DOKUZ EYLÜL ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

İKİ BOYUTLU LABİRENTLERDE ARAMA ALGORİTMALARININ İNCELENMESİ

Merve MENEVŞE

Eylül, 2024

İZMİR

İKİ BOYUTLU LABİRENTLERDE ARAMA ALGORİTMALARININ İNCELENMESİ

Dokuz Eylül Üniversitesi Fen Bilimleri Enstitüsü

Yüksek Lisans Tezi

Bilgisayar Bilimleri Anabilim Dalı, Bilgisayar Bilimleri Programı

Merve MENEVŞE

Eylül, 2024

İZMİR

YÜKSEK LİSANS TEZİ SINAV SONUÇ FORMU

MERVE MENEVŞE tarafından **DR.ÖĞR.ÜYESİ AYŞE ÖVGÜ KINAY** yönetiminde hazırlanan **" İKİ BOYUTLU LABİRENTLERDE ARAMA ALGORİTMALARININ İNCELENMESİ"** başlıklı tez tarafımızdan okunmuş, kapsamı ve niteliği açısından bir Yüksek Lisans tezi olarak kabul edilmiştir.

Dr.Öğr.Üyesi Ayşe Övgü KINAY

Danışman

Prof. Dr. Murat Erşen BERBERLER

Jüri Üyesi

Dr. Öğr Üyesi Kazım ERDOĞDU

Jüri Üyesi

Prof. Dr. Okan FISTIKOĞLU

Müdür

Fen Bilimleri Enstitüsü

TEŞEKKÜR

Yüksek lisans eğitimim ve tez çalışmam boyunca önümde yeni ufukların açılmasına yardımcı olan, bilgi birikimi, deneyimi, sabrı ve anlayışı ile her zaman yanımda olan, beni yüreklendiren saygıdeğer danışman hocam Dr. Öğr. Üyesi Ayşe Övgü KINAY'a desteği ve katkısı için teşekkür ederim. Çalışmam süresince bana ayırdığı zaman ile bilgi ve görüşleriyle beni aydınlatan değerli hocam Prof. Dr. Murat Erşen BERBERLER'e önerileri ve katkıları için teşekkür ederim. Hayatımda her zaman beni destekleyen ve yanımda olan zorlukların üstesinden beraber geldiğimiz sevgili eşim Mehmet MENEVŞE'ye teşekkür ederim. Bu süreçte deneyimleriyle bana yardımcı olan arkadaşım Servet ÖZ'e teşekkür ederim. Son olarak sevgileri ve şefkatlerini her daim eksik etmeyen canım annem Nesrin ESENLİK, babam Süleyman ESENLİK'e ve kardeşim Burak ESENLİK'e sonsuz teşekkür ederim.

Merve MENEVŞE

İKİ BOYUTLU LABİRENTLERDE ARAMA ALGORİTMALARININ İNCELENMESİ

ÖZ

Arama algoritmaları, akıllı elektrikli süpürgeler, oyunlarda yol bulma ve arama kurtarma robotları gibi çeşitli uygulamalarda yaygın olarak kullanılmaktadır. Bunlardan labirent çözümü için kullanılan DFS (Derinlik Öncelikli Arama), BFS (Genişlik Öncelikli Arama) ve A* (A Yıldız) algoritmaları ise farklı yaklaşımlarla en kısa yolun bulunmasını amaçlar. Labirentlerde DFS algoritması, yolları gezerek bir çözüm bulduğu için daha kısa bir yol varsa bile bu yola ulaşmayabilir. BFS, başlangıç düğümünden hedef düğüme olan en kısa yolu daha etkili bir şekilde bulur ancak haritadaki tüm düğümleri keşfetmesi gerekebileceğinden yavaştır. A* algoritması ise BFS'ye benzer şekilde en kısa yolu bulmada etkilidir ancak önceden belirlenmiş bir sezgisel fonksiyon kullanarak hedefe daha hızlı ilerler. Bu fonksiyon, A* algoritmasının daha az düğümü ziyaret etmesini ve bu nedenle daha hızlı sonuçlar üretmesini sağlar. A* algoritması ise diğer iki algoritmaya göre daha etkin en kısa yol bulunmasını sağlar.

Bu tez çalışması kapsamında büyük labirent boyutları için bahsedilen algoritmalarının performansları incelenmiştir. Farklı boyutlarda ve zorluk seviyelerinde rastgele oluşturulan labirentlerde yol, yol uzunluğu ziyaret edilen hücre sayıları, bellek kullanımı ve süre incelenmiştir. Hesaplamalar Pycharm kod analiz aracı üzerinden Python program dili kullanılarak yapılmıştır. Yöntemler arası farklılıklar istatistiksel olarak anlamlı farklılık olup olmadığı spss ile incelenmiştir.

Anahtar kelimeler: A* algoritması, derinlik öncelikli arama, genişlik öncelikli arama, labirent, rota planlama, yapay zeka, çizge arama, simülasyon

INVESTIGATION OF SEARCH ALGORITHMS IN TWO-DIMENSIONAL MAZES

ABSTRACT

Search algorithms are commonly used in various applications, such as smart vacuum cleaners, pathfinding in games, and search and rescue robots. DFS (Depth First Search), BFS (Breadth First Search) and A* (A Star) algorithms used for maze solution aim to find the shortest path with different approaches. Since the DFS algorithm in mazes finds a solution by traveling paths, it may not reach that path even if there is a shorter path. BFS efficiently finds the shortest path from the starting node to the target node, but it can be slow because it might need to explore all nodes on the map. On the other hand, the A* algorithm is effective in finding the shortest path, similar to BFS, but it advances towards the target more quickly by using a predetermined heuristic function. This function allows the A* algorithm to visit fewer nodes and therefore produce faster results. The A* algorithm provides a more efficient shortest path than the other two algorithms.

In this thesis, the performances of the mentioned algorithms for large maze sizes were examined. The paths, path lengths, number of cells visited, ram and time using were analyzed in randomly generated mazes of varying sizes and difficulty levels. Calculations were made using the Python programming language via the Pycharm code analysis tool. The differences between the methods were analyzed for statistical significance using SPSS.

Keywords: A* algorithm, depth first search, breadth first search, maze, path planning, artificial intelligence, graph search, simulation

İÇİNDEKİLER

Sayfa

YÜKSEK LİSANS TEZİ SINAV SONUÇ FORMU ii
TEŞEKKÜR iii
ÖZ iv
ABSTRACT v
İÇİNDEKİLER vi
ŞEKİLLER LİSTESİ viii
TABLOLAR LİSTESİ x
SEMBOLLER LİSTESİ xii
KISALTMALAR xiii

BÖLÜM BİR - GİRİŞ 1

1.1 Çalışma Alanı 1
1.2 Amaç 1

BÖLÜM İKİ - ÖNCEKİ ÇALIŞMALAR 2

2.1 Klasik Yaklaşım 2
2.2 Çizge Arama Yaklaşımı 4
2.2.1 Robotik Alanı 4
2.2.2 Oyun Alanı 5
2.2.3 Labirent Alanı 6
2.2.4 Optimizasyon Alanı 7
2.3 Metasezgisel Yaklaşım 8

BÖLÜM ÜÇ - YÖNTEM & VERİ ANALİZİ 9

3.1 Yöntem 9
3.1.1 Labirent Oluşturma 9
3.1.2 DFS Algoritması 10
3.1.3 BFS Algoritması 12
3.1.4 A* Algoritması 13

3.2 Veri Analizleri 15
3.2.1 Yol Uzunluğu Analizi 16
3.2.2 Ziyaret Edilen Hücre Sayısı Analizi 22
3.2.3 Bellek Kullanım Analizi 31
3.2.4 Başlangıç ve Bitiş Noktalarının Konumlarının Analizi 36

BÖLÜM DÖRT - SONUÇLAR VE ÖNERİLER 51

KAYNAKLAR 53

ŞEKİLLER LİSTESİ

Sayfa

Şekil 3.1 Kare labirentlerde yol uzunluğu grafiği 16

Şekil 3.2 Dikdörtgen labirentlerde yol uzunluğu grafiği (1:2 ve 2:1 oranları) 18

Şekil 3.3 Dikdörtgen labirentlerde yol uzunluğu grafiği (1:3 ve 3:1 oranları) 19

Şekil 3.4 0.1 engel oranına göre ortalama yol uzunlukları 21

Şekil 3.5 0.2 engel oranına göre ortalama yol uzunlukları 21

Şekil 3.6 0.3 engel oranına göre ortalama yol uzunlukları 21

Şekil 3.7 Kare labirentlerde ziyaret edilen hücre sayısı grafiği 23

Şekil 3.8 Dikdörtgen labirentlerde ziyaret edilen hücre sayısı grafiği (1:2 ve 2:1 oranları) 24

Şekil 3.9 Ziyaret edilen hücre sayısı için kare labirentlerde Friedman iki yönlü varyans analizi 27

Şekil 3.10 Ziyaret edilen hücre sayısı için dikdörtgen labirentlerde Friedman iki yönlü varyans analizi 27

Şekil 3.11 Kare labirentlerin histogram grafikleri 28

Şekil 3.12 Dikdörtgen labirentlerin histogram grafikleri 29

Şekil 3.13 Dikdörtgen labirentlerde ziyaret edilen hücre sayısı grafiği (1:3 ve 3:1 oranları) 30

Şekil 3.14 Bellek kullanımında Friedman iki yönlü varyans analizi 33

Şekil 3.15 Bellek kullanım histogram grafikleri 34

Şekil 3.16 Yön kod satırları 36

Şekil 3.17 Farklı başlangıç ve bitiş durumlarının görseli 36

Şekil 3.18 BFS'nin versiyonlara göre labirenti arama görseli 45

Şekil 3.19 100x100'de farklı başlangıç ve bitişlerde bellek kullanımı 45

Şekil 3.20 100x50 ve 50x100'de farklı başlangıç ve bitişlerde bellek kullanımı 46

Şekil 3.21 150x50 ve 50x150'de farklı başlangıç ve bitişlerde bellek kullanımı 46

Şekil 3.22 Farklı başlangıç ve bitişler için ziyaret edilen hücre sayısı histogram grafiği 48

Şekil 3.23 Farklı başlangıç ve bitişler için bellek kullanımı histogram grafiği 49

TABLOLAR LİSTESİ

Sayfa

Tablo 3.1 Kare labirentlerde yol uzunluğu ortalamaları 16

Tablo 3.2 Küçük boyutlu labirentte yol uzunluğu örnekleri 17

Tablo 3.3 Dikdörtgen labirentlerde yol uzunluğu ortalamaları (1:2 ve 2:1 oranları). 17

Tablo 3.4 Dikdörtgen labirentlerde yol uzunluğu ortalamaları (1:3 ve 3:1 oranları). 18

Tablo 3.4 devamı 19

Tablo 3.5 Yol uzunluğu için Friedman testi sonuçları 20

Tablo 3.6 Kare labirentlerde ziyaret edilen hücre sayısı ve süre ortalamaları 22

Tablo 3.7 Dikdörtgen labirentlerde ziyaret edilen hücre sayısı ve süre ortalamaları (1:2 ve 2:1 oranları) 24

Tablo 3.8 Dikdörtgen labirentlerde ziyaret edilen hücre sayısı ve süre ortalamaları (1:3 ve 3:1 oranları) 25

Tablo 3.9 Ziyaret edilen hücre sayısı Friedman test sonuçları 26

Tablo 3.10 Ziyaret edilen hücre sayısı ikili karşılaştırma sonuçları 26

Tablo 3.11 Kare labirentlerde bellek kullanımı ortalamaları (%) 31

Tablo 3.12 Dikdörtgen labirentlerde bellek kullanımı ortalamaları (%) (1:2 ve 2:1 oranları) 32

Tablo 3.13 Dikdörtgen labirentlerde bellek kullanımı ortalamaları (%) (1:3 ve 3:1 oranları) 33

Tablo 3.14 100x100'de farklı başlangıç ve bitiş için A* algoritması ortalamaları 37

Tablo 3.15 100x50 ve 50x100'de farklı başlangıç ve bitiş için A* algoritması ortalamaları 37

Tablo 3.15 devamı 38

Tablo 3.16 50x150 ve 150x50'de farklı başlangıç ve bitiş için A* algoritması ortalamaları 38

Tablo 3.16 devamı..... 39

Tablo 3.17 100x100'de farklı başlangıç ve bitiş için DFS algoritması ortalamaları . 40

Tablo 3.18 100x50 ve 50x100'de farklı başlangıç ve bitiş için DFS algoritması ortalamaları 40

Tablo 3.18 devamı..... 41

Tablo 3.19 150x50 ve 50x150'de farklı başlangıç ve bitiş için DFS algoritması ortalamaları 41

Tablo 3.19 devamı..... 42

Tablo 3.20 100x100'de farklı başlangıç ve bitiş için BFS algoritması ortalamaları . 42

Tablo 3.20 devamı..... 43

Tablo 3.21 100x50 ve 50x100'de farklı başlangıç ve bitiş için BFS algoritması ortalamaları 43

Tablo 3.21 devamı..... 44

Tablo 3.22 150x50 ve 50x150'de farklı başlangıç ve bitiş için BFS algoritması ortalamaları 44

Tablo 3.23 Farklı başlangıç ve bitişler için ki-kare testi sonuçları 47

Tablo 3.24 Farklı başlangıç ve bitişler için yol uzunluğu Friedman test sonuçları ... 47

Tablo 3.25 Farklı başlangıç ve bitişler için bellek kullanımı ikili karşılaştırma testi sonuçları..... 48

SEMBOLLER LİSTESİ

V : Labirentteki toplam hücre sayısı (düğüm sayısı)

O : Zaman karmaşıklığı

E : Labirentteki geçerli komşuların (kenarların) toplamı

g(n) : Maliyet fonksiyonu

f(n) : Sezgisel fonksiyonu

h(n) : Tahmini maliyet fonksiyonu

KISALTMALAR

BFS : Breadth First Search (Genişlik Öncelikli Arama)

DFS : Depth First Search (Derinlik Öncelikli Arama)

A* : A Star (A Yıldız)

GA : Genetic Algorithm (Genetik Algoritma)

ACO : Ant Colony Optimization (Karınca Kolonisi Optimizasyonu)

LHR : Left Hand Rule

RHR : Right Hand Rule

SPSS : Statistical Package for the Social Sciences

UCS : Uniform Cost Search (Tekdüze Maliyetli Arama)

BÖLÜM BİR
GİRİŞ

1.1 Çalışma Alanı

Son zamanlarda yapay zekâ alanına ilgi artmaktadır. Haritada yol arama, oyun yol bulma, robot süpürge, arama kurtarma robotlarında karşımıza gelen arama algoritmaları bunlardan bazılarıdır (Rathnayake, Marzuk, Senadheera, Vijeyakumar ve Abeygunawardhana, 2021). Otonom mobil robot navigasyonu, depo robotları, sürücüsüz araçlar, akıllı tekerlekli sandalyeler ve kişisel asistan robotları gibi çeşitli uygulamalarda kritik bir rol oynar (Alamri vd., 2021). Bomba tespitinden enkaz altındaki insanları bulmaya, ev otomasyonunda birçok alanda kullanılabilmektedirler (Yadav, 2012). Ayrıca satış elemanı için en iyi rotanın değerlendirilmesi, belirli hastalıklar için en iyi ilacı tespit etmede ve varış noktasına ulaşmak için en iyi uçuş seçeneğini tahmin etmede arama algoritmaları üzerine çeşitli çalışmalar yapılmaktadır. Bu algoritmalar birçok uygulamada modern mühendislik ve pratik problemleri çözmek için kullanılmaktadır (Rathnayake vd., 2021).

1.2 Amaç

Labirent çözme algoritmaları üzerine yapılan bu çalışmanın amacı, farklı boyut ve engel oranlarına sahip rastgele labirentlerde DFS, BFS ve A* algoritmalarının performanslarını incelemektir. Bunun için 10x10, 100x100, 500x500, 1000x1000, 2000x2000 boyutlarında kare labirentler ile 1:2 ve 2:1 oranlarından 5x10, 10x5, 50x100, 100x50, 500x1000, 1000x500 ve 1:3 ve 3:1 oranlarından 5x15, 15x5, 50x150, 150x50, 500x1500, 1500x500 boyutlarında dikdörtgen labirentler seçilmiştir. Labirentlerde başlangıç noktası kuzey-batı, hedef noktası ise güney-doğu olarak belirlenmiş ve en az bir çözümü bulunanlar alınmıştır. Ek olarak, farklı başlangıç ve bitiş noktası konumlarına sahip dört ayrı durum belirlenmiş ve bu durumlar için de 100x100, 50x100, 100x50, 50x150 ve 150x50 boyutlarında labirentler üretilmiştir. Her senaryo için 0.10, 0.20 ve 0.30 engel oranlarında 30'ar farklı rastgele labirent çalıştırılarak ortalama değerler alınmıştır. Bu labirentlerde algoritmaların ortalama yol uzunluğu, ziyaret edilen hücre sayısı, bellek kullanımı ve süre analiz edilmiştir.

BÖLÜM İKİ
ÖNCEKİ ÇALIŞMALAR

Labirent arama algoritmaları, bilgisayar bilimleri ve yapay zekâ alanlarında önemli bir araştırma konusu olup, çeşitli problemlerin çözümünde kullanılmaktadır. Bu algoritmaların tarihçesi oldukça eskiye dayanır ve ilk çalışmayı belirlemek zor olabilir çünkü birçok araştırmacı ve matematikçi bu algoritmaların gelişimine katkıda bulunmuştur. Yol planlama problemi üzerine araştırmalar 1960'ların sonlarında başlamıştır. Bununla birlikte, çabaların çoğu daha yenidir ve 1980'li yıllarda yürütülmüştür (Latombe, 1991). Bu tarihsel bağlam, labirent çözme algoritmalarının evrimini ve bu süreçte ortaya çıkan çeşitli yaklaşımları anlamak için bir temel oluşturur. Labirent çözme algoritmaları Alamri vd. (2021)'ye göre ortamın bilindiği ve bilinmediği durumlara göre farklılık gösterir. Ayrıca, yol bulma algoritmaları klasik, çizge arama ve metasezgisel yaklaşımlar olarak üç kategoriye ayrılabilir (Koubâa vd., 2018).

2.1 Klasik Yaklaşım

Klasik yaklaşımlar yaygın olarak otonom robotik alanında kullanılmıştır. Otonom robot navigasyonu; depo robotları, sürücüsüz araçlar, akıllı tekerlekli sandalyeler ve kişisel asistan robotları gibi çeşitli uygulamalarda kritik bir rol oynar (Alamri vd., 2021).

Otonom robotik arama alanında çeşitli algoritmalar kullanılmıştır. Alamri vd. (2021) çalışmasında, bilinen ve bilinmeyen ortamlar için tasarlanmış labirent çözme algoritmalarının verimliliği değerlendirilmiştir. Bu bağlamda, çıkmaz doldurma (dead-end filling) algoritması, labirentteki tüm çıkmazları belirleyerek bir yol bulmaya çalışır. Ancak, labirenti tamamen dikkate aldığı için robotik sistemlerde verimsiz olarak kabul edilmiştir. Labirent yönlendirme (maze routing) algoritması, iki nokta arasındaki olası yolları bulur ve eğer varsa bağlantıyı garanti eder, fakat yoğun düzenlerde yavaş çalışır ve büyük bellek gerektirir. Soukup algoritması, labirent çözmede etkili olup başlangıç ve hedef noktalarının belirtilmesini gerektirir, yoğun bölgelerde optimal olmayan rotalar üretir ve en kısa yol garantisi vermez.

Bilinmeyen ortamlar için kullanılan duvar takip algoritması, robotun konumunu veya yönünü bilmeye gerek kalmadan sadece duvarı takip eder. Ancak, bu yöntem döngüler içeren labirentlerde veya labirentin içinden başlandığında etkili değildir. Tremaux'un algoritması, her zaman bir yol bulur ve doğrudan varış noktasına giden yolları keşfeder, fakat robotikte uygulanması zordur ve büyük bellek gerektirir. Rastgele fare algoritması ise rastgele kararlar vererek yolları kaydetmeden hareket eder; bu nedenle yavaş hız ve düşük doğruluk nedeniyle verimsizdir ve robot uzun süre yanlış yönlere sapabilir. Pledge algoritması, bir pusula kullanarak herhangi bir noktadan çıkışa yönelir ve engelleri aşabilir, ancak yolu tersine çeviremeyebilir.

Bu algoritmaların karşılaştırılmasını, bellek boyutu, algoritmanın işlenmesi için gereken toplam bellek, robotik uygulama, algoritmanın robotik sistemlerde uygulanabilirliği, algoritma karmaşıklığı, algoritmanın çalışması için gereken zaman ve en kısa yol bulma yeteneği gibi kriterler göz önünde bulundurularak yapmıştır. Kamera tabanlı sistemler, küçük ortamlarda etkili yol planlama ve güvenilir navigasyon sağlar. Ancak büyük labirentler ve gerçek dünya uygulamaları için uygun değildir. Sensör tabanlı sistemler ise maliyet etkin ve uygulanması kolaydır; özellikle mesafe ölçer tabanlı sistemler güvenilir, çizgi takip sistemleri maliyet açısından etkilidir, fakat ortamda ışık kaynağı olması beklenir.

Yapılan analiz çalışmaları incelendiğinde Chang vd. (2020) çalışmasında, duvar takip algoritmasının yol tanımlama ve kaydetme işlevi göstermediği belirtilmiştir. Kapalı döngü labirent problemlerinde çözümü optimize edememiştir. Bu nedenle, Arduino programlamasında taşırma (flood fill) algoritması kullanılmıştır. Gupta ve Sehgal (2014) ise basit bağlantılı labirentler için duvar takip tekniğini en makul yöntem olarak değerlendirmiştir; fakat günümüzde çözülmesi gereken labirentler genellikle karmaşıktır ve duvar takip algoritması ekstra yolculuk yapması gerektiğinde hedefe ulaşmak için daha fazla zaman harcamaktadır. Taşırma algoritması, ziyaret edilen piksellerin daha önce denenip denenmediğini tespit edecek bir bileşene sahip değildir ve uzay karmaşıklığına çok yüksektir. Ayrıca, yüksek derecede özyinelemelidir. Yine de taşırma algoritması, basit ve kullanışlı çalışma mantığıyla uygun bir teknik olarak belirtilmiştir.

Babula (2009) çalışmasında; Pledge, rastgele fare, en kısa yol bulma, Tremaux, duvar takip ve özyinelemeli geri izleme algoritmalarının içlerinden labirent çözme sistemleri için daha fazla araştırma ve uygulama için Tremaux algoritması önerilmiştir. Bienias (2016) çalışmasında, otonom labirent çözme robotu için duvar takip algoritması ve Tremaux algoritmasının birleşiminden türetilmiş bir algoritma kullanılmıştır. Bu algoritma, Arduino UNO tarafından kontrol edilen bir mobil platformda test edilmiştir ve elde edilen sonuçlar, geliştirilen algoritmanın labirentteki keşif verimliliğini büyük ölçüde artırdığını göstermiştir. Önerilen yaklaşımın, bilinen çözümlere kıyasla kritik avantajlar sağladığı belirtilmiştir.

2.2 Çizge Arama Yaklaşımı

Çizge arama metodu yol bulma problemlerinde kullanılan diğer iyi bilinen yaklaşımdan biridir. A*, Dijkstra, BFS ve DFS algoritmaları bu yönteme örnek olarak verilebilir (Koubâa vd., 2018). Bu algoritmalar; robotik, oyun, labirent çözme ve çeşitli optimizasyon problemlerinde kullanılmaktadır.

2.2.1 Robotik Alanı

Son yıllarda geliştirilen çizge arama algoritmaları, otonom robotların yol planlaması için test edilmiştir (Koubâa vd., 2018). Yadav, Verma ve Mahanta (2012) otonom robotların geniş uygulamaları olduğunu belirtmektedir. Robotun çevresindeki ortamı bilmediği ve dolayısıyla karar verme yeteneğine sahip olması gerektiği vurgulanmıştır. Bu bağlamda, robotun belirli bir görevi yerine getirmek için eylemlerini planlaması gerekmektedir.

Yadav vd. (2012) çalışmasında duvar takip, DFS ve taşırma algoritmalarının içinden yol uzunluğu ve zaman performansına göre DFS algoritmasının, en kısa veya en hızlı rotayı bulamama ve tüm labirenti keşfetmek için fazla zaman harcama gibi bazı dezavantajları olduğu belirtilmiştir.

Aqel vd. (2017) üç boyutta Bluetooth bağlantılı görüntü algılayan robot ile bilgisayar ortamında beş farklı labirentte arama yaptırılmıştır. Zaman olarak en iyi öncelikli arama algoritması hızlı olmasına karşın genişlik arama algoritması kısa verimli çalışmıştır. Süre ve yol uzunluğuna bakıldığında genişlik arama algoritması

önerilmiştir. Ayrıca, kamerayla görüntüyü yakalayan robot yardımıyla labirent bilgisayara Bluetooth ile gönderilmiştir. Labirent, bilgisayar ortamında BFS algoritmasıyla analiz edilmiştir ve robota hedefe ulaşması için yolu göndermiştir. Bu çalışma mantığı simülasyona benzemekte olup, yol yürüme kısmında sadece robot kullanılmaktadır.

Hii, Lee ve Chuah (2021), karşılaştırma değerlerine göre, ortalama süre için LHR algoritması, BFS ve A* algoritmasından daha uzundur. LHR algoritması diğer algoritmalara göre yol ve süre olarak en kötü sonucu vermiştir. BFS algoritması en kısa yolu orta sürede bulmuştur. Öte yandan, A* algoritması en kısa yolu hızlı sürede bulma bakımından üç algoritma arasından en uygun olanı seçilmiştir.

2.2.2 Oyun Alanı

Çizge yaklaşımı, oyunlarda yol aramak için yaygın olarak kullanılmaktadır. Yol bulma algoritmaları, video oyunlarında özellikle platform türündeki oyunlarda önem arz etmektedir. Bununla ilgili çeşitli çalışmalar yapılmıştır. Iskandar, Diah ve Ismail (2020), video oyunlarında yol bulma algoritmanın uygun olacağına odaklanmıştır. Bu algoritmaların, dikey hareketlerin olduğu iki boyutlu platform oyunlarında verimlilikleri araştırmıştır. Sonuç olarak, bu makalede ele alınan geçmiş araştırmaların ve belirlenen algoritmaların avantajlarının ışığında, video oyunlarında en yaygın olarak kullanılan algoritmalar A* algoritması ile Dijkstra algoritması olduğu görülmektedir. Ancak, iki boyutlu platform oyunlarında dikey hareketlerin sınırlı olduğu ve yer çekimi kuvvetlerinin simüle edildiği bir ortamda bu algoritmaların test edilmesi durumunda, algoritmaların performansı farklı şekillerde etkilenebilir ve hangi algoritmaların daha uygun olacağı farklılık gösterebileceği belirtilmiştir.

Permana, Bintoro, Arifitama ve Syahputra (2018), Maze Runner oyununda hangi en kısa yol algoritmasının uygun olduğunu incelemiştir. Tetris bloklarıyla engelleri farklı konumlandırarak üç farklı seviyede labirentler oluşturmuştur. Yol bulma süresi, yol uzunluğu, işlem hesaplama ve bellek kullanımına bakmıştır. Indriyono (2021) oyun haritasında BFS algoritmasının, adım adım gezerek en iyi rotayı bulacağını ve A* algoritmasının diğer algoritmalara göre optimal çözüm bulmada

daha iyi olduğu belirtilmektedir. Noori ve Moradi (2015) çeşitli algoritmaları bir labirent koşucusu senaryosunda simüle ederek, A* varyantlarının verimli olduğunu ancak performansın sezgisel fonksiyona bağlı olduğunu sonucuna varmışlardır.

Özetlendiğinde, popüler birçok oyunda yol bulma algoritmaları oldukça önemli bir konudur. A* algoritmasının buluşu oyun endüstrisi için kıymetlidir. Süre ve yol uzunluğu açısından bu algoritmanın en iyi olduğunu belirtmiştir. Bellek kullanımının azaltılması, temel arama alanın optimize edilmesi algoritmanın iyileştirme yolları arasındandır (Cui ve Shi, 2011).

2.2.3 Labirent Alanı

İki boyutta labirent arama algoritmaları üzerine yapılan çalışmalarda, çeşitli algoritmaların güçlü ve zayıf yönleri incelenmiştir. Debnath vd. (2019), DFS algoritmasını birden fazla çözümün olduğu senaryolarda veya en kısa yolun öncelikli olmadığı durumlarda avantajlı bulmuştur. Ancak, yalnızca bir çözüm varsa veya en kısa yol gerekiyorsa, DFS uygun değildir çünkü arama ağacının derin dallarında sıkışabilir. Iloh (2022) çalışmasında, DFS'nin hedefi bulurken yol uzunluğunun fazla olmasından dolayı etkili olmadığını belirtmiştir.

BFS algoritması, kaynak düğümden hedef düğüme en kısa yolu bulmak istendiğinde etkilidir. Debnath vd. (2019) BFS'nin çizgenin kenarlarının ağırlıklı olmadığı veya benzer ağırlıklara sahip olduğu durumlarda en kısa yolu bulduğunu ve çıkmazlardan kaçınmada iyi olduğunu belirtmiştir. Ancak, Iloh (2022) BFS algoritmasının yürütme süresinin yüksek olduğunu vurgulamıştır. Pathak, Patel ve Rami (2018) ise BFS'nin tüm ortalama maliyetler benzer olduğunda UCS yerine uygun olduğunu belirtmiştir.

A* algoritması, BFS ve açgözlü arama algoritmalarının avantajlarını birleştirerek en kısa yolu bulmada etkili bir yöntem sunar. A* algoritmasının yol uzunluğu ve süre açısından iyi bir yol bulma yöntemi olduğunu belirtilmiştir (Iloh, 2022). Karova, Penev ve Kalcheva (2016), A* algoritmasının büyük boyut ve yüksek engel oranlarında bile en kısa yolu bulduğunu göstermiştir. Pathak vd. (2018) ise A* algoritmasının UCS'den daha verimli ve optimal olduğunu belirtmiştir. Sezgisel

aramanın kör aramaya göre daha verimli ve kabul edilebilir olduğu sonucuna varmıştır. Buluşsal aramaları önermiştir.

Çift yönlü arama algoritmalarına bakıldığında özellikle çift yönlü BFS, küçük labirentlerde geleneksel BFS'ye kıyasla daha iyi performans göstermiştir. Bellek kullanımı fazla olmasına rağmen çift yönlü arama stratejilerinin iyi sonuçlar verdiğini ve özellikle çift yönlü BFS'nin büyük bir arama alanlı labirenti için daha iyi performans gösterdiği belirtilmiştir (Kumar ve Kaur, 2019).

2.2.4 Optimizasyon Alanı

Çizge arama algoritmaları, çeşitli problemlerde etkin bir şekilde kullanılmaktadır. Žunić, E., Djedović ve Žunić, B. (2016), satış elemanlarının çalışmalarının planlanması ve optimize edilmesinde DFS ve BFS algoritmalarının nasıl kullanıldığını detaylı bir şekilde açıklamıştır. Bu çalışmada, MATLAB simülatörü kullanılarak yapılan deneyler sonucunda, DFS'nin BFS'ye göre çok daha düşük bellek gereksinimlerine sahip olduğu belirlenmiştir. Bu özellik, DFS'nin büyük bir avantajıdır çünkü her seviyedeki tüm çocuk işaretçilerini saklaması gerekmemektedir.

Rathnayake vd. (2021), “8 Bulmaca Problemi” için DFS, BFS, açgözlü ve A* algoritmalarıyla bellek kullanımı, işlemci kullanımı ve hedef duruma ulaşmak için geçen süre niteliklerini karşılaştırmıştır. Labirentte ilerlemek ve bitiş noktasını belirlemek için DFS tercih edilebilir olsa da döngülerde DFS’nin iyi olmadığı belirtilmiştir. Açgözlü algoritması belirli bir problem için optimum çözümü üretemeyebilir lakin küresel değerlere daha yakın bir yerel optimum çözüm sağladığı belirtilmiştir. A* algoritması bu problem için iyi sonuçlar göstermiştir. Hidayat, Susanti ve Wijaya (2021) da "8 Bulmaca Problemi" üzerine yaptıkları çalışmada, A* algoritmasının bilinçli arama algoritmaları (Best First Algorithm) kategorisinde yer aldığını ve bilinçsiz arama algoritmalarına (BFS, DFS) göre daha iyi sonuçlar verdiğini gözlemlemiştir. Bilinçsiz arama algoritmalarının tutarsız performans sağladığı da belirtilmiştir.

2.3 Metasezgisel Yaklaşım

Metasezgisel algoritmalar son yıllarda çalışmalara dâhil edilmiştir. Labirent çözme problemleri, yapay zekâ alanında oldukça ilgi çekici bir konudur. Bu problemler, genetik algoritmalar (GA), tabu arama ve karınca kolonisi optimizasyonu (ACO) gibi metasezgisel yöntemlerle çözülebilir. Bu yöntemler, farklı yaklaşımları ve avantajları temsil ederler ve literatürde geniş bir şekilde incelenmişlerdir.

GA, doğal seleksiyon ve genetik miras prensiplerini taklit ederek çözümler üreten bir optimizasyon tekniğidir. Tabu arama, bir çözüm alanında gezinirken daha önce ziyaret edilen noktaları hatırlayan ve tekrar ziyaret etmeyi engelleyen bir optimizasyon tekniğidir. Labirent çözme problemlerinde tabu arama, çözüm alanını keşfetme yeteneği ve hafıza yönetimi ile etkili olabilir. ACO, karıncaların yiyecek kaynaklarını bulmak için kullandığı doğal bir yöntemi taklit eder. Labirent çözme problemlerinde ACO, karıncaların izlediği izleri takip ederek çözüm alanında yol bulma yeteneğine sahiptir. Literatürde genetik, tabu arama ve ACO algoritmalarının labirent çözme problemlerindeki kullanımı ve performansı incelenmiş ve diğer optimizasyon teknikleriyle karşılaştırılması da değerlendirilmiştir (Gordon ve Matley, 2004; Koubâa vd., 2018).

Sonuç olarak, Koubâa vd. (2018)'e göre klasik yaklaşımlar yolu bulmak için çok zaman harcar. Metasezgisel algoritmalar yaklaşıktır ve genellikle deterministik değildir. Çizge arama yöntemleri yol planlama problemini çözmek için kullanılan diğer iyi bilinen yaklaşımlardır.

BÖLÜM ÜÇ
YÖNTEM & VERİ ANALİZİ

3.1 Yöntem

Bu tez çalışmasında, çizge arama algoritmalarından olan DFS, BFS ve A* algoritmalarının farklı boyutlu labirentlerde çözme performansları incelenmiştir. Çalışmanın ana hedefi, her bir algoritmanın farklı boyutlu ve karmaşık labirentlerde nasıl performans gösterdiğini analiz ederek, algoritmaların etkinliklerini ve verimliliklerini belirlemektir.

3.1.1 Labirent Oluşturma

Labirent oluşturma sürecinde, matris tabanlı bir yaklaşım benimsenmiştir. Belirli bir genişlik ve yükseklikte matrisler oluşturularak, labirentler bu matrisler üzerinde temsil edilmiştir. Matris elemanları sıfır ve bir değerlerinden oluşmaktadır; sıfır, boşluk hücresini temsil ederken bir, engel alanını temsil etmektedir. İlk aşamada tüm hücreler sıfır olarak oluşturulmuş, ardından satır, sütun ve değiştirilebilen engel oranı çarpımı kadar birler rastgele olarak matris içine yerleştirilmiştir. Bu işlem sırasında Python'un random kütüphanesi kullanılmıştır.

Labirent oluşturma adımları şu şekildedir:

Adım 1. Labirent matrisi oluşturulur. Bu matris, yükseklik ve genişlik boyutlarına sahip ve tüm hücreler sıfır değeri ile başlatılır. Sıfır değeri boş bir hücreyi temsil etmektedir.

Adım 2. Labirentin toplam hücre sayısı hesaplanır. Bu değer, yükseklik ve genişlik değerlerinin çarpımına eşittir.

Adım 3. Hedef hücre sayısı hesaplanır. Bu değer, labirentin toplam hücre sayısının engel oranıyla çarpılmasıyla elde edilir.

Adım 4. Labirentte rastgele seçilen hücreleri takip etmek için kullanacak boş bir küme oluşturulur. Labirentin içinde rastgele bir hücre(x, y koordinatları) seçilir. Eğer seçilen hücrenin değeri 0 ise (yani henüz işaretlenmemiş bir hücre ise), bu hücre 1 olarak işaretlenir (duvar oluşturulur) ve seçilen hücre kümesine eklenir. Seçilen

hücre kümesinin boyutu, hedef hücre sayısına eşit olana kadar döngü çalışır. Döngü tamamlandığında, oluşturulan labirent matrisi döndürülür.

Bu adımlar, belirli bir yoğunlukta rastgele bir labirent oluşturmak için izlenir. Oluşturulan labirent, daha sonra çeşitli algoritmalarla çözülebilir veya analiz edilebilir. Ayrıca, labirenti dosyaya yazdırma komutları şu şekildedir:

```
dosya = Document()

dosya.add_heading('yükseklik, "x", genişlik, "engeloranı=", engeloranı', level=1)

for satir in labirent:        # Labirent matrisini Word dosyasına ekle

    satir_str = ' '.join(map(str, satir))  # Satırı string haline çevir

    dosya.add_paragraph(satir_str)  # Her satırı yeni bir paragraf olarak ekle

dosya.save("labirent.docx")        # Dosyayı kaydet
```

Labirentin başlangıç noktası kuzey batı, bitiş noktası güney doğu alınmıştır. Başlangıç noktası (0, 0) ve hedef noktası (yükseklik - 1, genişlik - 1) olarak atanmıştır. Başlangıç ve bitiş noktaları sıfır noktası olarak atanmıştır. En az bir çözümü bulunan rastgele üretilen labirentler üzerinden 10x10, 100x100, 500x500, 1000x1000, 2000x2000 boyutlarında kare labirentler ile 1:2 ve 2:1 oranlarından 5x10, 10x5, 50x100, 100x50, 500x1000, 1000x500 ve 1:3 ve 3:1 oranlarından 5x15, 15x5, 50x150, 150x50, 500x1500, 1500x500 boyutlarında 0.1, 0.2 ve 0.3 engel oranında olacak şekilde kolay orta ve zor seviyede labirentler üzerinden inceleme yapılmıştır. 0.3'den daha büyük engeller oranlarında algoritmalar sonuç bulamama durumları görülmektedir. Ayrıca farklı başlangıç ve bitiş noktaları analiz edilmiştir.

3.1.2 DFS Algoritması

DFS (derinlik öncelikli arama) algoritmasının labirentler için kullanımı, bilgisayar bilimleri alanındaki gelişmelerle birlikte ortaya çıkmıştır. 19. yüzyılda Fransız bilim insanı Charles Pierre Trémaux tarafından labirent çözümünde kullanılmıştır. DFS, çizge veri yapıları üzerinde kullanılan bir algoritmadır (Tarjan, 1972). Bir düğümün tüm dallarını keşfetmeden bir dalın en derin noktasına ulaşmaya çalışır. Başlangıç

noktasından bitiş noktasına kadar hedefi bulana kadar aramaya devam eder. Bu algoritma, düğümün komşularını işlerken, genellikle komşuları bir yığın veri yapısında saklar. Bu durumda yığından alınan elemanlar en son eklenen komşuları temsil eder. Bu algoritmada, son giren, ilk çıkar (LIFO) prensibine dayalı bir veri yapısı olan yığın kullanır (Charniak, 1985). Yani, komşular yığına ters sırayla eklenir. Örneğin, bir düğümün komşuları sağa, aşağıya, sola ve yukarıya eklenir. Bu durumda, sağa gitmek önce eklenir ve yığından en son çıkarılan yani işlenen öğe olur. Yani, kodda önce sağa gidilir, ardından aşağıya, sonra sola ve en son olarak yukarıya gidilir.

DFS algoritmasının kaba kodu şu şekildedir:

1. Başlangıç (0, 0) ve bitiş noktalarını (son satır, son sütun) belirle.
2. Yığını oluştur ve başlat (yığın = deque([("", başlangıç)])).
3. Ziyaret edilenler kümesini başlat (ziyaretedilen = {}).
4. Labirenti çizgeye dönüştür (labirentiçizgeyedönüştür(labirent)).
5. Yığın boş değilse tekrar et:
5.1 Yığından sondaki yolu ve mevcut hücreyi çıkar (yığın.pop()).
5.2 Eğer mevcut hücre bitiş noktasıysa:
5.2.1 Yolu, ziyaret edilen hücre sayısını, bellek kullanımını ve süreyi döndür.
5.3 Eğer mevcut hücre ziyaret edildiyse:
5.3.1 Bir sonraki adıma geç.
5.4 Mevcut hücreyi ziyaret edilenler kümesine ekle.
5.5 Mevcut hücrenin tüm komşuları için:
5.5.1 Her komşuyu, yeni yol ve yön ile birlikte yığının sonuna ekle (yığın.append(yol + yön, komşu)).
6. Eğer yığın boşaldıysa ve hedefe ulaşılamadıysa:
6.1 "Yol bulunamadı!" mesajı, ziyaret edilen hücre sayısı, bellek kullanımı ve süreyi sıfır olarak döndür.

Derinlik arama algoritması için “from collections import deque” kütüphanesi kullanılmıştır. DFS algoritması, oluşturulan çift uçlu kuyruğu kullanarak labirenti tarar. Her adımda bir hücreye gidilir ve o hücrenin komşuları bu çift uçlu kuyruğa eklenir.

DFS, bir yola kadar inip, orada ilerleyemediğinde geri döner. Bu nedenle, başlangıç noktasından bitiş noktasına kadar yolu bulmak için labirenti tarar.

DFS, her düğümü ve her kenarı bir kez ziyaret eder. Bu nedenle, DFS'nin zaman karmaşıklığı, düğüm sayısı (V) ve kenar sayısına (E) bağlıdır.

DFS algoritması her düğümü ve kenarı bir kez ziyaret ettiği için zaman karmaşıklığı O(V + E) olarak ifade edilir (Sedgewick ve Wayne, 2011).

3.1.3 BFS Algoritması

BFS (Genişlik Öncelikli Arama), 1950'lerin sonlarında, onu bir labirentten çıkan en kısa yolu bulmak için kullanan EF Moore tarafından icat edildi (Moore, 1959). BFS algoritması, esas olarak GPS navigasyon sisteminde karşımıza sıkça gelmektedir (Çakraborti, 2019).

BFS algoritması, çizge yapısındaki düğümler arasındaki en kısa yolu bulmak için kullanılır. Bir ağacı başlangıç düğümden başlayarak seviye seviye her düğümü gezer. BFS algoritması, bir düğümün komşularını işlerken, genellikle komşuları bir kuyruk (queue) veri yapısında saklar (Charniak, 1985). BFS başlangıç düğüme en yakın olan düğüme ilk önce gider. Ve bir sonraki kademede önce o seviyedeki tüm düğümleri inceler. Yani, en önce eklenen komşu önce işlenir. BFS algoritmasında prensibi “ilk giren ilk çıkar” (FIFO)’dur (Tucker, 1996). Komşular kuyruğa ters sırayla eklenir. Örneğin, bir düğümün komşuları sağa, aşağıya, sola ve yukarıya eklenir. Bu durumda, sağa gitmek önce eklenir ve kuyruktan en önce çıkarılan, yani işlenen öğe olur. Yani, BFS algoritmasında önce sağa gidilir, ardından aşağıya, sonra sola ve en son olarak yukarıya gidilir. BFS algoritmasını kullanarak labirentte başlangıç ve bitiş noktaları arasında bir yol bulmaya çalışır.

BFS algoritmasının kaba kodu şu şekildedir:

1. Başlangıç (0, 0) ve bitiş noktalarını (son satır, son sütun) belirle.
2. Kuyruk oluştur ve başlat (kuyruk = deque([("", başlangıç)])).
3. Ziyaret edilenler kümesini başlat (ziyaretedilen = {}).
4. Labirenti çizgeye dönüştür (labirentiçizgeyedönüştür(labirent)).
5. Kuyruk boş değilse döngüye devam et:

5.1 Kuyruğun başından mevcut yolu ve hücreyi al (kuyruk.popleft()).

5.2 Eğer mevcut hücre hedefse:

5.2.1 Yolu, ziyaret edilen hücre sayısını, süreyi ve bellek kullanımını döndür.

5.3 Eğer mevcut hücre ziyaret edildiyse atla.

5.4 Mevcut hücreyi ziyaret edilenler kümesine ekle.

5.5 Mevcut hücrenin tüm komşuları için:

5.5.1 Her komşuyu, yeni yol ve yön ile birlikte kuyruğun sonuna ekle (kuyruk. append(yol + yön, komşu)).

6. Eğer kuyruk boşaldıysa ve hedefe ulaşılamadıysa:

6.1 "Yol bulunamadı!" mesajı, ziyaret edilen hücre sayısı, bellek kullanımı ve süreyi sıfır olarak döndür.

BFS, başlangıç noktasından başlayarak komşu hücreleri keşfeder, sonra onların komşularını keşfeder ve bu şekilde devam eder. Bu nedenle, başlangıç noktasından bitiş noktasına kadar olan en kısa yolu bulmak için labirenti tarar. BFS algoritmasının zaman karmaşıklığı O(V + E) şeklindedir. Bu, çizgenin tüm düğümlerini ve kenarlarını bir kez ziyaret ettiği için geçerlidir (Sedgewick ve Wayne, 2011).

3.1.4 A Algoritması*

A* algoritması genellikle bir başlangıç noktasından hedefe olan en kısa yolu bulmak için kullanılır. İlk olarak 1968 yılında düzenlenmiştir (Hart, 1968). Bu algoritmada sezgisel hesaplamalar yapılmaktadır (Yao, Lin, Xie, Wang, ve Hung, 2010). Hem en iyi arama algoritmasının genişleme özelliğini hem de Dijkstra'nın en kısa yol bulma özelliğini birleştirir. Özellikle büyük ve karmaşık çizgelerde veya haritalarda bu algoritma, her adımda tahmin yapmak için sezgisel fonksiyon kullanır ve hedefe olan gerçek en kısa yolun tahmin edilen uzunluğunu belirlemesine yardımcı olur (Debnath vd., 2019). Bu algoritma, her bir düğümün maliyetini ve tahmini toplam maliyetini içeren bir öncelik kuyruğu kullanır. Her iterasyonda, öncelik kuyruğundan en düşük tahmini maliyete sahip düğüm alınır ve bu düğümün komşuları incelenir. Komşular, toplam maliyetlerine göre öncelik kuyruğuna eklenir. Tahmini maliyet için manhattan uzaklığı ile mevcut hücre ile hedef noktası arasındaki mesafe hesaplanır. Her bir düğümün komşuları, tahmini toplam

maliyetlerine göre sıralanır. A* algoritması, her adımda en uygun yolu aramak için tahmini maliyetleri kullanır. Algoritmanın zaman karmaşıklığı O((V + E) * log V) olarak hesaplanır.

A* algoritmasının kaba kodu şu şekildedir:

1. Başlangıç (0, 0) ve bitiş noktalarını (son satır, son sütun) belirle.

2. Öncelikli kuyruk oluştur ve başlat (heappush(kuyruk, (0 + heuristic(başlangıç, bitiş), 0, "", başlangıç)). Burada ilk eleman yolun tahmini maliyetini içerir.

3. Ziyaret edilenler kümesini başlat (ziyaretedilen = {}).

4. Labirenti çizgeye dönüştür (labirentiçizgeyedönüştür(labirent)).

5. Kuyruk boş değilse döngüye devam et:

5.1 Kuyruğun başından mevcut maliyeti, yolu ve hücreyi al (heappop(kuyruk)).

5.2 Eğer mevcut hücre bitiş noktasıysa:

5.2.1 Yolu, ziyaret edilen hücre sayısını, süreyi ve bellek kullanımını döndür.

5.3 Eğer mevcut hücre ziyaret edildiyse atla.

5.4 Mevcut hücreyi ziyaret edilenler kümesine ekle.

5.5 Mevcut hücrenin tüm komşuları için:

5.5.1 Her komşuyu, yeni yol ve yön ile birlikte kuyruğa, toplam maliyet ve tahmin edilen maliyet ile ekle (kuyruk.append((yeni_maliyet + tahmini_maliyet, yol + yön, komşu))). Kuyruğu toplam maliyete göre sırala.

6. Eğer kuyruk boşaldıysa ve hedefe ulaşılamadıysa:

6.1 "Yol bulunamadı!" mesajı, ziyaret edilen hücre sayısı, bellek kullanımı ve süreyi sıfır olarak döndür.

Bu algoritmada A* için iki ana maliyet hesaplanır:

Sezgisel fonksiyon f(n);

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

Yeni maliyet g(n): Başlangıç düğümünden mevcut düğüme kadar ulaşma maliyeti.

Tahmini maliyet h(n): Mevcut düğümden hedef düğüme ulaşması için tahmini (Manhattan) mesafe.

Bu algoritma için “heapq” kütüphanesi kullanılmıştır. Ayrıca, Python'da bellek yönetimi genellikle otomatik olarak yapılır ve gereksiz nesneleri temizlemek için çöp toplama (garbage collection) mekanizması bulunur. Yine de A*, DFS ve BFS için bellek temizleme komutuna başvuruldu. Bellek sızıntılarını önleme amaçlı “gc” python kütüphanesi vasıtasıyla gc.collect() komutu kullanıldı.

3.2 Veri Analizleri

Bu çalışmada farklı boyutlu labirent boyutları üzerinden arama algoritmalarının verileri incelenmiştir. Farklı boyutlarda ve zorluk seviyelerinde rastgele labirentlerde DFS, BFS, A* algoritmaları için yol, yol uzunluğu, ziyaret edilen hücre sayısı, bellek kullanımı ve süre sonuçları elde edilmiştir. Bilgisayar özellikleri, Intel(R) Core(TM) i7-3630M CPU @ 2.20GHz 16 GB ram, 64 bit işletim sistemi özeliklerinde dizüstü bilgisayardan alınmıştır. 10x10, 100x100, 500x500, 1000x1000, 2000x2000 kare labirentlerden ve 1:2 ve 2:1 oranlarından 5x10, 10x5, 50x100, 100x50, 500x1000, 1000x500 boyutları ile 1:3 ve 3:1 oranlarından 5x15, 15x5 50x150, 150x50, 500x1500 ve 1500x500 boyutlarda dikdörtgen labirentlerde A*, DFS ve BFS algoritmalarının sonuçları incelenmiştir. Her bir boyut için engeller 0.10, 0.20 ve 0.30 engel oranınına göre labirentte rastgele konumlandırılmaktadır. Her bir durum için 30’ar farklı rastgele labirent üretilmiş ve ortalama değerleri alınmış ve veriler analiz edilmiştir. Her labirent için başlangıç noktası kuzey-batı hedef noktası güney-doğu kabul edilmiştir. Oluşturulan labirentlerde en az bir çözüm bulunmaktadır. Bunun haricinde başlangıç ve bitiş noktaları farklı 4 durum belirlenmiş 100x100, 50x100, 100x50, 50x150 ve 150x50 labirentleri üzerinden 0.1, 0.2 ve 0.3 oranlarında ayrı ayrı 30’ar farklı rastgele labirent çalıştırılmıştır. Elde edilen verilerdeki bağımlı gruplar arasında sıralamaya dayalı farkları anlamak için Friedman testi uygulanmıştır. Veriler normal dağılıma uymadığından Friedman testi tercih edilmiştir. Bu test spesifik olarak, aynı denekler (veya nesneler) üzerinde birden fazla koşulda yapılan ölçümlerin sıralamalarını karşılaştırarak, bu koşullar arasında anlamlı bir fark olup olmadığını test eder.

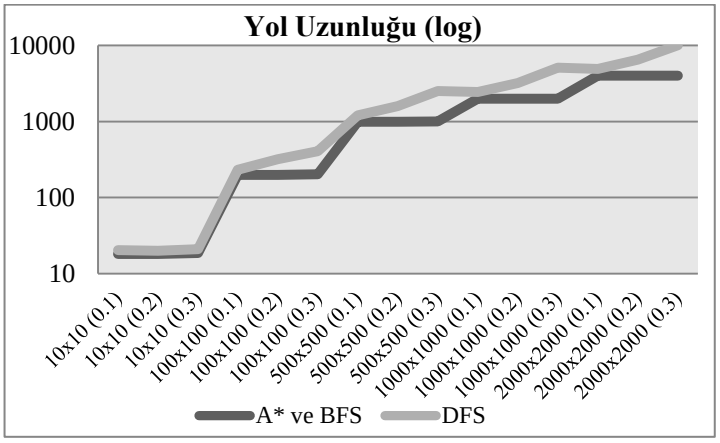
3.2.1 Yol Uzunluğu Analizi

Farklı labirent çözme algoritmalarının performansını değerlendirmek amacıyla yol uzunluğuna odaklanılmaktadır. Bu metrik, algoritmaların verimliliğini ve arama alanındaki etkinliğini belirlemek için kritik bir göstergedir.

Tablo 3.1 Kare labirentlerde yol uzunluğu ortalamaları

Boyut	Engel Oranı	A* ve BFS	DFS
10x10	0.1	18.00	20.26
	0.2	18.00	19.86
	0.3	18.60	21.00
100x100	0.1	198.00	230.93
	0.2	198.06	320.46
	0.3	201.80	407.73
500x500	0.1	998.00	1201.80
	0.2	998.00	1605.93
	0.3	1002.60	2511.00
1000x1000	0.1	1998.00	2454.86
	0.2	1998.00	3211.53
	0.3	2003.67	5144.10
2000x2000	0.1	3998.00	4947.06
	0.2	3998.00	6551.31
	0.3	4005.06	10114.13

Tablo 3.1'de kare labirentlerin farklı engel oranlarına karşılık üç algoritmanın yol uzunluklarının ortalama değerleri alınmıştır.



Şekil 3.1 Kare labirentlerde yol uzunluğu grafiği

Şekil 3.1 ve Tablo 3.1'de A* ve BFS algoritmalarının engel oranı 0.1'den 0.2'ye artmasına rağmen alternatif yollar ile en kısa yol değerleri aynı gelmiştir. Örneğin 500x500 boyutunda 0.1 ve 0.2 engel oranı A* ve BFS yol uzunluğu 998.00'dir. 0.3 engel oranında oluşan duvarlardan dolayı güzergâhlar uzamış ve yol uzunluğu diğer engel oranlarına göre daha yüksek gelmiştir. DFS algoritmasında üç engel oranında da ortalama değerler farklı çıkmıştır.

Tablo 3.2 Küçük boyutlu labirentte yol uzunluğu örnekleri

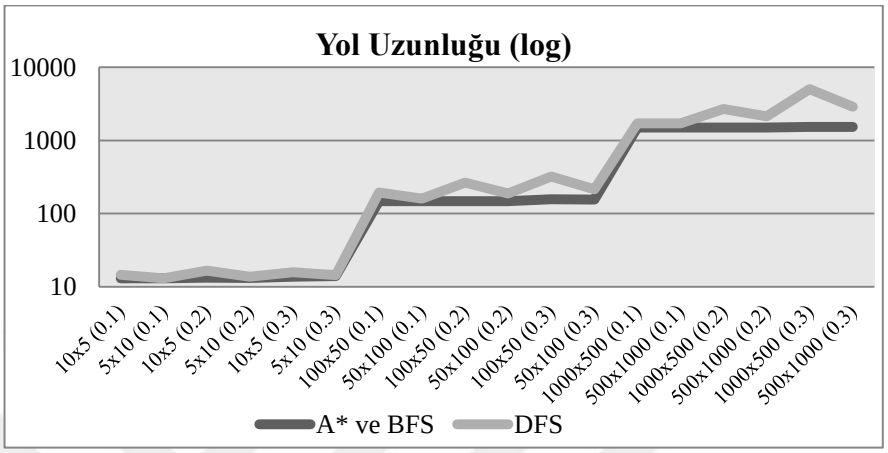
Labirentler	**Örnek 1**	**Örnek 2**	**Örnek 3**
Boyut	20x20	15x15	10x10
Engel Oranı	0.1	0.2	0.3
A*, BFS, DFS	38	28	18

DFS'nin Tablo 3.2'deki gibi boyut düşükken en kısa yolu bulduğu durumlar olmuştur. Ancak 1000x1000 gibi büyük boyutlarda DFS algoritması en kısa yolu bulamamıştır (DFS'nin değerleri 2454.86, A* ve BFS'nin değerleri 1998.00'dir).

Tablo 3.3 Dikdörtgen labirentlerde yol uzunluğu ortalamaları (1:2 ve 2:1 oranları)

Boyut	**Engel Oranı**	**A* ve BFS**	**DFS**
10x5	0.1	13.00	14.60
5x10		13.00	13.06
10x5	0.2	13.26	16.66
5x10		13.13	13.73
10x5	0.3	13.66	15.80
5x10		13.93	14.46
100x50	0.1	148.00	193.53
50x100		148.00	159.86
100x50	0.2	148.13	266.13
50x100		148.00	187.40
100x50	0.3	156.40	320.00
50x100		154.80	214.40
1000x500	0.1	1498.00	1958.34
500x1000		1498.00	1707.13
1000x500	0.2	1498.06	2696.86
500x1000		1498.13	2145.20
1000x500	0.3	1531.73	5012.53
500x1000		1531.86	5063.86

Tablo 3.3'te kare labirentlerdeki bazı sonuçlar dikdörtgen labirentlerden de görülmektedir. Burada, farklı engel oranlarına göre yol uzunluklarının değişiklik gösterdiği belirtilmiştir. DFS 10x5 ve 5x10 boyutlarında en kısa yolları bulduğu durumlar mevcuttur.



Şekil 3.2 Dikdörtgen labirentlerde yol uzunluğu grafiği (1:2 ve 2:1 oranları)

Şekil 3.2'de Tablo 3.2'deki değerler grafiğe aktarılmıştır. Şekil 3.2'de görüldüğü gibi küçük boyutlu (10x5 ve 5x10) haritalarda, düşük engel oranlarında (0.1) yol uzunlukları benzerlik gösterirken, engel oranı arttıkça bu benzerlik azalmaktadır. Büyük boyutlu (1000x500 ve 500x1000) haritalarda, yol uzunlukları daha belirgin bir şekilde artış göstermektedir. Engel oranı 0.3'e çıktığında A* ve BFS ve DFS sonuçları arasındaki farklar daha belirgin hale gelmiştir.

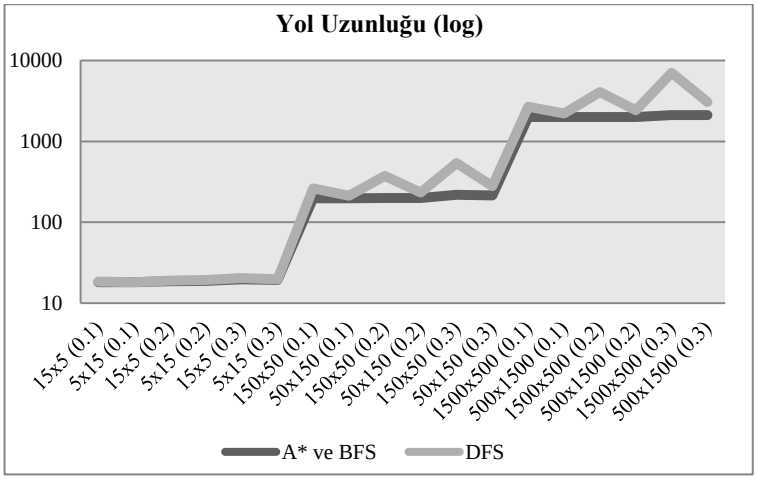
Tablo 3.4 Dikdörtgen labirentlerde yol uzunluğu ortalamaları (1:3 ve 3:1 oranları)

Boyut	Engel Oranı	A* ve BFS	DFS
15x5	0.1	18.13	18.40
5x15		18.06	18.06
15x5	0.2	18.66	18.93
5x15		18.86	19.26
15x5	0.3	19.60	20.46
5x15		19.26	19.66
150x50	0.1	198.00	261.00
50x150		198.00	211.80

Tablo 3.5 devamı

Boyut	Engel Oranı	A* ve BFS	DFS
150x50	0.2	199.00	375.26
50x150		198.93	233.26
150x50	0.3	219.00	537.33
50x150		214.13	280.26
1500x500	0.1	1998.00	2681.31
500x1500		1998.00	2210.60
1500x500	0.2	1999.73	4073.93
500x1500		1999.66	2437.33
1500x500	0.3	2115.06	7035.20
500x1500		2116.20	3087.20

Tablo 3.1, Tablo 3.3 ve Tablo 3.4'te de kare labirentlerde 0.1 engel oranında yol uzunluğu yüksekliğin ve genişliğin çarpımının bir eksiği kadar iken dikdörtgen labirentlerde yatay ve dikey farkı arttıkça genişlik ve yükseklik kenar uzunluğu toplamından daha fazla yol uzunluğu değerleri oluşmaktadır. Kare labirentlerde farklı alternatif kısa yollar üretmek daha kolay iken dikdörtgenlerde engellerden kurtulmak ve en kısa yollar bulmak için daha uzun rotasyonlar kullanması gerekebilmektedir. Ayrıca, 0.1 engel oranında; 1000x500 ve 500x1000'de DFS 1958.34 ile 1707.13, 1500x500 ve 500x1500'de DFS 2681.31 ile 2210.60'dir. Dikdörtgen labirentlerde genişlik ve yükseklik farkı arttıkça ve zorluk seviyeleri artışlarında DFS'nin yol uzunlukları daha yüksek gelmektedir.



Şekil 3.3 Dikdörtgen labirentlerde yol uzunluğu grafiği (1:3 ve 3:1 oranları)

Şekil 3.2 ve Şekil 3.3'te DFS yol uzunlukları daha fazla geldiği görülebilmektedir. A* ve BFS yol uzunlukları eşit çıkmaktadır ve en kısa yolu bulmada verimlidirler.

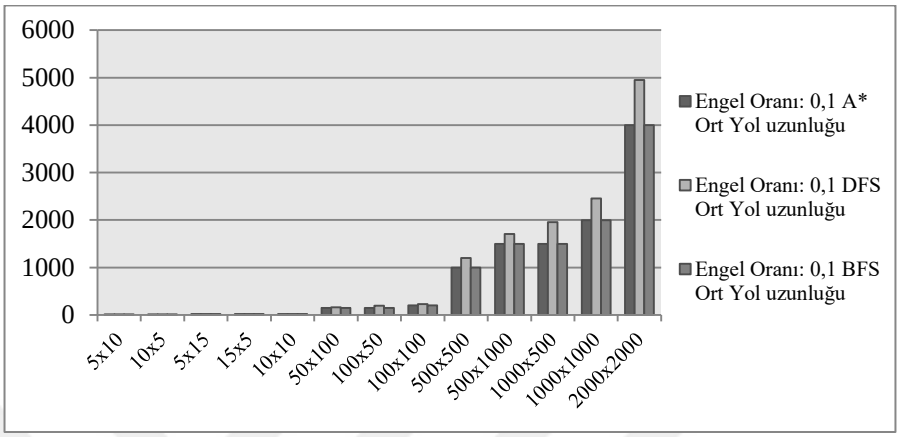
Tablo 3.6 Yol uzunluğu için Friedman testi sonuçları

Analizler	**Kare Labirentler**	**Dikdörtgen Labirentler 1:2 ve 2:1 oranlı**	**Dikdörtgen Labirentler 1:3 ve 3:1 oranlı**
A* veBFS ortalama sıralama	1.00	1.00	1.03
DFS ortalama sıralama	2.00	2.00	1.97
N	15.00	18.00	18.00
Ki-Kare	15.00	18.00	17.00
Serbestlik derecesi	1	1	1
Anlamlılık	<.001	<.001	<.001

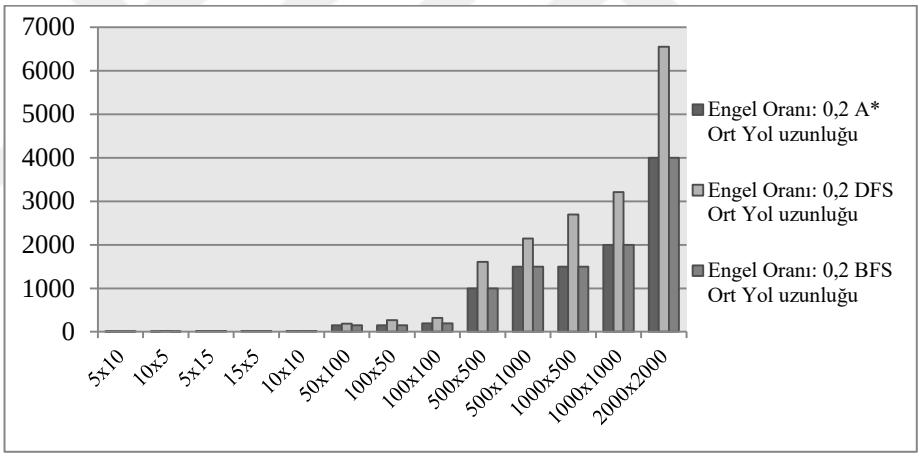
İstatistiksel değerlendirmek için Tablo 3.1, Tablo 3.3 ve Tablo 3.4'teki kare ve dikdörtgen labirentlerin yol uzunluğu değerlerine Friedman testi uygulandı. Elde edilen sonuçlar Tablo 3.5'te gösterildi. Tablo 3.5'e göre, ki-kare değerlerinin yüksek olmasından dolayı gruplar arasında belirgin bir fark olduğunu görülmektedir. p-değeri 0.001'den küçük çıkmıştır. Bu durumda, sıfır hipotezi (grupların dağılımlarının aynı olduğu hipotezi) reddedilir. Bu da gruplar arasında istatistiksel olarak anlamlı farklar olduğunu göstermiştir. A* ve BFS ile DFS yol uzunluklarının performansları veya sıralama dağılımları arasında belirgin bir ayrım olduğunu ortaya koymaktadır. A* ve BFS yol uzunluğu, DFS yol uzunluğundan daha iyi performans göstermektedir.

Genel değerlendirildiğinde, DFS, bir yol bulana kadar bir dalı derinlemesine araştırır ve bu nedenle dalların derinliği arttıkça daha fazla zaman harcar. Labirentin büyümesiyle birlikte, DFS algoritması daha derin dallara ulaşmak için daha fazla adım atmak zorunda kalabilir ve bu genellikle daha uzun sürecek bir arama sürecine yol açar. Özellikle labirent büyük ve karmaşık hale geldiğinde, DFS'nin daha derin dallara girmesi ve bu dalları araştırması daha uzun zaman alabilmektedir. Bu durumda, DFS algoritması tüm yol seçeneklerini tam olarak keşfetmeden birçok dalda ilerleyebilir, bu da kısa yolu bulmasını zorlaştırabilir. Ayrıca, DFS algoritması labirentte bir döngü olması durumunda sonsuz döngülere de takılabilir. Büyük bir labirentte döngülerin olma olasılığı artar ve DFS algoritması bu döngülere takılarak

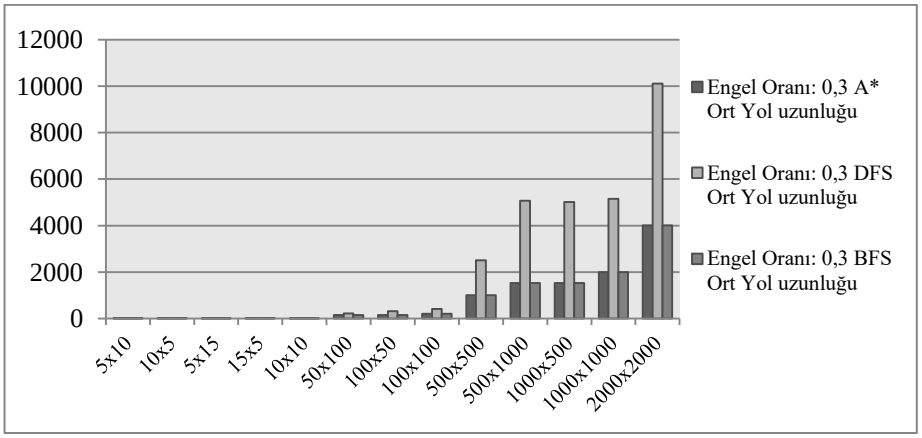
kısa yolu bulamayabilir. Bununla birlikte, DFS algoritması özellikle labirentin daha az dallandığı ve hedefin daha derinlerde bulunduğu durumlarda daha etkili olabilmektedir. Ancak genel olarak, labirentin büyümesiyle birlikte DFS'nin kısa yolu bulma yeteneği azalmaktadır.



Şekil 3.4 0.1 engel oranına göre ortalama yol uzunlukları



Şekil 3.5 0.2 engel oranına göre ortalama yol uzunlukları



Şekil 3.6 0.3 engel oranına göre ortalama yol uzunlukları

Şekil 3.4-3.6'da görüldüğü üzere labirent boyutu büyüdükçe, DFS algoritmasının kısa yolu bulmada verimi düşmektedir. Boyut büyüdükçe bu algoritmanın bulduğu yol uzunluğu ile en kısa yol arasındaki fark artmaktadır. Özellikle dikdörtgen labirentlerde daha büyük boyutlarda daha uzun yollarla karşılaşılabilmektedir. Hem kare hem de dikdörtgen labirentlerde A* ve BFS algoritmaları en kısa yolu bulmaktadır. Bu algoritmaların hedefe ulaşmak için en optimize edilmiş yolu bulma yetenekleri, labirentin şekline bağlı olarak büyük ölçüde değişmez.

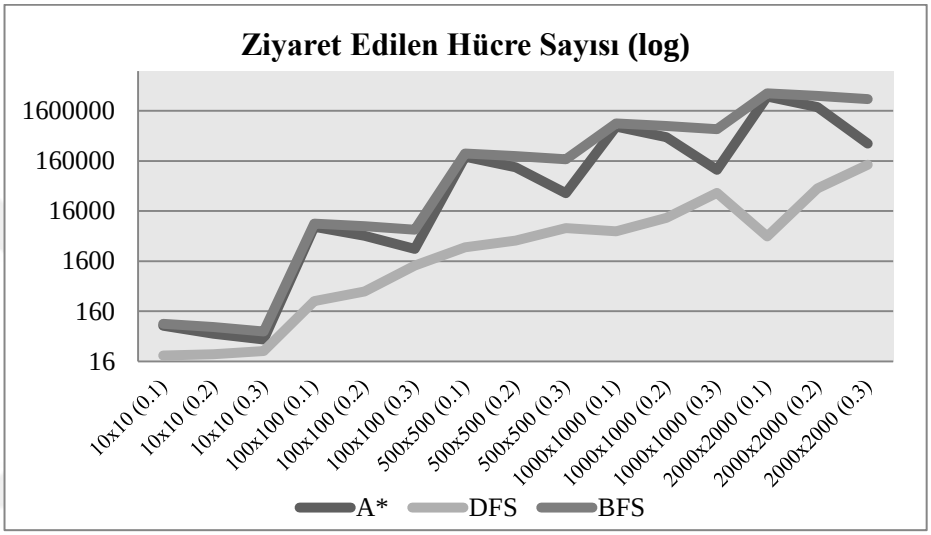
3.2.2 Ziyaret Edilen Hücre Sayısı Analizi

Bu analizde, labirent çözme algoritmalarının performansını ölçmek için ziyaret edilen hücre sayısı ve süre incelenmektedir. Ziyaret edilen hücre sayısı ve süre, algoritmanın ne kadar verimli çalıştığını ve hedefe ulaşmak için ne kadar fazla alanı taradığını ve zaman harcadığını gösterir. Bu metrikler, farklı algoritmaların etkinliklerini karşılaştırmak ve en uygun çözüm yöntemini belirlemek için önemlidir.

Tablo 3.7 Kare labirentlerde ziyaret edilen hücre sayısı ve süre ortalamaları

Boyut	**Engel Oranı**	**Ziyaret Edilen Hücre Sayısı**			**Süre (milisaniye)**		
		A*	**DFS**	**BFS**	**A***	**DFS**	**BFS**
10x10	0.1	82.43	20.6	89.16	1.00	0.01	0.01
	0.2	57.13	22.03	77.23	0.40	0.03	0.16
	0.3	43.43	25.70	63.23	0.27	0.01	0.13
100x100	0.1	7719.63	255.43	8997.76	46.76	0.37	18.39
	0.2	5108.13	397.33	7976.00	27.42	0.79	14.39
	0.3	2799.16	1306.26	6827.96	14.84	2.59	11.62
500x500	0.1	193905.00	3023.66	224974.70	1696.09	15.83	621.15
	0.2	120013.20	4127.43	199563.60	920.90	21.02	494.95
	0.3	36404.73	7266.73	171833.20	303.69	30.70	394.07
1000x1000	0.1	775384.40	6273.20	899902.66	12054.61	113.83	3905.17
	0.2	476147.30	11717.93	798318.50	4777.94	2247.72	2454.42
	0.3	106102.40	36615.80	687663.40	1102.93	6245.31	1859.13
2000x2000	0.1	3107138.00	5012.23	3599613.00	85882.61	47.81	16708.88
	0.2	1913913.00	43851.23	3193319.00	39193.57	91595.36	13718.23
	0.3	352665.70	134228.80	2751490.00	5010.24	131922.00	10403.59

Tablo 3.6’da kare labirentlerin ziyaret edilen hücre sayılarına göre küçük boyutlarda (10x10) düşük engel oranlarında (0.1), A* algoritması 82.43 hücre ziyaret ederken, DFS 20.76 ve BFS 89.16 hücre ziyaret etmektedir ve BFS daha fazla değerlere sahiptir. Büyük boyutlu labirentlerde de (2000x2000) BFS algoritmalarının ziyaret edilen hücre sayıları ve süreleri oldukça yüksekken, A* algoritması daha az hücre ziyaret eder.

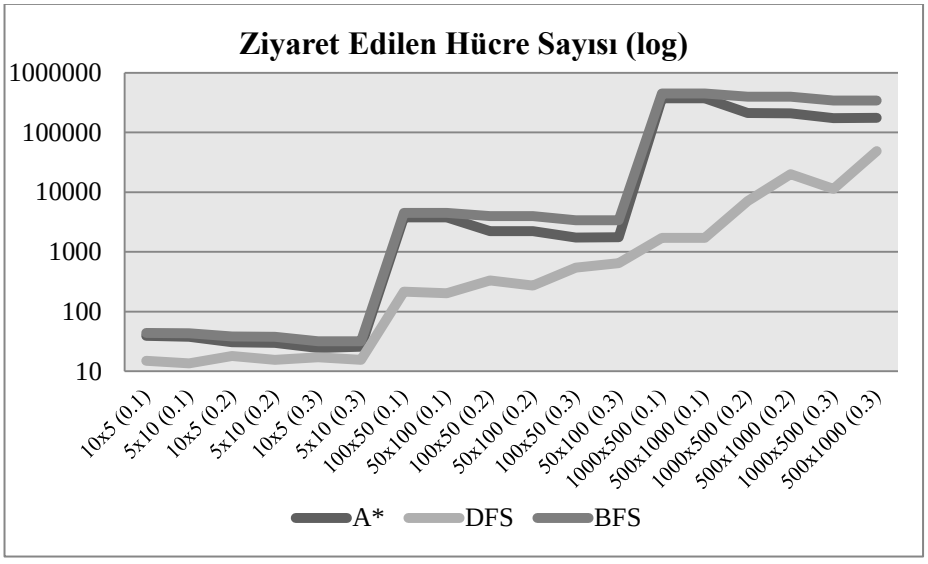


Şekil 3.7 Kare labirentlerde ziyaret edilen hücre sayısı grafiği

Tablo 3.6 ve Şekil 3.7'de görüldüğü gibi, kare labirentlerde boyutlar aynı kalırken engel oranı arttıkça A* ve BFS algoritmalarında ziyaret edilen hücre sayısı ve süre azalmıştır, DFS algoritmasında ise artmıştır. Bunun sebebi, A* ve BFS algoritmalarının olası tüm durumları kontrol etmesidir. Boş bir labirentte kısa yol bulmak için daha fazla olasılık oluşur ve en iyi yolu seçmek için her hücreye bakmaları gerekir. Engel oranı arttıkça hem ziyarete açık hücreler azalmakta hem de kapalı bölgeler oluşmaktadır. DFS algoritmasında ise derinlik araması yapıldığından, boş ya da az engelli labirentlerde ziyaret edilen hücre sayısı oldukça azdır. Engel oranı arttıkça, arama sırasında karşısına duvarlar çıkacağından arama dallanmakta ve yolu bulmak için daha fazla hücre ziyaret edilmesi gerekmektedir. Bu da benzer şekilde süreyi etkilemektedir.

Tablo 3.8 Dikdörtgen labirentlerde ziyaret edilen hücre sayısı ve süre ortalamaları (1:2 ve 2:1 oranları)

Boyut	Engel Oranı	Ziyaret Edilen Hücre Sayısı			Süre (milisaniye)		
		A*	DFS	BFS	A*	DFS	BFS
10x5	0.1	39.10	15.03	44.00	0.23	0.01	0.20
5x10		37.90	13.53	43.33	0.13	0. 03	0.13
10x5	0.2	30.63	17.93	38.23	0.20	0. 10	0.01
5x10		29.70	15.60	37.57	0.23	0. 03	0.07
10x5	0.3	24.60	17.10	31.90	0.17	0.06	0. 01
5x10		25.60	15.46	31.80	0.07	0.03	0. 06
100x50	0.1	3795.53	216.60	4498.23	21.63	0.20	8.50
50x100		3758.67	202.43	4498.23	21.75	0. 30	9.16
100x50	0.2	2226.87	330.03	3988.27	11.69	0. 54	6.77
50x100		2216.37	273.93	3986.73	11.47	0. 50	7.21
100x50	0.3	1729.80	546.93	3390.67	9.26	0. 87	5.36
50x100		1748.13	648.53	3395.87	8.99	1.26	4.99
1000x500	0.1	372829.20	2271.21	449949.30	3641.88	5.21	1278.41
500x1000		375970.20	1721.97	449948.10	3570.84	4.90	1251.76
1000x500	0.2	211372.10	7113.50	399143.50	1795.93	40.34	1047.93
500x1000		209588.70	19988.83	399118.70	1793.43	2945.92	1026.89
1000x500	0.3	173295.50	20982.13	343734.40	1875.58	168.26	903.90
500x1000		175999.00	48879.37	343745.80	2044.16	6226.91	999.16



Şekil 3.8 Dikdörtgen labirentlerde ziyaret edilen hücre sayısı grafiği (1:2 ve 2:1 oranları)

Tablo 3.7 ve Şekil 3.8'de dikdörtgen labirentlerin ziyaret edilen hücre sayısı ve süre değerleri verilmiştir. Küçük boyutlarda (10x5 ve 5x10) süreler birbirine yakındır, düşük engel oranlarında (0.1), A* algoritması 39.10 ile 37.90 hücre ziyaret ederken, DFS 15.03 ile 13.53 hücre ziyaret etmektedir. Dikdörtgen labirentlerde orta boyutlarda(100x50 ve 50x100) da benzer bir eğilim gözlemlenmektedir. A* algoritması 1729.37 ile 3795.53 hücre ziyaret ederken, DFS 202.43 ile 648.53 arasında kalmaktadır. Büyük boyutlarda(1000x500 ve 500x1000) A* algoritması, 173295.50 ile 375970.20 hücre ziyaret ederken, DFS 1721.97 ile 48879.37 arasında kalmaktadır. DFS'nin ziyaret edilen hücre sayısı ve süresi A*'dan azdır.

Tablo 3.9 Dikdörtgen labirentlerde ziyaret edilen hücre sayısı ve süre ortalamaları (1:3 ve 3:1 oranları)

Boyut	**Engel Oranı**	**Ziyaret edilen hücre sayısı**			**Süre (milisaniye)**		
		A*	**DFS**	**BFS**	**A***	**DFS**	**BFS**
15x5	0.1	59.23	19.80	67.03	0.33	0.01	0.06
5x15		55.86	19.06	66.40	0.47	0. 03	0.29
15x5	0.2	42.63	21.53	56.20	0.20	0.07	0.13
5x15		46.83	22.13	57.56	0. 39	0.06	0.13
15x5	0.3	36.90	22.13	49.13	0.10	0. 03	0.09
5x15		37.70	21.23	49.06	0.16	0. 01	0.01
150x50	0.1	5319.24	268.34	6748.41	31.89	0. 53	13.66
50x150		5399.30	284.83	6748.06	325.26	0. 47	14.09
150x50	0.2	3207.56	464.53	5980.56	18.49	0.66	10.89
50x150		3071.73	808.53	5980.40	17.23	2.032	10.59
150x50	0.3	3299.86	851.70	5087.90	17.97	1.30	7.99
50x150		3156.16	1435.86	5101.06	16.49	2.87	8.17
1500x500	0.1	527682.30	3729.65	674923.10	6913.80	18.48	2373.74
500x1500		524909.00	15848.07	674923.80	5991.54	7924.60	2188.38
1500x500	0.2	278037.40	6748.56	598698.40	3459.61	28.07	1625.83
500x1500		279767.30	31228.20	598718.80	3228.76	18056.67	1584.39
1500x500	0.3	332588.40	18965.07	515777.80	3156.76	123.31	1291.62
500x1500		337559.00	77777.70	515730.40	3206.89	35739.50	1259.70

Tablo 3.6–3.8'de görüldüğü üzere arama algoritmaları genel olarak dikdörtgen veya kare bölgelerde verimli şekilde çalışmıştır. Kare labirentlerde yollar genellikle

daha düzenli ve simetrik olurken, dikdörtgen labirentlerde yollar daha uzun ve dolambaçlı olabilir. Dikdörtgen labirentlerde, genişlik ve yükseklik oranlarının farklı olması nedeniyle yol uzunluğu ve ziyaret edilen hücre sayısında bazı küçük farklılıklar görülmektedir. DFS algoritmasında labirentin genişliği arttıkça daha fazla zaman harcanmaktadır.

Kare ve dikdörtgen labirentlerden elde edilen ziyaret edilen hücre sayıları arasında anlamlı farklılıklar olup olmadığı incelenmiştir.

Tablo 3.10 Ziyaret edilen hücre sayısı Friedman test sonuçları

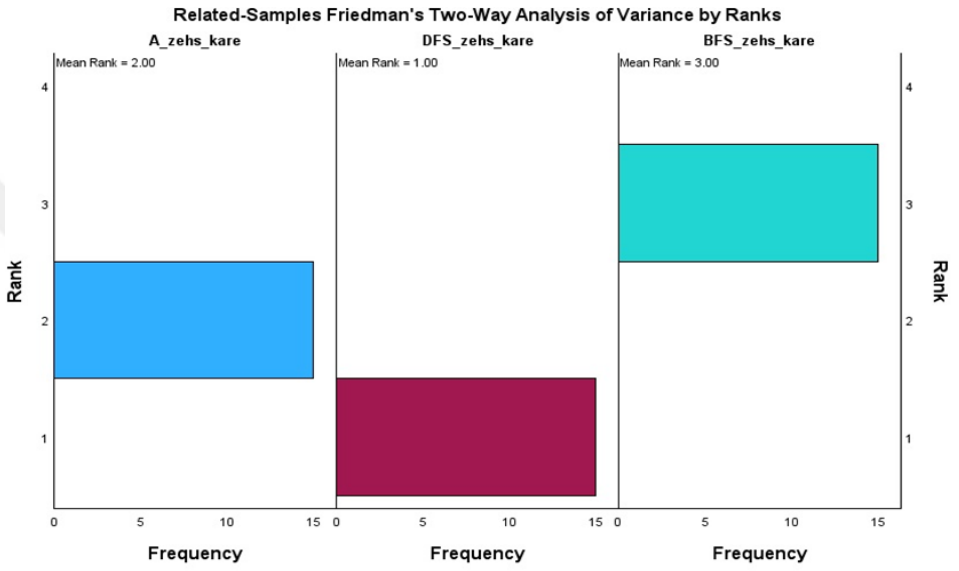
Labirentler	N	Ki-kare	df	Anlamlılık	Algoritmalar	En Yüksek Performans
Kare	15.00	30.00	2	<.001	A*, DFS, BFS	DFS
Dikdörtgen (1:2 ve 2:1 oranları)	18.00	36.00	2	<.001	A*, DFS, BFS	DFS
Dikdörtgen (1:3 ve 3:1 oranları)	18.00	36.00	2	<.001	A*, DFS, BFS	DFS

Tablo 3.9'da A*, DFS ve BFS algoritmalarının performans farklılıklarını değerlendirmek için Friedman iki yönlü sıralama varyans analizi yapılmıştır. Ki-kare değerleri yüksek ve anlamlılık değerleri 0.001'den küçük çıkmıştır. Analiz sonuçlarına göre bu algoritmaların istatistiksel olarak anlamlı farklılık tespit edilmiştir.

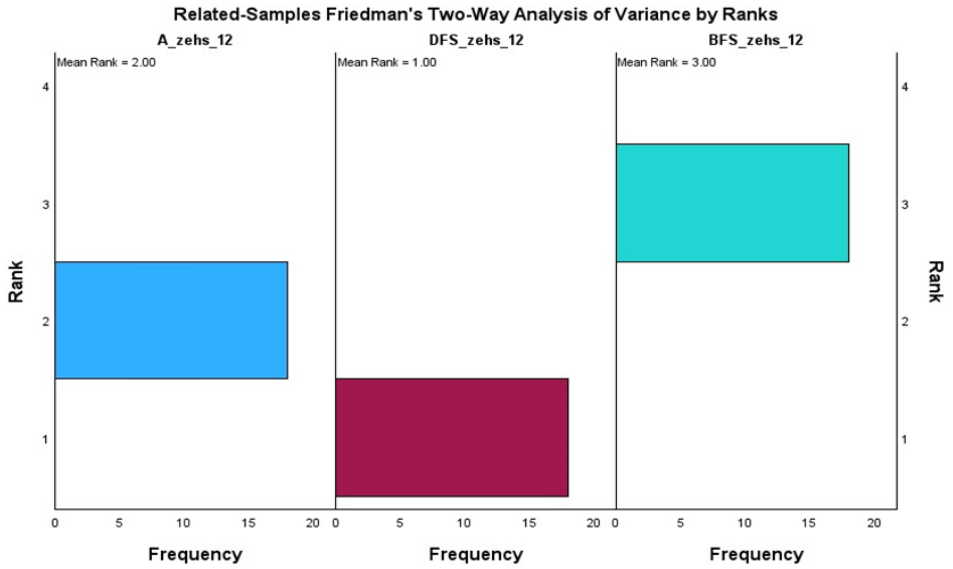
Tablo 3.11 Ziyaret edilen hücre sayısı ikili karşılaştırma sonuçları

Labirentler	Karşılaştırma	Test İstatistiği	Sig.	Adj. Sig. (Bonferroni)
Kare	DFS - A*	1.000	0.006	0.019
	DFS - BFS	-2.000	<.001	0.000
	A* - BFS	-1.000	0.006	0.019
Dikdörtgen (1:2 ve 2:1 oranları)	DFS - A*	1.000	0.003	0.008
	DFS - BFS	-2.000	<.001	0.000
	A* - BFS	-1.000	0.003	0.008
Dikdörtgen (1:3 ve 3:1 oranları)	BFS - A*	1.000	0.003	0.008
	BFS - DFS	-2.000	<.001	0.000
	A* - DFS	-1.000	0.003	0.008

Tablo 3.10’da ise kare ve dikdörtgen labirentlerde ikili karşılaştırmalar ile Bonferroni düzeltmesinin sonuçları 0.05’den düşük çıkmasından dolayı algoritmaların ikili karşılaştırmaları arasında anlamlı farklar görülmektedir (Bonferroni düzeltmesi anlamlılık düzeyi 0.05’dir). Ayrıca test istatistik değerlerine göre BFS'nin hem A* hem de DFS ile karşılaştırıldığında belirgin bir farka sahip olduğunu görülmektedir ve DFS ile A* grubu arasında da anlamlı bir fark bulunmaktadır.

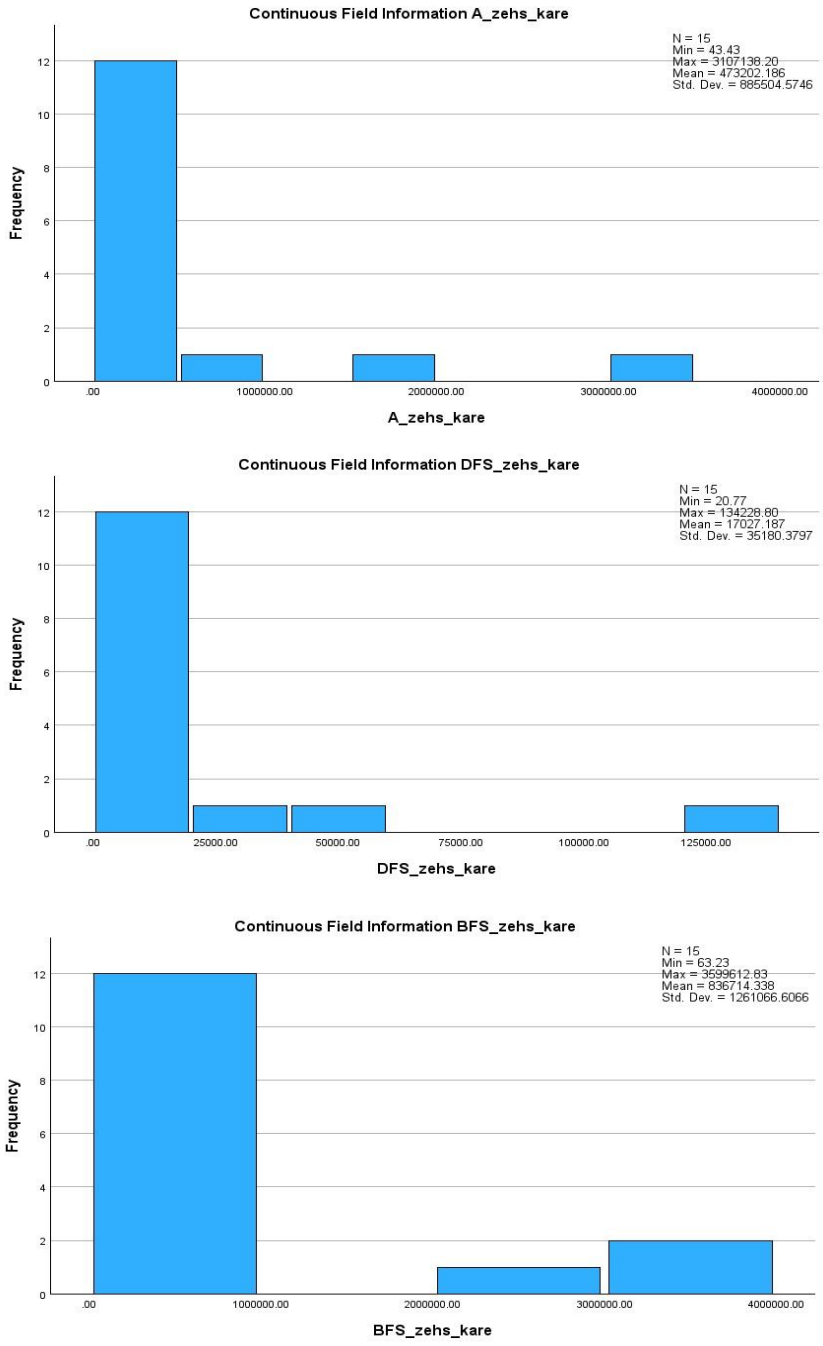


Şekil 3.9 Ziyaret edilen hücre sayısı için kare labirentlerde Friedman iki yönlü varyans analizi



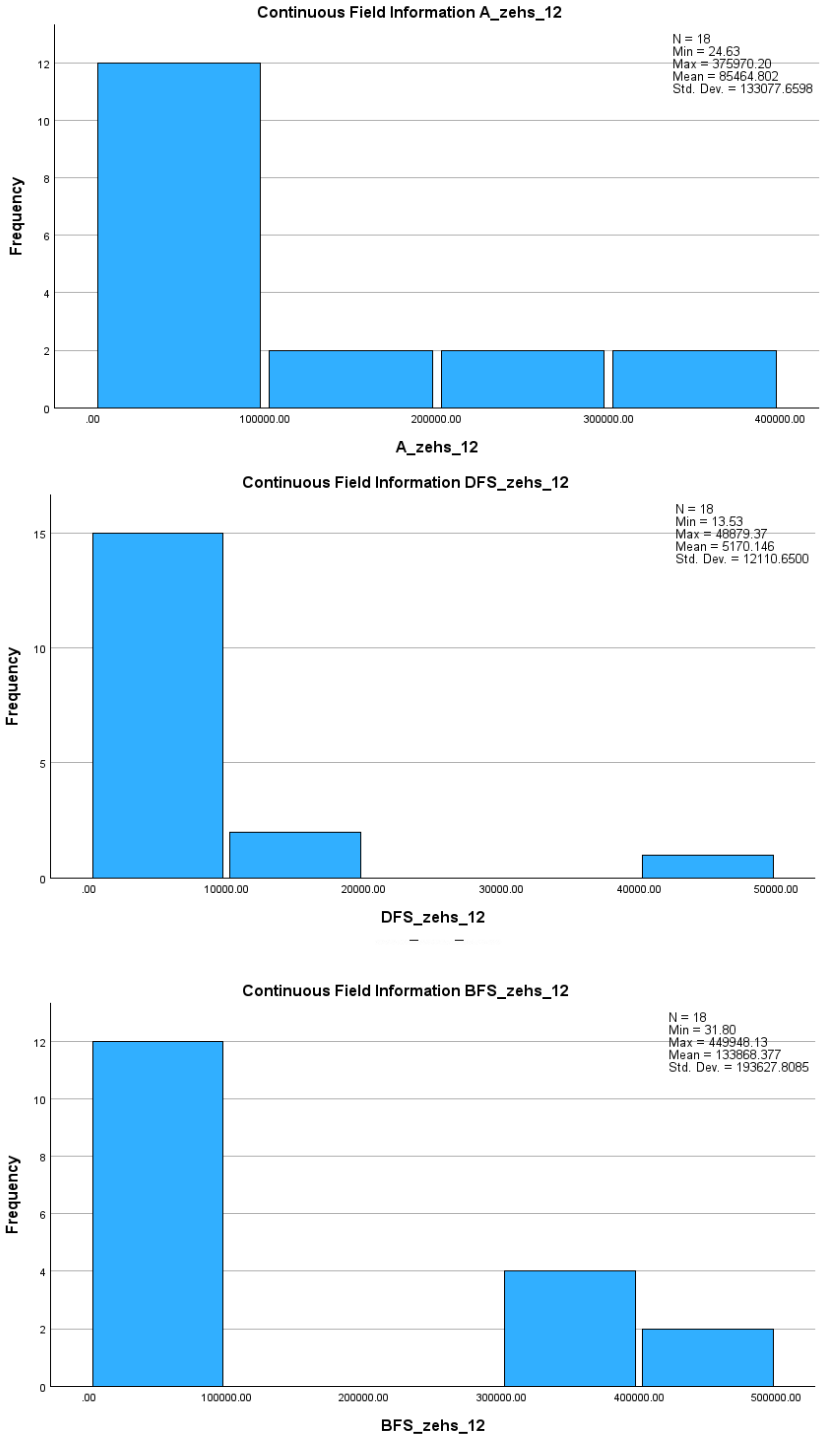
Şekil 3.10 Ziyaret edilen hücre sayısı için dikdörtgen labirentlerde Friedman iki yönlü varyans analizi

Şekil 3.9 ve Şekil 3.10'a göre DFS algoritmasının en yüksek, BFS algoritmasının en düşük performansa sahip olduğu görülmektedir. Ayrıca, Şekil 3.9 ve Şekil 3.10'da kare ve dikdörtgen labirentler arasında belirgin bir fark bulunmamaktadır; her iki labirent türü için de benzer ortalama sıralama değerleri ve frekans dağılımları mevcuttur. Bu durum, sıralama dağılımlarının labirent tipi değişse bile tutarlı kaldığını ve sıralamaların benzer şekilde dağıldığını göstermektedir.



Şekil 3.11 Kare labirentlerin histogram grafikleri

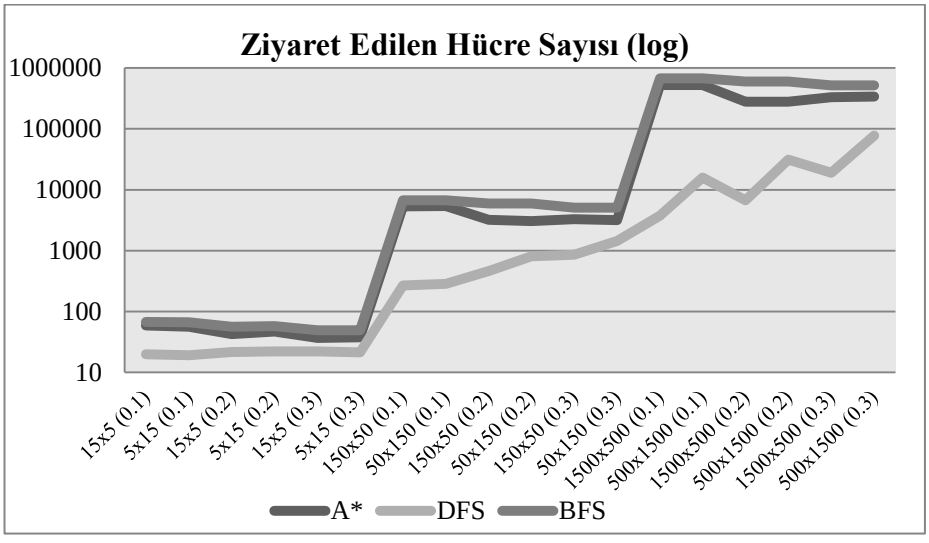
Şekil 3.11 ve Şekil 3.12'de hem kare hem de dikdörtgen labirentlerin verilerinin büyük bir kısmı sıfır veya sıfıra yakındır. Yüksek değerlere sahip veriler nadiren görülmektedir ve bundan dolayı yüksek değerler ortalama ve standart sapmayı önemli ölçüde etkilemektedir.



Şekil 3.12 Dikdörtgen labirentlerin histogram grafikleri

Şekil 3.11 ve Şekil 3.12, kare ve dikdörtgen labirentlerin verilerinin genel dağılım özelliklerini ve frekans yoğunlaşmalarını anlamak için faydalıdır. Kare labirentlerdeki veriler daha yüksek ortalama değerlere ve daha geniş dağılımlara sahiptir. Dikdörtgen labirent verileri bazı grafiklerde daha dar ve düşük değerlere sahiptir. Özellikle, kare labirentlerdeki BFS grafiğinde bu durum belirgindir. Dikdörtgen labirentlerde, BFS ve A* grafiklerinde de yüksek ortalamalara ve standart sapmalar görülmektedir. Bu geniş dağılımlara işaret etmektedir. DFS grafiklerinde ise daha düşük ortalama ve standart sapma değerlerinden dolayı daha dar dağılımlar görülmektedir.

Genel değerlendirildiğinde, Şekil 3.13'te görüldüğü gibi, ziyaret edilen hücre sayısı ve süre açısından üç algoritma arasında en yüksek değere sahip olan BFS algoritmasıdır. Onu A* algoritması takip etmektedir. En az ziyaret edilen hücre sayısına sahip olan ise DFS algoritmasıdır. Bu da DFS algoritmasının en hızlı olduğunu göstermektedir. Aynı boyutlarda ise engel oranı arttıkça iki durum ortaya çıkmaktadır. Eğer yol derinliği varsa, ziyaret edilen hücre sayısı artmaktadır ve algoritmalar daha fazla zaman harcamaktadır. Algoritmalar daha derin dallara ulaşmak için daha fazla adım atmak zorunda kalabilmektedir ve bu genellikle daha uzun sürecek bir arama sürecine yol açmaktadır. Diğer taraftan, dallanmanın az olduğu ve kapalı yolların çoğaldığı durumlarda ziyaret edilen hücre sayısı azalmaktadır.



Şekil 3.13 Dikdörtgen labirentlerde ziyaret edilen hücre sayısı grafiği (1:3 ve 3:1 oranları)

Ayrıca, labirent büyük ve karmaşık hale geldiğinde, algoritmaların daha derin dallara girmesi ve bu dalları araştırması daha uzun zaman alabilir. Bu durumda, algoritmalar tüm yol seçeneklerini tam olarak keşfetmeden önce çok sayıda dallanma yapabilir ve bu da kısa yolu bulmalarını zorlaştırabilir. Özellikle DFS algoritması, büyük labirentte sonsuz döngülere de takılabilir ve kısa yolu bulamayabilir. Bununla birlikte, bu algoritma, özellikle labirentin daha az dallandığı durumlarda ve hedefin daha derinlerde bulunması durumunda daha etkili olabilir. Ancak, labirentin büyümesiyle birlikte DFS'nin kısa yolu bulma yeteneği azalabilir. En kısa yolu bulmada BFS ve A* algoritması daha başarılıdır. Hem çözme hızı hem de en kısa yolu bulma açısından A* algoritması avantajlıdır.

3.2.3 Bellek Kullanım Analizi

Bu bölümde, farklı labirent çözme algoritmalarının bellek kullanım analizi yapılmaktadır. Bellek kullanımındaki veriler, algoritmaların çalışması sırasında RAM'in ne kadarını sistem tarafından kullanıldığını yüzde olarak ifade eder. Büyük ölçekli labirentlerde hafıza kullanımı performans açısından kritik bir faktördür.

Tablo 3.12 Kare labirentlerde bellek kullanımı ortalamaları (%)

Boyut	Engel Oranı	A*	DFS	BFS
10x10	0.1	13.47	13.43	13.53
	0.2	13.45	13.42	13.46
	0.3	13.42	13.41	13.42
100x100	0.1	14.89	14.78	14.90
	0.2	14.53	14.48	14.52
	0.3	14.22	14.09	14.14
500x500	0.1	17.79	16.38	16.45
	0.2	16.67	15.99	16.11
	0.3	15.23	15.17	15.24
1000x1000	0.1	33.68	26.20	26.41
	0.2	26.65	21.17	21.27
	0.3	17.66	17.10	17.36
2000x2000	0.1	94.68	40.16	42.33
	0.2	88.09	37.20	39.42
	0.3	31.79	25.46	27.99

Tablo 3.11'de kare labirentlerden bellek kullanım verileri alınmıştır. A* algoritması maliyet hesapları gerçekleştirmektedir ve tüm olası yolları hafızada tutmaktadır, bu da işlenen veri miktarını artmaktadır. Bu sebeple 2000x2000 boyutlarındaki labirentte bu algoritmanın bellek kullanım değerleri yüksektir. Diğer taraftan, BFS önceki seviyelerdeki düğümleri işledikten sonra bellekten çıkarır. BFS, genellikle daha az bellek tüketen bir algoritma olmasına rağmen, A* gibi bir tahmin yapmadığı için bazı durumlarda daha fazla adım atabilmektedir.

Tablo 3.13 Dikdörtgen labirentlerde bellek kullanımı ortalamaları (%) (1:2 ve 2:1 oranları)

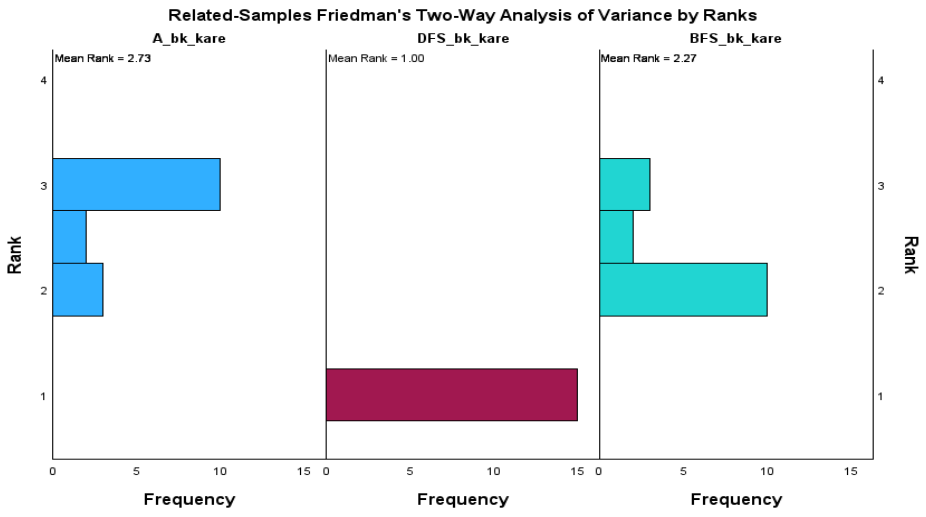
Boyut	Engel Oranı	A*	DFS	BFS
10x5	0.1	12.90	12.70	12.79
5x10	0.1	12.82	12.71	12.81
10x5	0.2	12.76	12.57	12.75
5x10	0.2	12.75	12.52	12.75
10x5	0.3	12.36	12.30	12.34
5x10	0.3	12.36	12.28	12.34
100x50	0.1	14.06	13.96	14.02
50x100	0.1	14.01	13.80	13.99
100x50	0.2	13.50	13.36	13.45
50x100	0.2	13.51	13.40	13.50
100x50	0.3	13.01	12.90	13.00
50x100	0.3	13.19	13.08	13.18
1000x500	0.1	20.64	18.20	18.34
500x1000	0.1	20.55	17.92	18.04
1000x500	0.2	20.06	18.11	18.17
500x1000	0.2	19.86	18.15	18.29
1000x500	0.3	15.84	15.77	15.81
500x1000	0.3	17.47	17.34	17.43

Tablo 3.12'de daha geniş (örneğin 1000x500) veya daha uzun (500x1000) labirentlerde A*, DFS ve BFS algoritmaları bellek kullanımında farklı sonuçlar göstermiştir. Özellikle A* ve BFS algoritmalarında büyük boyutlu labirentlerde daha fazla hücre ziyaret edildiği için bu bellek kullanımını etkilemiştir.

Tablo 3.14 Dikdörtgen labirentlerde bellek kullanımı ortalamaları (%) (1:3 ve 3:1 oranları)

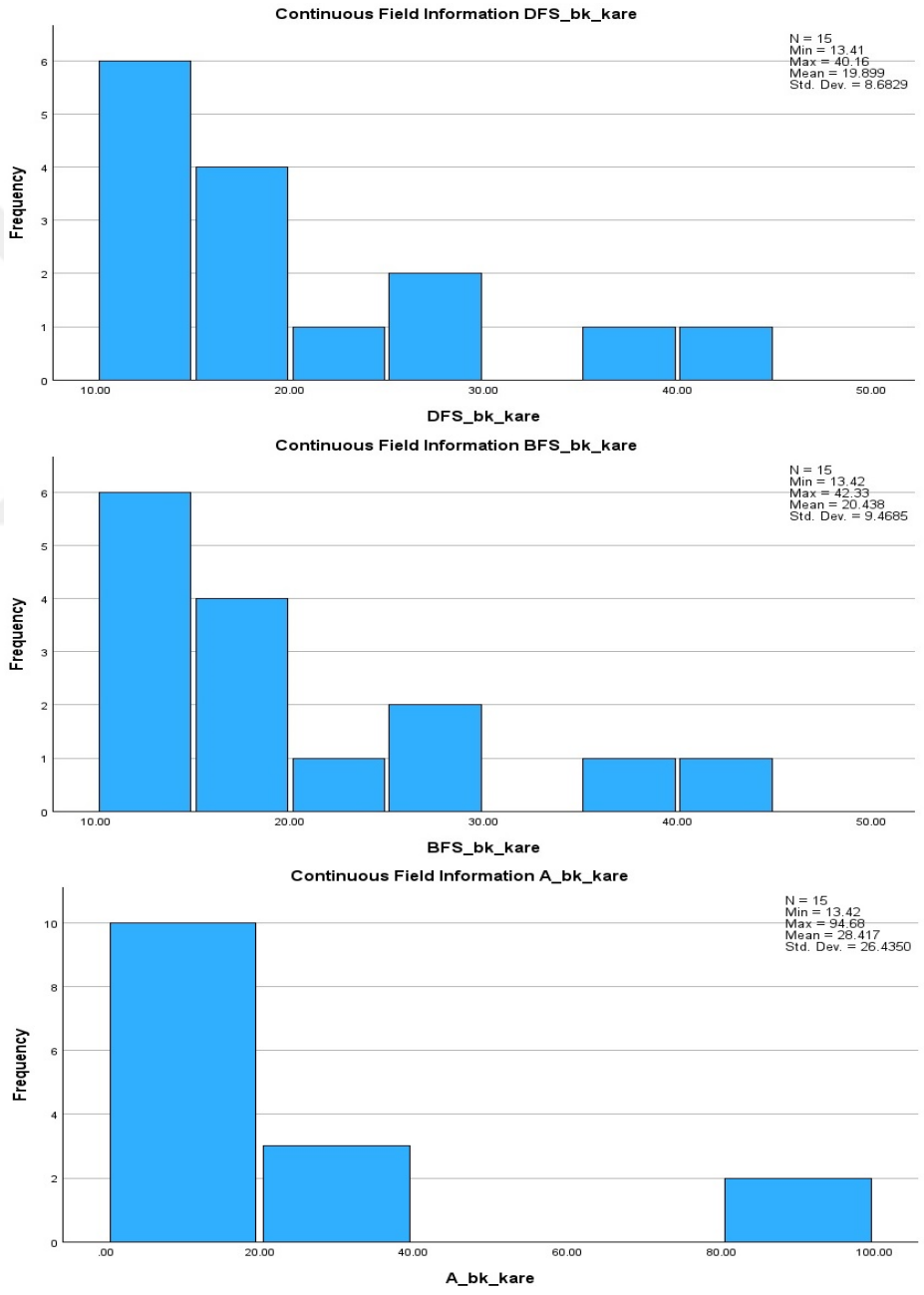
Boyut	Engel Oranı	A*	DFS	BFS
15x5	0.1	12.98	12.79	12.82
5x15	0.1	13.00	12.94	12.98
15x5	0.2	12.98	12.72	12.90
5x15	0.2	12.96	12.76	12.86
15x5	0.3	12.69	12.49	12.68
5x15	0.3	12.73	12.52	12.72
150x50	0.1	14.71	14.54	14.68
50x150	0.1	14.69	14.52	14.64
150x50	0.2	14.71	14.51	14.70
50x150	0.2	14.67	14.50	14.60
150x50	0.3	14.20	14.10	14.19
50x150	0.3	14.20	14.18	14.19
1500x500	0.1	28.77	23.93	24.13
500x1500	0.1	28.32	22.77	22.91
1500x500	0.2	22.19	20.31	20.36
500x1500	0.2	18.62	16.95	16.99
1500x500	0.3	17.47	17.44	17.37
500x1500	0.3	17.51	17.27	17.33

Tablo 3.11-3.13'e göre kare ve dikdörtgen labirentlerde küçük boyutlarda üç algoritmanın da ortalama bellek kullanımları birbirine yakın gelmektedir.



Şekil 3.14 Bellek kullanımında Friedman iki yönlü varyans analizi

Bellek kullanımı değerlerine Friedman testi uygulanmıştır. Şekil 3.14'te üç algoritmanın bellek kullanımı açısından performanslarını karşılaştırmalı olarak sunmaktadır. Sonuçlara göre, gruplar arasında anlamlı farklar bulunmaktadır. DFS grubu, en düşük ortalama sıralama ile en iyi performansı gösterirken, A* grubu en yüksek ortalama sıralama ile en kötü performansı göstermektedir. BFS grubu ise orta seviyede bir performans sergilemektedir. Bu bulgular, Friedman testi sonuçları ve çift yönlü karşılaştırmalar ile tutarlıdır.



Şekil 3.15 Bellek kullanım histogram grafikleri

Şekil 3.15’te DFS ve BFS algoritmalarının ortalama ve standart sapma değerleri yakındır ve her ikisi de benzer dağılım göstermektedir. Değerlerin çoğu düşük aralıkta yoğunlaşmış olup bazı orta ve yüksek değerler de mevcuttur. Her iki grafikte de 10 değeri en yüksek frekansa sahiptir. A* ise yüksek ortalama ve standart sapma değerlerine sahiptir; değerlerin çoğunluğu küçük aralıkta yoğunlaşmışken çok sayıda yüksek değer de bulunmaktadır. Genel olarak, A*, DFS ve BFS algoritmaları arasında tutarlılık gözlemlenmiştir.

Tablo 3.11-3.13'te görüldüğü üzere, aynı boyuttaki labirentlerde karmaşıklık arttıkça bellek kullanımında azalma gözlenmiştir. Özellikle küçük ve orta boyutlu labirentlerde bellek değerleri birbirine yakın olduğundan, algoritmaların performans karşılaştırmalarında bellek kullanımı, yol uzunluğu ve ziyaret edilen hücre sayısına göre daha az etkilenmiştir. Ancak büyük boyutlu labirentlerde bellek kullanım değerleri belirgin şekilde farklılık göstermektedir. Bununla birlikte, aynı karmaşıklıkta boyut büyüdükçe bellek kullanımı artmaktadır.

A*, DFS ve BFS algoritmaları, hem kare hem de dikdörtgen labirentlerde benzer bellek kullanım aralıkları sergilemektedir. Dikdörtgen labirentlerde, özellikle çok uzun veya geniş olanlarda, bazı algoritmaların performansları daha belirgin hale gelmektedir.

DFS ve BFS algoritmalarının bellek kullanımları birbirine yakın olsa da, BFS daha yüksek bellek kullanımı gerçekleştirmektedir. BFS, tüm düğümleri genişlik önceliğiyle taramak için bir kuyruk kullanır, bu da özellikle büyük labirentlerde veya sonsuz dallı ağaçlarda bellek kullanımını artırır. DFS ise derinlik önceliğiyle tarama yapar ve bir yığın kullanır. Bu sayede BFS'ye göre daha az bellek kullanır, ancak sonsuz dallı veya derin ağaçlarda bellek kullanımı hızla artabilmektedir. Sonuç olarak, DFS algoritması basit bir yapıya sahiptir ve bellek kullanımı düşüktür, ancak çizge çok büyük veya sonsuz döngülere sahipse, DFS algoritmasının bellek kullanımı artabilir. A* algoritması ise hedefe yönelik aramalarda oldukça etkilidir. Ancak her düğüm için maliyet ve tahmini toplam maliyet hesaplaması yapması gerektiğinden, daha fazla bellek kullanır. Bu nedenle, özellikle büyük ve karmaşık problemler için A* algoritması daha fazla bellek tüketmektedir.

A*, DFS ve BFS algoritmalarının RAM kullanımını karşılaştırmak genellikle zordur. Bunun nedeni, kullanılan dil, programın yapısı ve algoritmanın nasıl uygulandığı gibi faktörlere bağlı olmasıdır. Büyük labirentlerde yolları hafızada tutmak için daha fazla RAM gereksinimi ihtiyaç duymaktadır.

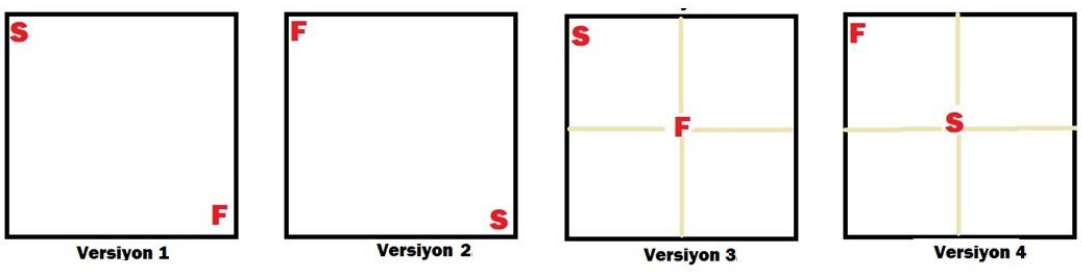
3.2.4 Başlangıç ve Bitiş Noktalarının Konumlarının Analizi

Bu bölümde, labirentlerde farklı başlangıç ve bitiş noktalarının algoritma performansı üzerindeki etkileri analiz edilmektedir. Bu analizde, çeşitli başlangıç ve bitiş noktası konfigürasyonları incelenerek yol uzunlukları, ziyaret edilen hücre sayıları, bellek kullanımı ve sürenin performans metrikleri karşılaştırılmaktadır.

```
for row, col in graph.keys():
    if row < height - 1 and not maze[row + 1][col]:
        graph[(row, col)].append(("D", (row + 1, col)))
        graph[(row + 1, col)].append(("U", (row, col)))
    if col < width - 1 and not maze[row][col + 1]:
        graph[(row, col)].append(("R", (row, col + 1)))
        graph[(row, col + 1)].append(("L", (row, col)))
```

Şekil 3.16 Yön kod satırları

Şekil 3.16'da yön için kod satırları yukarıda verilmiştir. Bu çalışmada A* ve BFS kuyruk çalışma mantığıyla aşağı, sağ, yukarı ve sol sırayla çalışırken DFS yığın prensibiyle ise sağ, aşağı, sol, yukarı sırasıyla çalışmaktadır. Algoritmaların yön sırası labirentlerin başlangıç ve bitiş noktalarının değiştiği durumlarda yol uzunluğu, ziyaret edilen hücre sayısını, bellek kullanımını ve süreyi etkilemektedir. Analiz için Şekil 3.17'deki görüldüğü gibi başlangıç ve bitiş noktalarının yerleri değiştirilerek 4 farklı versiyon belirlendi.



Şekil 3.17 Farklı başlangıç ve bitiş durumlarının görseli

Şekil 3.17'deki gibi versiyon 1 ve 2'de başlangıç veya bitiş noktaları haritaların en uçlarında iken versiyon 3 ve 4'te başlangıç veya bitiş noktaları ortalarda olacak şekilde tasarlanmıştır. S başlangıç noktasını, F bitiş noktasını ifade etmektedir.

Tablo 3.15 100x100'de farklı başlangıç ve bitiş için A* algoritması ortalamaları

Labirentler	Engel Oran	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 1	0.1	198.00	7719.63	14.89	46.76
	0.2	198.06	5108.13	14.53	27.42
	0.3	201.80	2799.16	14.22	14.84
Versiyon 2	0.1	198.00	7770.80	19.41	43.01
	0.2	198.00	4907.80	19.42	24.96
	0.3	201.20	2366.50	19.32	12.37
Versiyon 3	0.1	100.00	2035.80	18.08	11.96
	0.2	100.00	1297.83	18.12	6.90
	0.3	104.33	899.76	18.16	4.64
Versiyon 4	0.1	100.00	2089.33	17.51	11.09
	0.2	100.40	1326.26	17.59	6.13
	0.3	103.66	763.10	17.56	3.67

Tablo 3.14'te 100x100 boyutlarında her versiyon için üç farklı zorluk seviyesinde ayrı ayrı 30'ar kez çalıştırılmış ve üç algoritmanın verilerinin ortalama değerleri alınmıştır.

Tablo 3.16 100x50 ve 50x100'de farklı başlangıç ve bitiş için A* algoritması ortalamaları

Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 1	100x50	0.1	148.00	3795.53	14.06	21.63
	50x100		148.00	3758.66	14.00	21.75
	100x50	0.2	148.13	2226.86	13.50	11.70
	50x100		148.00	2216.36	13.51	11.47
	100x50	0.3	156.40	1729.80	13.01	9.26
	50x100		154.80	1748.13	13.19	9.00

Tablo 3.17 devamı

Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 2	100x50	0.1	148.00	3748.13	19.22	31.06
	50x100		148.00	3779.46	19.25	30.67
	100x50	0.2	148.13	2173.90	18.70	16.14
	50x100		148.13	2339.36	18.81	16.10
	100x50	0.3	157.40	1786.66	18.12	15.28
	50x100		155.86	1778.66	18.29	14.00
Versiyon 3	100x50	0.1	75.00	990.06	17.23	8.90
	50x100		75.00	1007.33	17.01	7.43
	100x50	0.2	75.46	679.70	16.50	6.13
	50x100		75.33	660.46	16.52	5.47
	100x50	0.3	79.93	518.26	16.11	4.54
	50x100		79.33	534.16	16.29	4.00
Versiyon 4	100x50	0.1	75.00	1007.80	16.06	7.34
	50x100		75.00	1019.83	16.00	8.24
	100x50	0.2	76.00	701.90	15.58	4.80
	50x100		76.00	650.16	15.61	4.54
	100x50	0.3	79.40	527.23	15.01	3.27
	50x100		83.46	630.20	15.19	4.83

Tablo 3.15'te 100x50 ve 50x100 boyutlarında her versiyon için üç farklı zorluk seviyesinde ayrı ayrı 30'ar kez çalıştırılmış ve üç algoritmanın verilerinin ortalama değerleri alınmıştır.

Tablo 3.18 50x150 ve 150x50'de farklı başlangıç ve bitiş için A* algoritması ortalamaları

Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 1	150x50	0.1	198.00	5319.24	14.71	21.63
	50x150		198.00	5399.30	14.68	21.75
	150x50	0.2	199.00	3207.56	14.70	11.70
	50x150		198.93	3071.73	14.67	11.47
	150x50	0.3	219.00	3299.86	14.19	9.26
	50x150		214.13	3156.16	14.20	9.00

Tablo 3.19 devamı

Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 2	150x50	0.1	198.00	5365.63	19.96	33.09
	50x150		198.00	5332.70	19.92	41.58
	150x50	0.2	199.60	3321.30	19.82	24.32
	50x150		199.20	3206.06	19.84	28.60
	150x50	0.3	217.60	3320.26	19.63	21.25
	50x150		219.80	3362.03	19.61	23.66
Versiyon 3	150x50	0.1	100.00	1419.80	17.83	10.47
	50x150		100.06	1445.13	17.53	15.69
	150x50	0.2	101.53	974.76	17.52	7.61
	50x150		101.53	985.86	17.58	7.74
	150x50	0.3	112.60	1071.73	17.21	7.73
	50x150		110.26	935.66	17.33	7.67
Versiyon 4	150x50	0.1	100.06	1370.76	16.60	9.93
	50x150		100.06	1410.66	16.71	9.93
	150x50	0.2	101.53	1000.23	16.46	7.06
	50x150		101.60	959.50	16.59	7.20
	150x50	0.3	112.40	1036.73	16.29	7.04
	50x150		111.86	1050.20	16.19	7.03

Tablo 3.16'da 150x50 ve 50x150 boyutlarında her versiyon için üç farklı zorluk seviyesinde ayrı ayrı 30'ar kez çalıştırılmış ve üç algoritmanın verilerinin ortalama değerleri alınmıştır.

Tablo 3.14-3.16'da görüldüğü üzere A* algoritmasında başlangıç ve bitiş arasındaki uzaklık fazla olduğundan sırasıyla versiyon 1 ve 2'de ziyaret edilen hücre sayısı, süre ve yol uzunluğu sırasıyla versiyon 3 ve 4'e göre daha yüksektir. Bu algoritma sezgisel çalışma mantığıyla bilinen bir algoritma olduğundan başlangıç ve bitiş noktalarının değişmesi versiyon 1 ile 2 aynı şekilde versiyon 3 ile 4 hakkında kıyaslanabilir sonuçlar mevcut değildir.

Tablo 3.20 100x100'de farklı başlangıç ve bitiş için DFS algoritması ortalamaları

Labirentler	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 1	0.1	230.93	255.43	14.78	0.37
	0.2	320.46	397.33	14.48	0.80
	0.3	407.73	1306.26	14.09	2.60
Versiyon 2	0.1	8261.86	8881.03	19.68	45.30
	0.2	6206.12	7594.64	19.56	28.54
	0.3	3504.00	5789.96	19.33	15.20
Versiyon 3	0.1	4037.26	4454.70	18.12	16.80
	0.2	2886.13	3895.96	18.15	11.76
	0.3	1615.80	3034.46	18.18	6.83
Versiyon 4	0.1	8204.26	8838.60	17.8	47.44
	0.2	6044.46	7490.93	17.71	28.47
	0.3	3300.46	5704.63	17.60	14.63

Tablo 3.17'de kare labirentlerde DFS algoritması versiyon 2 ve versiyon 4'te yol uzunluğu, süre ve ziyaret edilen hücre sayısı daha fazladır, versiyon 1'de düşük gelmiştir.

Tablo 3.21 100x50 ve 50x100'de farklı başlangıç ve bitiş için DFS algoritması ortalamaları

Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 1	100x50	0.1	193.53	216.60	13.96	0.20
	50x100		159.86	202.43	13.80	0.30
	100x50	0.2	266.13	330.03	13.36	0.54
	50x100		187.40	273.93	13.40	0.50
	100x50	0.3	320.00	546.96	12.90	0.87
	50x100		214.40	648.53	13.08	1.27
Versiyon 2	100x50	0.1	4085.00	4424.80	18.96	22.55
	50x100		4137.13	4444.30	18.81	20.28
	100x50	0.2	3021.06	3758.96	18.36	15.44
	50x100		3085.67	3800.43	18.55	15.71
	100x50	0.3	1569.13	2881.53	17.91	7.97
	50x100		1818.06	3025.60	18.07	8.58

Tablo 3.22 devamı

Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 3	100x50	0.1	2040.13	2156.20	16.95	9.48
	50x100		1911.40	2070.00	16.70	7.07
	100x50	0.2	1533.60	1863.76	16.37	6.33
	50x100		1161.66	1597.13	16.20	5.18
	100x50	0.3	904.80	1697.46	15.91	4.73
	50x100		660.40	1646.36	16.10	4.20
Versiyon 4	100x50	0.1	4057.46	4411.76	16.00	19.59
	50x100		4033.13	4418.33	15.93	19.01
	100x50	0.2	2968.06	3735.70	15.39	14.17
	50x100		2922.13	3731.00	15.52	13.50
	100x50	0.3	1565.13	2868.60	14.96	7.93
	50x100		1581.80	2944.56	15.06	8.30

Tablo 3.18'de dikdörtgen labirentlerde DFS algoritması versiyon 1'de yol uzunluğu, süre ve ziyaret edilen hücre sayısı daha düşüktür. En yüksek yol uzunluğu, süre ve ziyaret edilen hücre sayısı versiyon 2'dedir.

Tablo 3.23 150x50 ve 50x150'de farklı başlangıç ve bitiş için DFS algoritması ortalamaları

Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 1	150x50	0.1	262.00	268.34	14.54	0.53
	50x150		211.80	284.83	14.52	0.47
	150x50	0.2	375.26	464.53	14.51	0.67
	50x150		233.26	808.53	14.50	2.03
	150x50	0.3	537.33	851.70	14.10	1.30
	50x150		280.26	1435.86	14.18	2.87
Versiyon 2	150x50	0.1	6104.00	6644.20	19.46	36.35
	50x150		6229.00	6668.26	19.56	37.77
	150x50	0.2	4539.46	5626.73	19.50	23.91
	50x150		4650.00	5694.23	19.51	28.30
	150x50	0.3	2381.00	4332.86	19.12	11.75
	50x150		2715.40	4497.36	19.24	14.33

Tablo 3.24 devamı

Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 3	150x50	0.1	3112.46	3297.80	17.66	14.21
	50x150		2532.40	2809.10	17.41	14.16
	150x50	0.2	2438.0	3061.40	17.43	11.77
	50x150		1677.86	2565.00	17.44	9.58
	150x50	0.3	1364.86	2430.36	17.11	7.34
	50x150		897.66	2059.93	17.21	5.57
Versiyon 4	150x50	0.1	6120.53	6631.90	16.49	42.40
	50x150		6034.46	6621.70	16.59	46.72
	150x50	0.2	4424.00	5615.56	16.22	27.23
	50x150		4413.33	5632.40	16.39	28.59
	150x50	0.3	2239.46	4319.43	16.11	13.04
	50x150		2452.66	4337.63	16.17	13.94

Tablo 3.19'da versiyon 2 ve versiyon 4'ün yol uzunluğu, süre ve ziyaret edilen hücre sayısı daha fazla gelmiştir. DFS algoritması LIFO ve rekürsif olarak çalışmaktadır. Dolayısıyla, DFS algoritmasında versiyon 1, versiyon 2'ye göre ve versiyon 3, versiyon 4'e göre ziyaret edilen hücre sayıları, süreleri ve yol uzunlukları bakımından daha kısadır. Başlangıç ve bitiş noktaları uç noktalarda iken versiyon 1'de oldukça verimli sonuçlar alınmaktadır. En kötü sonuçlar versiyon 2 ve versiyon 4'te görülmektedir. Aynı engel oranında; versiyon 1, versiyon 2'ye göre çok daha verimli sonuçlar vermektedir. Benzer şekilde, versiyon 3'ün versiyon 4'ten daha iyi olduğu gözlemlenmiştir. Ayrıca, versiyon 3 ve versiyon 4'te başlangıç ve bitiş noktaları ortada olduğundan, labirent içinde bitiş noktasına giden yol bulunurken alternatif yollar denendiği için ziyaret edilen hücre sayısı, süre ve yol uzunluğu yüksek gelmektedir.

Tablo 3.25 100x100'de farklı başlangıç ve bitiş için BFS algoritması ortalamaları

Labirentler	Engel Oran	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 1	0.1	198.00	8997.76	14.90	18.40
	0.2	198.06	7976.00	14.51	14.39
	0.3	201.80	6827.96	14.14	11.62

Tablo 3.26 devamı

Labirentler	Engel Oran	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 2	0.1	198.00	8997.63	16.36	18.42
	0.2	198.00	7978.12	18.85	15.52
	0.3	201.20	6843.20	22.49	15.20
Versiyon 3	0.1	100.00	4495.63	16.37	9.20
	0.2	100.00	3737.33	16.97	7.10
	0.3	104.33	2846.20	15.50	4.57
Versiyon 4	0.1	100.00	8998.00	16.96	18.83
	0.2	100.40	7973.90	18.45	15.70
	0.3	103.66	6786.36	16.69	11.60

Tablo 3.20'de kare labirentlerde BFS algoritması yol uzunluğu versiyon 1 ve versiyon 2'de benzer iken versiyon 3 ve versiyon 4'te de kendi aralarında benzerdir.

Tablo 3.27 100x50 ve 50x100'de farklı başlangıç ve bitiş için BFS algoritması ortalamaları

Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 1	100x50	0.1	148.00	4498.23	14.02	8.53
	50x100		148.00	4498.33	13.99	9.16
	100x50	0.2	148.13	3988.26	13.45	6.77
	50x100		148.00	3986.73	13.50	7.20
	100x50	0.3	156.40	3390.56	13.00	5.36
	50x100		154.80	3395.86	13.18	5.00
Versiyon 2	100x50	0.1	148.00	4497.83	19.11	12.63
	50x100		148.00	4498.50	19.01	12.48
	100x50	0.2	148.13	3987.30	18.68	11.76
	50x100		148.13	3987.83	18.74	10.27
	100x50	0.3	157.40	3390.06	18.12	9.17
	50x100		155.86	3388.26	18.13	8.23
Versiyon 3	100x50	0.1	75.00	2261.03	17.01	6.67
	50x100		75.00	2259.80	16.86	6.70
	100x50	0.2	75.46	1951.50	16.47	5.44
	50x100		75.33	1953.50	16.51	5.51
	100x50	0.3	79.93	1563.90	16.02	4.13
	50x100		79.33	1577.80	16.22	3.63

Tablo 3.28 devamı

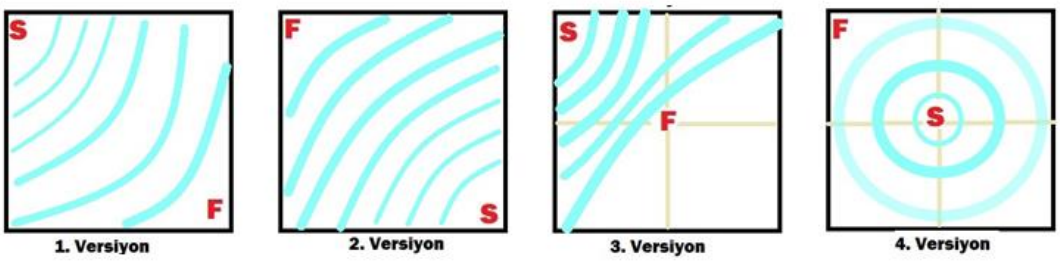
Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 4	100x50	0.1	75.00	4498.46	16.10	13.96
	50x100		75.00	4497.76	15.99	13.68
	100x50	0.2	76.00	3985.90	15.53	10.65
	50x100		76.00	3985.80	15.55	9.63
	100x50	0.3	79.40	3326.73	15.00	7.81
	50x100		83.46	3351.30	15.17	9.47

Tablo 3.20-22'de kare ve dikdörtgen labirentlerde yol uzunluğu versiyon 3 ve versiyon 4'te daha düşüktür.

Tablo 3.29 150x50 ve 50x150'de farklı başlangıç ve bitiş için BFS algoritması ortalamaları

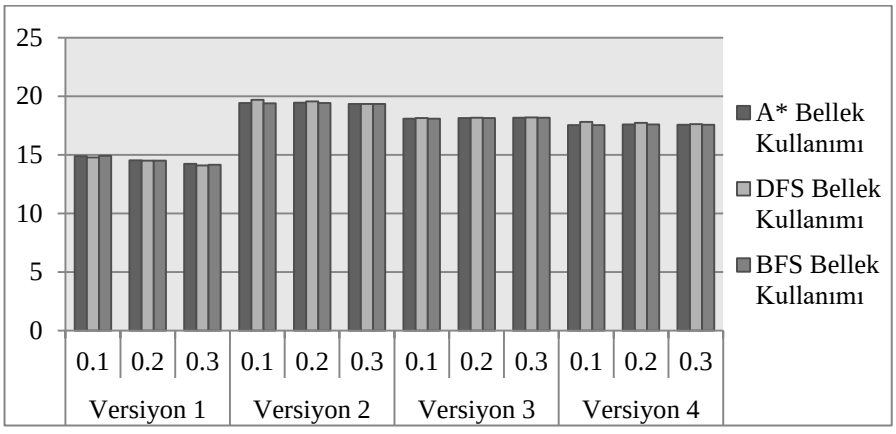
Labirentler	Boyut	Engel Oranı	Yol Uzunluğu	Ziyaret Edilen Hücre Sayısı	Bellek Kullanımı (%)	Süre (milisaniye)
Versiyon 1	150x50	0.1	198.00	6748.41	14.68	13.66
	50x150		198.00	6748.06	14.64	14.09
	150x50	0.2	199.00	5980.56	14.70	10.90
	50x150		198.93	5980.40	14.60	10.60
	150x50	0.3	219.00	5087.90	14.19	8.00
	50x150		214.13	5101.06	14.19	8.17
Versiyon 2	150x50	0.1	198.00	6747.53	19.75	16.69
	50x150		198.00	6747.70	19.71	19.12
	150x50	0.2	199.60	5979.70	19.59	14.31
	50x150		199.20	5980.20	19.59	16.60
	150x50	0.3	217.60	5083.13	19.19	9.45
	50x150		219.80	5102.30	19.49	12.91
Versiyon 3	150x50	0.1	100.00	3375.10	17.78	16.97
	50x150		100.06	3376.63	17.50	16.93
	150x50	0.2	101.53	2920.50	17.46	15.07
	50x150		101.53	2915.83	17.44	16.35
	150x50	0.3	112.60	2489.70	17.19	11.97
	50x150		110.26	2433.93	17.27	11.57
Versiyon 4	150x50	0.1	100.06	6747.90	16.56	9.07
	50x150		100.06	6747.13	16.63	11.43
	150x50	0.2	101.53	5975.46	16.41	7.16
	50x150		101.60	5972.36	16.46	8.33
	150x50	0.3	112.40	5060.26	16.19	5.77
	50x150		111.86	5045.86	16.19	5.80

Tablo 3.20-3.22'de ziyaret edilen hücre sayısı ve süre versiyon 1, versiyon 2 ve versiyon 4'te birbirine yakın çıkmıştır. Versiyon 3'te daha düşük çıkmıştır. Ziyaret edilen hücre sayısı bakımından BFS algoritması çalışma mantığı gereği kuyruk algoritması olsa da tüm haritayı taradığından versiyon 3 hariç değerler birbirine yakın çıkmıştır.

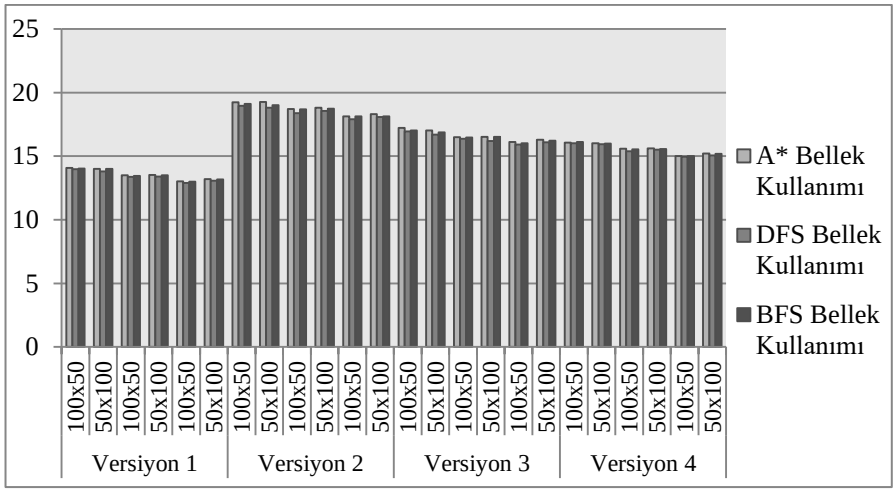


Şekil 3.18 BFS'nin versiyonlara göre labirenti arama görseli

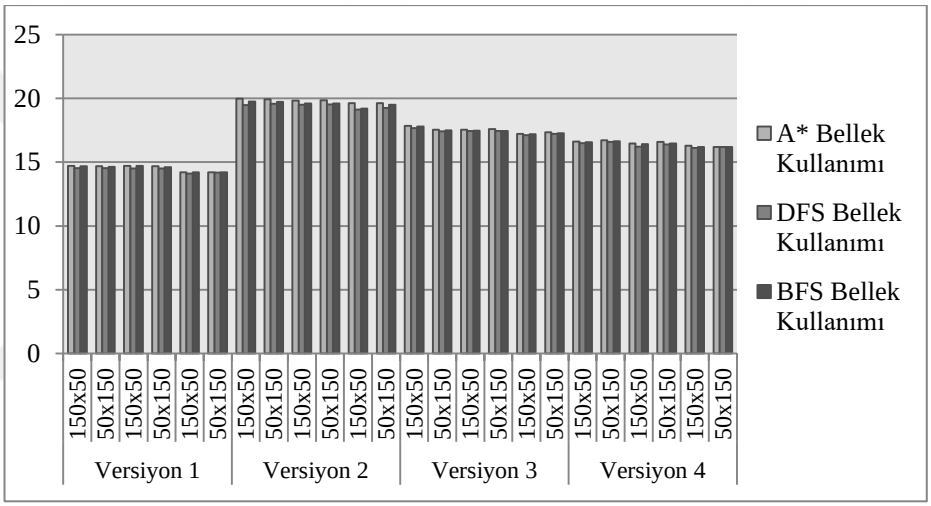
Şekil 3.18'de BFS'nin versiyon 3'te haritanın yarısı veya daha azını taraması hedefi bulması için yeterli olmuştur. Engel oranı arttıkça ziyarete kapalı bölgeler oluştuğundan ziyaret edilen hücre sayısı ve süre azalmıştır. Bunun yanında, A* ve BFS algoritmaları dört versiyonda da en kısa yolu bulmuştur.



Şekil 3.19 100x100'de farklı başlangıç ve bitişlerde bellek kullanımı



Şekil 3.20 100x50 ve 50x100'de farklı başlangıç ve bitişlerde bellek kullanımı



Şekil 3.21 150x50 ve 50x150'de farklı başlangıç ve bitişlerde bellek kullanımı

Şekil 3.19-3.21'de kare ve dikdörtgen labirentlerde ortalama bellek kullanım değerleri versiyon 1'de bellek kullanımı en az iken versiyon 2'de bellek kullanımı en yüksek gelmiştir. Çünkü versiyonda 2'de bu algoritmadaki yön sıralama stratejisi sebebiyle bitiş noktasına ulaşmak için daha çok adım atmaktır. Başlangıç ve bitiş noktaları birbirinden uzak oldukça algoritmalar daha da çok çaba göstermek durumunda kalmaktadır.

A*, BFS ve DFS algoritmalarının yol uzunluğu, ziyaret edilen hücre sayısı, bellek kullanımı ve süre tutarlılıkları SPSS yardımıyla incelenmiştir.

Tablo 3.30 Farklı başlangıç ve bitişler için ki-kare testi sonuçları

Gözlemler	Ki-Kare	df	Anlamlılık
Yol uzunluğu	24.00	1	<.001
Ziyaret edilen hücre sayısı	36.58	2	<.001
Bellek kullanımı	34.33	2	<.001

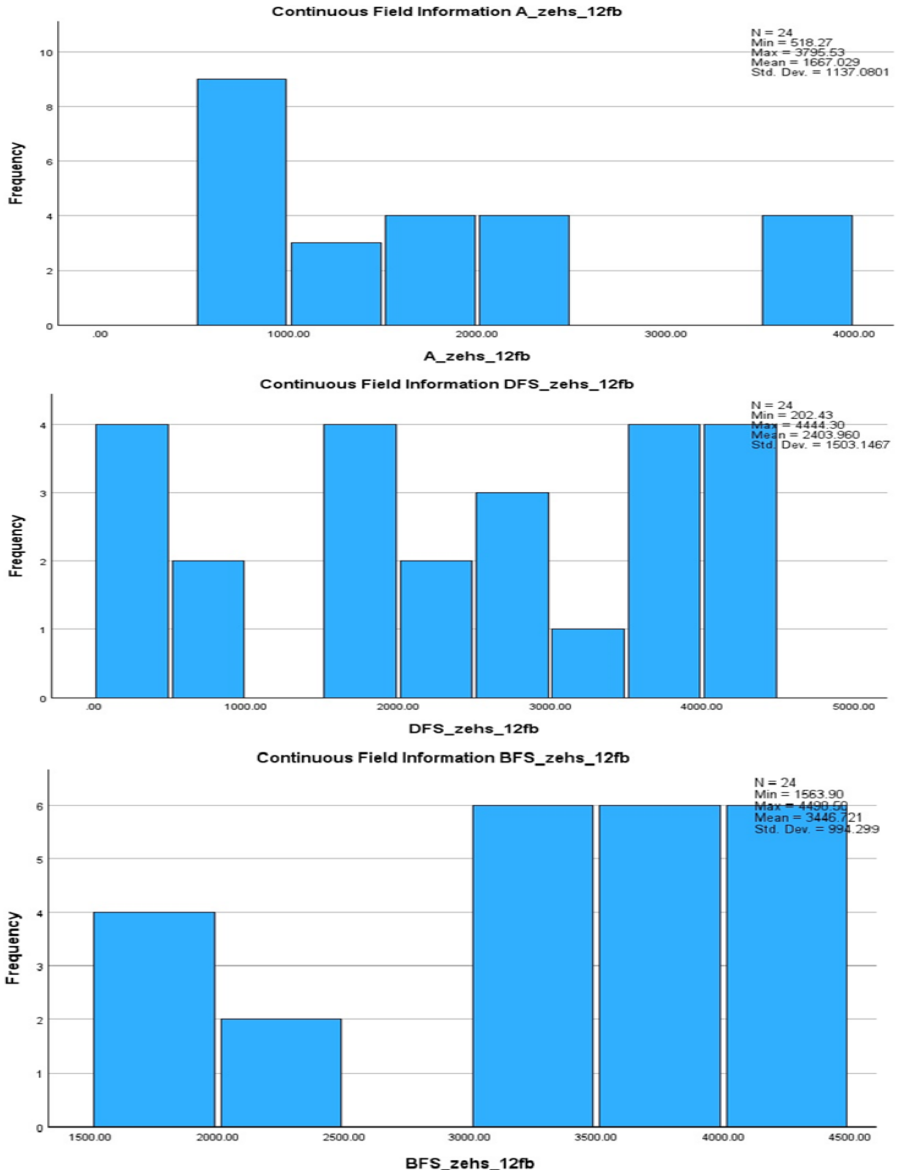
Tablo 3.23'te Friedman testi kullanılarak yapılan analiz sonucunda ki-kare değerleri yüksek ve asimptotik anlamlılık değerleri (Asymp. Sig.) <.001 olması algoritmalar arasındaki performans farkının istatistiksel olarak anlamlı olduğunu göstermektedir.

Tablo 3.31 Farklı başlangıç ve bitişler için yol uzunluğu Friedman test sonuçları

Yol Uzunluğu Ortalama Sıralama	
A* ve BFS	1.00
DFS	2.00

Tablo 3.24'te algoritmaların yol uzunluğu ortalama sıralamalarına bakıldığında, A* ve BFS algoritmalarının ortalama sıralama değeri 1.00 olarak hesaplanmıştır. Diğer yandan, DFS algoritmasının ortalama sıralama değeri ise 2.00 olarak belirlenmiştir. Bu bulgular, bakıldığında, A* ve BFS algoritmalarının tercih edilmesi gereken algoritma olduğunu göstermektedir.

Şekil 3.22'de ziyaret edilen hücre sayılarına göre BFS algoritmasında en yüksek ortalamalar ve en düşük standart sapmalar görülmektedir. DFS algoritmasında ise en düşük ortalamalar ve en yüksek standart sapmalar gözlemlenmektedir. Grafiklerde genel olarak farklı dağılımlar ve yayılımlar görülmektedir. DFS daha geniş dağılım ve BFS daha dar dağılım özellikleri göstermektedir. A* algoritmasında dağılımlar daha çeşitli ve düşük değerlere sahiptir.

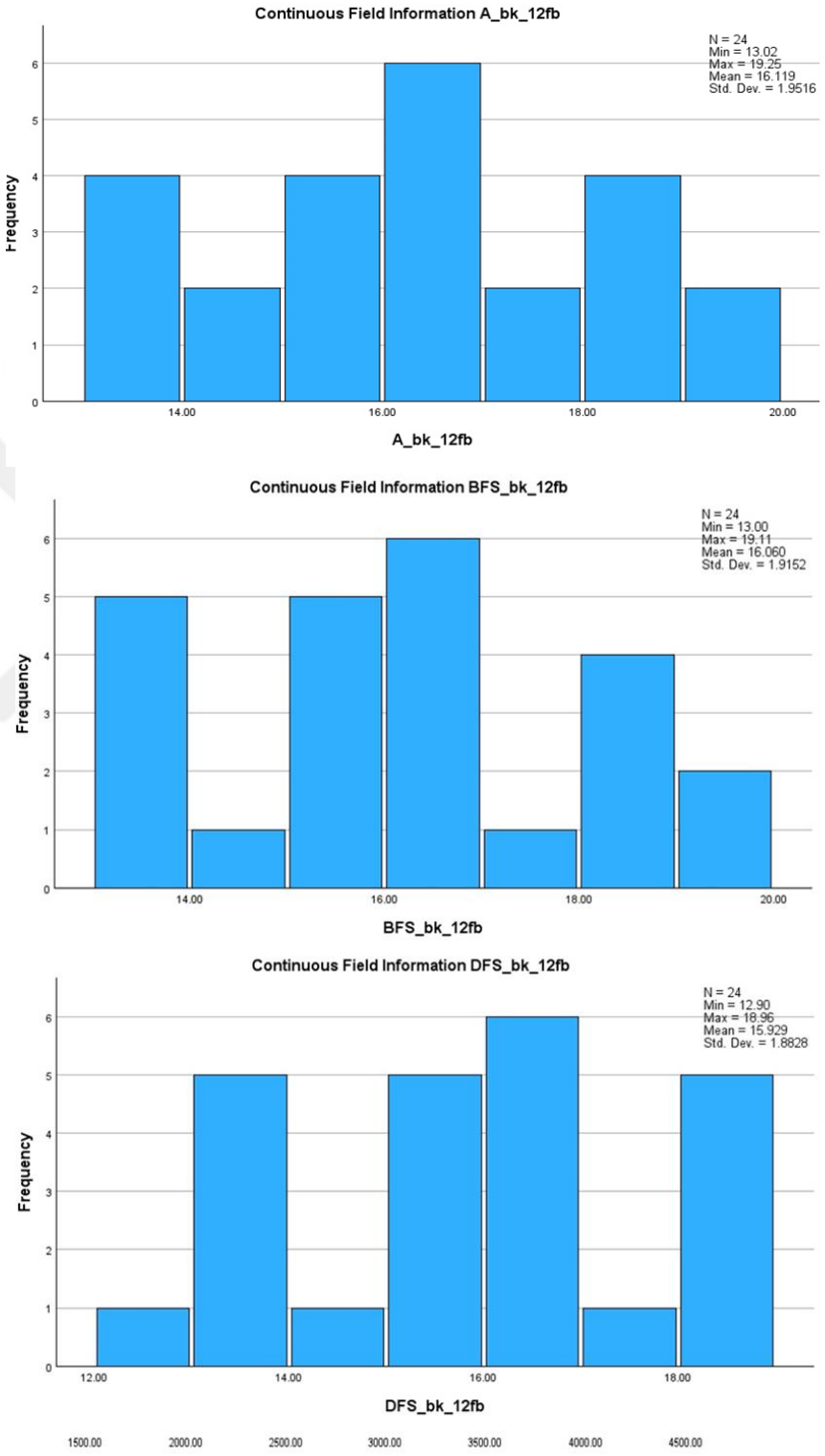


Şekil 3.22 Farklı başlangıç ve bitişler için ziyaret edilen hücre sayısı histogram grafiği

Tablo 3.32 Farklı başlangıç ve bitişler için bellek kullanımı ikili karşılaştırma testi sonuçları

Algoritma ikilileri	Test İstatistiği	St. Sapma	St. Test İstatistiği	Sig.	Adj. Sig.[a]
DFS-BFS	-1.042	0.289	-3.608	<.001	0.001
DFS-A*	1.958	0.289	6.784	<.001	0.000
BFS-A*	0.917	0.289	3.175	.001	0.004

Tablo 3.25'e göre bellek kullanımında DFS algoritması, hem BFS hem de A* algoritmalarına göre anlamlı derecede daha iyi performans göstermektedir. BFS algoritması, A* algoritmasına göre daha iyi performans göstermektedir.



Şekil 3.23 Farklı başlangıç ve bitişler için bellek kullanımı histogram grafiği

Şekil 3.23'e göre üç algoritma için bellek kullanımların ortalama, standart sapma ve frekans dağılımları yakın olmakla birlikte küçük farklılıklar mevcuttur. DFS, BFS ve A* algoritmalarının dağılımı belirli aralıklarda yoğunlaşmış olup, belirli değerlerde yüksek frekansa sahiptir. Standart sapmalarının düşük olması, veri setinde önemli ölçüde aykırı değerlerin az olduğunu göstermektedir. DFS'nin sıralama, ortalama, standart sapma ve frekans dağılımları BFS ve A* algoritmalarına nazaran daha iyidir. DFS algoritması daha iyi performans göstermektedir. Özetle, analiz sonuçlarına göre yol uzunluğu, ziyaret edilen hücre sayısı, bellek kullanımı ve süre veri setlerinde tutarlı ve güvenilir bir dağılım sergilemektedir.

Genel olarak, başlangıç ve bitiş noktalarının farklı konumları labirentin yapısının keşfedilmesi ve çözümün bulunması sürecini de etkileyebilmektedir. Başlangıç ve bitiş noktaları arasındaki uzaklık labirentin çözülmesi için gereken adımları etkilemektedir. Daha uzak bir başlangıç noktasından başlamak, daha fazla adım atılmasını gerektirmekte ve bu da çözümün zamanını ve zorluğunu artırmaktadır. Ayrıca, bazı durumlarda başlangıç noktasından bitiş noktasına ulaşmak için farklı yollar denenmesi gerekebilmektedir, bu da problemi daha karmaşık hale getirebilmektedir. Başlangıç ve bitiş noktalarının farklı olması, labiretin çözümünü çeşitli açılardan düşünmeye ve farklı stratejiler geliştirmeye teşvik etmektedir.

BÖLÜM DÖRT
SONUÇLAR VE ÖNERİLER

Analizler, Intel(R) Core(TM) i7-3630M CPU @ 2.20GHz 16 GB ram, 64 bit işletim sistemi özeliklerinde dizüstü bilgisayardan alınmıştır. Python programla diliyle Pycharm idesi kullanılmıştır. Çalışmada çeşitli senaryolarda incelemeler yapılmış ve SPSS ile analiz edilmiştir.

DFS, BFS ve A* algoritmaları, labirentler gibi çizge tabanlı problemlerin çözümünde kullanılmaktadır. DFS algoritması, bir yolu sonuna kadar takip ederek hedefe ulaşmaya çalışır. Eğer DFS, fazla dallara girmeden çözüm bulabilirse, harcadığı zaman ve bellek miktarı daha az olur. Ancak, sonsuz uzunlukta bir dalda sıkışma riski vardır. Büyük boyutlu labirentlerde, doğru yolu bulmak için gereken adım sayısı artabilir ve bu zaman alabilir. Yine de, daha az bellek tüketir ve daha basit bir yapıya sahiptir. DFS, genel olarak en kısa yolu bulmada etkili değildir ancak yolları gezerek bir çözüm bulabilir. DFS algoritmasında ziyaret edilen hücre sayısı genellikle daha düşüktür ve daha hızlıdır. Küçük boyutlu labirentlerde alternatif çözüm bulma amaçlı kullanılabilir. Ancak, labirentlerin boyutu ve zorluk seviyeleri arttıkça DFS algoritmasının çözüm bulma süresi olumsuz etkilenir.

BFS algoritması, labirentteki her seviyedeki hücreleri sırayla ziyaret ederek hedefe ulaşmaya çalışır. Bu algoritma, hedefe doğrudan gitmek yerine tüm yolları deneyerek sonuca ulaşır, bu da hesaplama süresini artırabilir. DFS, bir yolda sonuna kadar ilerler ve ardından geri dönerken, BFS bir seviyedeki tüm hücreleri ziyaret eder ve ardından bir sonraki seviyeye geçer. DFS, genellikle daha az hücre ziyaret eder çünkü derinlemesine ilerledikçe daha az dallanma gösterir. Ancak bu, labirentin yapısına ve hedefin konumuna bağlıdır. BFS ise her seviyedeki hücreleri ziyaret ettiği için daha fazla hücre ziyaret eder. BFS mutlaka bir çözüm varsa bulur ve genellikle daha az adım gerektiren çözümü belirler. BFS, başlangıç noktasından hedefe olan en kısa yolu bulur. DFS ve BFS algoritmalarının bellek kullanımı benzer olmakla birlikte, BFS genellikle daha yüksek bellek kullanımı gerektirir.

A* algoritması, her adımda hedefe doğru ilerlerken, bir tahmin fonksiyonu kullanarak en uygun yolu belirler. Bu sayede, diğer algoritmalara göre daha verimli

bir şekilde sonuca ulaşabilir. Ancak, büyük boyutlu labirentlerde kaynak tüketimi artabilir. Yürütme hızı büyük ölçüde tahmin fonksiyonunun doğruluğuna bağlı olduğundan karmaşıklık sorunları vardır. Ancak en uygun yolu bulmak için en az sayıda düğümü genişletir. Ziyaret edilen hücre sayısı ve yol uzunluğu açısından en etkili yolu bulur. Hem en kısa yolu bulmada etkilidir hem de zaman açısından kabul edilebilir bir denge sunar. Süre ve kısa yol bulma açısından A* algoritması tercih edilebilir.

Labirentlerde A*, DFS ve BFS algoritmaları arasında seçim yaparken, labirentin boyutu, bellek kullanımı ve hedefe en kısa yolla ulaşma gerekliliği gibi faktörleri göz önünde bulundurmak önemlidir. A* algoritması, birçok problemde tercih edilen bir algoritmadır. Oyun haritaları gibi başlangıç ve bitiş noktası bilinen durumlarda A* algoritması kullanılabilir. Büyük boyutlu, kör yürüyüş yapılan haritalar için de BFS uygundur. Robotik alanında ise DFS algoritması tercih edilebilir.

Sonuç olarak, büyük boyutlu labirentlerde A*, DFS ve BFS algoritmalarının performansları farklılık gösterebilir. Her bir algoritmanın avantajları ve dezavantajları farklıdır ve uygulamanın gereksinimlerine göre seçilmelidir. Hangi algoritmanın kullanılacağı, uygulamanın gereksinimlerine, labirentin boyutu ve karmaşıklığı gibi faktörlere bağlıdır.

Günümüzde bilgisayar teknolojilerinin hızla gelişmesi, daha büyük ve karmaşık labirentlerin çözümüne yönelik araştırmalarda önemli fırsatlar sunmaktadır. Daha gelişmiş bilgisayarlar kullanılarak, büyük labirentlerde çeşitli algoritmaların performansını incelenebilir. Bu bağlamda, farklı algoritmaların büyük ölçekli labirentlerde nasıl davrandığını anlamak ve karşılaştırmak için kapsamlı çalışmalar yürütülebilir. Ayrıca, mevcut algoritmaların labirent içerisinde çözüm ararken daha verimli hale getirilmesi için modifikasyonlar yapılabilir. Algoritmaların iyileştirilmesi, çözüm sürelerini kısaltabilir, bellek kullanımını optimize edebilir ve genel performansı artırabilir. Bu tür iyileştirmeler, hem akademik araştırmalar hem de pratik uygulamalar için değerli katkılar sağlayabilir.

KAYNAKLAR

Alamri, S., Alshehri, S., Alshehri, W., Alamri, H., Alaklabi, A. ve Alhmiedat, T. (2021). Autonomous maze solving robotics: *Algorithms and systems. International Journal of Mechanical Engineering and Robotics Research, 10*(12), (668-675).

Aqel, M. O., Issa, A., Khdair, M., ElHabbash, M., AbuBaker, M. ve Massoud, M. (Ekim 2017). Intelligent maze solving robot based on image processing and graph theory algorithms. *In 2017 International Conference on Promising Electronic Technologies (ICPET)* (48-53). IEEE.

Babula, M. (2009). Simulated maze solving algorithms through unknown mazes. *Organizing and Program Committee, 13.*

Bienias, Ł., Szczepański, K. ve Duch, P. (2016). Maze exploration algorithm for small mobile platforms. *Image Processing & Communications, 21*(3), 15.

Chang, K. C., Zhou, Y., Shoaib, A. M., Chu, K. C., Izhar, M., Ullah, S. ve Lin, Y. C. (Mart 2020). Shortest Distance Maze Solving Robot. *In 2020 IEEE International Conference on Artificial Intelligence and Information Systems (ICAIIS)* (283-286). IEEE.

Charniak, E. (1985). Introduction to Artificial Intelligence. *Addison-Wesley Publishing Company, USA,*, (260-267).

Cui, X. ve. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security, 11*(1), (125-130).

Çakraborti, A. (27 Ağustos 2019). Applications of DFS and BFS in Data Structures https://www.tutorialspoint.com/applications-of-dfs-and-bfs-in-data-structures

Debnath, S. K., Omar, R., Latip, N. A., Shelyna, S., Nadira, E., Melor, C. K. N. C. K., Chakraborty, T. K. ve Natarajan, E. (2019). A review on graph search algorithms for optimal energy efficient path planning for an unmanned air vehicle. *Indonesian Journal of Electrical Engineering and Computer Science, 15*(2), (743-749).

Gordon, V. S. ve Matley, Z. (Haziran 2004). Evolving sparse direction maps for maze pathfinding. *In Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753) (Cilt 1, 835-838). IEEE.*

Gupta, B. ve Sehgal, S. (Eylül 2014). Survey on techniques used in autonomous maze solving robot. *In 2014 5th International Conference-Confluence The Next Generation Information Technology Summit (Confluence)* (323-328). IEEE.

Hart, P. E., Nilsson, N. J. ve Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics, 4*(2), (100-107).

Hidayat, W., Susanti, F. ve Wijaya, D. (2021). A comparative study of informed and uninformed search algorithm to solve eight-puzzle problem. *Journal of Computer Science, 17*(11), (1147-1156).

Hii, J. Y., Lee, J. ve Chuah, Y. (2021). Optimization and Simulation of A Navigation Robot in Mazes. *Int. J. Eng. Res. Technol,, 10*(05), (222-227).

Iloh, P. C. (2022). A Comprehensive and comparative study of DFS, BFS, and A* search algorithms in a solving the maze transversal problem. *International Journal of Social Sciences and Scientific Studies, 2*(2), (482-490).

Indriyono, B. V. (2021). Optimization of breadth-first search algorithm for path solutions in mazyin games. *International Journal of Artificial Intelligence & Robotics (IJAIR), 3*(2), (58-66).

Iskandar, U. A. S., Diah, N. M. ve Ismail, M. (Haziran 2020). Identifying artificial intelligence Pathfinding algorithms for Platformer games. *2020 International Conference on Automatic Control and Intelligent Systems (I2CACIS) IEEE*, (74-80).

Karova, M., Penev, I. ve Kalcheva, N. (Haziran 2016). Comparative analysis of algorithms to search for the shortest path in a maze. *In 2016 IEEE international black sea conference on communications and networking (BlackSeaCom)* (1-4). IEEE.

Koubâa, A., Bennaceur, H., Chaari, I., Trigui, S., Ammar, A., Sriti, M. F., Alajlan, M., Cheikhrouhou, O. ve Javed, Y. (2018). Robot path planning and cooperation (Cilt 772). *Cham, Switzerland: Springer International Publishing*.

Kumar, N. ve Kaur, S. (2019). Bidirectional Graph Search Techniques for Finding Shortest Path in Image Based Maze Problem. *International Research Journal of Engineering and Technology.*

Latombe, J. C. (1991). Robot motion planning. *The Springer International Series in Engineering and Computer Science*.

Moore, E. F. (1959). The shortest path through a maze. *Proc. of the International Symposium on the Theory of Switching.* içinde Harvard University Press.

Noori, A. ve Moradi, F. (2015). Simulation and Comparison of Efficency in Pathfinding algorithms in Games. *Ciência e Natura, 37*, (230-238).

Pathak, M. J., Patel, R. L. ve Rami, S. P. (2018). Comparative analysis of search algorithms. *International Journal of Computer Applications, 179*(50), (40-43).

Permana, S. H., Bintoro, K. Y., Arifitama, B. ve Syahputra, A. (2018). Comparative analysis of pathfinding algorithms a*, dijkstra, and bfs on maze runner game. *IJISTECH (International J. Inf. Syst. Technol, 1*(2), 1.

Ponguillo Intriago, R. A., Khazaelpour, P., Querol Puchal, I., Semanjski, S., Ochoa, D., Gautama, S. ve Semanjski, I. (2021). Analysis of minimal clearance and algorithm selection effect on path planning for autonomous systems. *In The Tenth International Conference on Data Analytics* (45-50). IARIA.

Rathnayake, B. R. M. S. R. B., Marzuk, H., Senadheera, R. I. A., Vijeyakumar, S. ve Abeygunawardhana, P. K. (2021, August). Analysis of Searching Algorithms in Solving Modern Engineering Problems. I*n 2021 10th International Conference on Information and Automation for Sustainability (ICIAfS)* (123-128). IEEE.

Sedgewick, R. ve Wayne, K. (2011). Algorithms. Addison-wesley professional.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM journal on computing, 1*(2), (146-160).

Tucker, A. (1996). *The Computer Science and Engineering Handbook.* Florida: CRC Press.

Yadav, S., Verma, K. K. ve Mahanta, S. (2012). The Maze problem solved by Micro mouse. *International Journal of Engineering and Advanced Technology (IJEAT) ISSN, 2249, 8958.*

Yao, J., Lin, C., Xie, X., Wang, A. J. ve Hung, C. C. (Nisan 2010). Path planning for virtual human motion using improved A star algorithm. *In 2010 Seventh*

international conference on information technology: new generations, (1154-1158). IEEE.

Žunić, E., Djedović, A. ve Žunić, B. (2016, May). Software solution for optimal planning of sales persons work based on Depth-First Search and Breadth-First Search algorithms. *In 2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (1248-1253). IEEE.