze oluşturulması
2) Seçilen maze üzerinde farklı çıkış noktaları ve engeller yerleştirilerek hem DFS hem A* search ile çözülmesi

Kullanıcıya ilk aşamada yeni bir maze oluşturmak isteyip istemediği sorulur. Kullanıcı yeni bir maze oluşturmayı seçtiğinde program DFS ile random bir şekilde maze oluşturmaktadır. Oluşturulan maze mevcut mazelerin bulunduğu yere eklenmektedir ve program oluşturulan maze sonucunu kullanıcıya dönmektedir. Eğer kullanıcı mevcut maze üzerinden devam etmek isterse program kullanıcıyı ikinci aşamaya yönlendirmektedir. Kullanıcı ikinci aşamaya geçtiğinde ilk önce program elinde bulunan maze örneklerini kullanıcıya sıralamaktadır. Kullanıcı bunlardan birini seçmektedir. Daha sonrasında program kullanıcıya maze içinde kaç adet çıkış noktası bulunması gerektiğini sormaktadır. Bundan sonra ise seçilen maze görselini kullanıcıya göstermektedir. Kullanıcı maze üzerinden başlangıç ve bitiş noktalarını seçmektedir. Ardından maze üzerine istediği yerlere manuel olarak engeller veya duvarlar yerleştirmekte ve eğer isterse engeli veya duvarları geri alabilmektedir. Tüm bu işlemler tamamlandığında kullanıcı search işlemini başlatmak için enter tuşuna basmaktadır. Program A* search işlemini başlatmaktadır. A* search ile bir çözüm bulunursa kullanıcıya maze görseli üzerinde gösterilmektedir. Eğer bir çözüm yoksa herhangi bir path çizilmemektedir. Kullanıcı maze ekranını kapattıktan sonra aynı işlemler DFS için de yapıldıktan sonra kullanıcı DFS ekranını da kapattıktan sonra ekrana iki farklı search yönteminin kıyaslaması gösterilmektedir. Bunların içinde ise çözüm yolunun uzunluğu, çalışma süresince kaç adet node expand edildiği ve çalışma süresi gösterilmektedir. Eğer çözüm yoksa çözüm yolu uzunluğu sıfır olarak gösterilmektedir. Tüm bunların yanı sıra, oyunumuzu diğer benzer türleri ile karşılaştırıldığında çalışma zamanı ve heuristic seçimimizle başarılı bir sonuca ulaştığımızı raporumuz da değerlendirdik. Oyun seçim zamanından başlayarak projemizin bitimine kadar heuristic seçimimiz, implementasyonumuz, oyumuzun user-friendly olması ve çalışma zamanının gelişim süreçlerini göz önüne aldığımızda projemizin uygun ve ikna edici olduğuna karar verdik

KAYNAKLAR

1. https://medium.com/@arjunatlast/solving-the-maze-problem-varieties-cc908f802a3a
2. https://en.wikipedia.org/wiki/Maze
3. https://www.cantab.net/users/michael.behrend/repubs/maze_maths/pages/euler_en.html
4. https://en.wikipedia.org/wiki/Euclidean_distance