İSTANBUL MEDENİYET ÜNİVERSİTESİ

MÜHENDİSLİK VE DOĞA BİLİMLERİ FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
VERİ BİLİMİNE GİRİŞ DERSİ FİNAL PROJESİ

Advanced Maze Generator-Solver

24120205038,
Abdullah Yasin
YILMAZ
24120205076, Ahmet
MUSLEH
24120205028, Ahmet
IŞIK

Ders Sorumlusu

Dr. Öğr. Üyesi Muhammet Sinan BAŞARSLAN

Ocak, 2026

İstanbul Medeniyet Üniversitesi, İstanbul

İÇİNDEKİLER

10

1. GİRİŞ

Bu projeye ilk başladığımızda amacımız oyuncu kontrollü bir karakterle bir tür labirent oyunu yapmaktı fakat zamanla çizgi grafiklerinde farklı dolaşım algoritmalarının performansını gösteren eden bir simülasyona dönüştü. Burada amacımız bu konuyu yeni öğrenen veya bilip pratikte bu algoritmaların nasıl performans gösterdiğini merak edenler için kullanıcı dostu bir program yapmaktır.

Labirenti oluşturduktan sonra kullanıcı 4 farklı dolaşım algoritmasını kullanarak farklı performans testleri yapabilir, bu yöntemlerin teorik olarak nasıl çalıştığını görebilir ve yapılan tüm testleri karşılaştırabilir.

Şu anki haliyle programda kullanılan dolaşım algoritmaları aşağıda verilmiştir

- Breadth First Search
- Depth First Search
- Dijkstra
- A*

Bunların dışında farklı bağlamlarda test sonuçlarını görmek için labirent oluşturulmadan önce kullanıcıya 3 seçenek sunulur:

- Ağırlık: Oluşturulan labirentin bazı bağlantıları diğer yollara kıyasla 2 kat daha ağır olur. Bu Dijkstra gibi en ucuz yolu bulan algoritmaların bu faktörü görmezden gelen algoritmalara kıyasla ne kadar
  başarılı olduğunun görülmesini sağlar
- Dinamik Engel: Algoritmalar çalışırken labirentin duvarları değişir. Bu seçenek sayesinde her algoritma farklı bir labirente uygulanmış olur.
- Çoklu Hedef: Labirente 2 tane daha hedef koyulur, testin bitmesi için hepsi bulunmalıdır. Bu labirentin tamamının kullanılmasını ve testlerin daha uzun sürmesini sağlar ki böylece sonuçlar daha etraflıca olur.

2. MATERYAL METOT

Projeyi oluştururken kullandığımız programlar Visual Studio Code,Vim,NotePad++,Intellij ve GitHub oldu, bazılarımız kodu terminalden çalıştırırken bazılarımız VS Code'un derleyici eklentilerini kullandı. Sonrasında ise işi otomatikleştirmek için kendi .bat dosyamızı kullandık.
Altyapıyı için dil seçiminde birkaç seçenek arasından dersin de dili olan C'yi seçtik, düşük seviye bir dil olduğu için bu veri yapılarını tasarlarken diğer dillere kıyasla bize daha fazla özgürlük sağladı. Fakat C'de arayüz inşasında tecrübesiz olduğumuz için projenin o kısmı için Java'nın Java Swing kütüphanesini kullandık. Bu projede en zorlandığımız kısım C kod tabanı ile Java arayüzünü bağlayabilmekti.

2.1. Programın Akışı

Program çalıştırıldığında kullanıcıya 2 seçenek sunulur: "Test Oluştur" ve "Test Yükle". Test; oluşturulan labirent, seçilen ayarlar ve labirente uygulanan farklı algoritma testlerinin bir bütünüdür. "Test Oluştur" tuşuna basıldığında önceden bahsedilen 3 seçenek sunulur ve seçim tamamlandıktan sonra program Kruskal algoritmasının ağırlıktan bağımsız rastgele seçim yapan bir versiyonunu kullanarak bir labirent oluşturur ve kullanıcıya gösterir.

Buradan sonra kullanıcı farklı algoritma tuşlarına basarak onların işleyişini görebilir, kullanılan her algoritma sonucu arayüzde bulunan listeye o algoritma örneğinin özeti yazılır. Kullanıcı bu algoritma özetlerini algoritmanın yolu bulmak için gereksindiği adım sayısı veya bulduğu yolun ağırlık miktarına göre sıralayabilir.

Memnun kalındıktan sonra test bir JSON olarak kaydedilir ve tekrardan ana ekrana dönülür. "Test Yükle" seçeneği de bu .json dosyalarını kabul eder.

2.2 Kullanılan Veri Yapıları

Neredeyse kullandığımız tüm algoritmalar farklı veri yapılarına ihtiyaç duydu, bu sebeple hem veri yapılarını hem de bu yapıya ihtiyaç duyan algoritmaları sıralamak açıklayıcı olur.

- Dizi bazlı Queue: Arama algoritmalarından BFS
- Dizi bazlı Stack: Arama algoritmalarından DFS
- Dengeli Ağaç olan Heap (Priority Queue): Arama algoritmalarından Dijkstra ve A*
- Disjoint Set Union: Labirent oluşturmak için kullanılan Kruskal
- Kenar Listesi ve Komşuluk Matrisiyle Graph: Labirentin kendisi

2.3 Kullanılan Yardımcı Algoritmalar ve Harici Kütüphaneler

Projenin birçok yerinde kullanılan yardımcı algoritmalar aşağıda sıralanmıştır.

Algoritma özetlerinin sıralanması için QuickSort ve MergeSort, rastgele labirent jenerasyonu için Fisher-Yates Shuffle ve DSU verimi için path compression ve "Dinamik" ayarında labirentin yapısın bozulmadan değişmesi için BFS algoritmasından yararlanılmıştır.

Bunların dışında kullanılan tek üçüncü parti kütüphanesi C'de JSON oluşturup okunmasında kolaylık sağlayan cJSON[1] kütüphanesidir.

3. UYGULAMA

Bu bölümde programın başlıca algoritmaların açıklamaları ve burada nasıl uygulandıkları açıklanacaktır. En sonda Java Arayüzü ile C kod tabanının nasıl bağlandığının üzerinden geçilecektir

3.1 Algoritmalar

- **Kruskal**

Labirent oluşturmada kullandığımız ana algoritma, normalde 2 adımdan oluşur. Verilen graph'ın tüm kenarlarını azalan ağırlığa göre sırala ve DSU yardımıyla Minimum Spanning Tree yapısı bozulmayacak şekilde bu kenarları sırayla labirente ekle.
Biz ızgara şeklinde bir labirent istediğimiz için başta bu algoritmaya vereceğimiz graph her düğümün kare kabul edilip geometrik bir şekilde komşusu olduğu diğer düğümlere kenarı olan satranç tahtası gibi bir yapı.

Biz tamamen rastgele bir labirent istediğimiz için Kruskal'ı direkt bu haliyle uygulayamazdık, orijinal algoritma ağırlıklı bir graph'tan asgari miktarda ağırlık kullanarak bir tarama ağacı oluşturuyor ve bu birinci adımdaki sıralamadan kaynaklı deterministik bir durum. Bu sebeple eğer ilk adımdaki sıralamayı rastgele yaparsak Spanning Tree yapısını taşıyan tamamen rastgele bir labirent oluşturabiliriz.
Ağırlık hesaplarında her algoritmadan farklı sonuç almak istediğimiz için iki hücre arasında birden fazla yol oluşturmamız gerekti, bunun için labirent oluşturma işlemini en sonunda şansa bağlı olarak bazı hücreler arasında yeni yollar açtık.

Bu rastgele sıralamada kullandığımız algoritma Fisher-Yates Shuffle ya da diğer adıyla Knuth Shuffle'dır. Şekil 3.1'de bu algoritmanın kodunu görebilirsiniz

Şekil 3.1 Burada wallList tüm karelerin bağlı olduğu graph'ın kenar listesidir.

```
//random sort
for (int i = wallCount - 1; i > 0; i--) {
    int j = rand() % (i + 1);
    edge temp = wallList[i];
    wallList[i] = wallList[j];
    wallList[j] = temp;
}
```

- **BFS ve DFS**

4 arama algoritmasından bu ikisi uygulama ve teori bakımından en basit olmakla beraber aynı prensip üzerine dayalıdırlar, kaynak düğümünden başlayarak tüm komşuları algoritmaya bağlı olarak farklı veri yapısına eklemek ve bu yapıdan okunan her düğüm için aynı işlemi tekrar etmek.

Bunun yanında ziyaret edilen düğümleri kaydetmek için bir boolean dizisi kullanılır.

BFS ilk en yakın komşuları ziyaret ettiği için Queue kullanırken DFS sıra sıra tüm yolları bitirirdiği için Stack kullanır.

- **Dijkstra**

Bu algoritma ağırlıklı graphlarda en ucuz yolu bulma üzerine kurulmuştur ve bunun için her düğüme gitmenin ağırlığını gösteren bir uzaklık dizisi ile ağırlık bazlı bir Min Heap kullanır (Priority Queue).

BFS gibi bir kaynaktan başlar ve komşularını kenar ağırlıklarıyla beraber Heap'e ekler. Heap dolu olduğu sürece oradan aldığı her düğüm için uzaklık dizisini günceller ve onun da komşularını Heap'e ekler. Böylece hedefi bulduğunda ona giden en hafif yolu bulmuş olur.

- **A***

A* algoritması, Dijkstra'nın maliyet odaklı kör arama yaklaşımını, hedefe yönelik bir tahmin mekanizmasıyla birleştiren gelişmiş bir arama algoritmasıdır. Dijkstra algoritması başlangıç noktasından itibaren her yöne eşit bir şekilde yayılarak "en ucuz" yolu ararken, A* algoritması hedefin konumunu dikkate alarak aramayı belirli bir yöne kanalize eder.

Bu yönlendirme işlemi her bir düğüm için bir toplam skor (maliyet) hesaplanarak yapılır. Bu skor iki temel bileşenden oluşur:

→ Gerçekleşen Maliyet (g): Başlangıç düğümünden mevcut düğüme kadar kat edilen toplam yol ağırlığıdır.

→ Tahmini Maliyet (h): Mevcut düğümden hedef düğüme kalan mesafenin kuş uçuşu (sezgisel) olarak tahmin edilmesidir.Projemizde labirent ızgara yapısında olduğu için sezgisel tahmin yöntemi olarak **"Manhattan Mesafesi**(dik uzaklık) kullanılmıştır. Bu yöntem, iki nokta arasındaki farkı mutlak değerce hesaplar: $h(n) = |x_{he}d_ef - x_n| + |y_{he}d_ef - y_n|$ formulü ile.

→Final Maliyet (f): $f(n) = g(n) + h(n)$ olarak hesaplanır.

Algoritma, bu iki değerin toplamı olan en düşük skora sahip düğümlere öncelik vererek ilerler. Projemizde sezgisel tahmin olarak "dik uzaklık" (Manhattan mesafesi) kullanılmıştır. Bu yaklaşımın temel avantajı, hedefle ilgisi olmayan ve aksi yönde kalan yolları daha en baştan eleyerek işlem yükünü ciddi oranda azaltmasıdır. Böylece A*, labirent içerisinde hedefe doğru çok daha kararlı ve dar bir alanda tarama yaparak Dijkstra ile aynı en kısa yolu çok daha az düğümü ziyaret ederek bulabilmektedir.

3.2 Seçenekler

Labirent oluşturulmadan önce kullanıcıya sunulan 3 seçenek az önce açıklanan algoritmaları değiştirirler. Bunlardan "Ağırlık" seçeneği sadece Kruskal algoritmasında bazı kenarları daha ağır yaparken "Çoklu Hedef" seçeneği aynı algoritmayı birden fazla kez çağırır. Çalışma mantıkları basit olduğu için bizi daha çok zorlayan "Dinamik" seçeneğini açıklamak istiyoruz.
Bu seçenek işaretlendiğinde 4 algoritma da her adımından sonra labirentin şeklini değiştirir. Bunu yapmak için ilk önce rastgele bir düğüm seçilir (u) ve eğer eklenebiliyorsa ona yeni bir kenar eklenir. Yeni komşu düğümden (v) DFS ile farklı bir yoldan ulaşılır. Bu yeni yolda (u)'ya komşu olan hücrenin (n) (u) ile bağlantısı kesilir ve böylece Spanning Tree yapısı korunacak şekilde labirent değişir. Arama algoritmasının hiçbir hücreyi atlamaması için bu (u) hücresinin yakınlığına bakılır, eğer ki ziyaret edilen düğümlere komşu ise tekrardan sıraya eklenir ve komşularına tekrardan bakılır. Uzakta ise algoritma normal şekilde çalışmaya devam eder.

3.3 Test Sonuçları

GUI'ye bağlantı ve saklama kolaylığı için her arama algoritması bir testResult yapısı döndürür. Şekil 3.2'de görüldüğü gibi bu yapı testin sonucunda elde edilen tüm gerekli bilgileri içerir.

```
typedef struct TestResult{//GUI'ye gonderilecek testResult yani res
    //GUI'de gözükecekler
    char* algo; //algoritmanın ismi
    int steps; //adım sayısı
    int weight; //üzerinden geçilen yolların ağırlık toplamı


    //GUI'ye gönderilecekler
    int* explored; //keşfetme sırası
    int exploredCount;//kaç hücre keşfedildi
    int* result; //bulunan yolun sırası
    int resultCount;//result yolunda kaç hücre var
    int* dynamicChangeIndexes; //duvarın değiştiği turların sırası
    MatrixUpdate* dynamicChangeUpdates; //Duvar değişeceğinde yapılan güncellemeler
    int dynamicChangeCount;//değişim sayısı

}TestResult;
```

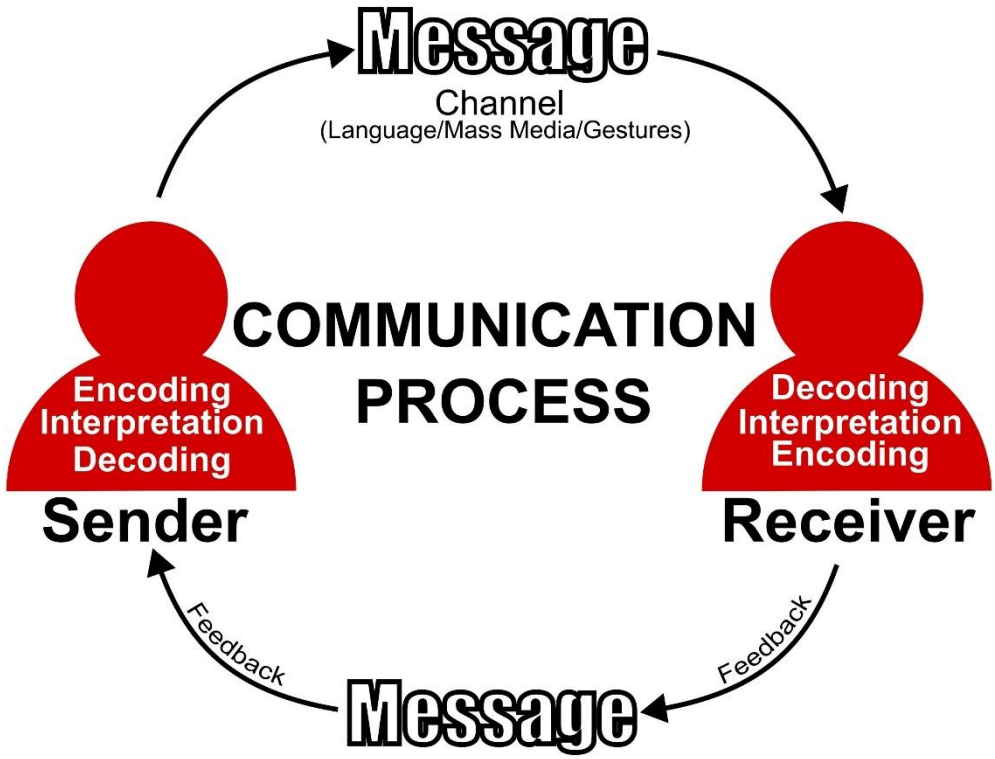
Oluşturulan bu sonuçlar bir listeye eklenir ve farklı şekillerle sıralanabilir.

Yapılan testler sonucunda elde edilen veriler (Bkz yukarıdaki resim), algoritmaların teorik çalışma prensiplerini doğrulamaktadır:

- **BFS (W: 59 | S: 372):** Ağırlık faktörünü dikkate almadan sadece en az hücreden geçen yolu seçmiştir.
- **Dijkstra ve A* (W: 56):** Her iki algoritma da en düşük maliyetli rotayı başarıyla bulmuştur. A* algoritmasının Dijkstra ile aynı maliyete sahip yolu 113 daha az hücre ziyaret ederek (S: 252 vs 365) bulması, sezgisel (heuristic) yaklaşımın başarısını kanıtlamıştır.
- **DFS (W: 62 | S: 51):** Bulduğu ilk yolu rapor eden DFS en yüksek maliyetli sonucu üreterek optimal çözüm garantisi sunmadığını göstermiştir.

3.4 Java C Bağlantısı ve Süreçler Arasındaki İletişim

Proje hızlı ve yüksek performanslı alogritma çalıştırabilme kapasitesi için C dilini backend ve kullanıcı arayüzü yetenekleri için Java Swing'i frontend alarak oluşturulmuştur. Bu ikisinin birlikte çalışabilmesi için Süreçler Arası İletişim(Inter-Process Communication - IPC) tekniklerinden olan Standart Girdi/Çıktı Yönlendirme(Standard I/O Redirection) yöntemi kullanılmıştır. Bu yöntem sıra sıra böyle çalışır:



1- Alt Süreç Oluşturma(Child Process Creation)

Java ana uygulama(parent process) olarak davranır ve C programını(demo.exe'yi) bir alt uygulama(child process) olarak başlatır.

2- İletişim Kanalı Standard I/O Pipes (Standart Akış Boruları)

Standart girdi (stdin) ve standart çıktı (stdout) kanalları üzerinden sağlanır. C programının stdout (standart çıktı) kanalı, Java'nın Input Stream (okuma kanalı) kanalına bağlanır bu şekilde C konsola çıktı atar ve Java okur

3- Veri Protokolü: Text-Based Custom Protocol (Metin Tabanlı Özel Protokol)

İki programın birbirini anlaması için String Parsing(Metin Ayrıştırma) ile Etiketleme kullanıldı Satır başına ayırt edici etiketler koyarak Örnek Olarak

MATRIX -> Harita verisi

DYNA -> Dinamik duvar değişimi

VISIT -> Algoritmanın gezdiği yerler

MAP_READY -> Senkronizasyon sinyali

Böylece Java String.split() ve startsWith() fonksiyonları ile C den gelen mesajları ayırt edebilir ve nesnelere dönüştürür.

4- Tampon Yönetimi

C'de printf metodunu kullandığımız zaman veriyi tamponda tutar Java'ya anlık ulaşması için C'de fflush komutu kullanılır böylece animasyon akıcı hale getirilir.

5- Asenkron G/Ç ve Çoklu İş Parçacığı

C programı çalışırken GUI(arayüz) donmaması için Java programında Demo.exe'yi dinleyen metot ayrı bir Thread(iş parçacığı) üzerinde çalıştırılır. C'den veri geldikçe SwingUtilities.invokeLater() kullanılarak arayüz güncellenir.

4. SONUÇ VE TARTIŞMA

Bu proje sürecinde, teorik olarak öğrendiğimiz veri yapıları ve algoritmaların pratik bir simülasyona dönüştürülmesi aşamasında teknik ve operasyonel anlamda birçok kazanım elde ettik. Projenin en önemli kazanımlarından biri, C dilinden oluşan arka yüz ile Java tabanlı ön yüzün aynı projede harmanlamak olmuştur. Farklı dillerin entegrasyonu ve ekip içi koordinasyonda Git/GitHub[2] kullanımı, profesyonel yazılım geliştirme süreçlerine dair kariyer boyu sürecek bir deneyim sağlamıştır.

Teknik açıdan yapılan testler ve gözlemler sonucunda şu sonuçlara varılmıştır:

- **BFS ve DFS:** Labirentin tüm yollarını keşfetme konusunda kararlı oldukları, ancak ağırlıklı yollar (maliyet faktörü) devreye girdiğinde optimal çözümü sunamadıkları gözlemlenmiştir.
- **Dijkstra:** Ağırlıklı bağlantıların olduğu senaryolarda en düşük maliyetli yolu bulma konusunda en güvenilir sonucu vermiş, ancak A* algoritmasına göre çok daha geniş bir alanı taradığı (ziyaret edilen düğüm sayısının fazlalığı) saptanmıştır.
- **A***: Sezgisel (heuristic) yaklaşımı sayesinde hedefe yönelimli bir arama gerçekleştirmiş ve Dijkstra ile aynı en kısa yolu çok daha az düğümü ziyaret ederek çok daha kısa sürede bulmuştur.
- **Dinamik Engel Modu:** Algoritmaların değişen çevre koşullarına (duvarların değişmesi) anlık olarak nasıl uyum sağladığı test edilmiştir. Bu modda, yolun tamamen baştan hesaplanması yerine etkilenen komşulukların güncellenmesi stratejisinin işlem maliyetini düşürdüğü tartışılmıştır.

Sonuç olarak projenin, bilgisayar bilimleri eğitiminde büyük bir eksiklik olan "algoritma görselleştirme" sorununa, Graph arama algoritmaları özelinde etkili ve kullanıcı dostu bir çözüm sunduğuna inanıyoruz. Proje; interaktif yapısı, JSON tabanlı test kaydetme/yükleme özelliği ve farklı senaryo seçenekleriyle (ağırlık, dinamik engel, çoklu hedef) hem öğrenciler hem de konuyu merak edenler için nitelikli bir ek kaynak niteliğindedir. Gelecekte bu çalışma, daha farklı labirent jenerasyon algoritmaları ve üç boyutlu görselleştirme araçlarıyla daha da ileriye taşınabilir.

Algoritma Karmaşıklığı (Big O Notation) Tablosu

İSİM	ZAMAN KARMAŞIKLIK (EN İYİ)	ZAMAN KARMAŞIKLIK (ORTA)	ZAMAN KARMAŞIKLIK (EN KÖTÜ)	UZAY KARMAŞIKLIK
enqueue() (Queue)	**O(1)**	**O(1)**	**O(1)**	**O(1)**
dequeue() (Queue)	**O(1)**	**O(1)**	**O(1)**	**O(1)**
push() (Stack)	**O(1)**	**O(1)**	**O(1)**	**O(1)**

pop() (Stack)	**O(1)**	**O(1)**	**O(1)**	**O(1)**
insert() (Binary Heap)	**O(1)**	**O(1)**	**O(logn)**	**O(1)**
extract() (Binary Heap)	**O(logn)**	**O(logn)**	**O(logn)**	**O(1)**
find() (DSU)	**O(α(n))**	**O(α(n))**	**O(α(n))**	**O(logn)**
union() (DSU)	**O(α(n))**	**O(α(n))**	**O(α(n))**	**O(1)**
Fisher-Yates Shuffle	**O(n)**	**O(n)**	**O(n)**	**O(1)**
QuickSort	**O(nlogn)**	**O(nlogn)**	**O(n^2)**	**O(logn)**
MergeSort	**O(nlogn)**	**O(nlogn)**	**O(nlogn)**	**O(n)**
BFS	**O(V^2)**	**O(V^2)**	**O(V^2)**	**O(V)**
DFS	**O(V^2)**	**O(V^2)**	**O(V^2)**	**O(V)**
Dijkstra	**O(V^2)**	**O(V^2)**	**O(V^2)**	**O(V)**
A*	**O(V^2)**	**O(V^2)**	**O(V^2)**	**O(V)**
Kruskal	**O(V* α(n))**	**O(V* α(n))**	**O(V* α(n))**	**O(V)**

6. KAYNAKÇA

[1] https://github.com/DaveGamble/cJSON

[2] https://github.com/ozymandias58/Advanced-Maze-Generator-Solver

[3] https://www.geeksforgeeks.org/c/c-programming-language