SUDOKU BULMACASININ KUYRUK LİSTE VERİ YAPISI TABANLI PARALEL ÖNCE-DERİNE ARAMA YÖNTEMİYLE ÇÖZÜLMESİ

Yüksek Lisans Tezi

Zeynep Feyza ESEN

Eskişehir, 2018

SUDOKU BULMACASININ KUYRUK LİSTE VERİ YAPISI TABANLI PARALEL ÖNCE-DERİNE ARAMA YÖNTEMİYLE ÇÖZÜLMESİ

Zeynep Feyza ESEN

YÜKSEK LİSANS TEZİ
Bilgisayar Mühendisliği Anabilim Dalı
Bilişim Yüksek Lisans Programı
Danışman: Doç.Dr. Cihan KALELİ

Eskişehir
Anadolu Üniversitesi
Fen Bilimleri Enstitüsü
Temmuz, 2018

JÜRİ VE ENSTİTÜ ONAYI

Zeynep Feyza ESEN'in "Sudoku Bulmacasının Kuyruk Liste Veri Yapısı Tabanlı Paralel Önce-Derine Arama Yöntemi Çözülmesi" başlıklı tezi 09/07/2018 tarihinde aşağıdaki jüri tarafından değerlendirilerek "Anadolu Üniversitesi Lisansüstü Eğitim-Öğretim ve Sınav Yönetmeliği"nin ilgili maddeleri uyarınca, Bilgisayar Mühendisliği Anabilim Dalı Bilişim Programında Yüksek Lisans tezi olarak kabul edilmiştir.

Jüri Üyeleri	**Unvanı Adı Soyadı**	**İmza**
Üye (Tez Danışmanı)	Doç. Dr. Cihan KALELİ	.....
Üye	Dr. Öğr. Üyesi Alper BİLGE	.....
Üye	Doç. Dr. Eyyüp GÜLBANDILAR	.....

Prof.Dr. Ersin YÜCEL
Fen Bilimleri Enstitüsü Müdürü
……

ÖZET

SUDOKU BULMACASININ KUYRUK LİSTE VERİ YAPISI TABANLI PARALEL ÖNCE-DERİNE ARAMA YÖNTEMİYLE ÇÖZÜLMESİ

Zeynep Feyza ESEN

Bilgisayar Mühendisliği Anabilim Dalı
Bilişim Programı

Anadolu Üniversitesi, Fen Bilimleri Enstitüsü, Temmuz 2018

Danışman: Doç. Dr. Cihan KALELİ

NP-Complete bir oyun olan sudoku bulmacası dünya genelinde oldukça ilgi gören bir bulmaca çeşididir. Sudokuya olan ilginin yoğunluğu ilk kez 2006'da şampiyona düzenlenmesine neden olmuştur. Sayıların dizilim ve kullanılan sayı miktarına bağlı olarak farklı zorluk seviyelerinde oluşturulabilen sudokunun kolay seviyeleri klasik kağıt kalem ile basitçe çözülebilirken, zor seviyesindeki sudoku bulmacaları için çeşitli deneme yanılma yöntemleri geliştirilmiştir. Bu tez çalışmasında sudoku bulmacasının kuyruk liste veri yapısı tabanlı paralel önce derine arama yöntemi ile çözülmesi amaçlanmıştır. Paralelleştirme yapılırken farklı sayıda thread'ler ve parametreler denenmiş ve bunların optimum değerleri bulunmaya çalışılmıştır. Kuyruk listesi veri yapısı tabanlı paralel önce derine arama yöntemi ile geleneksel önce derine arama algoritması karşılaştırılmıştır. Bu çalışmada iki farklı zorluk seviyesinden 2'şer adet sudoku bulmacası kullanılmıştır. Her bir sudoku bulmacası 10'ar kez çözülerek çözüm süresinin ortalamaları alınarak incelemeler yapılmıştır. Denemeler sonucunda kuyruk listesi veri yapısı tabanlı paralel önce derine arama yönteminin, belirli thread sayısı ve parametreler ile geleneksel önce derine arama algoritmasından daha hızlı çözdüğü görülmüştür.

Anahtar Kelimeler: Kuyruk Listesi Veri Yapısı, Paralel Önce-Derine Arama, Sudoku

ABSTRACT

SOLVING SUDOKU PUZZLE WITH PARALLEL DEPTH-FIRST-SEARCH METHOD BASED ON THE LIST OF QUEUES DATA STRUCTURE

Zeynep Feyza ESEN

Department of Computer Engineering
Informatics Program

Anadolu University, Graduate School of Sciences, July 2018

Supervisor: Assoc. Prof. Dr. Cihan KALELİ

The sudoku puzzle, an NP-Complete game, is a puzzle that is highly popular around the world. The intensity of the interest in the sudoku has resulted in the first championship being held in 2006. Sudokus can be in different levels of difficulty depending on the number of sequences and the number of digits used. Sudoku puzzles at easy levels can be solved with a classic paper-pencil. On the other hand, trial-and-error methods have been developed for puzzles in different levels of difficulty. The aim of this study was to solve sudoku puzzles by using the parallel depth-first-search method based on the list of queues data structure. Different numbers of threads and parameters were tested in parallelling process. Optimal thread and parameters were examined. The parallel depth-first-search method based on the list of queues data structure was compared with the traditional depth-first-search search algorithm. Two sudoku puzzles were used in each of two different levels of difficulty in the study. Each sudoku puzzle was solved ten times and analyses were performed by taking the averages of solution. As a result of the experiments, the parallel depth-first-search method based on the list of queues data structure was found to solve sudokus faster than the traditional depth-first-search algorithm with a certain number of threads and parameters.

Keywords: List of Queues Data Structure, Parallel Depth-First-Search, Sudoku

TEŞEKKÜR

Tez danışmanlığımı üstlenerek tezimin hazırlanma sürecinde desteğini esirgemeyen hocam Doç. Dr. Cihan KALELİ'ye teşekkür ederim.

Tez yazım ve sunum hazırlık aşamalarında yaptıkları değerli yorumlar için sevgili arkadaşlarım Ahmet AYDIN, Hakan YILDIRIM, Hakan ÖZBAŞARAN ve Esra SERTEL'e teşekkür ederim.

Tüm eğitim hayatım boyunca desteklerini esirgemeyen aileme ve en önemlisi tanıştığım ilk günden beri hayatımın her anında yanımda olan ve beni her konuda destekleyen canım eşim İbrahim ESEN'e teşekkürü bir borç bilirim.

Zeynep Feyza ESEN

Temmuz, 2018

09/07/2018

ETİK İLKE VE KURALLARA UYGULUK BEYANNAMESİ

Bu tezin bana ait, özgün bir çalışma olduğunu; çalışmamın hazırlık, veri toplama, analiz ve bilgilerin sunumu olmak üzere tüm aşamalarında bilimsel etik ilke ve kurallara uygun davrandığımı; bu çalışma kapsamında elde edilemeyen tüm veri ve bilgiler için kaynak gösterdiğimi ve bu kaynaklara kaynakçada yer verdiğimi; bu çalışmanın Anadolu Üniversitesi tarafından kullanılan "bilimsel intihal tespit programı'yla tarandığını ve hiçbir şekilde "intihal içermediğini" beyan ederim. Herhangi bir zamanda, çalışmamla ilgili yaptığım bu beyana aykırı bir durumun saptanması durumunda, ortaya çıkacak tüm ahlaki ve hukuki sonuçlara razı olduğumu bildiririm.

.....

Zeynep Feyza ESEN

İÇİNDEKİLER

ŞEKİLLER DİZİNİ

TABLOLAR DİZİNİ

KISALTMALAR DİZİNİ

P	: Polynomial Time
NP	: Nondeterministic Polynomial Time
DFS	: Depth First Search
LIFO	: Last in First Out
FIFO	: First in First Out
ACO	: Ant Colony Optimization
CPU	: Merkezi İşlem Birimi
GPU	: Grafik İşlemci Ünitesi
HPC	: Yüksek Başarımlı Hesaplama

1. GİRİŞ

Oyunlar belirli kurallar çerçevesi içerisinde insanların hoş vakit geçirip zevk almalarını sağlayan eğlencelerdir. Bir de bu oyunlara dilediğimiz yerde ve dilediğimiz zaman ulaşabiliyorsak bu oyunlar vazgeçilmezlerimiz arasında yer almaya başlar. Gelişen teknoloji ile birlikte birçok zeka oyununun (sudoku, tetris, satranç) bilgisayar ve mobil ortamlar için oyunları geliştirilmiştir.

Zeka oyunları hoşça zaman geçirmenin yanı bu tarz oyunları oynayan kişilerin kişisel ve zihinsel gelişimlerinde katkıları bulunmaktadır. Konsantrasyon gerektiren zeka oyunları oynayan kişide dikkat ve konsantrasyon seviyelerini artırmanın yanı sıra problem çözme, akıl yürütme, sorgulayıcı bakış açısı oluşturma, karar verme gibi becerilenin artmasını sağlanır. Alzheimer ve Demas gibi hastalıklara karşı beynin hızlı yaşlanmasına engel olur.

Karar verme problemleri olan zeka oyunları NP-Complete oyunlardır. NP-Complete oyunları ilgi çekici kılan eğlenceli olmalarıdır. Polinom zamanda çözülemeyen oyunlar her seferinde farklı şekilde sonuçlanırken polinom zamanda çözülebilen oyunlar bir kere çözüldüğü zaman sürekli aynı şekilde çözüldüğü için polinom zamanda çözülmeyen oyunlar kadar ilgi çekici değildir. Aynı zamanda oyunlar farklı zorluklarda olup çözüm için zeka gerektirmektedir. Merak uyandıran bu oyunlar bir süredir bilgisayar bilimlerinin bilimsel araştırma konuları arasında yer almaya başlamıştır. Matematiksel hesaplama gerektiren oyunlar, yapay zekanın gözde konularındandır. Karar verme ya da evet/hayır problemleri olan polinom zamanda çözülemeyen problemlerin çözümünde çeşitli arama algoritmaları ve metotlara başvurulmaktadır.

NP-Complete oyunlar/bulmacalar kolaylıkla çözülemeyen tiptedirler. Matematiksel zeka gerektiren bu oyunlar çeşitli metot ve algoritmalar ile daha hızlı çözüme ulaşmaktadır.

NP-Complete olarak sınıflandırılmış oyun ve bulmacalara örnek olarak şunlar verilebilir:

Battleship, Bejeweled, Candy Crush Saga, Minesweeper Consistency Problem, Peg Solitaire, Solitaire/Card Games, Sudoku, Super Mario Bros, Tetris, Verbal Arithmetic.

Bu tez çalışmasında bir NP-Complete bulmacası olan sudokunun çözümü ele alınmaktadır. Ülkemizde önceleri rakam yerleştirme olarak yayınlanan sudoku, otobüste, evde, kafede, mola saatinde çok rahatlıkla çözülebilecek bir bulmacadır. Bu da sudoku

bulmacasını tüm dünyada ilgi gören bir oyun yapmıştır. Birçok çeşidi olan bu zeka oyununda genellikle bir tabla üzerinde 3x3'lük 9 kare ve bu her bir karenin içerisinde 3x3'lük satır ve sütundan oluşan karelere 1'den 9'a kadar her sayının sadece 1'er kere yerleştirildiği ve başka bir kural içermeyen klasik sudoku oynanmaktadır. Bir sudoku bulmacasının tek bir çözümü olabilmesi için minimum gerekli ipucu sayısı 17'dir. Farklı zorluk seviyelerinin olduğu bulmacada seviyelerin belirlenmesini verilen ipuçlarının miktarı ve dizilişleri belirlemektedir.

Daha önce Sudoku bulmacasının çözümü için farklı algoritmalar ve kurallar kullanılarak çözüm yolları oluşturulmuştur. Bu çözüm yollarından bazılarında algoritmalar arası kıyaslama yapılmış, bazılarında da paralelleştirilerek performansları incelenmiştir.

Bu çalışmada ise önce-derine-arama yönteminin paralelleştirilebilmesi için bir kuyruk listesi veri yapısı oluşturularak çözüm süresi incelenecektir. Sudoku bulmacasının geleneksel önce-derine-arama algoritması ile kuyruk listesi veri yapısı tabanlı paralel önce-derine-arama yöntemlerinin performansları karşılaştırılacaktır.

Tezin amacı, paralel önce-derine-arama yönteminin sudoku gibi bir NP-complete problemin çözümünde nasıl bir etki yaptığını araştırmaktır.

2. ÖNBİLGİ

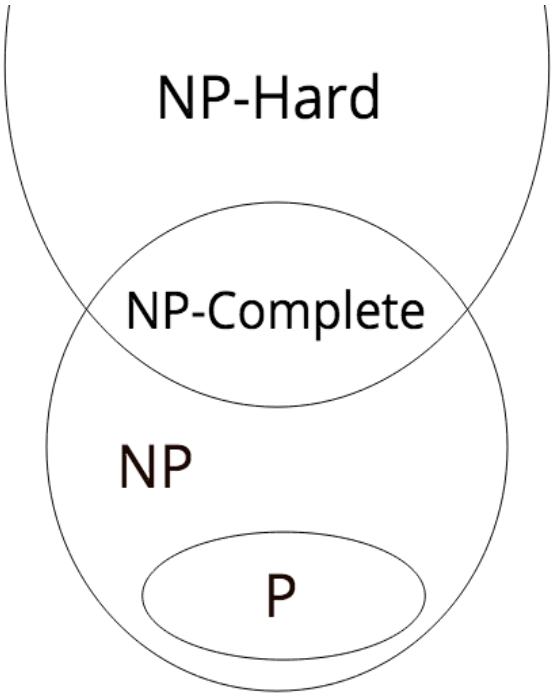
2.1. P - NP - NP-Complete - NP Hard Problemler

Polinomsal Zamanda Çözülebilen Problemler: Polinomsal zamanda çözülebilen problemler, problemi çözen algoritmaların çalışma süresinin, verinin büyüklüğüne bir polinom cinsinden bağlı olan problemlerdir. Bu tip problemlere kısaca P tipi problemler denir. Örneğin verilen sayıların en küçük ortak katını bulma.

Polinomsal Zamanda Çözülemeyen Problemler (Belirleyici Olmayan Polinom): Polinomsal zamanda çözülemeyen problemler ise, bu problemi çözen verinin büyüklüğüne göre polinomsal bir çalışma süresi olan bir algoritmanın bulunamadığı problemlerdir. Bu tip problemlere NP tipi problemler denir. NP tipi problemlerde eğer bir şekilde çözüm tahmin edilebiliyor ise, çözümün doğruluğunu verinin büyüklüğü ile bir polinom cinsinden bağlı olan çözen algoritmalar mevcuttur. Yani polinomsal zamanda çözülebilen problemler polinomsal zamanda çözülemeyen problemler içinde yer almaktadır.

Belirleyici Olmayan Zor Polinom (NP-Hard): NP-Hard, çözümü en az bir NP problem kadar zor olan problemler için yapılan sınıflandırmadır. NP-Hard problemlerin halen yeterince hızlı çözüm yolunun bulunup bulunmadığı konusunda tam bir bulgu yoktur.

NP-Complete (Nondeterministic Polynomial Complete): NP-Hard problemler NP problemlerden daha zor olabilir ancak NP problemler tahmin ile polinomsal zamanda doğrulama özelliğine sahip olabilirler. Eğer bir problem NP-Hard ve aynı zamanda NP ise bu problemler NP-Complete kategorisine girer. NP-Complete problemler için yapılan çözüm tahminleri, büyüklük ile polinomsal orantılı olan sürede doğrulanabilirler. Bu kategorizasyonu Şekil 2. 1.'le de anlatabiliriz.



Şekil 2. 1. *P - NP- NP-Complete- NP-Hard diyagramı*

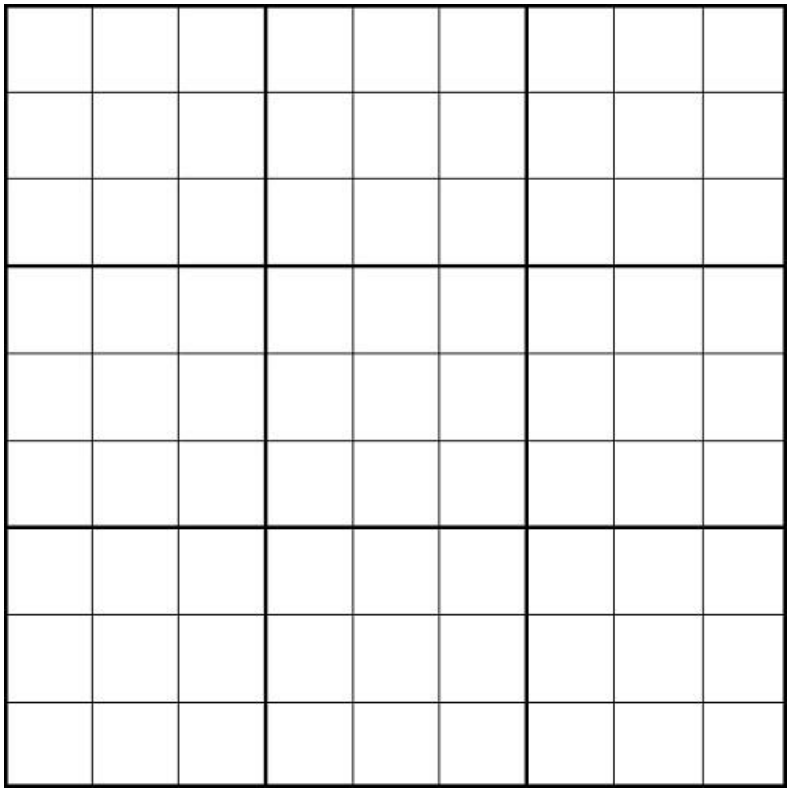
Bazı NP problemlerinin zorluğu çok yüksektir ki bu da polinomsal zamanda çözüme ulaşma girişimlerini sonuçsuz bırakmıştır. Bunlar NP-Complete sınıfı altında toplanmaktadır. Battleship, Bejeweled, Candy Crush Saga, Minesweeper Consistency Problem, Peg Solitaire, Solitaire/Card Games, Super Mario Bros, Tetris, Sudoku gibi birçok NP-Complete olarak sınıflandırılmış oyun ve bulmacalar bulunmaktadır.

2.2. NP-Complete Problemler ve Çözüm Yöntemleri

NP-Complete problemler farklı yollarla çözülebilir. Yaklaşık çözüm ile optimal olmasa da optimale yakın çözümler bulunabilir. Rastgele deneme çözümü ile rastgele çözümler denenir. Kısıtlama yöntemi ile verinin yapısına göre kısıtlama yapılarak daha hızlı algoritmalar kullanılabilir. Parametrik çözümlerde ise eğer bazı giriş parametreleri sabitlenir ise hızlı algoritmalar kullanılabilir. Deneysel çözümler ise genellikle en iyi çalışan yöntemlerdir.

2.3. Bir NP-Complete Bulmacası Sudoku

Klasik, Bölgesel, Ardışık, Tek/çift, Köşegen, Zincir, Toplamlı, Oklu gibi birçok çeşidi olan sudoku bulmacası üç temel kuraldan oluşur. Bu temel kuralların oluşturduğu sudokuna klasik sudoku olarak adlandırılır.



Şekil 2. 2. *9x9 sudoku bulmacası*

Şekil 2.2'de 9x9 boyutlarında boş sudokudur. 9x9 Klasik sudoku, 9 satır, 9 sütun ve 9 bölgeden oluşmaktadır. Klasik sudoku sorusu altta soldaki resimdeki gibidir. Soruda, 81 adet rakamdan bazıları yerleşmiş olarak verilir. Verilmeyen kısımlara ise 1'den 9'a kadar olan rakamlar öyle yerleştirilmelidir ki sudokunun her satırında 1'den 9'a kadar olan rakamların hepsi bulunmalı ve 1'den 9'a olan rakamlar sadece birer defa yer almalıdır. Aynı kural, sütunlar ve bölgelerde de sağlanmalıdır. Yukarıda bahsedilen 3 temel kural vardır.

Sudoku sorularında belirlenmiş miktarda rakam sudoku içine yerleştirilmiş olarak verilir. Daha sonra belirlenmiş 3 temel kurala uygun olacak şekilde rakamları yerleştirmemiz istenir.

• Her satırda tüm rakamlar bulunmalı ve bu rakamlar sadece birer defa yer almalıdır.

• Her sütunda tüm rakamlar bulunmalı ve bu rakamlar sadece birer defa yer almalıdır.

• Her bölgede tüm rakamlar bulunmalı ve bu rakamlar sadece birer defa yer almalıdır [1].

3. ARAMA ALGORİTMALARI ve VERİ YAPILARI

3.1. Arama Algoritmaları

Çeşitli büyüklüklerde veri yapılarında belirli bir bilginin daha az zaman, bellek, kaynak harcayarak kısa yoldan bulunmasını sağlayan algoritmalar arama algoritmalarıdır. Arama algoritmalarını temelde uninformed (blind) ve informed (Heuristic) olarak ikiye ayrılır. Uninformed arama algoritmaları her durumda aynı şekilde çalışan algoritmalarken informed arama algoritmaları problemin özelliklerine göre çalışan algoritmalardır.

Kaba Kuvvet Algoritması (Brute Force Algorithm)

Doğrusal arama olarak da adlandırılan basit bir eşleştirme algoritmasıdır. Kaba kuvvetler algoritması genelde şifre çözmek için kullanılır. Algoritmada muhtemel değerler tek tek denenerek doğruluğu test edilerek istenilen sonuca ulaşıp ulaşılmadığı kontrol edilir. Test sonucu mutlak bir değer döner. Değişkenlerin artması ile zaman fonksiyonun artış gösterir. Dolayısıyla problem büyüklüğü sınırlı olduğunda tercih edilir.

Geri İzleme Algoritması (Backtracking Algorithm)

Backtracking algoritması bir amaca ulaşmak için çeşitli olasılıkları bulunan bir problemin çözümünü deneme yanılma yöntemi ile bulur. Yanlış bir yola girdiğinde geri çıkarak belirli bir noktadan tekrar aramaya devam ettiği için geri arama algoritması denmektedir. Yanlış olduğunu düşündüğü yerden geri döndüğünde alt ihtimalleri eleyerek çözüme kısa sürede ulaşmayı sağlar. Tek bir denemede birçok ihtimali elediğinden kaba kuvvet algoritmaları ile karşılaştırıldığında daha hızlı bir algoritmadır.

Harmonik Search

Geem ve arkadaşları tarafından geliştirilen harmoni arama algoritması bir orkestra içindeki tüm enstrümanların kendi içerilerinde akorlar yaparak optimum senfoniyi elde etmesine dayanan bir prensibe sahiptir [2].

Harmoni arama yöntemi genetik algoritmaya benzer. Harmoni aramayı genetik aramadan ayıran en büyük fark yeniden üretim aşamasındaki varsayımlardan

kaynaklanmaktadır. Genetik algoritmada yeni bir birey oluşturulurken toplum içindeki iki birey kullanılırken harmoni'de ise tüm bireyler kullanılır.

Hormoni arama algoritması ile optimizasyon probleminin çözümü 5 adımda gerçekleşmektedir [3].

1. Problemin kurulması ve algoritma parametrelerinin belirlenmesi.
2. Harmoni belleğinin oluşturulması
3. Yeni harmoni oluşturulması
4. Harmoni belleğinin güncellenmesi
5. Durma koşulunun kontrolü

Genetik Algoritma

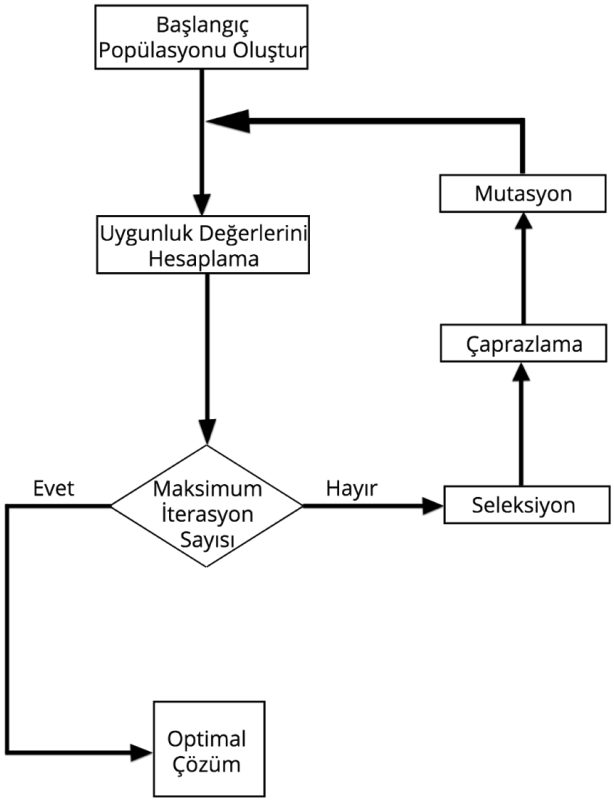
John Holland tarafından evrim sürecinin bilgisayara ortamına aktarılması ile çiftleşme, çoğalma, değişim gibi genetik süreçlerin sonrasında oluşan üstün yeni bireylerin elde edilebileceğini gösteren çalışması sonucunda geliştirdiği yöntemin adı "Genetik Algoritmalar"dır.

Genetik algoritmanın çözüm alanı oldukça geniş olmasına rağmen ilgili alanın bir kısmını tarayarak sonuca ulaşmaya çalışır. Bu da zamandan kazanç sağlar. Genetik algoritmalar problemi etkileyen çok faktör olduğunda kullanılır. Olasılık kurallarına göre çalışması algoritmanın ne kadar etkili çalışacağı önceden kestirilemez.

Genetik algoritmalar, problemler için tek bir çözüm üretmek yerine Şekil 3. 1.'de de görüldüğü üzere mutasyon, çaprazlama ve seleksiyon gibi işlemlerle içerisinde farklı birçok çözümün bulunduğu bir çözüm kümesi bulmaya çalışır. Çözüm kümesi içerindeki çözümleri eş zamanlı inceleyerek en iyi çözümü bulmaya çalışır.

Genetik algoritmalarda genellikle 5 adım bulunmaktadır [4]:

1. Rastlantısal olarak bir çözüm kümesi üretilmesi
2. Çözüm kümesinin uygunluk değeri hesaplanması
3. Uygunluk değerlerine göre ebeveyn seçilimi
4. Çaprazlama ve mutasyon ile yeni nesilleri üretmesi
5. Uygunluk değerine göre kötü olan kromozomları çözüm kümesinden çıkarılması



Şekil 3. 1. *Genetik algoritma akış diyagramı*

Kural Tabanlı (Rule-Based)

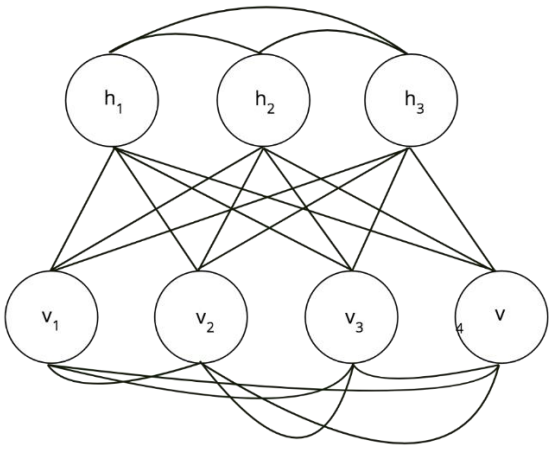
Genellikle yapay zeka araştırma ve uygulamalarında kullanılan kural tabanlı sistem bilgiyi doğru şekilde yorumlayıp işlemek için kullanılan bir yoldur. Yapı olarak normal bir insanın problem çözmesine benzer. Önceden belirlenen kurallar doğrultusunda çözüme ulaşılır.

Kural tabanlı sistemlerin 4 adımı vardır:

1. Kurallar listesinin oluşturulur.
2. Kurallar sürekli uygulanır.
3. Kurala uygun olmayan durum ile karşılaşıldığında rastgele değer ile devam edilip kurallar tekrar uygulanır.
4. Çözüm bulunana kadar devam eder.

Boltzmann Makinesi

Boltzmann makinesi 1985'te Geoffrey Hinton and Terry Sejnowski tarafından icat edilen olasılıksal tekrarlayan yapay sinir ağı ve Markof Rastgele Alan türüdür. Boltzmann makinesinin olasılıksal katman ve geri besleme bağlantıları Hopfield ağlarına benzemektedir. Boltzmann makinesi modelle seviyelerini araştırıp durum uzayında kavramları benzeterek birleştirmektedir. Bir Boltzmann makinesi simetrik olarak bağlı nöron benzeri bir ağı olup, açık veya kapalı olmasına ilişkin stokastik kararlar verir. Oluşan veri kümelerindeki ilginç özellikleri keşfetmelerini sağlayan basit bir öğrenme algoritmasına sahip olan Boltzmann makinesi arama problemleri ve öğrenme problemleri olarak iki farklı hesaplama problemini çözmek için kullanılırlar.

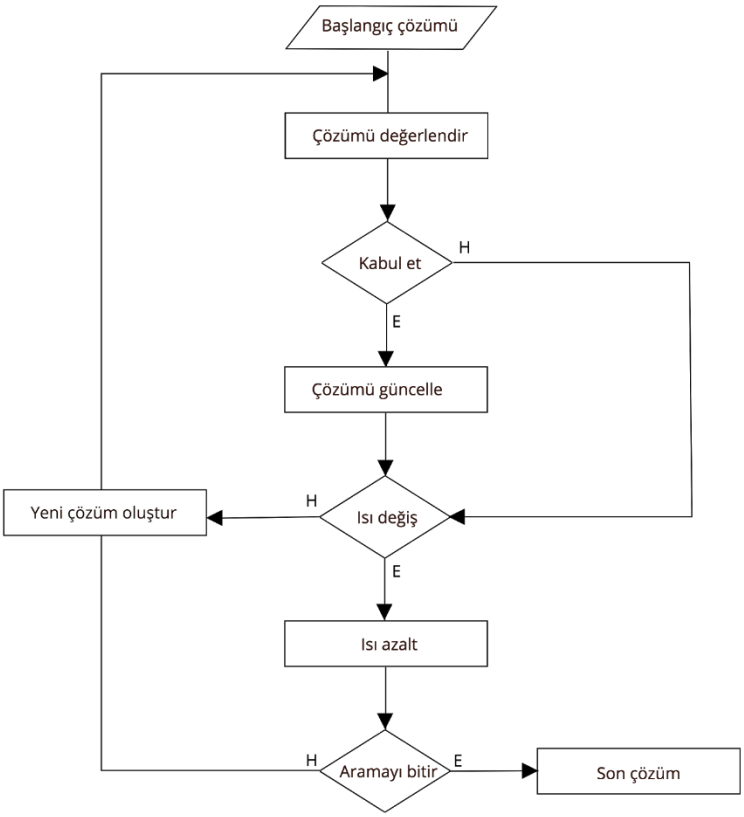


Şekil 3. 2. *Boltzmann Makinesi diyagramı*

Simulated Annealing

Stokastik bir arama algoritması olan simulated annealing genel iyileştirme elde etmeyi amaçlamaktadır. En büyük veya en küçük değeri bulmak için tasarlanmıştır. Özellikle matematiksel modellemelerde gösterilemeyen kombinasyonel problemlerin en iyileme uygulamalarında tercih edilir. Temel mantığı katıların ısıtılıp yavaş yavaş soğutularak kristalleşmesine dayanır. Yüksek bir sıcaklıkla başlatılan Simulated Annealing

algoritmasında sıcaklık yavaş yavaş düşürülür. Her sıcaklıkta belirli sayıda çözüm üretilir. Bu çözümlerin belirlenen kriterlere göre uygun olup olmadığı kontrol edilir. Eğer uygun çözüm bulunmuş veya sıcaklık minimum değere ulaşmış ise algoritma sonlandırılır. Akış diyagramı Şekil 3. 3.'de görülmektedir.



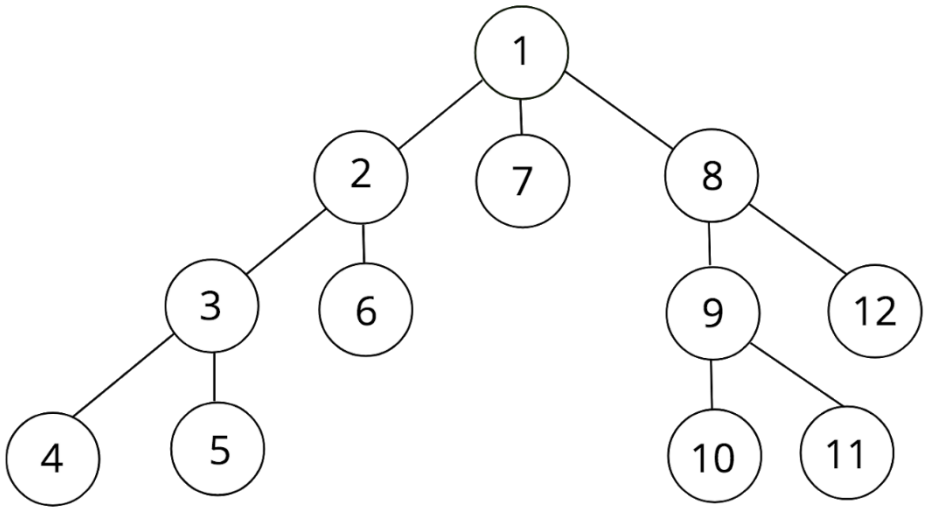
Şekil 3. 3. *Simulated Annealing akış diyagramı*

Önce Derin Arama (Depth First Search)

Önce Derin Arama ile backtracking birbirine benzese de backtracking daha genel bir algoritmadır. DFS backtracing in ağaç yapılarında aramanın daha özel biçimidir. DFS backtracking in daha özel bir formudur. DFS ağaç yapısı oluşturulurken kabul edilemeyen durumlar budama yapılır.

Ağaç üzerinde aranılan herhangi bir düğümü bulmak için kökten başlayarak sol veya sağ seçilerek ağacı en derinine kadar inilir. Son düğümden daha aşağıya herhangi bir çocuk

yok ise bir üst düğüme çıkılır ve yataydaki düğüm işlenmeye başlanır. Bu şekilde tüm düğümler gezilerek aranılan düğüm bulunur. DFS algoritması yığıt veri yapısı kullanmaktadır. Algoritmada ziyaret edilen düğümler yığıta eklenir. Gidilecek komşu bulunamadığında pop ile yığından eleman atılır ve ziyaret edilebilecek komşu aranır [5].



Şekil 3. 4. *Önce Derin Arama ağacı*

3.2. Veri Yapıları

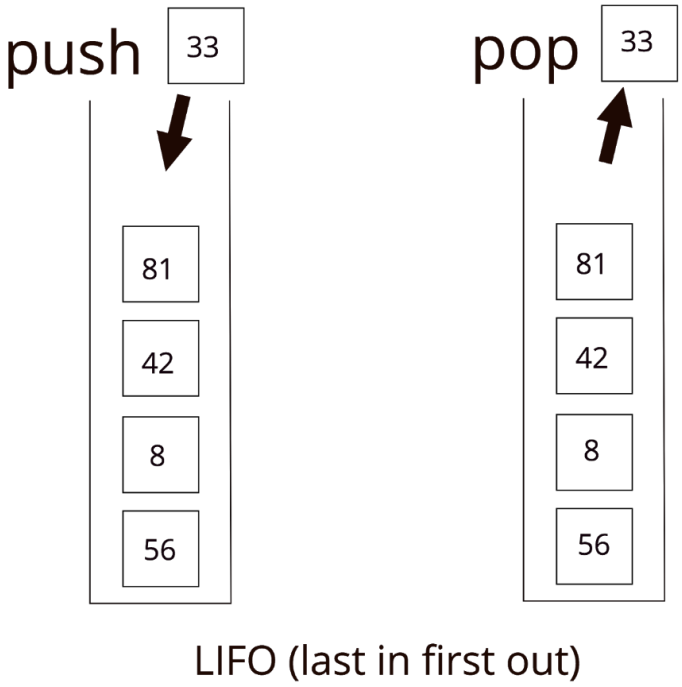
Bilgisayar ortamında verilerin düzenlenmesini ve saklanmasını sağlayan yapıların adıdır. İlkel veri tipi, bileşik veri tipi ve soyut veri tipi olarak üç grup altında incelenir. Temel veri tipleri; temel sayıları ifade eder. Basit algoritmalar için uygundur. Basit ve bileşik veri yapılarının oluşturulmasında kullanılırlar. Örneğin; Boolean, Character, Integer, String Bileşik veri tipleri; birbiri ile ilgili çok sayıda veriyi tek bir referans noktası ile bir arada tutabilen veri tipidir. Array, Structure, Union tipleri buna örnektir. Soyut veri tipleri; kullanan programcının veri tipinin uygulama detayları hakkında fazla bilgisi olmasını gerektirmeyen değerler kümesidir. Burada önemli olan soyut veri yapısını kullanılarak yapılacak işlemlerdir. Örneğin; Yığıt, Ağaç, Kuyruk, Grafik.

Yığıt (Stack): LIFO (last in first out), yani son giren ilk çıkar mantığı ile çalışan bir soyut veri tipidir. Bir nesne ekleneceği zaman yığıtın en üstüne konur. Çıkarılacağı zaman da yığıtın en üstündeki nesne çıkarılır. Tek taraflı giriş ve çıkışın olduğu bu yapıda en çok kullanılan üç işlem şunlardır [6].

Push; Yığının en üstüne yeni bir bilgi eklemek için kullanılan metottur.

Pop; Yığıtın en üstünde bulunan bilgiyi alarak silmek için kullanılır.

Top; Yığıtın en üstünde bulunan bilgiyi alıp okumaya yarar.

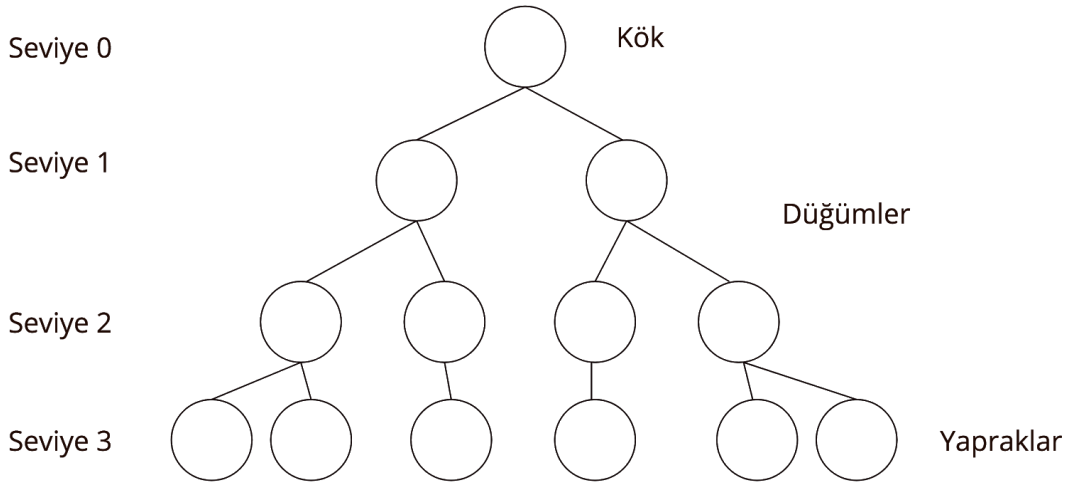


Şekil 3. 5. *Yığıt veri yapısı (LIFO)*

Ağaç (Tree): Veri tutma yöntemlerinden biri olan ağaç, bir kök (root) ve bu köke tıpkı ağacın dalları ve yaprakları gibi alt objeleri bulunan hiyerarşik bir yapıdır. Her nesne kendinin altındaki nesne ile üst-alt (parent-child) ilişkisi içindedir. Dallanan bir yapıya sahip olduğu için herhangi bir objeye ulaşılmak veri boyutu ile doğru orantılı değildir. Ağaç veri modeli

yüksek miktarda bellek alanına gereksinim duyar. Çünkü ağaç veri modelini kurmak için birden çok işaretçi değişkeni kullanır. Yürütme zamanından sağladığı getiri ve ağaç üzerinde işlem yapacak fonksiyonların rekürsif yapıda kolayca tasarlanması ve kodlanması ağaç veri modelini uygulamada önemli bir faktördür [6].

Şekil 3. 6.'deki ağaçta 1 tane başlangıç düğümü (kök) olmak üzere 13 düğüm (node) vardır. Yapraklarında (leaf) 6 düğüm bulunan ağacın seviyesi 3'dir.



Şekil 3. 6. *Ağaç veri yapısı*

Yığıt ve kuyruk veri yapılarını dolaşmak için kullanılırken, ağaç veri yapıları daha çok veri saklamak için kullanılır.

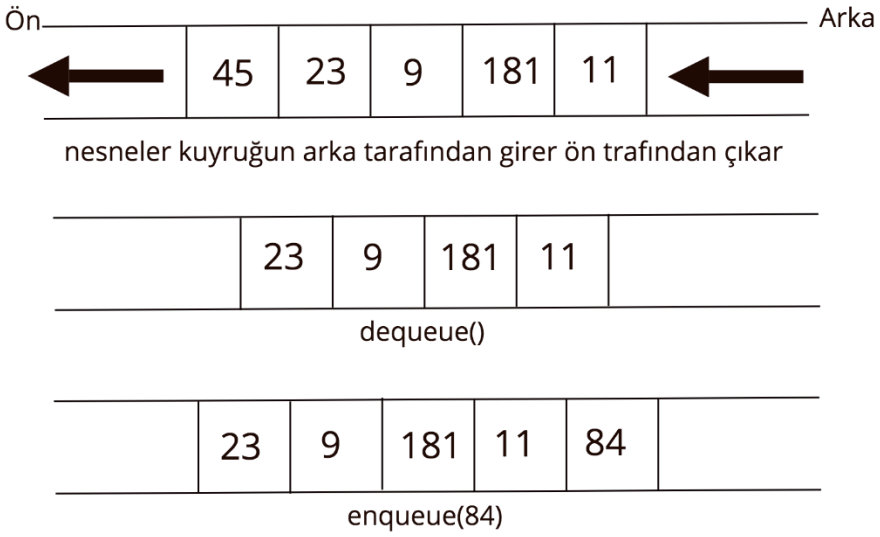
Kuyruk (Queue): Yığıt mantığına benzeyen bir yapıya sahiptir. Temel farkı tıpkı bir sıra gibi oluşan kuyruktan ilk girenin ilk çıkmasıdır (FIFO). Veri ekleme yığıttaki gibi kuyruğun sonuna eklenir. Fakat veri çıkarılacağı zaman kuyruğun sonundaki eleman değil başındaki eleman çıkarılır [6]. Başlıca kullanılan yapılar:

Enqueue; Kuyruğun en sonuna yeni bir değer eklemeyi sağlar.

Dequeue; Kuyruktaki ilk değeri alıp siler.

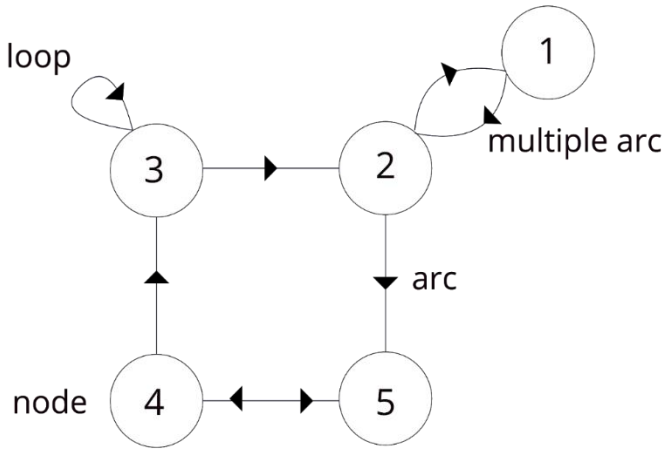
Peek; Kuyruktaki ilk değeri okur.

Count; Kuyrukta kaç değer olduğunu bulmaya yarayan metottur.



***Şekil 3. 7**. Kuyruk veri yapısı*

Çizge (Graph): Düğümler (node) ve bu düğümlerin birbirine olan ilişkilerinin kenarlardan oluştuğu bir ağ yapısıdır. Düğümlerin birbirine giriş çıkış yönleri vardır. Tek taraflı ya da çift taraflı olabilir. Bir düğümden çıkıp yine aynı düğüme giren bir kenar var ise buna döngü (loop) denir. Bir düğümden diğer bir düğüme giden iki farklı kenar aynı yönde veya yönsüz ise bunlara paralel kenarlar denir.



Şekil 3. 8. *Grafik veri yapısı*

4. İLGİLİ ÇALIŞMALAR

Sudoku bulmacası 1984 yılında ilk defa Japonyada oynanmıştır. Hızla Asya, Avrupa ve Kuzey Amerika'ya yayılarak tüm dünyada ilgi gören bir oyun olmuştur. Her yaştan insanın oynayabileceği ve kolay, orta, zor, çok zor olarak dört farklı seviyeye ayrılarak farklı zorluklarda insanların kendi sınırlarını zorlayarak beyin cimnastiği yapmasına olanak sağlaması bulmacanın herkes tarafından oynanmasına olanak sağlamıştır.

Sudoku bulmacası çözümü sırasında devreye giren dikkat ve karar verme alzheimer gibi beyinin işlevselliğini etkileyen hastalıkların ilerlemesini yavaşlatması da sudoku oyunun popüler olmasında etkisi olduğu söylenebilir [7].

Popüleritisini günden güne artmasıyla 2004 yılında The Times gazetesinde kendisine yer bulmuştur. Günümüzde sadece gazetelerde değil bilgisayarlarda, akıllı telefon ve tabletlerde de kendine yer bulmuştur. Amazonda en çok satan kitaplar arasında yer almaktadır. Düşünme ve yorumlama yeteneğini geliştiren sudoku bulmacasının büyük ilgi görmesiyle Dünya Bulmaca Federasyonu (World Puzzle Federation) ilk Dünya Sudoku Şampiyonasını italya'da 2006 yılında düzenlemiştir.

Bulmacalar arasında dikkat çeken sudoku bulmacası ile ilgili bilgisayar bilimlerinde de çeşitli çalışmalar yapılmaya başlanmıştır. Sudoku bulmacası için farklı çözüm algotirmaları denenmiştir. Çözüm algotimalarının hızlandırlımasına yönelik çalışmalar yapılmıştır. Bunlardan bazıları şu şekildedir;

García ve Palomino, Madue kurallarını kullanarak Sudoku bulmacalarının çözüleceğinin göstermişlerdir. Çözüm için kullandıkları teknik tarama, işaretleme ve analiz olarak bilinen üç işlemdir. Ana strateji olarak eliminasyon kullanmışlardır. Ayrıca what-if ve birkaç olasılık stratejisi kullanmışlardır [8].

Mahdian ve arkadaşları, Latin Kare tamamlama yöntemini Sudoku karelerine uygulamış ve buradan çıkan klasik sonuçların doğal uzantılarını araştırmışlardır [9].

Soto ve arkadaşları, Sudoku bulmacasını çözmek için yeni bir hibrit AC3-tabu arama algoritması önermiştir. Klasik tabu arama algoritmasını arc-consistency 3 (AC3) algoritmasıyla birleştirip ihtimal sayısını düşürmüşlerdir.Tabu aramasında AC3'ün rolü tek bir ön işlem değil her iterasyonunda geçerli olan tamamen entegre bir prosedür olarak hareket etmesini sağlamışlardır. Bu entegrasyon daha hızı bir çözüm sürecine yol açmış. Arama

yöntemleri kullanılarak çözülen sonuçlardan daha iyi bir performans gösterdiğini deneysel olarak kanıtlamışlardır [10].

Maji ve arkadaşları, Sudoku bulmacasını çözmek için kullanılan minigridlerden sadece istenen permütasyonları üretmişlerdir. Backtracking algoritma tekniğini kullanmışlardır, fakat her bir hücre yerine mini karelere uygulamış ve zamandan kazanç sağlamışlardır [11].

Zong Woo Geem, harmonik aramayı farklı optimizasyon teknikleri kullanarak problemi 285 fonksiyon değerlendirmesinden sonra Intel Celeron 1.8 GHz işlemci ile 9 saniyede başarıyla çözebilmişlerdir [12].

Patrik Berggren ve David Nilsson, üç farklı algoritma ile sudoku bulmacasını çözmüştür. Araştırmalarında sadece çözüme değil, aynı zamanda zorluk derecesi, bulmaca üretebilme yeterliliği ve paralelleştirmeye uygunluğu daincelemiştir. Değerlendirilen algoritmalar Backtracking, Kural Tabanlı ve Boltzmann makineleridir. Sudoku bulmacalarının çözümünde bu üç algoritmayı kıyaslamışlar ve en verimli algoritmanın kural tabanlı olduğuna karar vermişlerdir. Paralelleştirme, tüm algoritmalara farklı ölçüde uygulanabilmiştir. Boltzmann makinelerinin daha iyi şekilde paralelleştirebileceğini fakat düğüm durumlarının senkron güncelleştirmeleri nedeniyle biraz sınırlı olduğu sonucuna varmışlardır [13].

Padma Bonde ve Akta Agrawal, sudoku bulmacasının çözümünü için Brute-force ve Backtracking algoritma kullanarak performans değerlendirmesi yapmışlardır. Değerlendirme yapılarken 4 farklı seviyedeki (kolay, orta, zor, çok zor) bulmacalar kullanmışlardır. Sonuç olarak her seviyede Brute-force algoritmasının daha hızlı çalıştığı görülmüştür [14].

Raktim Chakraborty ve arkadaşları, kombinatoryal problemleri daha kolay bir şekilde çözmek için optimize edilmiş bir Graph Referance metodu önermişlerdir. Önerilen algoritma Backtracking algoritmasının bir modifikasyonudur. Önerilen algoritmanın geleneksel algoritmalara üstünlüğünü ortaya konulmaktadır [15].

Henrik Viksten ve Viktor Mattsson sudoku bulmacasının çözümünde Brute-force, Simulated Annealing, Dancing Links algoritmalarını kullanmış ve algoritmaların performanslarını incelemiş, bulmaca büyüdükçe performanslarının nasıl değiştiğini araştırmışlardır. Brute-force ve Simulated Annealing tutarlı sonuçlar elde edemezken, Dancing Links algoritması en hızlı ve en iyi ölçeklenebilirliğe sahip olduğu sonucuna varmışlardır [16].

Timo Mantere ve Janne Koljonen, çalışmalarında sudoku bulmacası ile ilgili 3 farklı araştırma yapmışlardır. Birincisi, genetik algoritmasının optimizasyonunun Sudoku bulmacası için etkili bir çözüm olup olmayacağıdır. İkincisi, genetik algoritmasının yeni Sudoku bulmacaları üretmede kullanılabilirliğidir. Diğeri ise Sudoku bulmacasının zorluğunu genetik algoritma ile değerlendiren bir derecelendirme makinesi olarak kullanılabilirliğini araştırmışlardır. Araştırmalarının sonucu olarak sudoku bulmacalarının bir kombinasyonel genetik algoritma ile etkili bir çözüme ulaştığını ispatlamışlardır [17].

Muhammad Asif ve Rauf Baig, standart bir Karınca Kolonisi Optimizasyon algoritmasının (ACO) modifikasyonu ile sudoku bulmacasını çözmüş ve diğer çözüm teknikleri ile ACO çözüm kalitesi ve zaman performansını kıyaslamışlardır. Sonuçlar NP-Komple problemleri çözmede evrimsel algoritmaların (ACO) kullanımının optimuma çok daha yakın çözümler bulabileceğini kanıtlamışlardır. En önemlisi, bu tekniğin polinom zamanında sonuç vermesi ve bu tür kombinasyonel problemlerde son derece zor olmasıdır [18].

5. ÖNERİLEN ÇÖZÜM

Tez çalışmasında zor ve en zor seviyeli 9x9 boyutundaki sudoku bulmacaları kullanılmıştır. Bunun sebebi kolay bulmacalar tek olası değerle kolayca çözülebilmektedir. En az 17 karenin bilindiği ve birden fazla çözümü olmayan bulmacalar tez kapsamında ele alınmıştır.

Bulmacalar, önce geleneksel önce-derine-arama yöntemiyle çözülecek. Daha sonra Kuyruk Listesi veri yapısı kullanılarak birden fazla işlemci aynı anda kullanılarak çözüme ulaşılmaya çalışılacaktır. İki yöntemin sonuçları ve performansı analiz edilecektir.

Çözüm için Visual Studio üzerinde C# programlama dili ile bir masaüstü uygulaması geliştirilmiştir. Uygulama text formatında sudoku problemini input olarak alır. Sonuçları console ve arayüz çıktısı olarak gösterir. Uygulama içinde sudoku problemini çözmek için sınıflar yaratılmıştır. Bu sınıflar her iki çözüm yönteminde de kullanılmıştır.

"Cell" sınıfı her bir sudoku karesini temsil eder. İçinde değer ve muhtemel alabileceği değerler bilgisini tutar.

```
/// <summary>
/// Sudoku tahasındaki bir hücreyi temsil eden sınıftır.
/// </summary>
public class Cell {
/// <summary>
/// Hücrenin değeri, eğer hesaplanmamış ise '0' olur
/// </summary>
public byte Value { get; set; }
/// <summary>
     /// Hücrenin alabileceği muhtemel değerler listesidir. Hesaplandığı an bu listeye eklenir.
      /// </summary>
   public List<byte> PossibleValues {get; set;}
      /// <summary>
      /// Hücrenin bir kopyasını alan fonksiyondur. Değer ve muhtemel değerler de olduğu gibi
kopyalanır.
      /// Sudoku tahtası kopyalanırken kullanılır.
      /// </summary>
      /// <returns>Kopylanan yeni hücre nesnesi</returns>
      public Cell Copy()
```

```
        {
            Cell copy = new Cell {Value = Value, PossibleValues = new List<byte>()};
            copy.PossibleValues.AddRange(this.PossibleValues);
            return copy;
        }
```

Board sınıfı ise tüm sudoku tahtasını temsil eder. İçinde hücreler ile ilgili bilgi tutar. Board üzerinde yapılan kopyalama, işleme alma, hücrelerin muhtemel değerlerini doldurma işlemleri için fonksiyonlar vardır.

```
public class Board {
    /// <summary>
    /// Sudoku tahasındaki 81 tane hücreyi temsil eden iki boyutlu dizidir.
    /// </summary>
public Cell[,] Cells = new Cell[9,9]
/// <summary>
/// <summary>
    /// Tahta kopyalama işini yapar. Mevcut durumdaki tüm hücrelerini de kopyalayıp yeni bir tahta döndürür.
    /// </summary>
    /// <returns>Kopyalanmış tahta</returns>
    public Board Copy()
  {
  ...
}
    /// <summary>
    /// Tahtanın hücrelerinin alabileceği muhtemel değerleri doldurur.
    /// </summary>
    public void FillPossibleValues()
{
.....
}
    /// <summary>
    /// Sudoku tahtasının hücre değerlerinin kurallara uygun olup olmadığını kontrol eder, eğer uygun ise true döndürür.
    /// </summary>
    /// <returns>true, eğer tahta kurallara uygun ise, aksi halde false döndürür.</returns>
```

```
        public bool IsValid()
    {

    }
        /// <summary>
        /// DFS çözümü içinde kullanılan bir fonksiyondur. Değeri bilinmeyen ilk hücre için muhtemel değerlerini içeren tahta kopyalarının listesini döndürür.
        /// </summary>
        public List<Board> GetBoardsWithFirstPossibleValues()
```

5.1. Önce-Derine-Arama Yöntemi

Arama ağacının kökü, başlangıçta, önce derine algoritmasında veri yapısı olarak kullanılan yığıta (stack) atılır. Yığıt boşalıncaya kadar dönen bir döngüye girilerek, her seferinde yığıtın tepesindeki düğüm yığıttan çıkartılır ve bu düğümün çocukları hesaplanarak bu çocuklar yığıta atılır. Önce-derine algoritması son giren ilk çıkar (LIFO) türünde bir veri yapısına sahiptir. Bunun için son eklenen çocuk daha önce yığıtta var olan düğümlerden daha önce işlenir.

Yığıtın tepesine sudoku probleminin tablasının ilk hali konulmaktadır. Daha sonra yığıttan hiç eleman kalmayana kadar işlem yapılır. Yapılan işlem ise yığıttan gelen tablanın değeri bilinmeyen herhangi bir hücresinin olası değerlerini içeren tablaları yığıta atıp, sonucun bulunup bulunmadığını kontrol etmektir. Bu işleme başlarken ilk önce problemin ilk halindeki tabla yığıttan gelir. Bu tablada değeri bilinmeyen bir hücre seçilir. Hücrenin muhtemel alacığı değerler hesaplanır. Daha sonra her bir değer için tablanın kopyası oluşturulur ve yığıta atılır. Yığıttan okunan tabla çözüme ulaşana kadar bu işlem tekrarlanır. Bu işlemlerin yapılışı aşağıdaki kod parçacığındadır.

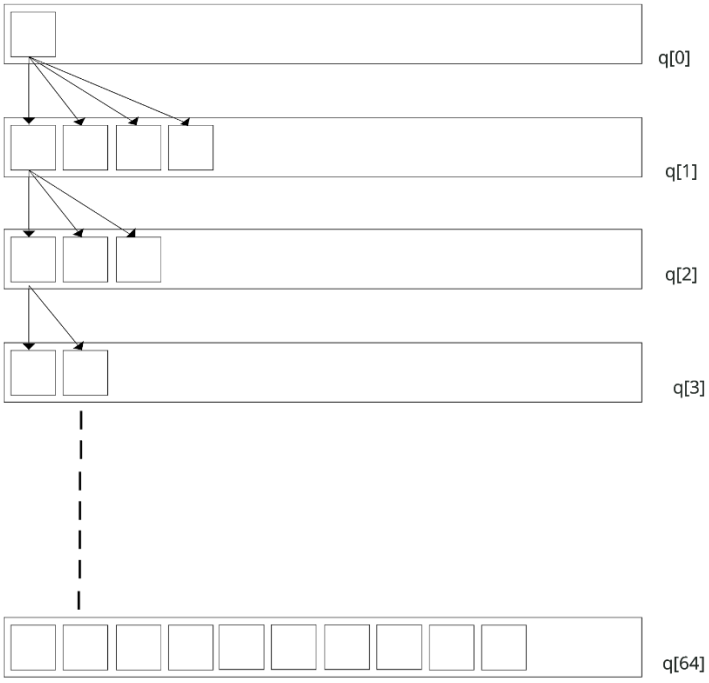
Yığıt döngüsü

```
        /// <summary>
        /// Tahtaların bulunduğu yığıt
        /// </summary>
        private Stack<Board> stack = new Stack<Board>();
```

```
        /// <summary>
        /// Verilen tahtayı DFS ile çözer ve sonuç olarak tüm hücrelerinin değeri dolu ve geçerli bir tahat nesnesi döndürür
        /// </summary>
        /// <param name="board">Sudoku problemi tahtası</param>
        /// <returns>Çözüm tahtası</returns>
        public Board SolveWithDFS(Board board)
        {
            stack.Push(board);
            while (stack.Count > 0)
            {
                Board current = stack.Pop();
                current.FillPossibleValues();
                if (current.IsSolved())
                {
                    // found solution
                    return current;
                }
                List<Board> possibleBoards = current.GetBoardsWithFirstPossibleValues();
                possibleBoards.ForEach(b => stack.Push(b));
            }
            return null;
        }
    }
```

5.2. Kuyruk Listesi Veri Yapısı Tabanlı Paralel Önce-Derine-Arama

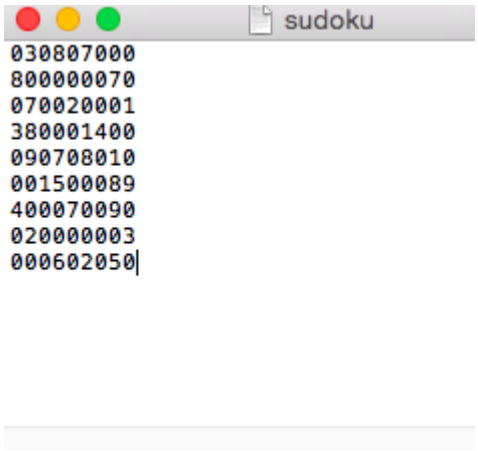
Bu yöntemde tek bir yığıt yerine, birden fazla kuyruk veri yapısı kullanılarak aynı anda farklı seviyelerdeki tablalar üzerinde işlem yapılır. Farklı seviyelerde aynı anda işlem yapıldığı için DFS deki gibi olası değerlerin kontrol işlemi birbirlerini beklemez, aynı anda birden fazla kontrol ve dallanma olur.



Şekil 5. 1. *Kuyruk listesi veri yapısı*

En fazla 65 adımda çözüldüğü için 65 adet boş kuyruk oluşturulur. Aynı anda sadece tek bir thread bir kuyruk üzerinde işlem yapar. Threadlerin karışıklık oluşturmaması için işlemden önce Monitor.TryEnter yöntemi ile senkronizasyon sağlanmıştır. Thread kuyruktan aldığı tahtanın ilk boş hücre için olası değerleri içeren tahtaları oluşturur. Oluşan tahtaları bir alttaki kuyruğa atar. Daha sonra diğer tüm boş hücreler için bu işlemi tekrarlar. Ancak bir alttaki kuyruğa bir anda tüm ihtimalleri atmamak için maxChildCount parametresi tanımlanmıştır. Bu değerden daha fazla elemanı alt kuyruğa atmaz. Eğer işlem sonrası tahtanın tüm olası değerleri için tahtalar oluşturulup alt kuyruğa atılmış ise tahta kuyruktan çıkartılır. MaxChildCount parametresi ile threadlerin daha kolay boşa çıkması sağlanabilir.

Thread sayısı da algoritma için diğer bir önemli parametredir. Bu parametre arttırıldığında her zaman çözümün hızlandığı gözlenmemiş, yavaşladığı da gözlenmiştir.



Şekil 5. 2. *Text dökümanı olarak sudoku*

Tez çalışmasında sudoku bulmacaları Şekil 5.2'de belirtilen formatta bir text dosyası olarak kaydedilip, programa çözmesi için verilir.

Bu text dosyasında 9 adet satır ve 9 adet kolon vardır. Sayılar hücrelerdeki değeri temsil eder, 0 değeri boş hücre değerini temsil eder.

Program hem önce derine hem de paralel önce derine ile çözüm yapar sonuçları çıktı olarak sunar ve aradaki fark görülür.

Program ayrıca thread sayısı ve maxChildCount parametrelerinin belirli bir aralıktaki tüm ihtimallerini deneyerek toplu olarak çalışıp tek bir rapor da üretir.

Üzerinde çalışılacak yazılım ve algoritmalar, Visual Studio üzerinde C# dili ile geliştirilmiştir.

Aşağıda bu çözümde kullanılan sınıflar mevcuttur:

```
/// <summary>
/// DFS queue çözümü için bir tahtayı temsil eden sınıf. Diğer tahta sınıfına ek olarak kuyruk index değeri bilgisi de mevcuttur.
/// </summary>
public class BoardForDFSQueue
{
    /// <summary>
    /// Sudoku tahtası nesnesi
```

```
    /// </summary>
    public Board Board { get; set; }
    /// <summary>
    /// <summary>
    /// Tahtanın bulunduğu kuyruk index numarası
    /// </summary>
    public int QueueIndex { get; set; }

public class LoQSolver : ISolver
 {
        /// <summary>
        /// Kullanılan thread sayısı
        /// </summary>
        private int _threadSayisi = 2;
        /// <summary>
        /// Bir sonraki kuyruğa aktarılabilecek maximum tahta kopyası sayısı
        /// </summary>
        private int _maxChildCount = 4;
        /// <summary>
        /// Çözüm değerleri
        /// </summary>
        private bool cozuldu = false;
        private Board cozumBoard = null;
        /// <summary>
        /// Kuyruk listesi, thread-safe olması için ConcurrentQueue kullanılmıştır.
        /// </summary>
        private List<Queue<BoardForDFSQueue>> queueList;
        /// <summary>
        /// Queue locks
        /// </summary>
        private object[] boardlocks = new object[65];

        public LoQSolver(int ThreadSayisi, int MaxChildCount)
        {
        _maxChildCount = MaxChildCount;
```

```
_threadSayisi = ThreadSayisi;
}
/// <summary>
/// Verilen tahtayı kuyruk listesi ile çözer ve sonuç olarak tüm hücrelerinin değeri dolu ve geçerli bir tahat nesnesi döndürür.
/// </summary>
/// <param name="board">Sudoku problemi tahtası</param>
/// <returns>Çözüm tahtası</returns>
public Board Solve(Board board)
{
Log.Information("LoQSolver başlıyor: Thread: {0}, MaxChildCount: {1}", _threadSayisi, _maxChildCount);
queueList = new List<Queue<BoardForDFSQueue>>(65);
for (int i = 0; i < 65; i++)
{
queueList.Add(new Queue<BoardForDFSQueue>());
boardlocks[i] = new object();
}
// ilk queue ya board u ekle
queueList[0].Enqueue(new BoardForDFSQueue { Board = board, State = BoardProcessState.Empty });
var options = new ParallelOptions { MaxDegreeOfParallelism = _threadSayisi };
List<Thread> threads = new List<Thread>();
for (int i = 0; i < _threadSayisi; i++)
{
threads.Add(new Thread(delegate ()
{
Coz();
}));
}
foreach (Thread t in threads)
{
t.Start();
}
Log.Information("LoQSolver threadler bekleniyor");
```

```
foreach (Thread t in threads)
{
t.Join();
}
Log.Information("LoQSolver bitti");
return cozumBoard;
}

public void Temizle()
{
cozuldu = false;
cozumBoard = null;
}

/// <summary>
/// Her bir çözüm thread inin çalıştırdığı fonksiyon
/// </summary>
private void Coz()
{

Log.Verbose("Started thread={0}", Thread.CurrentThread.ManagedThreadId);
while (!cozuldu)
{
for (int q = 64; q >= 0; q--)
{
Log.Verbose("Thread={0} trying queue: {1}", Thread.CurrentThread.ManagedThreadId, q);
if (cozuldu)
{
        Log.Information("Thread={0} cozuldu en basta: {1}",
Thread.CurrentThread.ManagedThreadId, q);
        break;
}

bool lockTaken = false;
Monitor.TryEnter(boardlocks[q], ref lockTaken);
```

```
                if (!lockTaken)
                {
                        break;
                }
                if (queueList[q].Count == 0)
                {
                        Log.Information("Thread={0} queuedan eleman peek edemedi: {1}",
Thread.CurrentThread.ManagedThreadId, q);
                        Monitor.Exit(boardlocks[q]);
                        continue;
                }

                if (boardDfsq.Board.IsSolved())
                {
                        Log.Information("Thread={0} cozuldu ortada: {1}, {2}",
Thread.CurrentThread.ManagedThreadId, q, boardDfsq.Id);
                        cozuldu = true;
                        cozumBoard = boardDfsq.Board;
                        Monitor.Exit(boardlocks[q]);
                        break;
                }
                int childCount = 0;
                bool exit = false;
                bool clear = true;
                if (boardDfsq.KilitlendiMi())
                {
                        queueList[q].Dequeue();
                        Monitor.Exit(boardlocks[q]);
                        continue;
                }
                for (var index = 0; index < 9 && childCount < _maxChildCount; index++)
                {
                        for (var indexy = 0; indexy < 9 && childCount < _maxChildCount; indexy++)
                        {
                        if (boardDfsq.Board.Table[index, indexy].Value == 0)
```

```
                    {

                    Log.Verbose("Thread={0} found empty value: [{1},{2}], possible count:{3}",
Thread.CurrentThread.ManagedThreadId,
                    index, indexy, boardDfsq.Board.Table[index, indexy].PossibleValues.Count);
                    var usedPossibleValueCount = 0;
                    for (int k = 0; k < boardDfsq.Board.Table[index, indexy].PossibleValues.Count;
k++)
                    {
                    clear = false;
                    byte possibleValue = boardDfsq.Board.Table[index, indexy].PossibleValues[k];
                    BoardForDFSQueue child = boardDfsq.Copy();
                    child.Board.Table[index, indexy].Value = possibleValue;
                        if (childCount == _maxChildCount)
                        {
                        exit = true;
                        break;
                        }

                        Log.Verbose("Thread={0} found empty value: [{1},{2}], valid, possible
value: {3} {4}", Thread.CurrentThread.ManagedThreadId,
                        index, indexy, true, possibleValue);
                        child.Board.FillPossibleValues();
                        child.State = BoardProcessState.Empty;
                        child.QueueIndex = q + 1;
                        queueList[q + 1].Enqueue(child);
                        childCount++;
                        usedPossibleValueCount++;
                    }
                    boardDfsq.Board.Table[index, indexy].PossibleValues.RemoveRange(0,
usedPossibleValueCount);
                    }
                    if (exit)
                    {
                    break;
```

```
                    }
                    }
                    if (exit)
                    {
                    break;
                    }
                }
                if (clear)
                {
                    queueList[q].Dequeue();
                }
                Monitor.Exit(boardlocks[q]);

                }
                }
                Log.Verbose("Ended thread={0}", Thread.CurrentThread.ManagedThreadId);
                }

    }
```

5.3. Pseudo Code ve Karmaşıklık Analizi

Karmaşıklı analizi bir problemin çözüme ulaşmadaki karmaşıklığını ölçer. Karmaşıklı analizi için pseudo code kullanılır.

5.3.1. Pseudo code

Sözde kod olarak da adlandırılan pseudo kod programın çalışma mantığını anlatmak için yazılan yol haritasıdır. Programlama dillerine has syntax yapısı taşımaz. Daha çok günlük konuşma dilline benzer. Pseudo kod'un standart bir söz dizilimi yoktur.

Sudoku bulmacasının kuyruk liste veri yapısı tabanlı paralel önce derine arama yönteminin pseudo kodu şu şekildedir.

```
Input board
Input threadSayisi
Input maxChildCount
```

```
Set queueList to empty list
Set threads to empty list
Set cozuldu to false
Set boardlocks to object array of 65
Call board.FillPossibleValues()
For i = 1 : 65
  Add a new Queue object to queueList list
  boardlocks[i] = new lock object
Enqueue board to queueList[0]
For i = 1 : threadSayisi
  Add a new thread to threads list
For i = 1 : threadSayisi
  Start threads[i] to work on Coz function
For i = 1 : threadSayisi
  Wait for threads[i] to complete
Return cozumBoard
```

Function Coz

```
  While cozuldu is false
        For i = 65 : 1
              If cozuldu is true
                    Break from loop
              Set lockTaken to false
              Try to get lock for boardLocks[i]
              If lockTaken is false
                    Continue loop
              If Size of queueList[i] is 0
                    Release lock boardLocks[i]
                    Continue loop
              Set boardDfsq to result of queueList[i].Peek()
              If boardDfsq.IsSolved() is true
                    Set cozuldu to true
                    Set cozumBoard to boardDfsq
                    Release lock boardLocks[i]
                    Break from loop
```

```
                    Set childCount to 0
                    Set exit to false
                    Set clear to true
                    If boardDfsq.KilitlendiMi() is true
                        Dequeue board from queueList[i]
                        Release lock boardLocks[i]
                        Continue loop
                    For index = 1 : 9
                        For indexy = 1 : 9
                            If boardDfsq.Table[index, indexy] is 0
                                Set usedPossibleValueCount to 0
                                For k = 0 : boardDfsq.Table[index,
indexy].PossibleValues.Count
                                    Set clear to false
                                    Set possibleValue to boardDfsq.Table[index,
indexy]. PossibleValues[k]
                                    Set childBoard to boardDfsq.Copy()
        Set childBoard.Table[index, indexy].Value = possibleValue;
        If childCount is equal maxChildCount
        Set exit to true
        Break from loop
        Call childBoard.FillPossibleValues()
        Enqueue childBoard to queueList[i + 1]
        Set usedPossibleValueCount to usedPossibleValueCount + 1
        Remove first usedPossibleValueCount elements from boardDfsq.Board.Table[index,
indexy].PossibleValues
        If exit is true
        Break from loop
                        If exit is true
                            Break from loop
                    If clear is true
                        Dequeue board from queueList[i]
                    Release lock boardLocks[i]
```

Function Board.FillPossibleValues

```
While True
        Set fillAgain to true
        For index = 1 : 9
                For indexy = 1 : 9
                        Set cell to Table[index, indexy]
                        If cell.Value is 0
                                Set cellValues to empty list
                                For i = 1 : 9
                                        Set cell.Value = i
                                        If IsValid() result is true
                                                Add i to cellValues list
                                        Set cell.Value to 0
                                Set cell.PossibleValues to cellValues
                                If cell.PossibleValues count is 1
                                        cell.Value = cell.PossibleValues[0]
                                        Set fillAgain to true
                                        Break from loop
                If fillAgain is true
                        Break from loop
        If fillAgain is true
                Continue loop
        Break from loop
```

Function Board.IsValid

```
Set row to Cell array of 9
For rowIndex = 1 : 9
        For colIndex = 1 : 9
                Set row[colIndex] = Table[rowIndex, colIndex]
        If ArraySayiKontrol(row) result is false
                Return false
For colIndex = 1 : 9
        For rowIndex = 1 : 9
Set row[rowIndex] = Table[rowIndex, colIndex]
                If ArraySayiKontrol(row) result is false
```

```
                    Return false
  For index = 1, index <= 9, index = index + 3
        For y = 1, y <= 9, y = y + 3
              Set rowIndex to 0
              For i = index, i < index + 3, i = i + 1
                    For k = y, k < y + 3, k = k + 1
                          Set row[rowIndex] = Table[i, k]
                          Set rowIndex = rowIndex + 1
              If ArraySayiKontrol(row) result is false
                    Return false
  Return true
```

Function Board.ArraySayiKontrol

```
  Input cells array
  Set numbers to boolean array of 9, default value of false
  Foreach cell in cells
        If cell.Value is 0
              Continue loop
        Set value to numbers[cell.Value - 1]
        If value is true
              Return true
        Set numbers[cell.Value - 1] to true
  Return true
```

Function Board.KilitlendiMi

```
  For index = 1 : 9
        For indexy = 1 : 9
              Set cell to Table[index, indexy]
              If cell.Value is 0 and cell.PossibleValues.Count is 0
                    Return true
  Return false
```

Function Board.Copy

```
  Set board to new Board object
  Set newTable to array of Cell object (9, 9)
```

```
  For index = 1 : 9
        For indexy = 1 : 9
              newTable[index, indexy] = Table[index, indexy].Copy()
  board.Table = newTable
  Return board

Function Board.IsSolved
  For index = 1 : 9
        For indexy = 1 : 9
              Set cellValue to Table[index, indexy].Value
              If cellValue is 0
                    Return false
  Return true

Function Cell.Copy
  Set copy to new Cell object
  Set copy.Value to Value
  Add all PossibleValues copy.PossibleValues array
  Return copy
```

5.3.2. Karmaşıklık analizi

Karmaşıklık analizi bir algoritmanın çözüme ulaşmak için nasıl bir karmaşıklık seviyesine sahip olduğunu ölçmek için kullanılır. Bu ölçü aynı zamanda algoritmanın performansı hakkında bize önemli bir bilgi verir. Karmaşıklık için birçok farklı birim kullanılır. En sık kullanılan birim büyük-O (big-oh) dediğimiz birimdir. Bu birim algoritmanın en kötü ihtimalle hangi performansta çalışacağını hakkında bize fikir verir. Büyük-O değeri problemin girdi sayısına orantılı olarak bir formül ile gösterilir. Çünkü girdi sayısına göre algoritmalar farklı çalışma performansı gösterebilir.

Sudoku problemindeki girdi sayısı tahtadaki bilinmeyen hücre sayısıdır. Değeri bilinmeyen hücre sayısına bağlı olarak sudoku probleminin çözüm süresi değişmektedir. Kuyruk listesi çözümü analiz edildiğinde Büyük-O değerinin n^2 olduğu sonucuna varılmıştır. Bunun sebebi threadlerin değeri bilinmeyen hücreler üzerinde işlem yapması ve her işlem

sonrası tüm diğer bilinmeyen hücreleri alabileceği muhtemel değerlerin değişmesi sebebiyle tekrar hesaplanmasıdır.

6. DENEYLER

Önerilen algoritmanın test edilmesi için 3,40 GHz hızında, 4 çekirdek bulunan Intel Core i7-4770 CPU işlemcili bir bilgisayar kullanılmıştır. Deneylerde hem önce derine hem de paralel önce derine ile çözümleri yapılmıştır. Deneylerde alınan değerler 10 adet deneme sonucunun ortalaması olarak hesaplanmıştır. 2'şer adet iki farklı zorluk seviyesinde toplam 4 adet sudoku problemi denenmiştir. Deneylerden kullanılan farklı sudoku zorluk seviyeleri Hard ve xxHard olarak adlandırılmıştır ve sonuçları aşağıdaki Tablo 6.1. ve Tablo 6.2.'de gösterilmektedir.

Tablo 6. 1. *Zorluk seviyesi Hard olan sudoku bulmacalarının farklı thread ve MaxChild değerleri ile çözülme süreleri*

Thread Sayısı	MaxChilds Sayısı	Hard I		Hard II	
		DFS (ms)	Paralel DFS (ms)	DFS (ms)	Paralel DFS (ms)
2	2	81,4	46,5	10,6	3,4
	3	81,4	55,8	10,6	3,3
	4	81,4	74,9	10,6	3,3
	5	81,4	59,8	10,6	5,3
	6	81,4	96,7	10,6	7,8
	7	81,4	117,9	10,6	8,7
	8	81,4	148,0	10,6	9,9
	9	81,4	181,8	10,6	9,9
	10	81,4	230,7	10,6	12,5
3	2	81,4	41,9	10,6	3,3
	3	81,4	52,9	10,6	3,3
	4	81,4	81,2	10,6	3,4
	5	81,4	48,1	10,6	5,4
	6	81,4	77,4	10,6	7,3

Tablo 6. 2. (Devam) *Zorluk seviyesi Hard olan sudoku bulmacalarının farklı thread ve MaxChild değerleri ile çözülme süreleri*

	7		81,4	104,0		10,6	7,9
	8		81,4	125,6		10,6	9,8
	9		81,4	143,5		10,6	10,7
	10		81,4	192,7		10,6	12,2
4	2		81,4	43,4		10,6	4,0
	3		81,4	57,5		10,6	3,9
	4		81,4	82,4		10,6	3,8
	5		81,4	57,6		10,6	6,0
	6		81,4	89,9		10,6	7,6
	7		81,4	95,3		10,6	8,0
	8		81,4	120,3		10,6	10,5
	9		81,4	136,5		10,6	10,6
	10		81,4	188,2		10,6	12,9
8	2		81,4	54,6		10,6	5,3
	3		81,4	74,4		10,6	5,2
	4		81,4	119,0		10,6	5,2
	5		81,4	76,8		10,6	7,4
	6		81,4	120,0		10,6	9,7
	7		81,4	149,6		10,6	9,5
	8		81,4	194,2		10,6	11,6
	9		81,4	208,5		10,6	11,5
	10		81,4	337,0		10,6	14,1

Tablo 6. 3. (Devam) *Zorluk seviyesi Hard olan sudoku bulmacalarının farklı thread ve MaxChild değerleri ile çözülme süreleri*

16	2		81,4	117,3		10,6	5,5
	3		81,4	90,7		10,6	14,7
	4		81,4	156,2		10,6	16,0
	5		81,4	92,1		10,6	16,2
	6		81,4	187,5		10,6	9,2
	7		81,4	210,2		10,6	9,6
	8		81,4	283,7		10,6	11,1
	9		81,4	417,1		10,6	12,6
	10		81,4	1183,3		10,6	24,0
32	2		81,4	150,8		10,6	7,8
	3		81,4	126,2		10,6	16,8
	4		81,4	201,2		10,6	7,6
	5		81,4	124,6		10,6	8,2
	6		81,4	210,3		10,6	9,4
	7		81,4	304,2		10,6	10,1
	8		81,4	423,8		10,6	11,5
	9		81,4	556,1		10,6	31,1
	10		81,4	2167,3		10,6	14,7

Tablo 6. 4. *Zorluk seviyesi xxHard olan sudoku bulmacalarının farklı thread ve MaxChild değerleri ile çözülme süreleri*

Thread Sayısı	MaxChilds Sayısı	xxHard I		xxHard II	
		DFS (ms)	Paralel DFS (ms)	DFS (ms)	Paralel DFS (ms)
2	2	82,1	44,8	17,0	9,7
	3	82,1	58,7	17,0	15,0
	4	82,1	77,5	17,0	13,8
	5	82,1	59,7	17,0	17,4
	6	82,1	98,4	17,0	13,3
	7	82,1	122,4	17,0	20,7
	8	82,1	156,7	17,0	23,8
	9	82,1	182,5	17,0	13,6
	10	82,1	235,8	17,0	27,8
3	2	82,1	43,9	17,0	10,8
	3	82,1	53,8	17,0	14,4
	4	82,1	82,4	17,0	14,8
	5	82,1	49,1	17,0	17,4
	6	82,1	74,1	17,0	22,9
	7	82,1	103,8	17,0	32,3
	8	82,1	121,7	17,0	40,4
	9	82,1	139,6	17,0	42,0
	10	82,1	194,6	17,0	46,4
4	2	82,1	41,9	17,0	13,8
	3	82,1	58,7	17,0	19,6
	4	82,1	84,8	17,0	19,9

Tablo 6. 5. (Devam) *Zorluk seviyesi xxHard olan sudoku bulmacalarının farklı thread ve MaxChild değerleri ile çözülme süreleri*

	5		82,1	57,3	17,0	22,5
	6		82,1	90,8	17,0	26,0
	7		82,1	97,8	17,0	31,5
	8		82,1	117,5	17,0	38,0
	9		82,1	146,0	17,0	39,0
	10		82,1	201,8	17,0	47,6
8	2		82,1	60,2	17,0	13,4
	3		82,1	75,0	17,0	19,4
	4		82,1	115,4	17,0	17,7
	5		82,1	74,6	17,0	24,5
	6		82,1	115,8	17,0	33,1
	7		82,1	156,8	17,0	46,9
	8		82,1	201,3	17,0	49,6
	9		82,1	204,7	17,0	55,7
	10		82,1	315,7	17,0	57,3
16	2		82,1	58,8	17,0	13,9
	3		82,1	106,9	17,0	20,4
	4		82,1	172,3	17,0	17,5
	5		82,1	92,4	17,0	25,4
	6		82,1	137,8	17,0	37,2
	7		82,1	243,0	17,0	52,1
	8		82,1	256,3	17,0	48,6
	9		82,1	368,3	17,0	55,8
	10		82,1	472,2	17,0	62,7

Tablo 6. 6. (Devam) *Zorluk seviyesi xxHard olan sudoku bulmacalarının farklı thread ve MaxChild değerleri ile çözülme süreleri*

32	2		82,1	68,9	17,0	15,8
	3		82,1	95,1	17,0	21,4
	4		82,1	148,2	17,0	18,6
	5		82,1	102,0	17,0	26,9
	6		82,1	167,5	17,0	34,8
	7		82,1	215,1	17,0	52,6
	8		82,1	280,3	17,0	58,1
	9		82,1	443,1	17,0	61,3
	10		82,1	488,7	17,0	60,4

Deneylerde kuyruk listesi veri yapısı tabanlı paralel önce derine aramasında, farklı sayıda maxChildCount ve farklı sayıda thread değerleri kullanılmıştır.

İncelemeler sonucunda önce derine arama ile kuyruk listesi veri yapısı tabanlı paralel önce derine arama karşılaştırıldığında kuyruk listesi veri yapısı tabanlı paralel önce derine aramasının önce derine arama algoritmasına göre daha hızlı çözebildiği kanıtlanmıştır.

Önce derine arama tek bir stack kullanıldığı için, aynı anda sadece tek bir işlem yapılabilmektedir, paralel önce derine arama yönteminde ise birden fazla thread farklı kuyruklarda iş yapabildiği için performansta artış olabilmiştir.

Thread'lerin ve maxChildCount'ların artışı ile belirli bir yere kadar önce derine arama algoritmasının çözüm süresinin altında bir sürede çözüldüğü görülmüştür. Fakat thread ve maxChildCount'ların artışları çözüm süresinin yavaşlattığı da ortadadır. Deneyler sonucunda ideal maxChildCount ve thread'ler belirlenmiştir.

Kuruk listesi veri yapısı kullanılarak sudoku bulmacasının ideal maxChildCount sayısı 2,3 ve 4 olup bu sayı sudokunun karakteristik özelliğini göstermektedir.

En iyi thread performansı thread sayısı 2 olduğunda gözlenmiştir. Thread sayısının artışı işlemciye ek yük getirdiğinden çözüm süresi yavaşlamaktadır.

7. SONUÇ ve ÖNERİLER

Bu çalışmada, sudoku bulmacasının çözümü için kuyruk listesi veri yapısı oluşturularak önce derine arama algoritmasının paralelleştirilmesi sağlanmıştır. Kuyruk liste veri yapısı tabanlı paralel önce derine arama yönteminin geleneksel önce derine arama algoritması yöntemine göre performansı değerlendirilmiştir. En fazla 65 adımda çözülen sudoku problemleri için 65 adet kuyruk oluşturulmuş ve her bir kuyruk başlangıçta boş olarak belirlenmiştir. 65 satırın her birini yalnızca 1 thread in işleyebilmesi için kilitler oluşturulmuştur. Her bir satırın bir thread tarafından işlenirken tüm threadlerin dolu olması durumunda yeni bir satır oluşursa threadlerden birinin işini bitirmesini beklenmiştir. Bu bekleme süresi hızlı çözümü geciktireceği için maxChildCount ile sınırlandırma yapılmıştır. MaxChildCount boş bir hücrenin alabileceği değerleri belirli bir sayı ile sınırlandırarak alt kuyruğa atılmasını sağlanmıştır. Birden fazla thread aynı anda en alttaki 65. Kuyruk listesinden çalışmaya başlatılmıştır. Kuyruk listesinde bir değer olması durumunda threadlerden biri o kuyruğu kilitleyerek işlemeye başlamıştır. Kuyruğu kilitleme ve açma işlemi Monitor.TryEnter() metodu ile sağlamıştır. Diğer threadler ise alt kuyruklara yeni değerler gelmesini beklemiştir.

Kuyruk liste veri yapısı tabanlı paralel önce derine arama yönteminde farklı sayılarda thread ve farklı miktarlarda olası sayı değerleri kuyruğa atılarak incelendiğinde belirli bir seviyeye kadar arttırılan thread ve maxChildCount sayıları ile geleneksel önce derine arama yöntemine göre daha hızlı çözdüğü görülmüştür.

Sudoku bulmacasının Kuyruk listesi veri yapısını kullanarak çözülmesindeki ideal MaxChildCount sayıları 2-3 ve 4'tür. Thread sayısı 2 olduğunda en hızlı şekilde çalışmaktadır.

Deneyde 2 farklı seviyede ve her 2 seviyede 2'şer faklı bulmacada denemiştir. Deney farklı zorluk seviyelerindeki sudoku bulmacaların çözümleri ile denenebilir.

Sadece 4 çekirdek bulunan Intel Core i7-4770 CPU ile çözülen sudoku bulmacalarının performansı farklı ortamlarda tekrar incelenebilir. Bunlardan bazıları;

Farklı CPU ile çözülerek hız performansları incelenebilir.

Başka bir programlama dili.

Yüksek paralel yapılı kompleks algotirmalar için CPU lardan daha verimli çalışan GPU'larda

HPC ortamlarıdır.

KAYNAKÇA

[1] https://sudoku.matematiktutkusu.com/32-sudoku-kurallari.html (29.03.2018)

[2] Lee, K. S., & Geem, Z. W. (2004). A new structural optimization method based on the harmony search algorithm. Computers & structures, 82(9-10), 781-798.

[3] Akkoyunlu, M. C. (2013). Bulanık paralel çok işlemcili makina problemlerinin çözümünde harmoni arama algoritması (Doctoral dissertation, Selçuk Üniversitesi Fen Bilimleri Enstitüsü).

[4] Mitchell, M. (1998). *An introduction to genetic algorithms*. MIT press.

[5] http://www.algoritmauzmani.com/algoritmalar/dfs-algoritmasi-derin-oncelikli-arama-konu-anlatimi-c-kodu/

[6] http://web.karabuk.edu.tr/hakankutucu/BLM227/VER%C4%B0%20YAPILARI%20v.6.3.pdf (7.02.2018)

[7] http://www.algoritmauzmani.com/algoritmalar/dfs-algoritmasi-derin-oncelikli-arama-konu-anlatimi-c-kodu/ (29.01.2018)

[8] https://uskudar.edu.tr/assets/uploads/basin/8729.jpg?31 (15.02.2018)

[9] Santos-García, G., & Palomino, M. (2007). Solving Sudoku puzzles with rewriting rules. Electronic Notes in Theoretical Computer Science, 176(4), 79-93.

[10] Soto, R., Crawford, B., Galleguillos, C., Monfroy, E., & Paredes, F. (2013). A hybrid ac3-tabu search algorithm for solving sudoku puzzles. Expert Systems with Applications, 40(15), 5817-5821.

[11] Maji, A. K., Jana, S., & Pal, R. K. (2013). An algorithm for generating only desired permutations for solving Sudoku puzzle. Procedia Technology, 10, 392-399.

[12] Geem, Zong Woo (2007). "Harmony Search Algorithm for Solving Sudoku."

[13] Berggren, P., & Nilsson, D. (2012). A study of Sudoku solving algorithms. Royal Institute of Technology, Stockholm

[14] Agrawal, A., & Bonde, P. (2015). Study on the Performance Characteristics of Sudoku Solving Algorithms. International Journal of Computer Applications, 122(1).

[15] Chakraborty, R., Paladhi, S., Chatterjee, S., & Banerjee, S. (2014). An Optimized Algorithm for Solving Combinatorial Problems using Reference Graph. IOSR Journal of Computer Engineering, 16(3), 1-7.

[16] Viksten, H., & MATTSSON, V. (2013). Performance and scalability of Sudoku solvers.

[17] Mantere, T., & Koljonen, J. (2007, September). Solving, rating and generating Sudoku puzzles with GA. In Evolutionary Computation, 2007. CEC 2007. IEEE Congress on (pp. 1382-1389). IEEE.

[18] Asif, M., & Baig, R. (2009, October). Solving NP-complete problem using ACO algorithm. In Emerging Technologies, 2009. ICET 2009. International Conference on (pp. 13-16). IEEE.

ÖZGEÇMİŞ

Adı Soyadı	: Zeynep Feyza ESEN
Yabancı Dili	: İngilizce
Doğum Yeri ve Yılı	: Eskişehir / 1984
E-Posta	: zfm@anadolu.edu.tr

Eğitim Geçmişi

- 2004 - 2009, Lisans, Başkent Üniversitesi, Eğitim Fakültesi, Bilgisayar ve Öğretim Teknolojileri Öğretmenliği Bölümü
- 1999 – 2003, Lise, Türk Telekom Mesleki ve Teknik Anadolu Lisesi, Bilişim Teknolojileri

Mesleki Geçmiş

2014 - …. Öğretim Görevlisi, Eskişehir Osmangazi Üniversitesi, Uzaktan Eğitim Uygulama ve Araştırma Merkezi

2012 – 2014, Yazılım Geliştirici, Anadolu Üniversitesi, Bilgisayar Araştırma ve Uygulama Merkezi

2009 – 2010, Bilgisayar Öğretmeni, Sabiha Gökçen Mesleki ve Teknik Anadolu Lisesi