MTM2602
YAPAY ZEKAYA GİRİŞ
DÖNEM PROJESİ RAPORU

SAYISAL LABİRENTTE EN KISA YOL

ANIL ERĞAN
BEDİA SULTAN DEMET
ÖMER KIRAÇ
İBRAHİM ÇAKMAK
A. SELÇUK KINALI

1. GİRİŞ

Bu raporda, Yıldız Teknik Üniversitesi Matematik Mühendisliği lisans programı dahilinde MTM2602 kodlu Yapay Zekaya Giriş dersinin kapsamı altında grup çalışması ile yürütülen "Sayısal Labirentte En Kısa Yol" probleminin çözümü için geliştirilen yazılım projesinin detayları yer almaktadır. Çalışma kapsamında, Bilgisiz (Uninformed) ve Bilgili (Informed) arama algoritmaları Python programlama dilinde, ilgili problemin çözümü gözetilerek kodlanmıştır.

2. PROBLEMİN TANIMI

Problem kabaca, sayısal bir labirent içerisinde belli bir konumdan başlayan ajanın her bir durumda, en fazla içerisinde bulunduğu konumun kısıtlandırdığı sayı kadar birim yol alacak şekilde yukarı sağa aşağı ve sola amaçlanan konuma gelene dek hareket etmesi olarak tanımlanabilir. Problem, daha teknik bir ifade ile aşağıdaki şekilde tanımlanabilir.

Labirentler sabit birer matris olarak problemin başında tanımlanabilir. Belirtilmelidir ki, indeks numaraları sıfırdan başlayacak şekilde tanımlanmıştır. Dolayısıyla Şekil 1'de ele alınan matriste en sol üst köşe (0,0) indeksine karşılık gelecektir. Şekil 1'de gösterildiği gibi, her bir matris indeksi, belli bir sayı barındırmaktadır. Bu sayılar, problemin tanımında bahsedilen hareket edilecek birim kısıtlamalarıdır. '-1' ise, amaç duruma karşılık gelmektedir. '0'lar ise labirentin içerisindeki duvarlardır. Bu duvarların özelliği, ajanın o indekste bulunamayacağını yahut herhangi bir durumda o duvarlar üzerinden atlayamayacağı anlamına gelir. Dolayısıyla labirenti kısıtlayıcı bir etkileri vardır.

Ajanın içerisinde bulunduğu durum ve başlangıç durumu bir indeks numarası ile gösterilmiştir. Bu durumlar 'tuple' veri tipinde '(1,2)', '(4,2)' formunda saklanmıştır. Python'da tuple veri yapısının özelliği, içeriğinin bir kez oluşturulduktan sonra değiştirilemez olmasıdır.

Aksiyonlar, her bir durum için doğrultu ve adım birimi bilgilerini döndürmelidir. Bir durum içerisinde gerçekleşebilecek tüm aksiyonlar, o an ki durumun çocuk düğümleri olarak düşünülebilir. Mesela, aşağıda verilen (Şekil 1) matris göz önünde bulundurulduğunda ajanın (8,1) konumunda bulunduğu durumda, gerçekleştirebileceği olası aksiyonlar [(9,1), (0,8)] olacak şekilde içerisinde tuple tipinde indeks içeriği olan liste şekilde tutulmaktadır. Dikkat çekilmesi gereken bir başka nokta, ajanın aksiyonları araştırırken, doğrultu bazında saat yönünde önceliklendirme yapması (yukarı, sağa, aşağı, sola), adım birimi bazında ise kısa mesafeye öncelik vermesidir. Örneğin ajan bir konumda 2 birim hareket etme şansı olsa bile bir birim hareket etme şansı olduğu konuma öncelik vermektedir.

```
MAZE_L = array([
    [1,2,4,2,0,0,0,2,0,3,1,1,2,2,1,5],
    [1,5,1,1,2,1,1,4,0,1,2,0,2,1,1,1],
    [1,0,4,0,0,0,1,1,1,1,0,1,3,0,0,1],
    [5,0,2,1,1,1,0,2,1,0,1,4,1,4,1,0],
    [0,1,2,0,2,0,1,1,0,2,1,2,1,0,1,4],
    [3,5,1,0,1,1,2,0,2,1,0,0,0,5,0,1],
    [2,0,3,1,2,1,0,2,5,0,1,2,2,1,0,1],
    [1,2,0,1,0,2,2,1,0,2,1,0,0,2,0,2],
    [2,5,0,0,1,1,0,2,3,4,0,1,0,1,0,1],
    [4,0,5,1,1,0,1,1,2,0,1,1,1,3,2,0],
    [2,1,1,0,2,2,0,2,4,2,0,0,2,2,1,1],
    [0,0,3,4,3,1,1,3,0,4,1,1,1,0,1,-1]
])
```

Şekil 1

3. DOSYALAR VE FONKSİYONLAR

Program, klasör içerisinde yer alan '.py' uzantılı iki dosya ve aynı dizinde yer alan Informed, Uninformed klasörleri içerisinde yer alan '.py' uzantılı toplamda 7 adet algoritma kısaltma ismi ile oluşturulmuş dosyayı kapsamaktadır. ,

3.1 'main.py' Dosyası

Bu dosya, programın çalıştırılacağı ana dosyadır. Öncelikle diğer dosyaların içerisindeki algoritmaları içe aktarır. Sayısal Labirentler bu dosya içerisinde manuel olarak oluşturulmuştur. Bu labirentlerin oluşturulmasında 'numpy' kütüphanesindeki array metodundan faydalanılmıştır. İlgili algoritma sınıflarından oluşturulan objeler, sınıfların miras aldığı ana sınıf olan 'Searching' içerisinde yer alan 'train' metodunu içerisine labirent parametresini matris formunda vererek çağırır. Böylece optimizasyon işlemi başlamış olur.

3.2 'search.py' Dosyası

Bu dosya, içerisinde tüm algoritmaların miras aldığı ana sınıfı barındırır. Bu sınıf 'Searching' ismi ile tanımlanmıştır. Searching sınıfı, Python programlama dilinin sözdiziminde 'def __init__()' şeklinde tanımlanan yapıcı metodunda tüm algoritmaların ihtiyaç duyduğu değişkenleri tanımlar. Bu değişkenler Python programlama dilinde 'self' ifadesini başına alarak tanımlanır. Böylece sınıfın gerek kendi içerisinde gerekse de miras verdiği sınıflar içerisinde bu değişkenler kullanılabilir. Yazının devamında bu değişkenler kısaca açıklanacaktır.

MAZE, sınıfın tanımlanırken parametre olarak aldığı bir değişkendir, sayısal labirentin kendisidir. Numpy Array (Matris) formundadır. Algoritmaların farklı labirentler üzerinde kolayca uygulanabilmesi adına parametre olarak tanımlanmıştır.

ACTIONS, tuple veri tipinde ve saat yönünde olacak şekilde aksiyon doğrultuları string tipinde tanımlamıştır.

ML, 'Maze Limit' kısaltması olarak bu ismi almıştır. Labirent'in köşe sınırlarını göz önünde bulundurarak aksiyonların sınırlandırılması işleminde kullanılmak üzere tasarlanmıştır.

agent, tuple veri tipinde bir koordinat bilgisi içermektedir. Ajanın içerisinde bulunduğu koordinatı betimlemektedir.

path, algoritmanın şimdiye dek geçtiği tüm yolların tutulduğu liste bir veri tipidir. Bu yollar tuple veri tipindedir ve içlerinde yine tuple tipinde koordinat bilgileri barındırmaktadır. Bu tuple tipindeki yolların içeriğindeki tuple tipinde koordinat bilgileri soldan sağa doğru eski koordinattan güncel koordinata gidecek şekilde sıralanmıştır. Örnek olması açısından aşağıda bir duruma ait path bilgisi örneklenmiştir.

path = [((0,0), (1,0), (2,0)), ((0,0), (1,0), (1,1))]

Yukarıda yer alan bu örnekte, path değişkeni henüz iki adet yolu içermektedir. Bu yollardan ilki ((0,0), (1,0), (2,0)) yolu, ikincisi ise ((0,0), (1,0), (1,1)) yoludur. İlk yol üzerinden örneklenmesi gerekirse, ajan ilk olarak (0,0) konumundan başlamış, (1,0) konumundan geçmek suretiyle (2,0) konumuna varmıştır.

value, ajanın içerisinde bulunduğu indeksin sayısal değeridir. Program, ajanın bir sonraki olası durumlarını hesaplarken adım sayısı bu değer ile kısıtlanır.

frontiers, arama algoritmalarında 'frontier' olarak anılan kavrama karşılık gelmektedir. Aramanın önceliğini belirtir. Farklı algoritmalar bu değişkeni farklı şekillerde biçimlendirecektir.

explore_set, frontierde olan fakat çıkartılan değişkenlerin tutulduğu bir yapıdır. Algoritmanın geçişini hatırlamasını sağlar. Python programlama dilinde 'set' olarak anılan veri tipindedir. Sırasızdır. Zira sırasının da pek bir önemi yoktur. Altı çizilmesi gereken nokta, bu yapının hash_table veri tipinde oluşturulmasının daha optimal olabileceğidir.

epoch, her bir algoritmanın kullandığı döngüsel yapının kaçıncı iterasyonda olduğunu tanımlar.

node_created, algoritmanın çalışma zamanı boyunca oluşturduğu toplam düğüm sayısını tarif eder.

Searching sınıfı, aynı zamanda tüm algoritmaların ortak olarak kullandığı bir takım fonksiyonları barındırır. Bu fonksiyonlar aşağıda kısaca açıklanmıştır.

move; acion ve value parametrelerini alan ve bunlara göre integer tipinde bir doğrultu ve tuple tipinde bir adım sayısı bilgisi döndüren fonksiyondur. Örnek olarak, action = 'DOWN' ve value = 3 parametrelerini almasına karşın, 'DOWN' matris üzerinde satır sayısını değiştireceğinden doğrultu olarak 0, ve adım sayısı olarak (1,2,3) değerlerini tuple tipinde döndürür. Burada adım sayısı değerleri negatif değerler olabilmektedir. 'DOWN' doğrultusuna hareket etme işleminin satır sayısında pozitif bir değişim yaratıyor olması bu örnek için pozitif değerler almasına sebep olmuştur.

frontier_elimination; olası aksiyonların gerekli koşulları sağlaması dahilinde legal hale getirilerek frontier'a eklenmeye hazır sinyalini vermesini sağlar. nf, acts ve dir parametrelerini alır 'nf' new frontier isminin kısaltması olan bir parametredir, 'acts' içerisinde bulunulan durum için 'dir' parametresi yönünde olası tüm aksiyonları içeren bir tuple'dır.

wall_detector; frontier elimination metodu tarafından çağrılır. Aksiyonun legal olması için gerekli koşullardan birinin sağlanıp sağlanmadığını kontrol eder ve bool tipinde bir değer döndürür. Bu koşul, aksiyon doğrultusunda adım sayısı boyunca ajanın önüne bir duvarın çıkıp çıkmayacağıdır. Bu durum söz konusu ise, aksiyonları kısıtlar.

action_path; bu fonksiyon 'current_state' ve 'next_state' parametrelerini alır. Bu iki indeks değeri arasında kalan yolu tuple tipinde döndürür. frontier_elimination fonksiyonu içerisinde duvar yol boyunca bir duvar olup olmadığının tespit edilmesi işleminde kullanılmak üzere tasarlanmıştır.

straight_line_distance; bu fonksiyon bir heuristic fonksiyona karşılık gelir (h(x)). 'Açgözlü En İyi Öncelikli Arama' ve 'A* Arama' algoritmaları tarafından kullanılmak için tasarlanmıştır. Belli bir indeks değerindeki hücrenin amaç hücreye (en sağ alt indeksli hücre) olan en kısa uzaklığını ölçer ve döndürür.

path_cost; bu fonksiyon başlangıç durumunun indeksinin içerisinde bulunulan indekse uzaklığını tanımlar ve döndürür. 'A* Arama' ve Sabit Maliyetli Arama' tarafından kullanılmak üzere tasarlanmıştır.

train; bu fonksiyon tüm algoritmaların eğitilmesi için tasarlanmış ana fonksiyondur. İçerisinde sonsuz bir döngüde bir arama başlatır ve gerekli koşul(lar) sağlandığında döngüyü sonlandırır. Böylece arama tamamlanmış olur. Algoritmanın doğası gereği, nihayetinde ya bir çözüm ya da bir hata döndürür. Tüm bu döngü süresince her bir durumda; frontier bilgisini, frontierdan en son çıkartılan koordinat bilgisini, explore_set'i (keşif seti) döndürür ve iterasyon sayısını ekrana bastırır. Şayet algoritma amaç durumuna ulaştıysa, algoritmanın çalıştırılması süresince geçen zamanı, bellekte tutulan toplam düğüm sayısını ve çözüm yolunu ekrana bastırır.

3.3 Uninformed ve Informed Klasörleri İçerisinde Yer Alan Algoritmalar

Informed klasörü içerisinde yer alan 'AStar.py', 'GBFS' isimli algoritma dosyaları sırasıyla A* Arama Algoritması, Aç Gözlü En İyi Öncelikli Arama algoritmalarına karşılık gelmektedirler. Uninformed klasörü içerisinde yer alan 'BFS.py', 'DFS.py', 'DLS.py', 'IDS.py', 'UCS.py' isimli algoritma dosyaları sırasıyla Yayılım Öncelikli Arama, Derin Öncelikli Arama, Derin Limitli Arama, İteratif Derinleşen Derin Öncelikli Arama, Sabit Maliyet Arama algoritmalarına karşılık gelmektedir. Programlamanın her aşamasında olduğu gibi dosya isimlendirmelerinde de algoritmalarının İngilizce terminolojide anılan isimlerinin kısaltmaları

kullanılmıştır. Her bir dosya, aldığı algoritma isminin açılımını isim olarak almış sınıfları içerisinde barındırır.

Tüm bu sınıflar, içerisinde temelinde iki ana fonksiyon bulundurur. Bu fonksiyonlar 'update_frontiers' ve 'update_path' olarak isimlendirilmiştir. Bu fonksiyonlar, her ne kadar tüm algoritma sınıflarında bulunsa da belli başlı temel farklılıkları içerisinde barındırır. Bu sebeple her bir algoritma için özel olarak oluşturulmuştur. Burada belirtilmelidir ki, daha ileri seviye bir programlama sergilene bilmesi adına fonksiyonlar, ortak algoritmik basamakları ana süper sınıf içerisinde dekoratör metot kullanımı ile içerisinde barındırabilir. Lakin, kodda bu ileri seviye adımlar sergilenmemiştir.

Gözlemlenebileceği üzere bazı algoritma dosyaları bu iki ana metot dışında, farklı metotları da içerisinde barındırmaktadır. Aşağıda her bir algoritma içerisinde gerçekleşen işlemler, farklılıklar vurgulanarak kısaca açıklanmıştır.

Uninformed klasörü içerisindeki dosyalar:

'BFS.py' dosyası içerisinde gerçekleştirilen frontier ve yol güncellemesi, frontier'ın içerisinde ilk eklenen koordinatın çıkartılması ve onun alt düğümlerinin frontier içerisinde sondan eklenmesi şeklinde tanımlanabilir.

'UCS.py' dosyası içerisinde gerçekleştirilen frontier ve yol güncellemesi, frontier'ların oluşturulurken, her bir koordinat için maliyet bilgisini de barındırmasını sağlar. Bu maliyet bilgisi, Searching içerisinde yer alan 'path_cost' metodu ile hesaplanır. Akabinde, oluşturulmuş frontier, maliyet bilgisini gözeterek en küçük maliyetli koordinat bilgileri en solda olacak şekilde sıralanır. frontier'dan eleman çıkarma işlemi, ve yeni elemanların eklenmesi işlemi BFS'deki ile aynıdır. Tabi ekleme işleminden hemen sonra bir sıralama işlemi de gerçekleştiği bir farklılık olarak yeniden vurgulanmalıdır.

'DFS.py' dosyası içerisinde gerçekleştirilen frontier ve yol güncellemesi, frontier'ın içerisinde ilk eklenen koordinatın çıkarılması işlemini kapsar. Lakin BFS'den farklı olarak, yeni eklenen frontier'lar frontier listesinin sonuna değil başına eklenir.

'DLS.py' dosyası içerisinde gerçekleştirilen frontier ve yol güncellemesi, DFS ile neredeyse birebir aynıdır. Tek fark, DLS yeni alt düğümler oluşturmadan önce, alt düğümler oluşturacağı yolun uzunluğuna bakar. Eğer uzunluk, parametre olarak alınan limit bilgisinden küçük eşitse DFS ile aynı işlemi uygular. Aksi taktirde bu yol için yeni alt düğümler oluşturmayacaktır ve bu düğümü (maksimum derinliğe ulaşmasına rağmen çözümü bulamadığı için) path listesinden çıkaracaktır.

'IDS.py' dosyası içerisinde gerçekleştirilen frontier ve yol güncellemesi, DLS ile algoritmik açıdan birebir aynıdır. Fakat DLS'de olduğu gibi dışarıdan limit parametresi almaz. Limit değerini 1'den başlatarak her bir optimizasyon işlemi bitiminde arttırır ve algoritmayı yeniden başlatır. Günün sonunda, algoritma bir çözüme ulaşırsa durur, ve çözümü kaçıncı limit değerinde ulaştığı bilgisini de beraberinde bastırır.

Informed klasörü içerisindeki dosyalar:

'GBFS.py' dosyası içerisinde gerçekleştirilen frontier ve yol güncellemesi, UCS'de yapılan işlemlerin benzeridir. Algoritmik olarak bir farklılık içermemesine karşın GBFS, maliyeti Searching içerisinde yer alan 'straight_line_distance' metodu ile hesaplar.

'AStar.py' dosyası içerisinde gerçekleştirilen frontier ve yol güncellemesi, diğer maliyet tabanlı arama (GBFS, UCS) algoritmalarına benzemesine karşın, bu algoritmaların kullandığı maliyet hesaplama yöntemlerinin ortalamasını alır.

5. SİMÜLASYONLAR

Bu bölümde, iki farklı labirent için (Bölüm 6) simülasyon sonuçlarına dair detaylar yer almaktadır. Her bir simülasyon sonucu, algoritmanın 5 kez çalıştırılması sonucunda elde ettiği değerleri ve ortalamasını kapsamaktadır.

Tablo 1'de yer alan sonuçlar algoritmaların sonuca ulaşma zamanlarını temsil etmektedir. Her bir eğitim modelin yeniden çalıştırılması anlamına gelmektedir. Sonuca ulaşma süreleri genellikle bir saniyenin altında olduğundan her bir modelin beş farklı defa çalıştırılması suretiyle çalıştırma sürelerinin ortalaması alınmıştır. Tablo ayrıca, modellerin çalışması süresince ürettiği ve bellekte tuttuğu düğüm sayılarını vermiştir. Burada dikkat çekilmesi gereken nokta, derin arama temelli algoritmaların (DFS, DLS, IDS) normal şartlarda bir düğüm silme işlemi gerçekleştiriyor olmaları gerekmesine karşın, kodda bu durum işlenmemiştir. Algoritmalar yorumlanırken bu durum göz önünde bulundurulacaktır.

L Labirentinin eğitim süreleri ve düğüm sayıları göz önünde bulundurulduğunda sırasıyla aşağıdaki gibi bir sıralama yapılabilmektedir.

GBFS < DLS < DFS < AStar < UCS < BFS < IDS

GBFS < DLS = IDS < DFS < AStar < BFS < UCS

XL Labirentinin eğitim süreleri ve düğüm sayıları göz önünde bulundurulduğunda sırasıyla aşağıdaki gibi bir sıralama yapılabilmektedir.

GBFS < DLS < DFS < AStar < UCS < BFS < IDS

GBFS < DLS = IDS < DFS < AStar < BFS < UCS

Bu sıralamalar üzerinden yapılabilecek en temel yorum, sıralamanın başını çeken algoritmaların bellek avantajları ve çalışma süreleri avantajları birbirlerine paralel ilerlemiştir. Sıralamak öyle göstermektedir ki, derin arama tabanlı algoritmalar (DLS, DFS) ilgili problemin çözümünde gerek düğüm sayısı gerekse de çalışma süreleri açısından avantajlıdır. Aç gözlü yaklaşım ise her iki labirent için gerek bellek gerekse de zaman açısından açık ara en avantajlı seçimdir. Bu, problemin doğasını gereği pek çok farklı senaryoda çözüme ulaşabilmesinin ürünüdür. Labirent, aç gözlü yaklaşımları cezalandırılacak şekilde de tasarlanabilir. İki labirent için de BFS, UCS'ye göre eğitim açısından dezavantajlı olsa da bellek tüketimi açısından avantaj sağlamıştır

L LABIRENT							
	1. Eğitim	2. Eğitim	3. Eğitim	4. Eğitim	5. Eğitim	Ortalama	Düğüm Sayısı
BFS	0.13052773	0.134619	0.12374163	0.16015744	0.11332059	0.13247	132
UCS	0.14408088	0.12476301	0.11639071	0.1367619	0.13295937	0.13099	136
DFS	0.09888864	0.07078099	0.07115817	0.0719924	0.10758543	0.08408	112
DLS (22)	0.06122255	0.06645322	0.08725834	0.06117749	0.07875848	0.07097	96
IDS	1.08324862	1.0826807	1.67472315	1.13409686	1.18112969	1.23118	96
GBFS	0.02491283	0.02335906	0.0204215	0.02739811	0.02068329	0.02335	58
AStar	0.07232523	0.07711744	0.08113337	0.10362577	0.08803082	0.08445	114
XL LABIRENT							
	1. Eğitim	2. Eğitim	3. Eğitim	4. Eğitim	5. Eğitim	Ortalama	Düğüm Sayısı
BFS	0.84151578	0.9874568	1.20392132	1.25760722	1.06011605	1.07012	411
UCS	1.02083492	0.86201787	1.01989841	0.94216347	0.95731091	0.96045	418
DFS	0.52045512	0.640414	0.5429554	0.69309807	0.62920547	0.60523	372
DLS (51)	0.48093486	0.45774817	0.4664948	0.74314165	0.6940093	0.56847	339
IDS	20.02282619	18.50650954	19.92092919	22.02378392	20.23514771	20.14184	339
GBFS	0.1799047	0.30217052	0.17437959	0.24337745	0.28180456	0.23633	219
AStar	0.82424688	0.87987757	0.83719778	0.96994996	0.88853621	0.87996	403

Tablo 1

6. SONUÇ

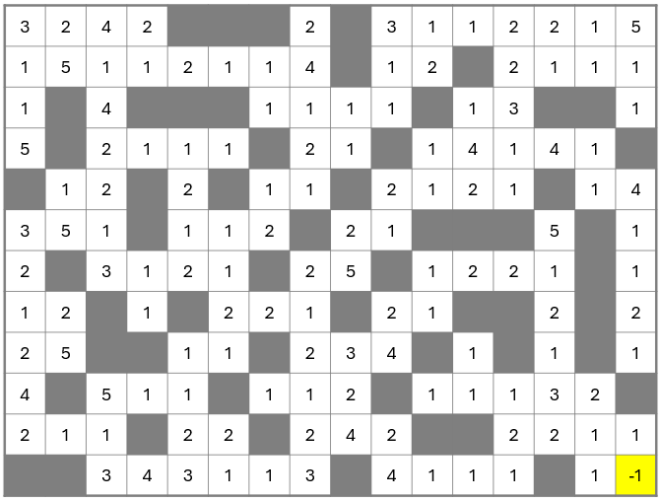
Bu rapor, Yıldız Teknik Üniversitesi Matematik Mühendisliği lisans programı dahilinde gerçekleştirilen "Sayısal Labirentte En Kısa Yol" problemi üzerine yapılan çalışmanın detaylarını içermektedir. Çalışma kapsamında, MTM2602 kodlu Yapay Zekaya Giriş dersinin bir parçası olarak grup çalışmasıyla geliştirilen yazılım projesi incelenmiştir. Problem tanımı, dosya ve fonksiyonlar hakkında detaylı bilgiler sunulmuştur. Ayrıca, Uninformed ve Informed arama algoritmalarının Python programlama dili kullanılarak nasıl kodlandığına dair açıklamalar bulunmaktadır.

Simülasyon sonuçlarına göre, çeşitli algoritmaların performansı değerlendirilmiştir. Bu değerlendirme, algoritmaların çalışma süreleri ve bellek kullanımları üzerinden yapılmıştır. Elde edilen sonuçlar, derin arama tabanlı algoritmaların belirli senaryolarda avantajlı olduğunu, ancak aç gözlü yaklaşımların genelde en avantajlı seçenek olduğunu göstermektedir.

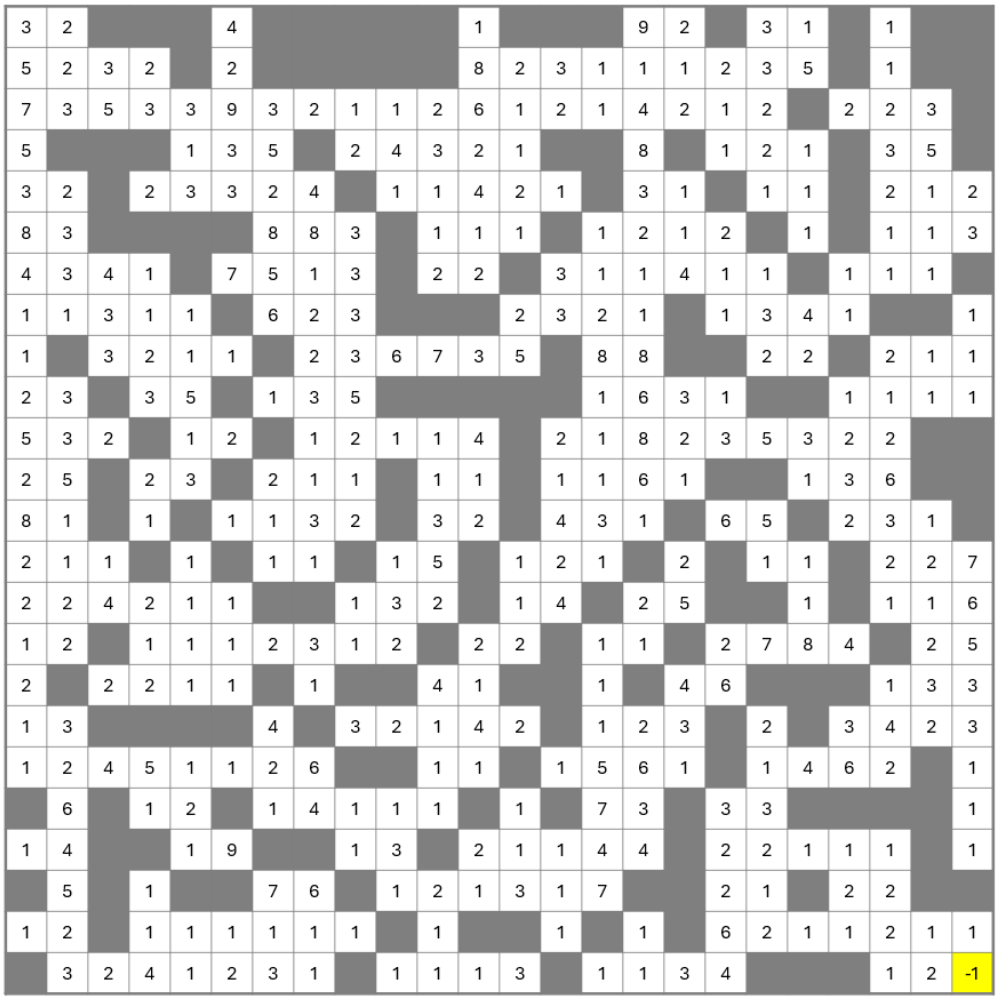
Sonuç olarak, bu çalışma yapay zekâ ve algoritmaların pratik uygulamalarıyla ilgilenenler için değerli bir kaynak sunmaktadır. Ayrıca, farklı arama algoritmalarının avantajları ve dezavantajlarına dair önemli bir içgörü sağlamaktadır.

7. LABİRENTLER

Aşağıda algoritmaların uygulandığı labirent prototipleri yer almaktadır.



Şekil 2, L Labirent (12x16)



Şekil 3, XL Labirent (24x24)