YAPAY ZEKA UYGULAMALARINDA KULLANILAN ARAMA ALGORİTMALARININ KIYASLANMASI

Ali ihsan BENZER

YÜKSEK LİSANS TEZİ
BİLGİSAYAR EĞİTİMİ

GAZİ ÜNİVERSİTESİ
BİLİŞİM ENSTİTÜSÜ

MAYIS 2007
ANKARA

Ali ihsan BENZER tarafından hazırlanan YAPAY ZEKA UYGULAMALARINDA KULLANILAN ARAMA ALGORİTMALARININ KIYASLANMASI adlı bu tezin Yüksek Lisans tezi olarak uygun olduğunu onaylarım.

Yrd. Doç. Dr. Benian TEKİNDAL
Tez Yöneticisi

Bu çalışma, jürimiz tarafından oy birliği ile Bilgisayar Eğitimi Anabilim Dalında Yüksek Lisans tezi olarak kabul edilmiştir.

Başkan : Prof. Dr. Ali Paşa AYDIN

Üye : Prof. Dr. Ömer Faruk BAY

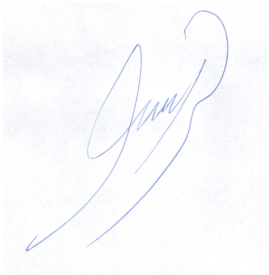
Üye : Yrd. Doç. Dr. Benian TEKİNDAL

Tarih : 23 / 05 / 2007

Bu tez, Gazi Üniversitesi Bilişim Enstitüsü tez yazım kurallarına uygundur.

TEZ BİLDİRİMİ

Tez içindeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edilerek sunulduğunu, ayrıca tez yazım kurallarına uygun olarak hazırlanan bu çalışmada orijinal olmayan her türlü kaynağa eksiksiz atıf yapıldığını bildiririm.



Ali ihsan BENZER

YAPAY ZEKA UYGULAMALARINDA KULLANILAN ARAMA ALGORİTMALARININ KIYASLANMASI

(Yüksek Lisans Tezi)

Ali ihsan BENZER

GAZİ ÜNİVERSİTESİ

BİLİŞİM ENSTİTÜSÜ

Mayıs 2007

ÖZET

Arama işlemi yapay zeka alanın önemli bir parçasıdır. Bir problemin çözümü için aramak ve en uygun çözümü bulmak gerekir. Genel olarak arama algoritmaları iki ana başlık altında toplanmıştır. Bunlar; uninformed ve informed aramalardır. Uninformed aramalara kör aramalar da denmiştir. Bunun sebebi arama yaparken herhangi bir bilgi kullanmamasıdır. Bu arama algoritmaları; Breadth-first search, Depth-first search, Bidirectional (BF) Search`dır. Informed algoritmaları ise arama yaparken daha başarılı olmaktadır. Bunun sebebi arama yaparken bazı bilgileri kullanmasıdır. Bu arama kategorisine sezgisel (heuristic) aramalar da denmektedir. Bu kategorideki algoritmalar Best-first -Greedy arama, A* Aramalar örnek olarak verilebilir. Arama algoritmalarının çalışmalarını incelemek için birbirleriyle kıyaslama işlemi yapılmıştır. Arama algoritmalarının birbirleriyle kıyaslanması için 8-puzzle problemi kullanılmıştır. Kıyaslama işlemi için tam rasgele ve mantıksal rasgele yöntemleri kullanılarak 1000`er adet 8-puzzle başlangıç durumu örneği oluşturulmuştur. Bu oluşturulan örnekler 5 dakikalık süre içerisinde BFS, DFS ve A* algoritmaları tarafından, hedef duruma ulaşmak üzere çözümlendirilmeye çalışılmış ve çözüm için açtıkları durum sayıları bir metin dosyasına kayıt edilmiştir. Elde edilen sonuçlar tek yönlü varyans analizi tekniği ile analize tabi tutulmuştur.

Farklı grupların tespitinde ise Tukay testi kullanılmıştır. Oranların karşılaştırılmasında ise Z testi kullanılmıştır.

Bu çalışmanın esas amacı arama algoritmalarından en çok kullanılan BFS, DFS ve A* algoritmalarının etkinliğinin araştırılması ve birbirleriyle kıyaslamaktır.

Bilim Kodu : 702.3.006
Anahtar Kelimeler : Yapay Zeka, Arama algoritmaları, Sezgisellik
Sayfa Adedi : 100
Tez Yöneticisi : Yrd. Doç. Dr. Benian TEKİNDAL

COMPARING SEARCH ALGORITHMS IN USED ARTIFICIAL INTELLIGENCE APPLICATIONS

(M.Sc. Thesis)

Ali ihsan BENZER

GAZI UNIVERSITY
INFORMATICS INSTITUTE
May 2007

ABSTRACT

Search implementation is an important part of artificial intelligence. The most suitable solution must be found and searching is essential for solution of problem. Typically, Search algorithms are separated two importants groups. These are uniformed search and informed search. Uninformed search is also called blind search. This is because, Uninformed doesn`t use any information that about the problem. These algorithms are Breadth-first search, Depth-first search and Bidirectional (BF) Search. Informed Algorithms are more succesfully than uninformed search in search implementation. This is because, informed algorithms use some information that about the problem. Informed search is also called heuristic search. These algoritms are Best-first – Greedy search, A* search. 8-puzzle problem have been used for compare search algorithms. For this process 1000 8-puzzle start state samples have been generated by using complete random and logical random methods. This generated samples have been tried to solve by BFS, DFS and A* algorithms and the expanded total state numbers for solution have been saved into text file. The results that were obtained have been undergone of statistical analysis with one-way variance analysis tecnique. Tukay test have been used to determine different groups. Z test have been used to compare of proportions.

This study`s main aim is that explore and compare the effectiveness of well-known search algorithms, BFS, DFS and A* algorithms.

Science Code : 702.3.006
Key Words : Artificial Intelligence, Search Algorithms, Heuristic
Page Number : 100
Adviser : Assistant Prof. Dr. Benian TEKİNDAL

TEŞEKKÜR

Çalışmalarımda her türlü desteği esirgemeyen ve beni yönlendiren danışmanım ve değerli hocam Yrd. Doç. Dr. Benian TEKİNDAL'a, istatistik analizleri, değerlendirme ve yorumlarda bana yardımcı olan Ankara Üniversitesi, Ziraat Fakültesi, Biyometri İstatistik öğretim üyesi Prof. Dr. Fikret GÜRBÜZ'e ve aynı bölümde Araştırma Görevlisi Özgür KOŞKAN'a, çalışmalarımda sürekli bana destek olan GenelKurmay Başkanlığı MEBS Başkanlığında görevli Dr.Müh.Yzb. Recep BENZER`e teşekkürü bir borç bilirim.

İÇİNDEKİLER

Sayfa

ÇİZELGELERİN LİSTESİ

Çizelge **Sayfa**

ŞEKİLLERİN LİSTESİ

Şekil **Sayfa**

SİMGELER VE KISALTMALAR

Bu çalışmada kullanılmış bazı simgeler ve kısaltmalar, açıklamaları ile birlikte aşağıda sunulmuştur.

Simgeler	Açıklama
b	Dallanma faktörü
o	Algoritma karmaşıklığı
d	Hedef düğüm derinliği
g(n)	Maliyet tahmini fonksiyonu
f(n)	En ucuz çözüm değerlendirme fonksiyonu
m	Maksimum derinlik
l	Derinlik sınırı
h	Sezgisel fonksiyon

Kısaltmalar	Açıklama
A*	A Star Search
AI	Artificial Intelligence
BF	Bidirectional
BFS	Breadth –First Search
DFS	Depth –First Search
DLS	Depth Limited Search
IDA	Iterative Deepening A* Search
IDS	Iteratif Derinleşme Search
RBFS	Recursive Best-First Search
SLD	Straight-line Distance
SMA	Simplified Memory-Bounded A*
YZ	Yapay Zeka

1. GİRİŞ

Bilgisayar teknolojisindeki ilerlemelerin, son yıllarda baş döndürücü bir hıza erişmesi, beraberinde yeni çalışma alanlarını da gündeme getirdi. İnsanoğlunun bilgisayar teknolojisini ortaya koyuncaya kadar kullandığı araçların büyük bir çoğunluğu, kol gücüne dayanan çalışmaları kolaylaştırmaya yöneliktir. Bilgisayar teknolojisinin ortaya çıkması ile bilgi derleme, değerlendirme, saklama ve benzeri beyin gücü gerektiren faaliyetlere yardımcı olan araçlar ve algoritmalar geliştirilmiştir.

Arama pek çok yapay zeka uygulamasının önemli bir parçasıdır. Arama pek çok alanda kullanılmaktadır. Örneğin oyun oynama programları, optimizasyon, yol bulma (path-finding), planlama, web etmenleri(agents), ağ sistemleri, DNA sıra diziminde. Kısaca, arama yapay zekanın kalbidir [1].

Bazı bilim adamlarına göre yapay zeka iki ana bölüme ayrılmaktadır. Bunlar; Bilgi(Knowledge) ve Arama(Search) [2].

Arama, en uygun cevabı vermek için bilgi alternatiflerini araştırmaktır. Arama bilginin kalitesini artırır. İstenmeyen bilgileri atarak kaynakları sınırlı kullanarak, gereksiz bilgi miktarını azaltır. Bunun en iyi örneği Deep Blue ile Kasparov arasında geçen satranç oyunudur. Bilgisayar saniyede 200 milyon pozisyonda arama yaparak en iyi hamleleri arama algoritmaları sayesinde yapmıştır [2].

Bu çalışmanın amacı yapay zeka uygulamalarında sıklıkla kullanılan arama algoritmalarını, örnek bir problem üzerinde etkilerini ve etkinliklerini araştırmaktır. Bu amaçla kullanılan örnek problem 8-puzzle oyunudur. Bu oyunun amacı rasgele yerleştirilen 8 taşı boşluğu kullanarak sağa, sola, yukarı ve aşağı hareket ettirerek istenilen duruma ulaştırmaktır. Bu işlemi gerçekleştirmek için BFS, DFS, A* (h1) ve A* (h2) algoritmaları kullanılmıştır.

Algoritmalardan BFS ve DFS kör arama grubu A* (h1) ve A*(h2) ise sezgisel arama grubu algoritmalarıdır.

Karşılaştırma işlemi için 1000 adet 8-puzzle örneği oluşturulmuş ve hedef durum sabit tutulmuştur. Her bir algoritma üretilen 1000 problemi çözmeye çalışmışlardır. Çözüme ulaşana kadar açtıkları durumlar, maliyeti belirlemektedir. Ne kadar az durum açarak çözüm bulunursa algoritma o kadar etkindir. Algoritmaların çözüm için açtıkları durumlar bir dosyaya kaydedilerek istatistiksel analize tabi tutulmuşlardır.

Bu çalışma bittiğinde algoritmaların çözümü ne kadar durum açarak buldukları ile, hangi algoritmanın problem çözmede daha etkin olduğu tespit edilmeye çalışılacaktır.

Literatürde yapılan araştırmalarda, algoritmaların genel özellikleriyle ilgili bilgiler bulunmuş olup istatistiksel teknikler kullanılarak algoritmaların karşılaştırma işleminin yapılmadığı gözlenmiştir.

Çalışmada VC++ yazılmış 8-puzzle problemi kullanılmıştır. Bu problemi çözmek için BFS, DFS, A* (h1) ve A* (h2) algoritmaları kullanılmaktadır. Algoritmaların çözümü bulana kadar açtıkları durumlar bir txt dosyasına kayıt edilmiş olup bu bilgiler istatistiksel analize tabi tutulmuştur. Bunun için tek yönlü varyans analizi ve Z testi teknikleri kullanılmıştır. Varyans analizi ortalamarı, Z testi ise oranları karşılaştırmaktadır.

Bu çalışmada; 2. bölümde literatür ve kuramsal temeller üzerine değinilmiş, yapılmış çalışmalardan bahsedilmiştir. 3. Bölümde yapay zeka ve arama ile ilgili kavramlar, yapay zeka uygulamalarında kullanılan arama algoritmaları ve çalışma mantıklarından bahsedilmiştir. 4. Bölümde bölümde 8-puzzle uygulaması ile ilgili bilgiler verilerek, kullanılan materyal ve metotlardan bahsedilmiştir. Son bölümde yapılan çalışma sonucunda elde edilen bilgiler ışığında algoritmaların etkinlikleri tartışılmıştır.

2. KURAMSAL TEMELLER ve KAYNAK ARAŞTIRMASI

Genel olarak arama algoritmaları problem çözme algoritmaları olarak da isimlendirilir. Yaygın kullanım alanları yol bulma, oyun oynama, robot yön hareketleri, ağ topolojilerinde yön bulmadır.

Pek çok durum uzayı problemlerinin literatürde oldukça eski bir geçmişi vardır. Misyoner ve yamyam problemi Amarel (1968) tarafından analiz edilmiştir.

1957`de Newell ve Simon Mantık teorisi üzerine çalışmalar yapmışlardır ve aynı yıl genel problem çözücüyü yazdılar [3].

Knoblock (1990) problem gösterimi ve çıkartımı işlemlerini içeren AI programları yapmıştır.

8 puzzle 15 puzzle`ın küçük kuzenidir. 15 puzzle ünlü amerikan oyun tasarımcısı Sam Loyd (1959) tarafından keşfedilmiştir. İlk keşfedildiğinde pekçok matematikçinin yoğun ilgisini çekmiştir. 8-puzzle`ın teferruatlı analizi Schofield (1967) tarafından bilgisayar yardımıyla yapıldı. Rather ve Warmuth (1986) tarafından NP-complete problem türüne ait 15- puzzle`ın genel n x n versiyonu ortaya koyulmuştur [4].

8-vezir problemi ilk olarak Alman satranç magazin dergisinde 1848 yılında basıldı. Netto n vezir problemini genelleştirdi ve Abramson ve Yung bir O(n) algoritmasını buldu [4].

Problem çözmek için kullanılan Uninformed arama algoritmaları klasik bilgisayar biliminin bir merkez noktasıdır. Ve araştırmacılar Dreyfus (1969), Deo ve Pang (1984) ve Gallo ve Pallottino (1988) bu konuda pek çok incelemelerde bulunmuştur. Breadth-first arama Moore (1959) tarafından labirent çözümleri için formüle edilmiştir. Dinamik programlama metodu

(Bellman ve Dreyfus, 1962) graflarda bir breadth-first şekli olarak görünür. İki nokta arasındaki en kısa yol Dijkstra (1959) tarafından bulunmuş ve uniform maliyet aramanın çıkış noktası olmuştur[4].

İteratif deepening versiyonu satranç zamanını daha etkili kullanabilmek amacıyla ilk olarak Slate ve Atkin (1977) tarafından Satranç 4.5 oyun programında kullanılmıştır. Fakat uygulamanın en kısa yol aramasını Korf (1985) yapmıştır.

İki yönlü arama Pohl (1969, 1971) tarafından bulunmuş ve bazı durumlarda oldukça etkili olmuştur.

Kısmen görülebilir ve belirsiz çevrelerde problem çözme yaklaşımıyla büyük derinliklerde çalışılamaz. Genesereth ve Nourbakhsh (1993) tarafından böyle arama durumlarında bazı başarılar elde edilmiştir.

Koenig ve Simmons (1998) bilinmeyen başlangıç durumundan robot yön hareketleriyle ilgili çalışmalar yapmışlardır.

Erdmann ve Mason (1988) arama kullanarak robotları sensörsüz idare etme üzerine çalışmalar yapmışlardır.

George Polya “How to Solve it” (1945) yılında yayınladığı kitabında heuristic ile ilgili ilk çalışmaları yapmıştır.

Sezgisel bilgilerin problem çözmede kullanımları ilk olarak Simon ve Newel (1958) tarafından yapılmıştır [4].

“Sezgisel arama” deyimi ve amaca uzaklığı hesap eden sezgisel fonksiyonların kullanımı Newel ve Ernst (1965) yapmışlardır [4].

Doran ve Michie (1966) özellikle 8-puzzle ve 15 puzzle gibi bir örnek probleme uyguladığı sezgisel aramaların geniş deneysel çalışmalarını yapmışlardır.

Sezgisel aramaya yol uzunluk bilgisini ekleyen A* algoritma Hart, Nilsson ve Raphael (1968) tarafından düzenlenmiştir[4].

Hart tarafından A* algoritmaya bazı düzeltmeler yapılarak geliştirilmiştir[4].

Dechter ve Pearl (1985) A* optimal verimliliğini gösteren çalışmalar yapmışlardır [4].

Lawler ve Wood tarafından A* ve diğer durum-uzayı arama algoritmaları araştırmalarda yaygın olarak kullanılan dallanma ve budama teknikleriyle ilişkilendirildi [4].

1985 yılında Korf tarafından Bellek sınırlı IDA* algoritması geliştirildi.

Zhou ve Hansen (2002) bellek sınırlı A* graf arama ile ilgili çalışmalar yapmıştır [5].

Russel tarafından 1992 yılında daha etkili bellek sınırlı A* algoritması SMA* geliştirildi [5].

3. YAPAY ZEKA NEDİR?

Yapay zeka kavramından önce zeka kavramının anlaşılması gerekir. Zekayı anlamak, beynimizin çalışma prensibini, yaptığı işlevleri ve işlemleri anlamaktan geçer. Bugün bilim beynimiz ile ilgili bir çok bilinmeyeni ortaya koymuş olsa da daha bilinmeyen bir çok husus vardır [6].

Zeka ile ilgili olarak, bir çok bilim adamı ilgi alanlarına göre farklı tanımlar yapmıştır. Bu tanımlardan bazıları:

- "iyi akıl yürütme, hüküm verme ve kendini iyileştirme kapasitesi",
- "soyut düşünebilme süreci",
- "algılama, sorgulama, yaratıcılık",
- "gayeli davranma, mantıklı düşünme ve çevresiyle ilişkilerinde etkili olma kapasitesi",
- "düşüncesini yeni durumlara bilinçli olarak uydurabilme yeteneği",
- "çevreye uygun tepkilerde bulunabilme",
- "öğrenme, problem çözme, yeni ürünler ortaya çıkarma ve iletişim kurma kapasitesi" [6].

D. Lenat ve E. Feigenbaum'un tanımlarına göre zeka "Karmaşık bir problemi çözmek için gerekli bilgileri toplayıp birleştirme kabiliyetidir" veya "Karmaşık bir problemi, çözüm arama alanını daraltarak kısa yoldan çözebilme kabiliyetidir" [7].

Yapay Zeka; zeka ve düşünme gerektiren işlemlerin bilgisayarlar tarafından yapılmasını sağlayacak araştırmaların yapılması ve yeni yöntemlerin geliştirilmesi hususunda çalışan bilim dalıdır . Daha geniş anlamda YZ:

- "bilgisayarların bilgi edinme, algılama, görme, düşünme ve karar verme gibi insan zekasına özgü kapasitelerle donatılması bilimi"

veya

- “programlanmış bir bilgisayar düşünme girişim”

olarak da tarif edilmektedir. YZ tanımının zeka tanımında olduğu gibi kolayca tarif edilemeyeceği ortadır [6].

Yapay zeka, zeka kavramından yola çıkarak, us yapısı teoremi ışığında geliştirilen algoritmaların, bilgisayar donanım ve yazılımlarına uygulanarak, düşünce üretim sistemleri oluşturma çalışmalarının bütünüdür.

Son zamanlarda popüler olmaya başlayan yapay zeka çalışmaları, bilgisayar bilimine yeni boyut kazandırmış, yeni bir bakış açısı geliştirmiştir [8].

Düşünce üretim sistemi, karakutu yaklaşımı ile göstertilmektedir.

Çizelge 3.1. Düşünce üretim sisteminin gösterimi.

GİRDİLER -->	SÜREÇLER -->	ÇIKTILAR
Şekilsel bilgi	Kavrama	Sistemler
Sembolik bilgi	Değerlendirme	Sınıflar
Semantik bilgi	Bellek	Çıkarsamalar
Davranışsal bilgi	Yaratıcılık	Mantık Yürütme

Günümüzde yapay zekanın kullanım alanları aşağıdaki gibi sıralanabilir [9,10].

- Problem Çözme, Arama ve Sezgisel Programlama
- Bilgi ve Çıkarsama
- Planlama
- Oyunlar (satranç, dama, strateji, diğer...)
- Yapay Yaşam (Hayat)
- Teorem İspatlama (Prolog, Paralel Prolog, Cebrik Mantık Prog.)

- Doğal Dil Anlama ve Çeviri
- Bilgi Tabanlı Sistemler (Bilgi gösterimi, uzman sistemler, bilgi tabanlı
- simülasyon, genel bilgi sistemleri -CYC, EDR-)
- Makine Öğrenmesi (Bilgi düzeyinde öğrenme, sembol düzeyinde öğrenme, aygıt düzeyinde öğrenme)
- Makine Buluşları (Bilgi Madenciliği, Bilimsel buluşların Modellendirilmesi)
- Robotik (Görev planlama, robot görmesi)
- Şekil Tanıma (Nesne tanıma, Optik Harf Tanıma (OCR), Ses Tanıma)

Örnek AI Sistemi

1) Satranç Oynama (Chess Playing) Deep Blue (IBM) gibi.

- Algılama : Satranç tahtasının ileri özellikleri
- Eylemler : Hareket seçme
- Çıkarsama : Tahta konumlarını değerlendirme
- Sezgisi ve arama.

3.1. Algoritma ve Graflar

Algoritma, mekanik davranan kişiye veya bir makineye, birtakım verilerden yola çıkarak ve sonlu sayıda aşamalardan geçerek, belli bir problemi çözme imkanı veren, çok kesin komutlar bütününden oluşmaktadır. Bir algoritmanın çalışmasındaki mutlak zorunluluk, her türlü belirsizlikten arınmış olmasıdır. Bir algoritmanın yürütülmesi, her biri bir komutla belirlenen bir etkiler dizisi oluşturur ve bu dizi, önceki komutun yürütülmesinin sona ermesiyle birlikte yürütülmeye başlar [11].

Algoritma problemi nasıl çözüleceğini açıklayan komut dizileridir [4].

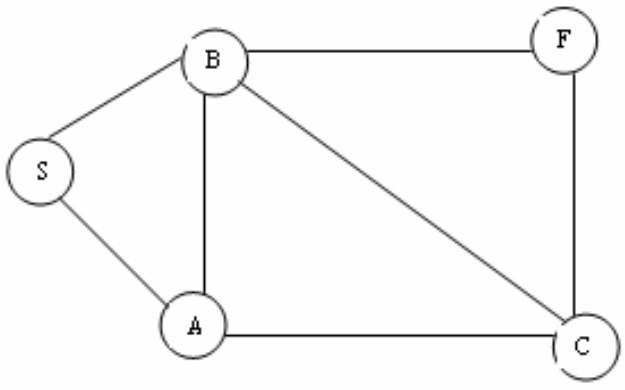
Kısaca, bazı görevleri başarmak için gerekli, iyi tanımlanmış adımlar dizisidir [12].

İnsan, çocukluğundan başlayarak birçok algoritmik yaklaşımı öğrenmekte ve gerçekleştirmektedir. Örneğin okula başlayan bir çocuk, başlangıçta ardışık olarak matematiğin temellerini kazanır. Bir denklemi, integral veya logaritma problemi çözümünde ardışık biçimde uygular [11].

Yapay zeka ile ilgili birçok problemde durum uzayının veya çözüm ağacının gösterilmesinde graf-ağaç yapıları kullanılmaktadır [13].

“Graf” kelimesi ilk kez 1822`de İngiliz matematikçi J.J. Sylvester tarafından kullanılmıştır. Graf, bir noktalar kümesi ile (düğümler) bu noktalar arasındaki ilişkileri ifade eden kenarlar yardımıyla tanımlanan bir yapıdır. Her kenar iki düğümü birleştirmektedir. Grafın her kenarının bir başlangıcı ve bir sonu varsa, bu graf yönlü olarak tanımlanır. Aksi halde graf yönsüz olarak kabul edilir. Yönsüz graflarda kenarlar bağ olarak adlandırılır.

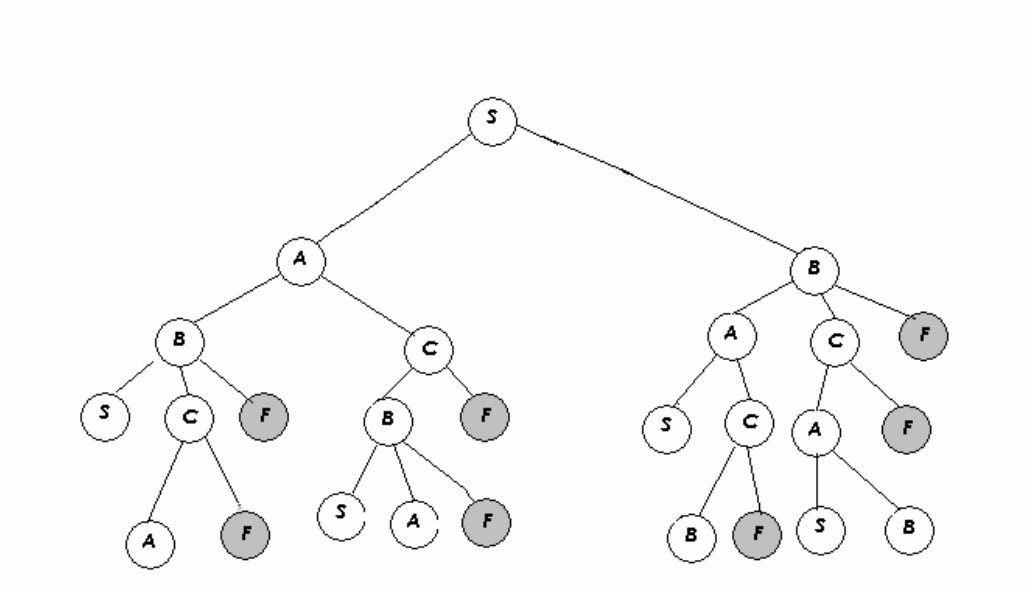
Graf yapıları genellikle bir problemin durum uzayını, nesneler arası ilişkileri (minimum yol, renkleme, vb) ifade etmek için kullanılmaktadır.



Şekil 3.1. Bir graf örneği [11]

Grafların özel biçimi olan ağaçlarda arama algoritmaların nasıl çalıştığı ileride incelenecektir. Ağaç, her çocuk düğüm için yalnız bir ebeveyn düğümü olan graflara denilir [14].

Ağaçlara döngü içermeyen yönlü graflar şeklinde bakılır [11].



Şekil 3.2. Bir ağaç örneği [11]

Bir ağacın en üst düzeyinde yer alan ve ebeveyn düğüme sahip olmayan düğüme ana düğüm (root node) denir. Ortada olan ve çocuk düğümlere sahip olmayan düğümlere de yaprak düğümler (leaf node) denir [15].

3.2. Yapay Zekada Arama

Arama, yapay zeka alanında problem çözmede yaygın olarak kullanılmaktadır. Bundan dolayı arama çoğu zaman "arayarak problem çözme" konusu olarak da literatürde geçmektedir. Problem çözme yapay zekanın önemli bir kısmıdır [16].

Arama algoritmaları tekli etmen (single agent) yapısında olduğundan etmen ve yapısı hakkında bilgi verilecektir.

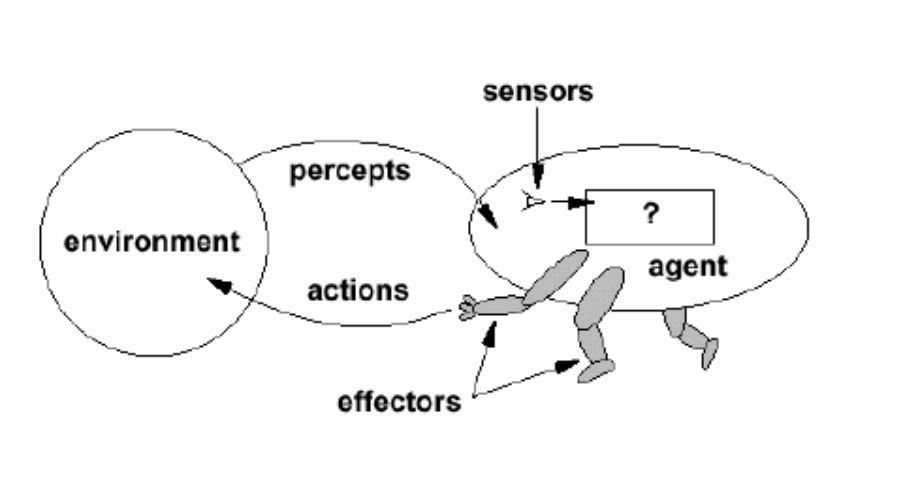
3.3. Arayarak Problem Çözme

Aramanın amacı, en uygun çözümü veya yakın çözümü çabucak bulmaktır [13].

Bu bölümde amaç tabanlı etmenlerin bir türü olan problem çözme etmenlerinden bahsedilecektir. Problem çözme etmenleri istenilen durumlara ulaştıracak hareket sırasını bularak ne yapacağına karar verir. Bir problemi ve bu problemin çözümünü oluşturan parçaları tam olarak anlatılıp ve bu tanımlamaları göstermek için birkaç örnek verilerek konuya başlanılacaktır. Daha sonra bu problemleri çözmek için kullanılan birkaç genel amaçlı arama algoritması anlatılacaktır. Sonraki bölümde anlatılacak olan algoritmalar problem hakkında problemin tanımından daha fazla bilginin verilmediği uninformed algoritmalardır. Daha sonraki bölümde ise çözüm için nereye bakılacağı hakkında bazı fikirleri olan informed arama algoritmaları anlatılmaktadır [4].

3.4. Problem Çözme Etmenleri

Etmen (Agent), “Sensor” leri yardımıyla çevreyi algılayan ve “effector” leri yardımıyla eylemde bulunan parçacıklara verilen isimdir.



Şekil 3.3. Etmen (agent) ın yapısı [4]

Dört önemli elemandan oluşur :

- *Algılamalar(Percepts)* : Etmenin aldığı bilgiler
- *Hareketler (Actions)* : Etmenin yaptıkları
- *Hedefler (Goals)* : Etmenin ulaşmaya çalıştığı – İstenen Durum
- *Çevre (Environment)* : Etmenin hareket ettiği yer, ortam.

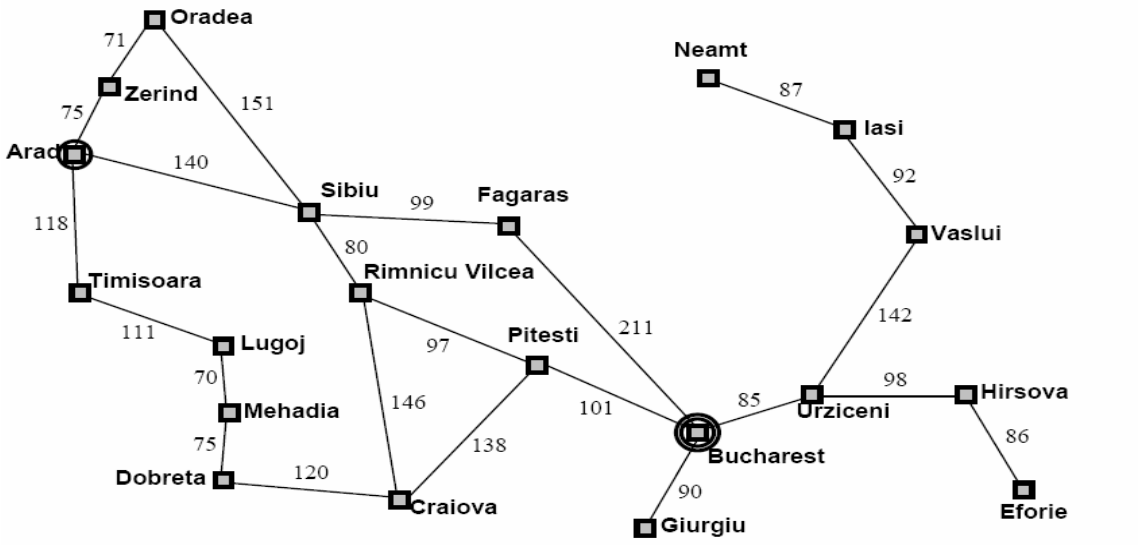
Zeki etmenlerden performans ölçüsünün maksimum olması beklenir. Bunu başarmak eğer etmen onu tatmin edecek yeterli amaç ve gaye edinmiş ise kolaylaşır. Şimdi bir etmen bunu nasıl ve niçin yapar bu incelenecektir [4].

Romanya`nın Arad şehrinde bir etmen düşünülsün ve bu etmen tatil turlarından hoşlanmakta olduğu bilinsin. Bu etmenin performans ölçüsü bazı faktörler içerir: O bronzlaşmak isteyebilir, Rumencesini geliştirmek isteyebilir, turistik yerleri görmek isteyebilir, gece yaşamından hoşlanabilir, içki sersemliğini önlemek isteyebilir ve böylece pek çok. Bu karar verme problemi pek çok konuyu içeren karmaşık bir problemdir. Ve dikkatlice incelenmesi gerekir. Şimdi de etmenin ertesi gün Bucharest`e uçmak için geri iade edilemez bir bileti olduğu varsayılsın. Bu durumda, Bucharest e varma amacını edinmek etmen için mantıklı olur. Bucharest`e zamanında

vardırmayacak hareketler daha fazla göz önünde bulundurulmayarak çıkartılır ve bu etmenin karar verme problemini oldukça basitleştirilir. Hedefler etmenin bu amaca ulaşmak için uğraştığı nesneleri sınırlandırarak davranışı organize etmede yardımcı olur. O anki durumları ve etmenin performans ölçüsünü temel alan Amacı formüle etmek problem (Goal formulation) çözümünde ilk adımdır.

Mevcut günümüz dünya durumlarını içeren bir hedef düşünülsün. Tam olarak hedefteki bu durumlar tatmin edici olacak. Etmenin görevi hedef durumuna ulaştıracak hareket sırasını öğrenmek. Bunu yapmadan önce üzerinde düşünülecek hareket ve durumların sıralamalarının nasıl yapılacağına karar verilmesi gerekmektedir. Eğer "sol ayağı bir inç ileri hareket" veya "direksiyonu bir derece sola çevirme" düzeyinde hareketler düşünülüyorsa, etmenin galiba yolu bulması bir şans dışında mümkün değildir. Çünkü detay düzeyinde dünyadaki belirsizlikler ve de çözüm sürecindeki adımlar oldukça fazladır. Problemi formüle etme (problem formulation) amaca ulaştıracak düşünülen hareket ve durumlar hakkında bir karar verme işlemidir. Bu işlem daha sonra ayrıntısıyla anlatılacaktır. Şimdi etmenin bir şehirden başka bir şehre gitme düzeyinde hareketler düşündüğünü varsayılsın. Düşünülecek durumlar bu yüzden belirli bir şehirde olmaya benzemektedir.

Etmen Bucharest`e ulaşma hedefini benimsemiş ve Arad `dan nereye gideceği üzerinde düşünmektedir. Arad`ın dışına çıkan 3 yol bulunmaktadır, bir yol Sibiu`ya, bir yol Timisoara`ya ve diğeri Zerind`e gitmektedir. Bunların hiç birisi amaca ulaştırmamaktadır, öyleyse etmen Romanya coğrafyasını çok iyi bilmeden hangi yolu izleyeceğine karar veremez. Diğer bir deyişle etmen mümkün olan hareketlerin hangisinin en iyi hareket olduğunu bilemez, çünkü her hareketi yaptıktan sonra meydana çıkan durum hakkında yeteri kadar bilgisi yoktur. Bu durumda yapılabilecek en iyi şey rast gele hareketlerden birini seçmektir.



Şekil 3.4. Romanya haritası [4]

Etmenin kağıdında veya belleğinde Romanya haritası olduğu farz edilsin. Haritadaki bir nokta etmene oluşabilecek durumlar hakkında bilgi sağlamakta ve yapabileceği hareketlerle ilgili bilgi vermektedir. Etmen bu bilgileri nihai olarak Buhcharest`e varacak bir yol bulmak için uğraşarak her üç şehir üzerinden varılacak varsayımlı seyahatin alt sıra durumları düşünmek için kullanabilir. Etmen bir kez Arad`da Bucharest`e ulaşabilecek yolu harita üzerinde tespit eder. Etmen hareketleri yerine getirerek amacına ulaşabilir. Genellikle bilinmeyen değerlerle ilgili pek çok yakın seçenekleri olan bir etmen bilinen değerlerin durumlarına götürecek mümkün olabilecek hareket sıralarını bir kez çalıştırarak ne yapmak istediğine karar verebilir ve en iyi hareketi seçebilir.

Böyle bir sıra arama işlemine *arama (search)* adı verilir. Bir arama algoritması, problemi bir girdi olarak alır ve bir hareket sırası şeklindeki bir çözümü çıktı olarak ortaya koyar. Çözüm bulunur bulunmaz önerilen hareketler yerine getirilir. Buna çalışma (execution) safhası denir. Bu şekilde bir etmen için basit bir “formüle etme (formulation), arama (search), çalıştırma (execution)” tasarımı yapılmış oldu. Bir hedef ve bir problem

formüle edildikten sonra etmen bir arama algoritmasını problemi çözmek için çağırır. Etmen hareketlerine rehberlik yapması için çözümü kullanır, çözüm neyi öneriyorsa örneğin bir sonra yapılacak şey ve sıradan bir hareketin kaldırılması gibi yapar. Çözüm bir kez çalıştırıldığında, etmen yeni amacı formüle edecektir [4].

3.4.1. İyi tanımlanmış problemler ve çözümleri

Problem genellikle 4 parçadan oluşur**;***Başlangıç Durumu, İzleme Fonksiyonu, Hedef Testi, Maliyet*

Etmenin çalışmaya başladığı *Başlangıç durumu* (initial state). Örneğin, Romanya`daki etmen için başlangıç durumu In (Arad) olarak ifade edilmiştir.

Etmenin sahip olduğu mümkün *hareketler* (action) ifadesi. Pek çok *izleme fonksiyonu* (successor function) bir fonksiyon kullanır. Örneğin bir x durumu verilmiş olsun. Bu durumda fonksiyon (action, successor) ikilisini değer olarak geri gönderir. Buradaki her action x durumundaki legal hareketleri ve her successor ise hareketler uygulanarak x den elde edilen bir durumdur. Örneğin, In (Arad) durumundan geçici fonksiyon Romanya problemi için aşağıdaki değerler çiftini döndürür.

{[Go(Sibiu), In(Sibiu)], [Go(Timisoara), In (Timisoara)], [Go(Zerind), In(Zerind)]}

Başlangıç durumu ve geçici fonksiyon beraber problemin *durum uzayını* (state space)- başlangıç durumundan erişilebilir tüm durum setleri- belirler.

Durum uzayı

Verilmiş herhangi bir Yapay zeka problemi izinli eylemleri içeren sonlu durumlar kümesi oluşturmaktadır. Bu kümenin elemanları birbiri ile ilişkili olup (bu ilişki diğer düğümler üzerinden de yapılabilmektedir) graf şeklinde ifade

edilmektedir. Bir problemin tüm izinli durumlarını içeren bu graflara problemin durum uzayı denmektedir [11].

Mümkün tüm çözümlerin uzayına (istenen çözümün aralarından bulunduğu çözümler kümesi) arama uzayı (durum uzayı) adı verilir [17].

Bu uzayı graf biçiminde ifade edilmesinin nedeni durumların düğümlere, izinli gidişlerin ise bağlara karşı düşmesidir. Problemin başlangıç ve erişilmesi gereken hedef durumları verilmezse bu graflar yönsüz graflar olmaktadır. Fakat çözümün hangi başlangıç durumdan başlayarak hedefe erişilmesi isteniyorsa graflar yönlü olmaktadırlar. Problemin çözümü ise bu graflarda minimum yolun aranmasına karşılık gelmektedir [11].

Durum uzayı bir graf oluşturur. Bu grafta düğümler durumları, düğümler arasında yer alan oklar ise hareketleri gösterir. (Bkz. Şekil 3.4) Romanya haritası bir durum uzayı grafiği olarak yorumlanabilir. Durum uzayındaki bir *yol* (path) bir hareketler sırasına bağlı olan bir durumlar dizisidir.

Hedef testi, verilen bir durumun amaç durum olup olmadığını test eden bir fonksiyondur [18].

Bazen mümkün amaç durumların dizisi açıkça var olabilir. Bu durumda hedef testi, verilen durumun bunlardan biri olup olmadığını kontrol eder. Etmenin amacı In (Bucharet) dir. Bazı durumlarda hedef açıkça durumlar dizisini sayarak değil de soyut bir özellik tarafından belirtilmiştir. Örneğin, satranç oyununda hedef rakibin kralının kaçamayacak bir şekilde yakalamak olan şah mat olarak adlandırılan duruma ulaşmaktır.

Yol maliyeti (path cost) fonksiyonu her yola bir sayısal maliyet tahsis eder. Problem çözme etmeni kendi performansını ölçen bir maliyet fonksiyonunu seçer. Etmen Bucharest e varmak için çalışırken zaman burada önemli bir ölçü esasıdır. Böylece yol maliyeti kilometre olarak uzunluğa bağlıdır. Bu

konuda yolun maliyetinin yol boyunca kişisel hareketlerin maliyetlerinin toplamı olduğunu varsayılmaktadır. X durumundan Y durumuna gitmek için yapılan A hareketin *adım maliyeti* (step cost) c(x,a,y) olarak gösterilmektedir. Romanya için adım maliyetleri yol mesafeleri olarak gösterilmiştir. Bu adım maliyetlerinin negatif olmadıkları kabul edilmektedir.

Parçalardan önce problem tanımlanabilir ve bir problem çözüm algoritmasını girdi olarak veren tek bir veri yapısı içerisinde bütün hepsi bir araya getirilebilir. Bir problem için *çözüm* (solution) başlangıç durumundan hedef durumuna ulaştıracak bir yoldur. Çözüm kalitesi yol maliyeti fonksiyonu tarafından ölçülebilir. En uygun çözüm yolu tüm çözüm boyunca en düşük yol maliyetine sahip çözümdür [4].

3.5. Problemleri Formüle Etme

Önceki konuda başlangıç durumu, geçici fonksiyon, hedef testi ve yol maliyeti terimlerinde Bucharest`e ulaştıracak bir problem formülasyonu önerildi. Bu formulasyon makul gözükmektedir, henüz gerçek dünya ile ilgili pek çok durumu ihmal etmektedir. Seçilmiş olan In (Arad) basit durum ifadesi içerisinde seyahat rehberi, radyoda ne olduğu, pencere dışındaki manzara, bir sonraki mola yerinin ne kadar uzakta olduğu, yol ve hava durumu ve bunun gibi benzer pek çok faktörlerini barındıran gerçek bir ülke seyahatine benzetilmiştir. Tüm bu faktörler durum ifadesinin dışında bırakılmıştır. Çünkü etmeni Bucharest`e ulaştıracak bir yol bulma problemiyle bunların bir ilgisi yoktur. Bir betimlemeden detayların atılması işlemine *çıkartım* (abstraction) denir.

Durum ifadelerinden çıkartım yapmaya ilave olarak, hareketlerin de kendi kendilerden çıkartılması gerekir. Araba sürme hareketinin pek çok etkileri, sonuçları vardır. Bunlar, aracın yerinin ve oturanların konumunun değişmesi, zaman alma, yakıtın tükenmesi, kirlilik oluşturma gibi. Formülasyonda yalnızca konumdaki değişiklik dikkate alınmıştır. Aynı zamanda, göz önüne

alınmayan pek çok harekette olacaktır. Bunlar; radyonun açık olup olmaması, pencereden dışarı bakılıp bakılmaması, trafik memurları için yavaşlayıp yavaşlamak gibi. Ve elbette "direksiyonu üç derece sola döndürme" seviyesinde hareketlerde belirtilmeyecektir.

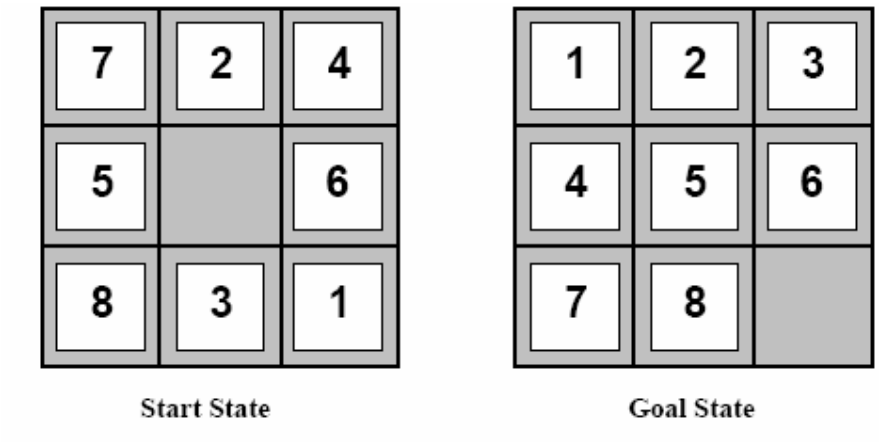
Uygun çıkartım düzeyinin tanımlamak için daha fazla hassas olunabilir mi ? Ayrıntılı geniş dünya durumları ve hareket sıraları dizisine karşılık olarak seçilecek çıkartım durumları ve hareketleri düşünülsün. Çıkartım problemi için bir çözüm tasarlansın. Örneğin, Arad`dan Sibiu`ya oradan Rimnic Vilcea`ya oradan da Pitesti ve oradan da Bucharest`e bir yol düşünülsün. Bu çıkartım çözümü daha ayrıntılı yollar sayılarına bağlıdır. Örneğin, Sibiu ve Rimnicu arasında radyo açık seyahat edip daha sonra yolculuğun geri kalanında radyo kapatıldı. Eğer daha ayrıntılı dünyada bir çözüm içerisinde her hangi bir çıkartım çözümü genişletebiliyorsa çıkartım geçerlidir. Çıkartım eğer çözümdeki her hareketin uygulanmasını orijinal problemden daha çok kolaylaştırsa kullanışlıdır. Bu durumda ortalama bir araba sürme etmeni tarafından daha fazla arama ve planlama yapılmaksızın uygulanabilir olmasıyla hareketler daha kolaydır [4].

Örnek problemler:

Problem çözme yaklaşımı çok büyük miktardaki iş çevrelerinde uygulanmıştır. Şimdi seçilmiş oyuncak (toy) ve gerçek dünya problemleri arasında en iyi bilinen bazı problem çözme yaklaşımları listelenecektir. Oyuncak problem çeşitli problem çözme metotlarını göstermek ve çalıştırmak için tasarlanmıştır. Bunun anlamı algoritmaların performanslarını karşılaştırmak için farklı araştırmacılar tarafından kolayca kullanılabilir olmasıdır. Gerçek dünya problemi ise insanların bazı şeyler hakkında gerçekten taşıdıkları kaygıların çözümlerinden biridir. Burada tek bir ifade yoktur. Fakat burada çözümlerin formüle edilmesinde genel niteliklere ulaşılmaya çalışılacaktır.

Oyuncak (Toy) Problemler

Şekil 3.5 `te gösterilen, sekiz tane numaralandırılmış taş ve bir boş alanın olduğu 3X3 lık bir oyun tahtası olan 8- puzzle örneği ele alınsın. Boş alanın yanında bulunan bir taş boşluğa doğru kaydırılabilir. Hedef, şeklin sağ tarafındaki gibi gösterilen belirtilmiş hedef duruma ulaşmaktır. Standart formülasyon aşağıdaki gibidir:



Şekil 3.5. 8 puzzle oyuncak problemi

Durumlar (States): Durum ifadesi sekiz taştan her birinin ve dokuz kareden birinde bulunan boş alanın yerini belirtir.

Başlangıç durumu (Initial State): Herhangi bir durum başlangıç durumu olarak tasarlanabilir. Verilen herhangi bir amaca muhtemel başlangıç durumlarının yarısından tamamen ulaşılabilir.

İzleyici fonksiyonu (Successor Function): Dört hareketin (boşluğu Sola, Sağa, Yukarıya ve Aşağıya hareket etmesi) yapılması sonucunda meydana çıkacak olan legal durumları oluşturur.

Hedef Testi (Goal Test): Durumun Şekil 3.5 de gösterilen hedef şekline uyup uymadığını kontrol eder. (Başka hedef şekilleri de mümkündür.)

Yol Maliyeti (Path cost): Her adımın maliyeti 1 dir. Böylece yol maliyeti yoldaki adımların sayısına eşittir.

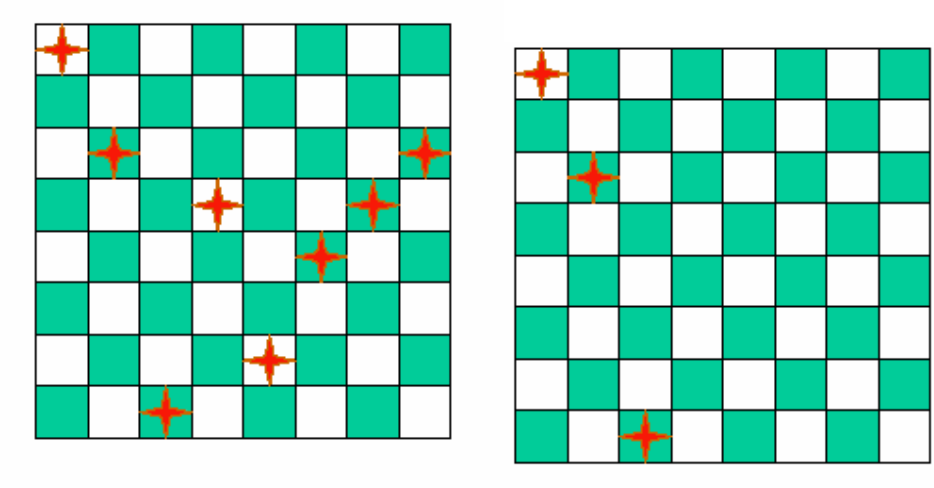
Buraya dahil edilebilecek çıkartımlar nelerdir? Hareketlerin bloğun kaydığı ortadaki yerleri atlanarak başlangıç ve son durumları çıkartılır. Burada tüm fiziksel etkileri önleyerek sadece puzzle kurallarına dikkat edilecektir. 8-puzzle AI de yeni arama algoritmaların testi için oldukça sık kullanılan kaydırma-block puzzle (sliding-block puzzles) ailesinin bir üyesidir.

8 puzzle örneğinde 9!/2=181 440 ulaşılabilir durum mevcut olup oldukça kolay bir şekilde çözülür. Yani her iki permütasyonda bir çözümlü durum mevcuttur [19].

15-puzzle örneği ise (4 X 4lük bir oyun tahtası) yaklaşık 1,3 trilyon olasılık mevcuttur ve rasgele seçilen örnekler en iyi arama algoritmaları tarafından birkaç mili saniyede uygun bir şekilde çözülebilmektedir. 24- puzzle (5 x 5 lik tahta) yaklaşık olarak 10^{25} olasılık mevcuttur ve böyle bir rasgele örneği mevcut makineler ve algoritmalarla uygun bir şekilde çözülmesi hala oldukça zordur.

8 vezir problemi

8 vezir probleminin amacı sekiz veziri satranç tahtasına biri diğerine hamlede bulunmayacak şekilde yerleştirmektir. (Bir vezir aynı satırda, sütunda veya köşegensel yerde bulunan her hangi bir taşa hamle yapabilir.) Başarısız bir deneme girişiminde vezir başka yerde olan vezire hamle yapabilir.



Şekil 3.6. Sekiz vezir problemi

Bu problem ve tüm n-vezir tipli problemleri çözmek için etkili özel amaçlı algoritmalar mevcut olmasına rağmen arama algoritmaları için ilginç bir test problemi olmaya devam etmektedir. İki ana tür formülasyon mevcuttur. Artan formülasyon (incremental formulation) durum ifadelerini boş bir durumdan başlayarak arttırmaktadır; 8 vezir probleminde bunun anlamı, her hareketin bir duruma bir vezir eklemesidir. Tam durum formülasyonda (complete-state formulation) satranç tahtasındaki 8 vezirin hepsiyle başlar ve onları etrafta hareket ettirir. Her iki durumda da yol maliyete gerek duyulmamaktadır. Çünkü yalnızca son durum sayılmaktadır. Birinci artan formülasyon aşağıdaki gibi çalışır.

Durumlar: Tahta üzerindeki 0 dan 8 e kadar vezirlerin her hangi yerleşimi bir durumdur.

Başlangıç durumu: Tahta üzerinde hiçbir vezir bulunmamaktadır.

İzleyici fonksiyon: herhangi boş bir kareye bir vezir koymak

Hedef testi: sekiz vezirin hepsi tahta üzerinde ve birbirlerine hamle yapamayacak düzende olması.

Bu formülasyonda, araştırma yapmak için yaklaşık olarak $64x63x……57 \approx 3x10^{14}$ mümkün kareye vardır. Daha iyi bir formülasyon, hamle yapılabilecek her hangi bir kareye veziri yerleştirmeyi yasaklayacaktır.

Durumlar: n vezirlerin ($0 \leq n \leq 8$) en soldaki n sütunların her bir sütununa vezirin diğerlerine hamle yapamayacak şekilde yerleşmeleri durumları oluşturur.

İzleyici fonksiyon: veziri diğer başka bir vezir tarafından hamle yapılamayacak olan en sol boş sütundaki herhangi bir kareye koyar.

Bu formülasyon 8 vezir durum uzayını $3x10^{14}$ ten tam 2 057 indirir. Ve çözümleri bulmayı oldukça kolaylaştırır. Diğer taraftan 100 vezir için geliştirilmiş formülasyon yaklaşık olarak 10^{52} durum mevcut iken başlangıç formülasyonu yaklaşık olarak 10^{400} duruma sahiptir. Bu büyük bir indirgemedir. Fakat bu bölümdeki algoritmalar için geliştirilmiş durum uzayı kullanılması için hala oldukça büyüktür [4].

3.6. Problem Çözme Performansının Ölçülmesi

Problem çözme algoritmasının çıktısı ya başarılıdır yada başarısızdır. Algoritmanın performansını değerlendirmek için 4 kriter kullanılmaktadır. Bunlar;

Tamlık(Completeness): Algoritmanın bir çözüm bulmayı garanti edip etmiyor mu? Eğer bir çözüm varsa hedef durumu bulmayı garanti ediyorsa bu algoritma tamdır denebilir. Tam olmayan algoritmaların çözüm bulamadığı ile ilgili raporlarına tam inanılmaz [16].

Optimalite: Strateji en uygun çözümü buldu mu? Algoritma mevcut çözümler içerisinde en uygun olanı garantiliyorsa bu algoritma optimaldir denebilir. Diğer bir deyişle en az adım sayısı ile hedef duruma ulaşan yol bulunması gerekir [16].

Zaman Karmaşası (Time complexity) : Bir çözümü bulmak ne kadar zaman aldı? Bir algoritmanın zaman karmaşıklığı çözüm durumunu buluncaya kadar geçen süreyle ilişkilidir. Algoritmaların karmaşıklığını ifade etmek için Büyük-O işareti kullanılmaktadır. Örneğin BFS algoritmasının zaman karmaşası $O(b^d)$ b ağaç dallanma faktörü ve d ise ağaçtaki hedef düğümün derinliğini gösterir [16].

Bellek durumu karmaşası (Space complexity): Aramayı gerçekleştirmek için gerekli bellek miktarı ne kadardır?

Zaman ve Bellek karmaşası arama algoritmalarını anlama konusunda önemli bir özelliktir. Bir arama algoritması küçük test problemleri için etkili olabilirken daha büyük problemlerde kabul edilemeyecek şekilde zaman karmaşıklığı yüksek olabilir. Çok hızlı arama algoritmaları daima en iyi çözümü bulamayabilir [16].

4. ARAMA ALGORİTMALARI

Arama algoritmaları iki ana başlıkta toplanabilir

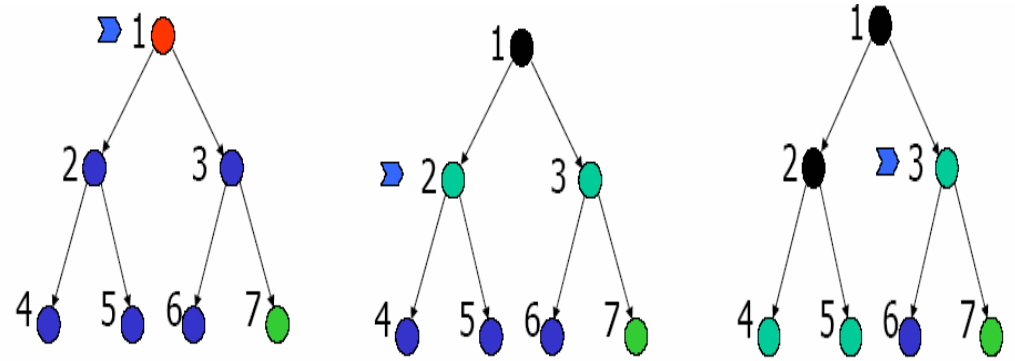
- Uninformed Arama (Blind-Kör)-Bilgiye Dayanmayan Arama
- Informed Arama (Heuristic-Sezgizel) - Bilgiye Dayalı Arama

4.1. Uninformed Arama (Blind-Kör)- Bilgiye Dayanmayan Arama

Diğer bir adı *kör (blind)* aramadır. Bunun anlamı problemin tanımında durumlar hakkında ek bir bilgi kullanmamasıdır. Kısaca durum bilgisinden yararlanmaz.

4.1.1. Breadth-first search (önce genişliğine arama)

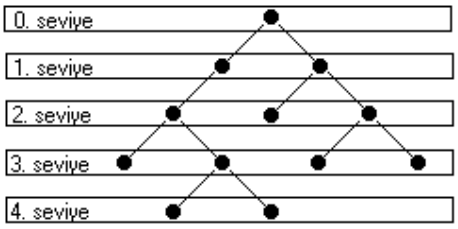
Başlangıç durumuna bağlı olarak seviye seviye tüm düğümleri açar ve kontrol eder. Bu arama algoritmasında kural “ilk giren ilk çıkar” (first-in/first-out) dır [18].



Şekil 4.1. Önce genişliğine arama algoritması çalışması

- Düğümleri, kuyruğun sonuna ekler.
- Düzeydeki tüm düğümler,
- (i+1).Düzeydeki tüm düğümlerden önce açılır. BFS, hedefe ulaştıran en kısa yolu bulmayı garantiler.

Bu arama metodu ile seviye seviye tarama yapılır [20].



Şekil 4.2. Önce genişliğine aramanın tarama gösterimi

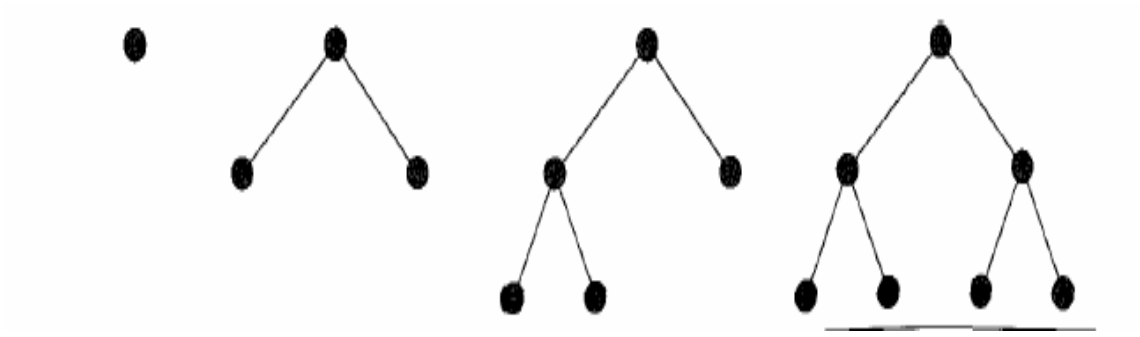
1. düğümü(node) işaretle
2. düğümü kuyruğa(queue) ekle
3. kuyruktan ilk elemanı al
4. elemanın ziyaret edilmeyen komşularını (çocuklarını) kuyruğa at
5. sıra boş değilse 3'e dön

BFS başlangıç düğümden başlayarak düğümleri hedef duruma olan uzaklıklarına göre sırasıyla kontrol eder [21].

BFS başlangıç düğüme en yakın olan düğüm ilk önce açar. Ve bir sonraki seviye geçmeden önce o seviyedeki tüm düğümleri ziyaret eder [22].

BFS bir düğümün mümkün alt durumları 1. derinlikte incelendikten sonra diğer durumlara geçilmektedir [11].

Eğer bir çözüm var ise bu yöntem çözümün bulunmasını garanti eder (tamlık). Birden fazla çözüm var ise en sığ olanı yani derinliği en az olanı verir (optimallik). Aşağıdaki şekilde önce genişlik aramalı ağacın gelişimi görülmektedir.



Şekil 4.3. Önce genişlik aramada 0, 1, 2 ve 3 düğümlerinin açılması [4]

Yöntemin zaman ve bellek karmaşıklığını incelemek için her durumun b yeni duruma açıldığını kabul edilsin. Yani dallanma faktörü (branching factor) *b* olsun. Arama ağacının kökü ilk seviyede b adet düğüm üretir, bu düğümlerin her biri de b tane yeni düğüm ürettiği varsayılsın. Bu durumda, ikinci seviyede b^2 , üçüncü seviyede b^3, .., düğüm üretilir. Yol uzunluğu veya derinliği d olan bir problemin çözümünü bulmadan önce açılacak düğüm sayısı aşağıdaki şekilde yazılabilir:

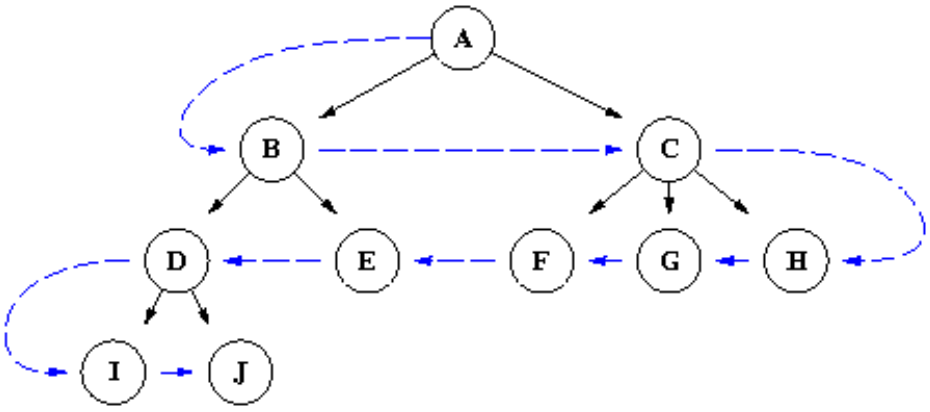
$$1 + b + b^2 + b^3 + ... + b^d$$

Böyle bir problemin algoritma karmaşıklığı $O(b^d)$ şeklindedir. (Bu algoritmalara örnek olarak haberleşme ve ses tanımada yaygın kullanılan Viterbi algoritması gösterilebilir.) Önce genişlik aramada dallanma faktörü b=10 için değişik derinliklerde zaman ve bellek karmaşıklığı aşağıdaki tabloda görülmektedir [11].

Çizelge 4.1. Önce genişlik aramada dallanma faktörü *b*=10 için değişik derinliklerde zaman ve bellek karmaşıklığı [4].

Derinlik	Düğümler	Zaman		Bellek	
0	1	1	msn	100	byte
2	111	0.1	sn	11	Kbyte
4	11111	11	sn	1	Mbyte
6	10^6	18	dk.	111	Mbyte
8	10^8	31	saat	11	Gbyte
12	10^{12}	35	yıl	111	Terabyte
14	10^{14}	3500	yıl	11111	Terabyte

Çizelgede dallanma faktörü b=10, 1000 düğüm/sn, 100 byte/düğüm alınmıştır.



Şekil 4.4. Önce genişliğine aramanın düğümleri arama şekli [23].

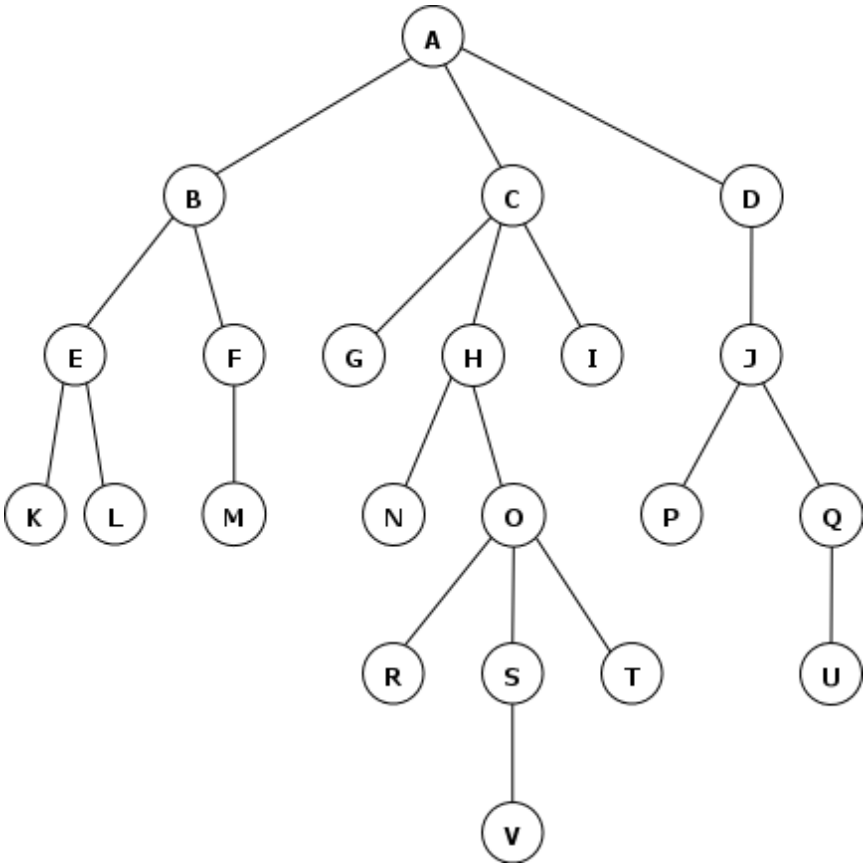
Bu yöntemde açılan tüm düğümler bellekte saklandığı için bellek karmaşıklığı zaman karmaşıklığı gibidir. Çizelgeye bakıldığında zaman, genişliğine arama yönteminde bellek gereksinimi icra süresinden daha büyük bir problem ortaya çıkarmaktadır. 6. Seviyedeki sonucu bulmak için 18 dakika beklenebilir ama 111 Mbyte RAM biraz zor bulunur.

Zaman gereksinimi de önemli bir faktördür. 14. Derinlikteki bir çözüm için 3500 yıl beklemek gerekiyor. 10 yıl sonra 100 kat daha hızlı bir bilgisayar alınırsa sonuç 35 yılda bulanabilir [4].

BFS algoritması, bulmaca ve oyunlar gibi yapay zeka problemlerine uygulanabilmektedir [11].

Örnek

Şekil 4.5 deki arama ağacına göre BFS algoritmasının nasıl çalıştığı açıklanmak istenirse;



Şekil 4.5. Önce genişliğine arama uygulaması için örnek arama ağacı

BFS algoritması düğümleri aşağıdaki gibi bir kuyruk oluşturarak açar ve inceler.

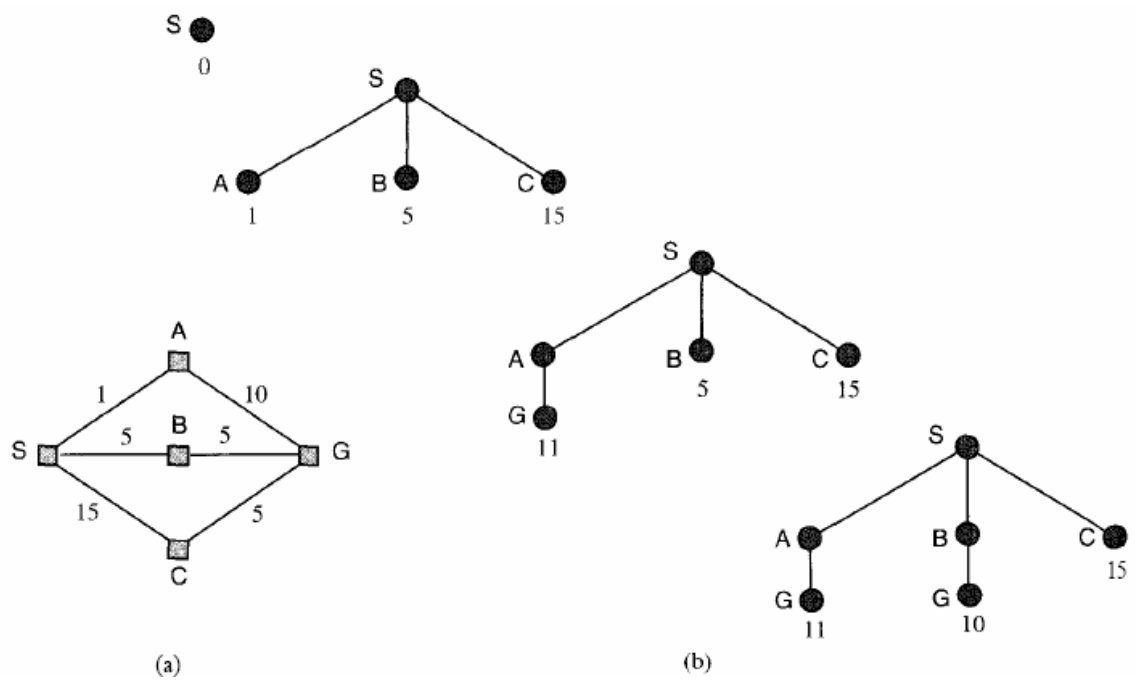
A
B C D
C D E F
D E F G H I
E F G H I J
F G H I J K L
G H I J K L M
H I J K L M
I J K L M N O
J K L M N O
K L M N O P Q
L M N O P Q
M N O P Q
N O P Q
O P Q
P Q R S T
Q R S T
R S T U
S T U
T U V
U V
V

Burada durumları oluşturma ve genişleme sırası ABCDEFGHIJKLMNOPQRSTUV şeklindedir. Eğer hedef düğümler M,V ve J düğümleri ise BFS algoritması en az derinliğe sahip olan J`yi bulacaktır.

4.1.2. Uniform-cost first

Genişliğine aramada en sığ hedef durum elde edilir. Ama çözüm en düşük maliyetli yol olmayabilir. Uniform maliyetli aramada en düşük derinlikteki düğüm yerine açılmayı bekleyen düğümlerden en düşük maliyetli olanı açılır. Maliyet fonksiyonu g(n) ve derinlik Derinlik(n) olarak tanımlanırsa g(n)=Derinlik(n) için Uniform maliyetli arama genişliğine aramaya eşit olur.

Eğer daha az maliyetli yollar varsa bu yollar daha önce açıldığı için ilk bulunan çözüm en düşük maliyetli çözüm olur. Aşağıdaki şekilde verilen S (Başlangıç- Start)'den G(Hedef-Goal) 'ye rota bulma problemini göz önüne alınsın:



Şekil 4.6. Uniform-cost first ile rota bulma problemi. a) her işlemin maliyetini gösteren durum uzayı, b) aramanın ilerleyişi.

Bu yöntemde; başlangıç durumu açılarak A, B ve C yolları sağlanır. En ucuz A olduğu için A açılarak SAG çözümü elde edilir. Maliyeti 11 olduğu için algoritma bunu çözüm olarak kabul etmez. Bir sonraki adımda SB açılarak SBG çözümü bulunur. Ağaçta daha düşük maliyetli bir yol kalmadığı için SBG problemin çözümü olarak geriye verilir. Yol maliyeti yol arttıkça azalmıyor ise Uniform maliyetli arama en ucuz çözümü verir [4].

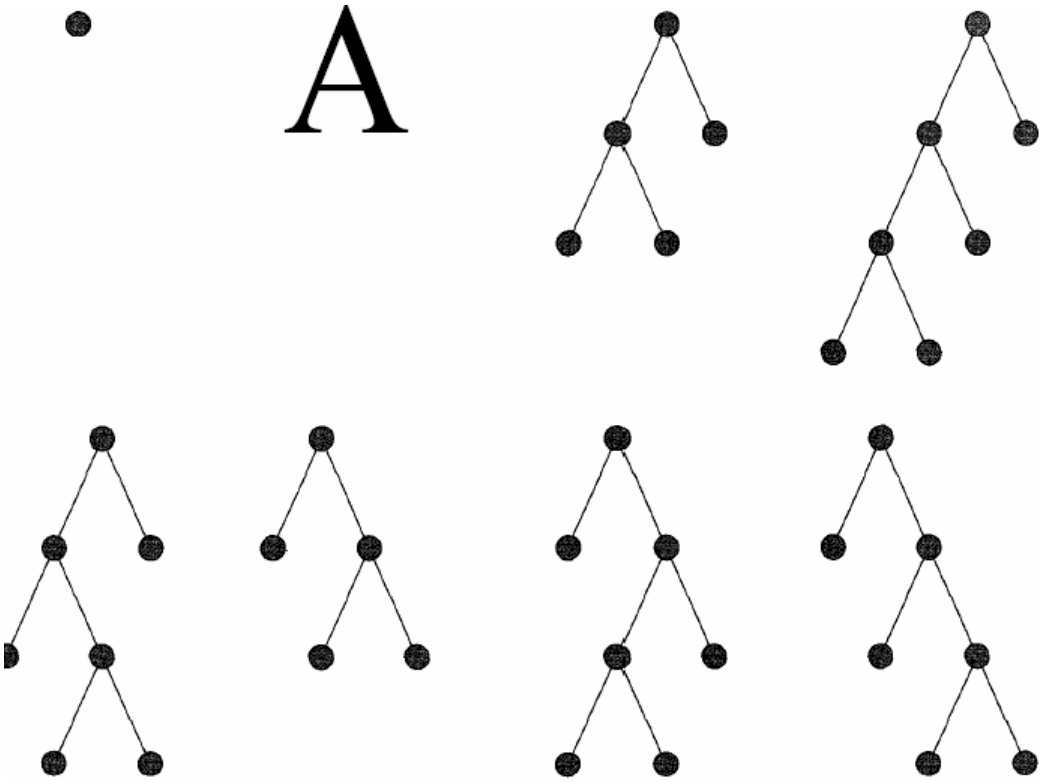
4.1.3. Depth-first search (derinlik öncelikli arama)

BFS algoritmasının tam tersidir [24].

DFS`de en son açılan düğüm takip edilerek yeni durumlar oluşturulur [11].

Derinlik öncelikli aramada daima ağacın en derin düğümlerinden biri açılır. Potansiyel çözümünün çok derinlerde olmadığı durumlarda yaygın olarak kullanılmaktadır [22].

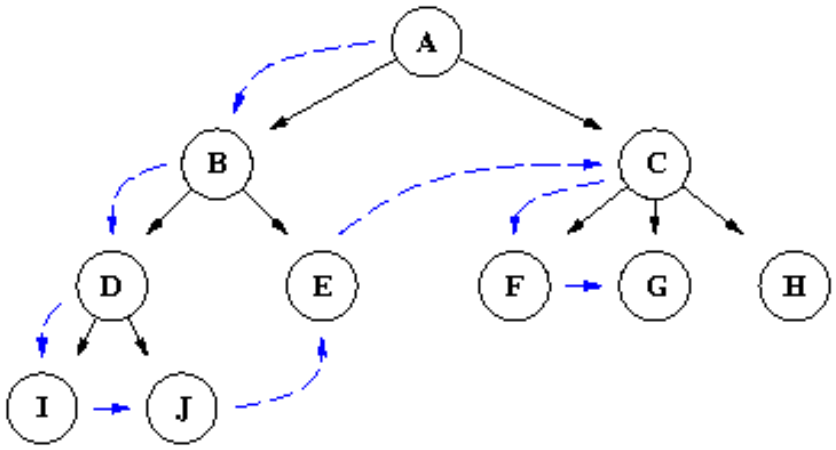
DFS aramada eğer amaçlanmayan düğüme erişilmiş ise veya açılacak düğüm kalmamış ise açma işlemine daha sığ seviyelerden devam edilir. Derinlik öncelikli aramanın ilerleyişi aşağıdaki şekilde görülmektedir.



Şekil 4.7. İkili arama ağacı için derinlik öncelikli arama ağaçları

DFS algoritması “son giren ilk çıkar” (Last-in-first-out) mantığına dayanmaktadır [18].

Derinliğine aramada kökten yaprağa kadar yalnız bir yol depolandığı için bellek gereksinimi azdır. Dallanma faktörü b ve maksimum derinliği m olan durum uzayında derinliğine aramada yalnız bm düğümün saklanması gerekir. Genişliğine aramada ise bu b^d idi. Burada, d en sığ çözümün bulunduğu derinliktir. 12. Derinlik için genişliğine aramada 111 terabyte gerekirken derinliğine aramada 12 kilobyte yeterlidir. 10 milyarda biri kadar bellek!

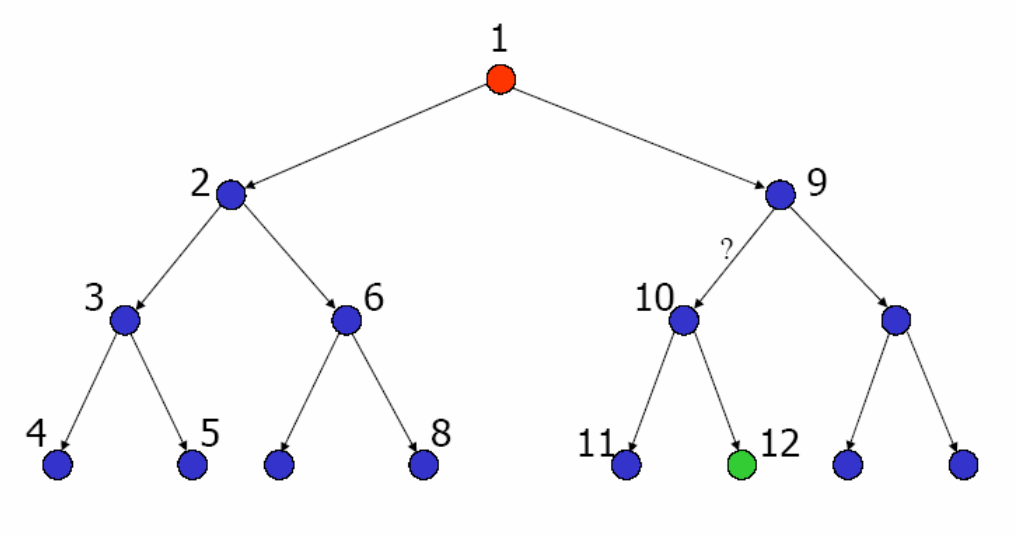


Şekil 4.8. Arama ağacı üzerinden derinlik öncelikli aramanın çalışması [23].

Anlaşılacağı gibi DFS algoritması özyineleme özelliği taşımaktadır. Bu ise onun mantıksal diller tarafından programlanmasını kolaylaştırmaktadır. Derinliğine aramanın zaman karmaşıklığı $O(b^m)$ dir [11].

Birden fazla çözümü olan problemlerde derinliğine arama genişliğine aramadan daha iyi sonuç verir. Sonsuz döngüye girme, fazla derine gitme veya daha sığ çözümleri bulamama gibi problemler nedeniyle derinliğine arama tam veya optimal değildir.

- DFS, birçok çözüm varsa BFS'den etkindir. Kuyruk kullanılmasına gerek yoktur. DFS, tam değildir.?
- Çözüm derinliğinin büyük olduğunda kullanılmamalıdır.



Şekil 4.9. Derinlik öncelikli aramanın en derin düğümü bulması

Örnek

Şekil 4.5 deki arama ağacına göre DFS algoritmasının nasıl çalıştığı açıklanmak istenirse;

DFS algoritması düğümleri aşağıdaki gibi bir kuyruk oluşturarak açar ve inceler.

A
B C D
E F C D
K L F C D
L F C D
F C D
M C D
C D
G H I D
H I D
N O I D
O I D
R S T I D
S T I D
V T I D
T I D

I D
D
J
P Q
Q
U

Eğer hedef düğümler M,V ve J düğümleri ise DFS algoritması M`yi bulacaktır.

4.1.4. Depth limited (derinlik sınırlı arama)

k derinliğinde kesilen (cutoff) DFS'dir.
k, altındaki düğümlerin açılamayacağı maksimum derinliktir.

Üç olası sonuç vardır :

- Çözüme ulaşılır (Solution)
- Çözüme ulaşılamaz (Failure)
- Cutoff (derinlik sınırları içinde sonuca ulaşılamaz)

Derinliğine aramada karşılaşılan problemi gidermek için yolun maksimum derinliği belirtilerek derinlik sınırlı arama yapılabilir. Örneğin 20 şehrin bulunduğu haritada bir şehirden diğerine gitmek için yapılacak bir aramada maksimum derinlik 19 olarak verilebilir. Yeni işlem formu "eğer A'da iseniz ve seyahat ettiğiniz şehir sayısı 19'dan küçük ise yol uzunluğu bir fazla olan B şehrinde yeni bir durum oluştur" şeklinde ifade edilebilir. Bu yeni işlem kümesiyle çözüm var ise bulunması garanti edilmiş olur. Ama en kısa olanın bulunduğu garanti edilemez. Sonuç olarak derinlik sınırlı arama tamdır ama optimal değildir. Eğer derinlik az seçilirse tamlık da tehlikeye girer. Derinliğine aramada olduğu gibi derinlik sınırlamalı aramada da zaman karmaşıklığı $O(b^l)$, bellek karmaşıklığı O(bl)'dir. Burada l derinlik sınırıdır.

Çizelge 4.2. DLS aramanın özellikleri

Zaman Karmaşası:	*O*(| *V* | + | *E* |)
Durum Uzayı Karmaşası:	
Optimal:	hayır
Tam	hayır

DLS arama algoritması

```
DLS(node, goal, depth)
{
  if (node == goal)
    return node;
  else
  {
    stack := expand (node)
    while (stack is not empty)
    {
      node' := pop(stack);
      if (node'.depth() < depth)
        DLS(node', goal, depth);
      else
        ; // no operation
    }
  }
}
```

4.1.5. Iterative deepening search (yineli derinleştirmeli arama)

DFS ve BFS yöntemlerinin iyi yönlerini birleştirir.

Durum uzayı çok büyük sayıda düğümler içeren problemlerde gereksiz işlemleri durdurmak için belirli bir derinlik sınırlaması yapılmaktadır. Yinelemeli derinine arama algoritması adı verilen bu yöntemde belirli bir derinliğe ulaşılana kadar önce derinine, sonrada genişlemesine olarak bütün düğümler incelenir. Sonra ise önceden belirlenmiş bir değere kadar derinlik tekrar büyütülerek diğer düğümlerde çözüm aranır [11].

Derinlik sınırlı aramanın en zor kısmı uygun derinliğin tespitidir. Örneğin harita probleminde derinlik sınırı 19 olarak alınmıştı ama harita incelendiğinde bir şehirden diğerine en fazla 9 adımda gidilebildiği görülmektedir. Bu sayı durum uzayının çapı olarak bilinir ve daha iyi derinlik sınırı verir. Ama bir çok problemde problem çözülmeden iyi derinlik sınırının ne olduğu bilinemez [4].

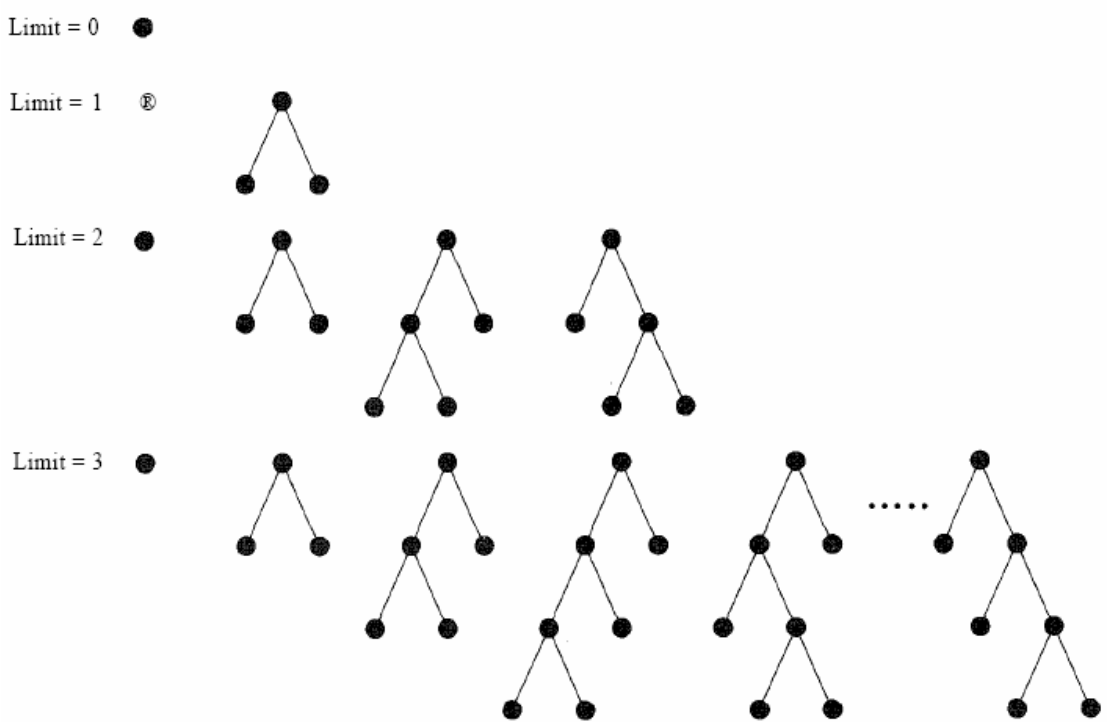
Yineli derinleştirmeli arama algoritması.

Function ITERATIVE- DEEPENING- SEARCH(problem) returns a solution, or failure

inputs: *problem*, a problem

for *depth* ← 0 to ∞ do
result ← DEPTH-LIMITED-SEARCH *(problem, depth)*
if result ≠ cutoff then return *result*

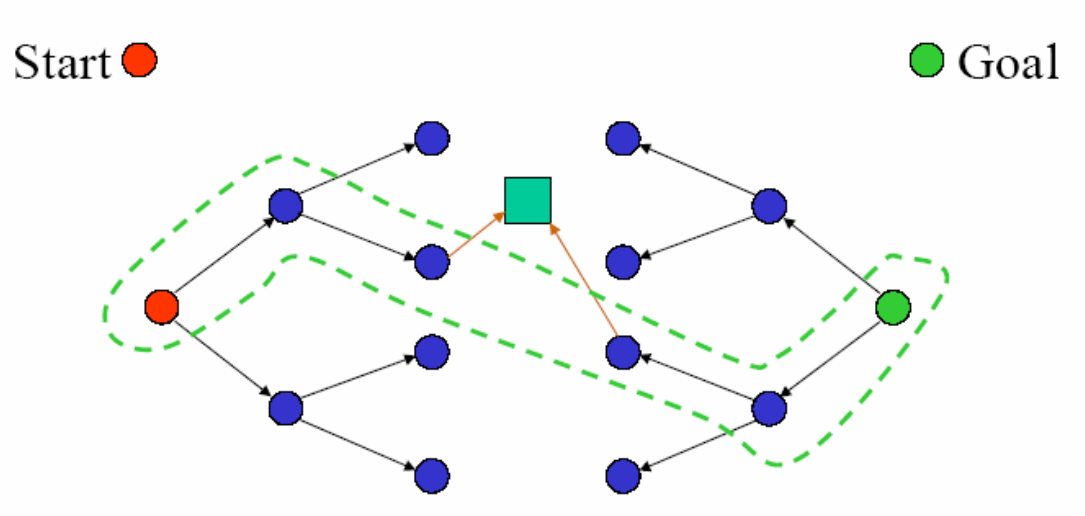
İteratif derinleşmede en iyi derinliğin seçimi bir kenara koyularak olası tüm derinlik sınırları denenir: önce 1, sonra 2, 3 .. şeklinde. Aslında iteratif derinleşme, derinliğine arama ve genişliğine aramanın faydalarını birleştirir. Genişliğine arama gibi optimal ve tamdır, derinliğine arama gibi az bellek gerektirir.



Şekil 4.10. Dört iterasyonlu yineli derinleştirmeli arama

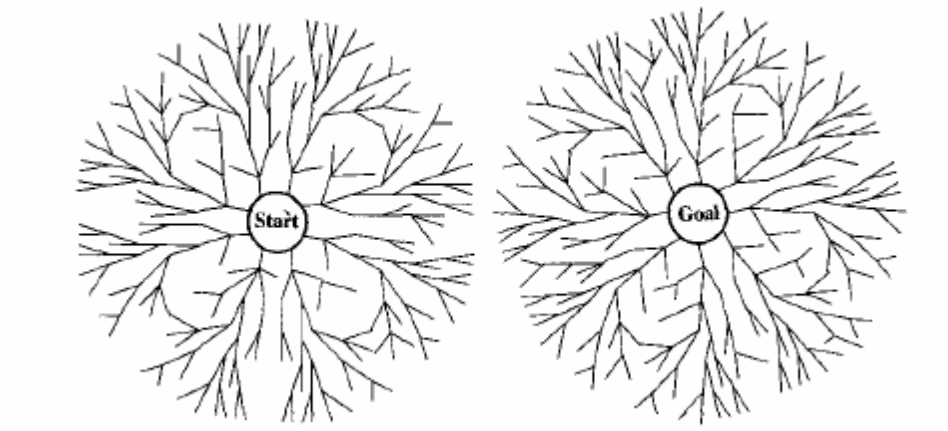
4.1.6. Bidirectional (bf) search (çift yönlü arama)

Arama işlemine eş zamanlı olarak biri başlangıç durumundan ileriye diğeri hedef durumundan geriye doğru aynı anda başlanır. Ve bu iki arama ortada karşılaştığı zaman işlem biter. Dallanma faktörü her iki yönde de b ise iki yönlü aramada karmaşıklık açısından büyük fark olur. Aşağıdaki şekilde Hareket b^d den daha az $b^{d/2} + b^{d/2}$ olur. İki küçük dairenin alanı başlangıcı merkezi ve amaca erişme ortada olan büyük bir dairenin alanından daha azdır.



Şekil 4.11. Arama ağacında çift yönlü arama

Çift yönlü arama her düğümü çift arama yaparak düğümü açmadan önce kontrol edilmesiyle uygulanmaktadır. Örneğin, bir problem çözüm derinliği d=6 ise ve her iki yöne breadth – first araması uygulanması durumunda en uygun durumda iki arama bütün düğümler açılmış ve derinliğin d=3 olduğu durumda karşılaşacaktır. Bunun anlamı b=10 durumunda 22 200 düğüm oluşturulmuşken standart tek yönlü breadth –first aramasında bu rakam 11 111 100 olmaktadır. Diğer arama ağacındaki üyelik için bir düğümün kontrolü bir hash tablo sayesinde sabit bir zamanda yapılmaktadır. Böylece zaman karmaşıklığı $O(b^{d/2})$ olmaktadır. En azından arama ağaçlarından biri hafızada saklanmaktadır. Böylece üyelik kontrolü yapılabilmektedir. Durum uzay karmaşıklığı ise $O(b^{d/2})$ olmaktadır. Bu durum uzayı ihtiyacı çift yönlü aramanın en önemli zayıflığıdır [4].



Şekil 4.12. Çift yönlü aramanın şematik bir gösterimi

4.1.7. Uninformed aramaların karşılaştırılması

- BFS, tamdır, optimaldir ancak yer karmaşıklığı yüksektir.
- DFS, yer karmaşıklığı etkindir, ancak tam da değildir, optimal de değildir.
- Yineli derinleştirme, asimptotik olarak optimaldir.

Problem uzayının içerdiği durumlar sayısı çok büyük olduğunda bu algoritmaların zayıf kaldıkları görülmektedir. Bu nedenle genişlemesine ve derinliğine aramaya körüne arama algoritmaları olarak tanımlanmaktadır. Çünkü söz konusu algoritmalar çözümün konumundan bağımsız biçimde çalışmaktadır. Bu açıdan sezgisel algoritmaların uygulanması yapay zeka problemlerinde büyük önem taşımaktadır [11].

Çizelge 4.3. Uninformed aramaların karşılaştırılması

Kriter	Genişlik öncelikli (Breadth)	Uniform Maliyet	Derinlik Öncelikli (Depth-First)	Derinlik Sınırlı (Depth – Limited)	İteratif Derinlik	Çift Yönlü (Bidirectional)
Zaman	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Bellek	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal	Evet	Evet	Hayır	Hayır	Evet	Evet
Tam	Evet	Evet	Hayır	evet	Evet	Evet

4.2. Informed(Heuristic-sezgizel)-Bilgiye Dayalı Arama

Yapay zeka kitaplarının yaklaşık hepsinde sezgisellik kelimesi ile karşılaşmak mümkündür. Sözlüklerde sezgisellik “gerçeğin deneye veya akla vurmadan, doğrudan doğruya kavranması” olarak geçmektedir [11].

Yapay zekada kullanılan “sezgisellik” sözü “heuristik” kelimesinin Türkçe eşdeğer anlamı olarak kullanılmaktadır. Bu kelime ise “Eureka” (Buldum!) kelimesinden gelmektedir [11].

Artık bir teoremi ispatlamak için tek bir reçete verilememektedir. Benzeri problemlerin çözümü için belirli özelliklerin ve benzerliklerin bulunması gerekmektedir. Çözüme ilişkin bir anahtar bulma işlemi bir sezgiselliktir. Çözüm aşamasında kullanılan sınırlamalar ise sezgisel kuralları oluşturur.

Yapay zekada problemlerin çözümünde genellikle sezgisel yöntemler kullanılmaktadır. Literatürlerde bu açıdan bir boşluk bulunmakta ve sezgisel algoritmaların her zaman çalışmadığı savunulmaktadır. Gerçekten de bir problem için geçersiz olan sezgisel yaklaşım, diğerinde başarılı sonuçlar verebilir.

Sezgisellik Fagenbaum ve Fieldman tarafından “problemin durum uzayı çok büyük olduğunda çözümün aranması kesin biçimde sınırlayan herhangi kural, strateji, hile, sadeleştirme ve etmenlerin kullanımıdır” şeklinde tanımlanmıştır.

Bilgili arama (informed search) veya sezgisel arama durum uzayı hakkındaki bilgileri kullanarak aramanın karanlıkta rasgele dolaşma şeklinde yapılmasını engeller. Bilgili arama yöntemleri çözüm bulmada bilgiye dayanmayan arama yöntemlerine göre daha etkili bulunurlar. Sezgisel aramada hedef en düşük maliyetle çözümü bulmaktır. Çünkü çözüm için yapılan aramalar neticesinde problem durumu üssel olarak büyümektedir [25].

4.2.1. Best-first /Greedy search (en iyi öncelikli arama)

En iyi öncelikli aramanın en basit şeklinden birisi amaca erişmek için tahmini maliyetin minimize edilmesidir. Yani, hedef duruma en yakın olduğu sanılan düğüm öncelikle açılır. Herhangi bir durumdan hedef duruma erişmenin maliyeti tahmin edilebilir ama kesin olarak saptanamaz. Maliyet tahminini hesaplayan fonksiyona sezgisel (heuristic) fonksiyon denir ve genellikle h ile belirtilir:

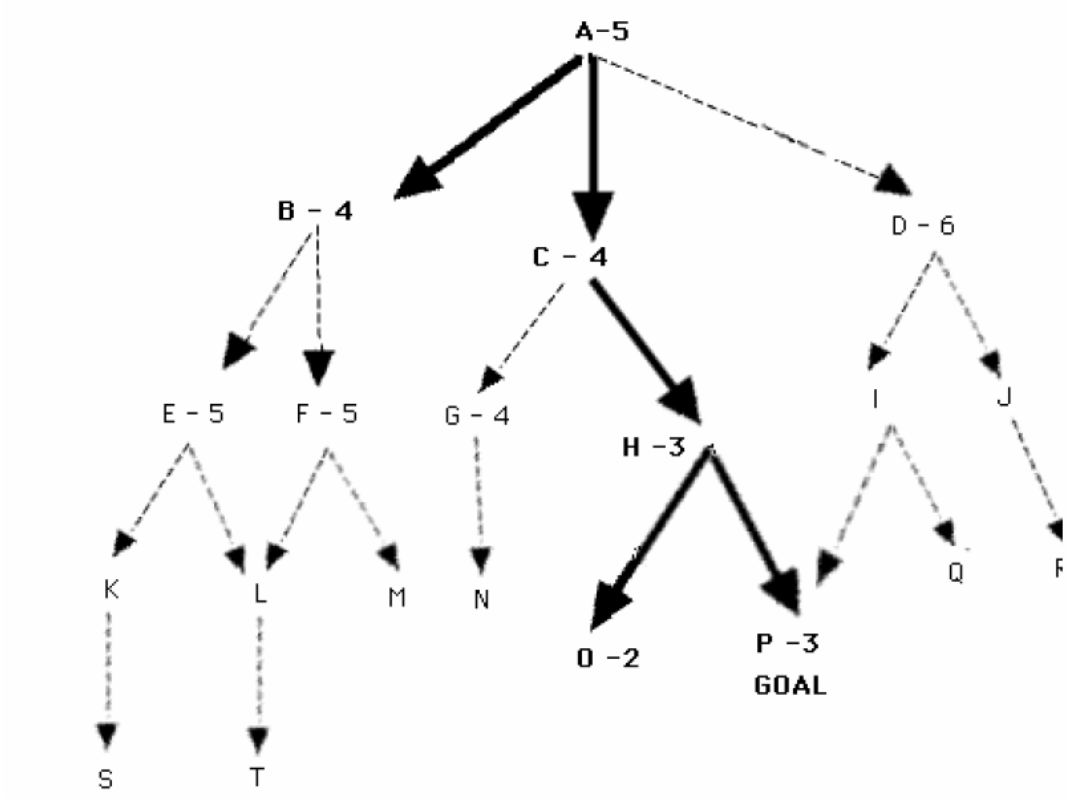
h(n) = n düğümündeki durumdan hedef duruma en ucuz yolun tahmini maliyeti.

Açılacak düğümü h'i kullanarak tespit eden en iyi öncelikli aramaya greedy arama adı verilir. H herhangi bir fonksiyon olabilir. n hedef ise h(n)=0 olması gerekir.

Hangi düğümün öncelikle açılacağını saptamada kullanılabilecek bilgi değerlendirme fonksiyonu (evaluation function) ile elde edilebilir. Düğümlerin değerlendirme fonksiyonu kullanarak sıralanması ve açma işlemine en iyi değeri veren düğümden başlanmasına en iyi öncelikli arama (best-first

search) yöntemi denir. En iyi olan değerlendirme fonksiyonuna göre tespit edilmektedir.

Değerlendirme fonksiyonu hatalı sonuçlar da verebilir. Aslında bu arama en iyi sanılanın öncelikle açılmasıdır. Amaç düşük maliyetli çözümler bulmak olduğu için bu algoritmalar maliyet için tahmini bir ölçü kullanarak onu minimize etmeye çalışır. Bu yöntemde iki yaklaşım kullanılmaktadır. Birincisi, amaca en yakın düğümü açmaya çalışır. İkincisi ise, en düşük maliyetli çözüm yolunda düğümü açmaya çalışır [4].



Şekil 4.13. Best first /greedy search çalışma şekli

1. AÇ = {A5}; KAPATILMIŞ = { }
2. evaluate A5; AÇ = {B4, C4, D6}; KAPATILMIŞ = {A5}
3. evaluate B4; AÇ = {C4, E5, F5, D6}; KAPATILMIŞ = {B4, A5}
4. evaluate C4; AÇ = {H3, G4, E5, F5, D6}; KAPATILMIŞ = {C4, B4, A5}

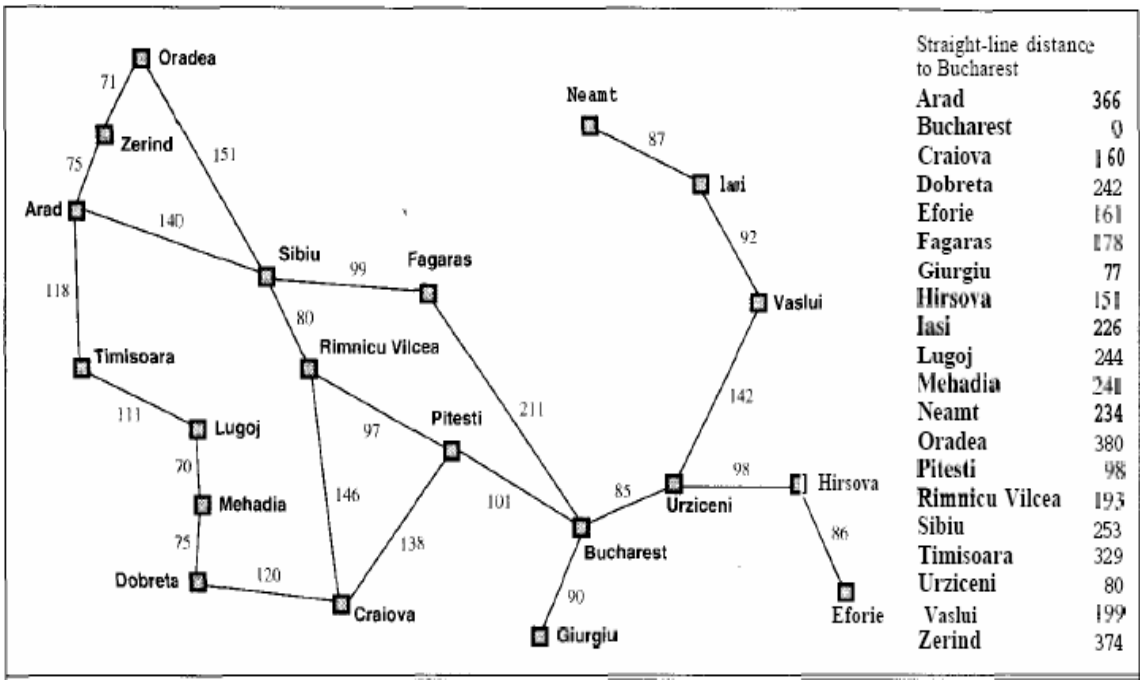
5. evaluate H3; AÇ = {O2, P3, G4, E5, F5, D6}; KAPATILMIŞ = {H3, C4, B4, A5}
6. evaluate O2; AÇ = {P3,G4,E5, F5, D6}; KAPATILMIŞ = {O2, H3, C4, B4, A5}
7. evaluate P3; çözüm bulundu [26].

Herhangi bir anda, sezgiye göre en umut verici düğüm açılır. Aşağı yukarı uniform-cost search'ün tersidir.

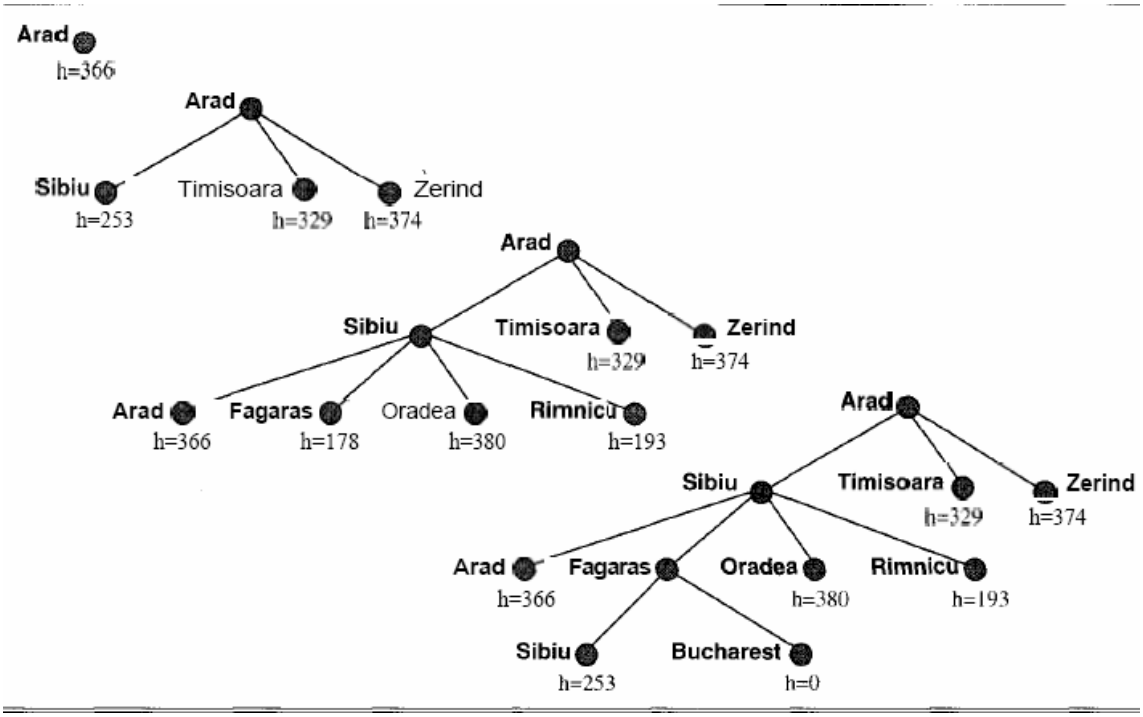
- DFS'ye benzer, ancak sezgi içerir.
- Sezgi, her zaman 0 ise, BFS'ye döner.
- Greedy Yöntemleri, uzun vadeli sonuçları önemsemeden, kısa vadeli avantajları en iyi duruma getirir.

Rota bulma problemleri için iki nokta arasındaki direkt mesafe (straight-line distance, SLD) iyi bir sezgisel fonksiyon olarak kullanılabilir. Yani $h_{SLD}(n) = n$ ile hedef yer arasındaki direkt mesafe.

Eğer şehirlerin koordinatları biliniyor ise hSLD değerleri hesaplanabilir. Bu şekilde ek bilgiler arama maliyetini azaltır. Daha önce verilmiş olan haritada şehirler arası mesafe ve bu şehirlerle B arasındaki direkt mesafe, h, aşağıdaki şekilde görülmektedir.



Şekil 4.14. Romanya haritası ve B (Bükreş) şehrine olan uzaklıkları



Şekil 4.15. B şehri için best-first /greedy aramanın aşamaları

Direkt mesafe fonksiyonu ile A'da açılacak ilk düğüm S olacaktır. Çünkü S B'ye Z veya T'den daha yakındır. Bir sonraki açılacak düğüm ise F olacaktır çünkü B'ye en yakın olan F'dir. F hedef durum olan B'yi üretir. Bu problem için sezgisel fonksiyon en az arama maliyetini sağlar. Çözüm yolunda olmayan düğümlerin hiç biri açılmamıştır. Ama tam olarak optimal değildir. Çünkü A□S□R□P□B yolu 32 kilometre daha kısadır. Bu yöntem amaca ulaşmak için en büyük adımı atmaya çalışır. Uzun vadeyi düşünmez.

I'dan F'ye gitme problemini göz önüne alınsın. Best-first/ Greedy arama önce N'yi açacaktır ama bu çıkmaz yoldur. Çözüm önce V'ye gitmektir. Bu durumda gereksiz bir düğüm açılmış oldu. Eğer tekrarlanan durumlara dikkat edilmez ise arama N ve I arasında salınım yapar [4].

Best-first/ Greedy arama, derinliğine arama yöntemi gibi çalışır. Çıkmaz bir yolla karşılaşınca geri döner. Derinliğine aramaya benzer şekilde ne tam ne de optimaldir.

- Tam değildir
- Optimal değildir.
- Zaman ve Yer Karmaşıklıkları kötüdür. $O(b^m)$ m durum uzayınının maksimum derinliğidir (üstel)

4.2.2. A* search (Toplam yol maliyetinin azaltılması)

Sezgisel aramada en iyi bilinen ve kullanılan algoritmalardan biri A*`dir [27].

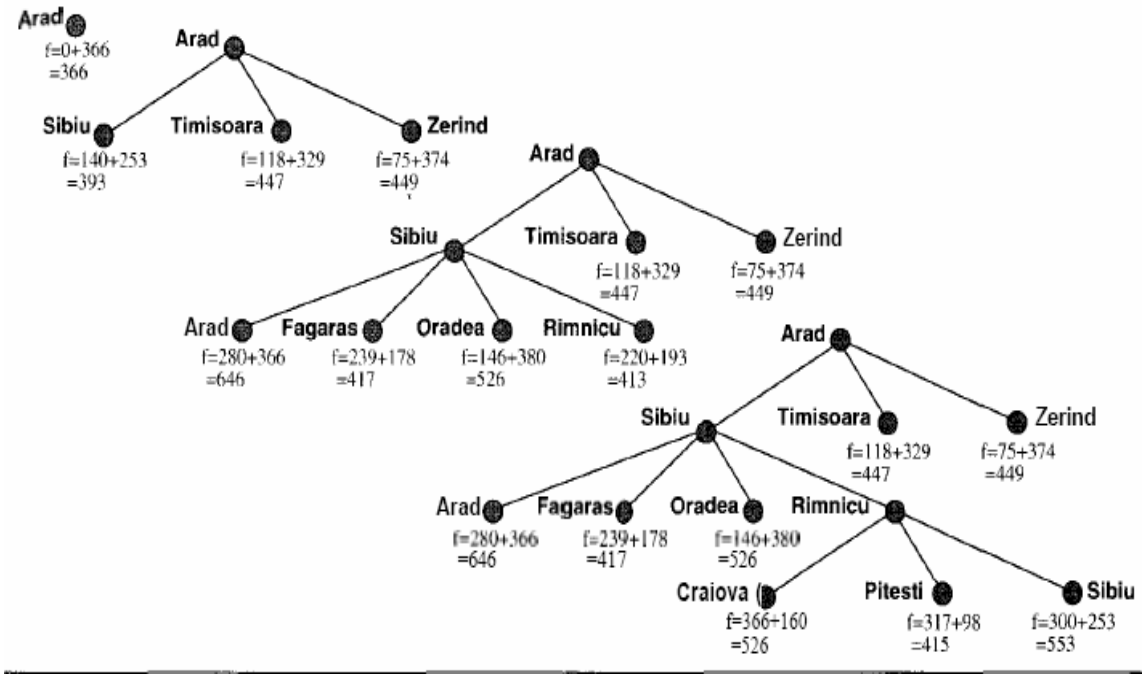
A* algoritması yol bulma-planlama, robot hareket planlama için yaygın olarak kullanılmaktadır [28].

Best-first/Greedy arama amaca erişmek için tahmini maliyeti minimize eder ve böylece arama maliyetini azaltır. Diğer yandan tam ve optimal olan

uniform arama o ana kadar olan yolun maliyetini, g(n), azaltır. İki yöntemin birleştirilmesi ile daha iyi arama yapılabilir. Bu işlem iki değerlendirme fonksiyonunu toplayarak yapılabilir:

f(n) = g(n) + h(n) ;n `den geçen en ucuz çözümün tahmini maliyeti

En ucuz çözümü bulmak için en düşük f değerli düğümü öncelikle denemek gerekir. h fonksiyonuna bir sınırlama getirerek yöntemin tam ve optimal olması sağlanabilir. Sınırlama, h fonksiyonun amaca erişmenin maliyetini asla fazla tahmin etmemesidir. Böyle h, kabul edilebilir sezgisel (admissible heuristic) olarak adlandırılır. Kabul edilebilir sezgisel fonksiyonu maliyeti gerçekte olduğundan az verdiği için iyimserdir. Bu iyimserlik f fonksiyonu için de geçerlidir. Eğer h kabul edilebilir ise f(n) n'den geçen en iyi çözümün gerçek maliyetini asla fazla tahmin etmez. En iyi öncelikli aramada f fonksiyonu değerlendirme fonksiyonu olarak kullanılıyorsa bu arama A* arama olarak isimlendirilir [4].



Şekil 4.16. B şehri için f maliyet hesaplamaları ve A* aramanın aşamaları

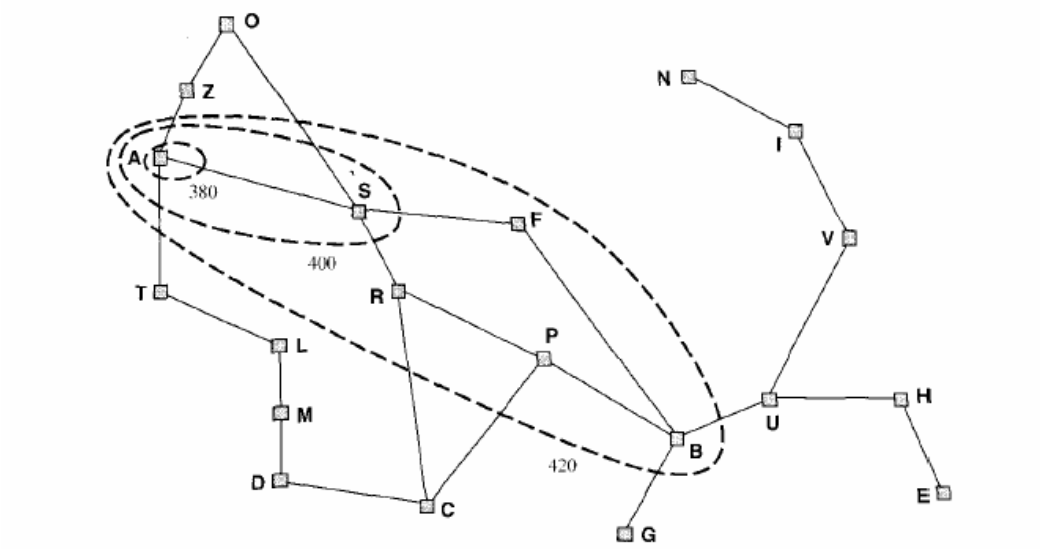
Verilen arama ağacına bakıldığı zaman, kökten başlayan herhangi bir yol boyunca f maliyeti asla azalmamaktadır. Burada sezgisel fonksiyon monotonik (monotonocity) özelliktedir. h, üçgen eşitsizliğine uyuyor ise monotoniktir. Örnekte direkt mesafe üçgen eşitsizliğine uyduğu için monotoniktir.

Eğer h monotonik değil ise küçük bir düzeltme ile monotik hale getirilebilir. n düğümü n' düğümünün ebeveyni olsun. g(n)=3, ve h(n)=4 olduğu kabul edilirse f(n)= 7 olur. Yani n'den geçen çözümün maliyeti en az 7'dir. g(n')=4 ve h(n')=2 olduğu kabul edilirse f(n')=6 olur. Bu örnek monotonik değildir. Ama n' düğümünden geçen yol aynı zamanda n düğümünden de geçmesi gerektiği için 6 anlamsızdır. Çünkü gerçek maliyetin en az 7 olduğu bilinmektedir. Bu nedenle yeni bir düğüm üretildiği zaman f-maliyetinin ebeveynin maliyetinden düşük olup olmadığı kontrol edilir eğer düşük ise ebeveynin maliyeti kullanılır:

f(n') = max(f(n), g(n')+h(n')).

Bu eşitlik *maksimum yol (pathmax***)** eşitliği olarak isimlendirilir. Bu eşitlik kullanılır ise kökten başlayarak her yolda f devamlı artar.

A'dan B'ye gitmek için A* arama. f=g+h ve h= A'ya direkt mesafe. Kökten başlayan herhangi bir yol boyunca f asla azalmıyor ise durum uzayında eş uzaklık eğrileri (contour) çizilebilir. Aşağıdaki şekilde 400 ile etiketlenen eğrinin içindeki tüm düğümlerde f(n) 400 den küçük veya eşittir.



Şekil 4.17. A başlangıç durumundan f=380, f=400 ve f=420 eğrileri.

A* arama en düşük f değerli yaprak düğümü açtığı için düğümler bir merkez etrafında bant şeklinde açılır. Uniform maliyetli aramada h=0 olduğu için bantlar başlangıç durumu etrafında dairesel olacaktır.

- Best-first Search'ün en büyük dezavantajı, uzak görüşlü olmamasıdır.
- g : şu anki duruma kadar maliyet (c) fonksiyonu
- h : uygun sezgi fonksiyonu olsun.
- Uygun olması, iyimser olması yani hedefe götüren maliyeti hiçbir zaman gerçek değerinden daha fazla tahmin edilememesi demektir.
- Best-first/ greedy search, yerine f=g+h kullanılarak yapıldığında, A*'a ulaşılır.

A Aramasının Özellikleri*

- Tamdır. Sonuçta, çözüm varsa bulunur. Kökten başlanarak gidilen herhangi bir yolda f tahmini her zaman artar (h, monoton olsa bile). Zaman karmaşıklığı Optimaldir.
- Zaman ve Yer karmaşıklıkları kötü olsa da iyi bir sezgi ile, düzelecektir.

- A* algoritması en uygun çözümü en düşük sayıda adımla bulmayı garanti etmektedir [29].

4.2.3. Bellek sınırlı arama (memory-bounded)

Zeki arama algoritmalarının bulunmasına karşın bazı problemlerin çözülmesi güçtür. Karşılaşılan güçlüklerden birisi bellek gereksinimidir. Kullanılabilecek bellek büyüklüğünü göz önüne alarak problemleri çözmeye çalışan üç algoritma bu kısımda incelenecektir; IDA* , RBFS, ve SMA*.

4.2.3.1. İteratif derinleşen A* arama (IDA*)

A* arama algoritması bellek gereksimi tarafından sınırlanmıştır. A* algoritma açtığı tüm düğümleri hafızada saklamaktadır [30].

Daha önceki bölümde iteratif derinleşme algoritmasının bellek gereksinimini azaltan yararlı bir algoritma olduğundan bahsedilmişti. Benzer şekilde, A* arama iteratif A* arama (IDA*) şekline dönüştürülebilir. Bu algoritmada her bir iterasyon derinliğine aramadır. Derinliğine aramada derinlik sınırı yerine f-maliyet sınırı kullanılır. Verilen eş uzaklık eğrileri içindeki arama tamamlanınca bir sonraki eğriye geçilir [4].

IDA* ın zaman karmaşıklığı bulma fonksiyonunun alacağı farklı değerlere bağlıdır. 8-bulmacasında, manhattan uzaklığı bulma fonksiyonu olarak alındığında, f herhangi bir çözüm yolunda 2 veya üç kez artar. Böylece IDA* 2-3 iterasyona girer. Birçok pratik problemin optimal çözümü ilk olarak IDA* ile bulunmuştur. IDA* birkaç yıl, tek optimal bellek-sınırlı bulma algoritması olarak kalmıştır.

IDA* karmaşık problemlerde iyi çözüm vermemektedir. Örneğin TSP probleminde bulma değeri her durum için farklıdır. Yani her eğri bir önceki eğriden yalnız bir fazla durum içermektedir. Eğer A* N düğüm açıyorsa IDA*

N iterasyon yapmak zorunda kalacaktır: 1+2+...+N=O(N^2). N bellek için büyük ise N^2 çok çok büyük olacaktır [4].

İteratif derinleşen A* arama (IDA*) algoritması

function IDA* (*problem*) bir çözüm sırası döndürür
inputs: problem, bir problem
local variables: *f-limit*, *o anki f-maliyet limiti,*
root, bir düğüm

root <-MAKE-NODE(INITIAL-STATE[*problem*])
f-limit ← f-COST (*root*)
loop do
solution,f-limit ← DFS-CONTOUR(root,f-limit)
if *solution* is non-null then return *solution*
if *f-limit* = ∞ then return failure; end

---

function DFS -CONTOUR (*node, f-limit*) bir çözüm sırası ve yeni bir f-maliyet limiti döndürür
inputs: node, *bir düğüm*
f-limit, *o anki f-maliyet limiti*

local variables: *next-f* , *bir sonraki eş uzaklık eğrileri için f-maliyet limiti, başlangıç olarak* ∞

if *f-COST* [node] > f-limit then return null, f-COST [*node*]
if GOAL-TEST [*problem*] (STATE[node]} then return node, f-limit
for each node *s* in *SUCCESSOR (node)* do
solution, new-f ← DFS-CONTOUR (*s, f-limit*)
if solution is non-null then return *solution, f-limit*
next-f ← MIN (*next-f,new-f*); end
return null, next-*f*

Bu durumu aşmak için f maliyet limiti her iterasyonda ε kadar artırılarak toplam iterasyon sayısı 1/ε ile orantılı hale getirilebilir. Bu arama maliyetini düşürür ama bulunan çözüm optimal çözümden ε kadar kötü olabilir. Bu algoritmalara kabul edilebilir ε (ε-admissible) algoritmalar denir.

IDA* (Iterative-Deepening A*)`in genel özellikleri

- DFS'ye dayanır. Bir düğümün çocuklarının açılma önceliğini belirlemede f değerinden yararlanır.
- f değeri sınırı vardır (derinlik limiti yerine)
- IDA*, A* ile aynı özellikleri taşır ama daha az bellek kullanır.
- Oldukça az bellek kullanır.
- IDA tam ve optimaldir.(A arama gibi aynı ikazlarla)

4.2.3.2. RBFS(Recursive best-first search)

Standart best –first arama işlemlerine uygulanan basit bir özyineleme algoritmasıdır. Fakat yalnızca lineer uzayını(space) kullanır. Yapı olarak özyinelemeli depth-first aramaya benzemektedir. Tanımlanmamış bir yola devam etmektense en iyi alternatif yol için f değerini kullanır. Eğer bulunan düğüm bu limiti aşarsa, özyineleme alternatif yol için tekrar çalışır. Bu şekilde, RBFS her bir düğümün ve onun alt çocuklarının f değerlerini yeniler. Bu şekilde, RBFS unutulmuş alt dallardaki en iyi yaprağın f değerini hatırlar ve böylece bir sonraki zamanda yeniden açılacak alt dalların kıymetine karar verir [4].

RBFS(Recursive best-first search) algoritması

function RECURSIVE-BEST-FIRST-SEARCH (*problem*) *çözüm döndürür veya başarısız olur*

    RBFS (problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS (*problem, node, f_limit*) *bir çözüm döndürür veya başarısız olur ve yeni f-maliyet limiti döndürür*

    if GOAL-TEST [*problem*](STATE[*node*]) then return *node*
    successors ← EXPAND (*node, problem*)
    if *successors* is empty then return failure, ∞
    for each s in successors do
        f[s] ← max(g(s) + *h*(s), *f*[*node*])
    repeat

best ← *izleyicideki en düşük f-değerli düğüm*
if f[best] > f_limit then return *failure*, f[*best*]
alternative ← *izleyicideki ikinci en düşük f-değerli düğüm*
result, f[*best*] ← RBFS (*problem*, *best*, min (*f_limit, alternative*))
if result ≠ failure then return *result*

RBFS bazı yerlerde IDA* dan daha etkilidir. Fakat hale aşırı düğüm yeniden üretiminde sıkıntı yaşamaktadır.

A* da olduğu gibi RBFS de eğer bulma fonksiyonu h(n) kabul edilebilirse optimal bir algoritmadır. Onun uzay karmaşası O (bd) dir. Hem IDA hem de RBFS de arama grafına bağlı olarak bu durum üssel bir şekilde artmaktadır. Çünkü her ikisi de tekrar eden durumları kontrol etmemekte bu da pek çok kez aynı durumlarla karşılaşılmasına yol açmaktadır.

IDA* VE RBFS oldukça az bellek kullanmaktan bazı sorunlar yaşamaktadırlar. İtarasyonlar arasında IDA* yalnızca tek bir sayı aklında tutar: o anki f-maliyet sınırı. RBFS ise bellekte daha fazla bilgi tutar, fakat yalnızca O(bd) bellek miktarını kullanır. Daha fazla bellek olsa bile RBFS bunu kullanacak bir tekniği yoktur.

En doğrusu mevcut belleğin tamamının kullanılmasıdır. Bunu yapan iki algoritma bulunmaktadır. Bunlar MA*(Memory-bounded A*) ve SMA* (Simplified MA*). Burada SMA anlatılacaktır [4].

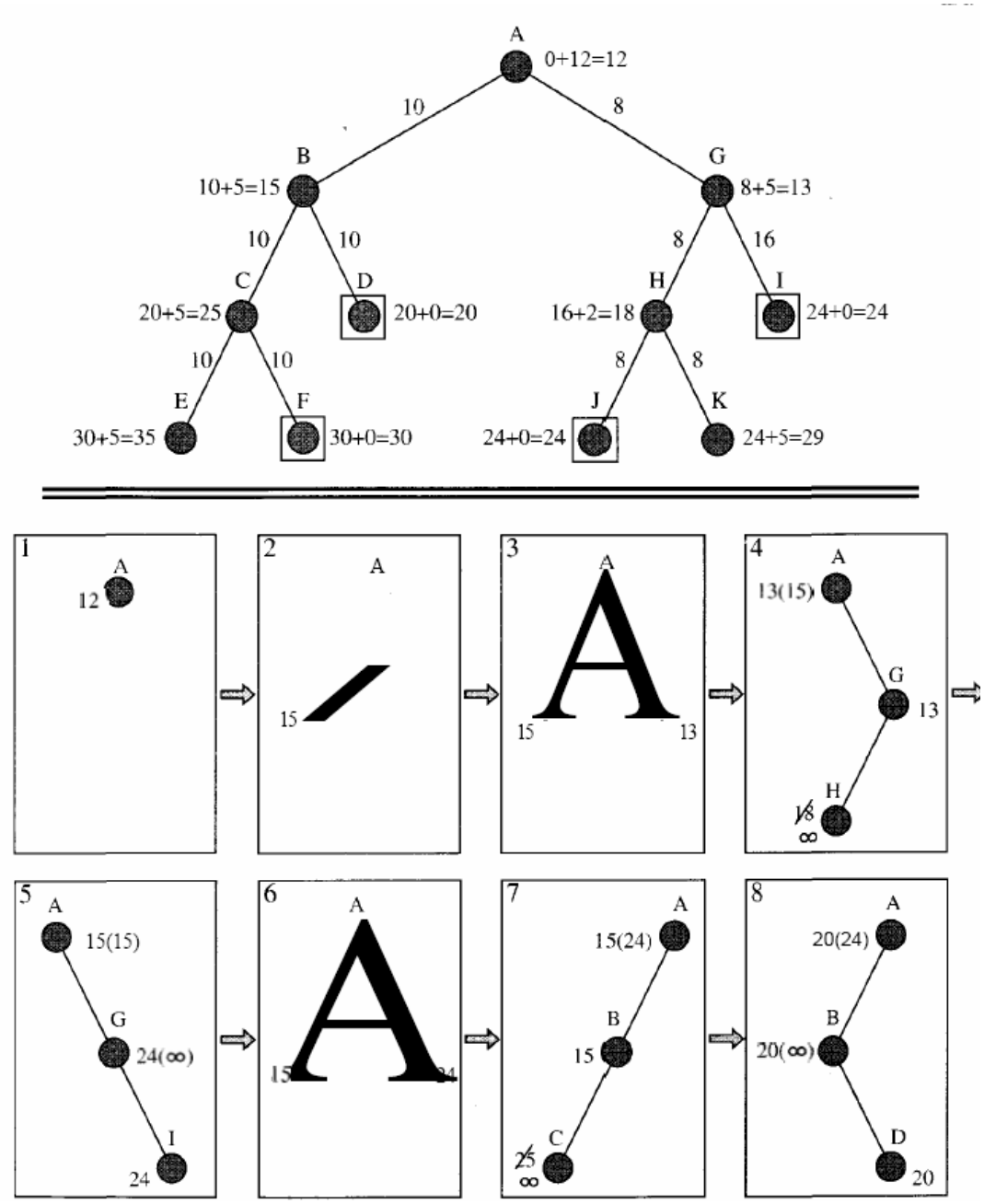
4.2.3.3. Basitleştirilmiş bellek sınırlı A* arama (SMA*)

IDA* iterasyonlar arasında sadece o anki f-maliyet değerini saklamaktadır. Açtığı düğümler saklanmamaktadır. Bu nedenle aynı düğümleri tekrar tekrar açması gerekebilmektedir. Diğer bir deyişle IDA* çok az bellek kullanmaya çalışmaktadır.

SMA*(Simplified Memory-Bounded A*) ise arama için var olan belleğin tamamını kullanmaktadır. Daha fazla bellek kullanmak sadece aramanın etkinliğini artırır. Gerektiği zaman bir düğümü hatırlamak onu tekrar üretmekten daha iyidir. SMA* ın özellikleri aşağıda verilmiştir:

- Ayrılan belleğin tamamını kullanır.
- Tekrarlanmış durumlardan sakınır.
- Eğer ayrılan bellek en sığ çözüm yolunu saklamaya yeterli ise tamdır.
- Eğer ayrılan bellek en sığ optimal çözüm yolunu saklamaya yeterli ise optimaldir.
- Tüm arama ağacı için yeterli bellek olduğunda arama etkindir.

Yeni bir düğüm üretileceği zaman bellekte yer kalmamış ise kuyruktaki bir düğüm çıkartılır. Bu düğüme *unutulmuş düğüm (forgetten node)* denir. Gelecek vaat etmeyen yani en yüksek f-maliyetli düğümlerin unutulması tercih edilir. SMA* örnek üzerinde açıklanacaktır [4].



Şekil 4.18. Arama ağacında basitleştirilmiş bellek sınırlı A* arama (SMA*)

Her düğüm f=g+h ile etiketlenmiştir. Hedef düğümler D,F,I ve J dikdörtgen içinde belirtilmiştir. Hedef yalnız üç düğüm saklanabilen bellek ile en düşük maliyetli hedef düğümü bulmaktır.

SMA* Algoritması aşağıdaki şekilde ilerler:

1. Her aşamada, en derin en düşük f maliyetli düğüme izleyen (successor) eklenir. İzleyen ağaçta olmamalıdır. A düğümüne B eklenmiştir.
2. f(A) hala 12 olduğu için G (f=13) eklenir. A'nın izleyenlerinin minimum maliyeti 13 olduğu için f(A) 13'e eşitlenir. Üç düğüm olduğu için bellek dolmuştur.
3. Düşük maliyetli G açılmadan önce bellekte yer boşaltılmalıdır. En sığ en yüksek değerli B yaprağı unutulur. A'nın en iyi unutulanın değeri f=15 parantez içinde gösterilir. Sonra H eklenir. f(H)=18'dir. H çözüm olmadığı için ve belleğin tamamı kullanıldığı için H'dan geçen bir çözüm bulunamaz. Bu nedenle f(H)= ∞'e eşitlenir.
4. H iptal edilerek I açılır. f(I)=24'dür. G'nin izleyenleri 24 ve ∞ değerlerine sahiptir. Bu nedenle f(G)=24 olur. 24 ve 15 değerlerinden küçük olan 15 f(A)'nın değeri olur. I çözüm olmasına karşın f(A)=15 olduğundan en iyi çözüm olmayabilir.
5. Beklentisi olan B düğümü tekrar üretilir.
6. B'nin ilk izleyeni, maksimum derinlikteki C hedef düğüm olmadığından f(C)= ∞ olur.
7. C'yi çıkartarak ikinci izleyen D'ye bakılır. f(D)=20'dir.
8. En derin en düşük maliyetli düğüm D'dir. D hedef düğüm olduğu için seçilerek arama işlemine son verilir.

SMA algoritmasının kısa bir bölümü*

```
Function SMA *(problem) bir çözüm sırası döndürür
        inputs: problem, bir problem
        local variables: Queue, f- maliyet tarafından sıralanmış düğüm
kuyruğu
Queue ← Make-Queue({MAKENODE(INITIALSTATE[problem])})
```

```
loop do
        if Queue is empty then return failure
        n ← Kuyruktaki en derin en az maliyetli f-maliyet düğümü
        if GOAL-TEST(n) then return success
        s ← NEXT-SUCCESSOR(n)
        if s is not a goal and is at maximum depth then
                f(s) ← ∞
        else
                f(s) ← MAX(f(n), g(s)+h(s))
        if all of n's successors have been generated then
        update n's f-cost and those of its ancestors if necessary
        if SUCCESSORS(n) all in memory then remove n from Queue
        if memory is full then
                delete shallowest, Kuruktaki en yüksek f-maliyetli düğüm
                remove it from its parent's successor list
                insert its parent on Queue if necessary
        insert s on Queue
end
```

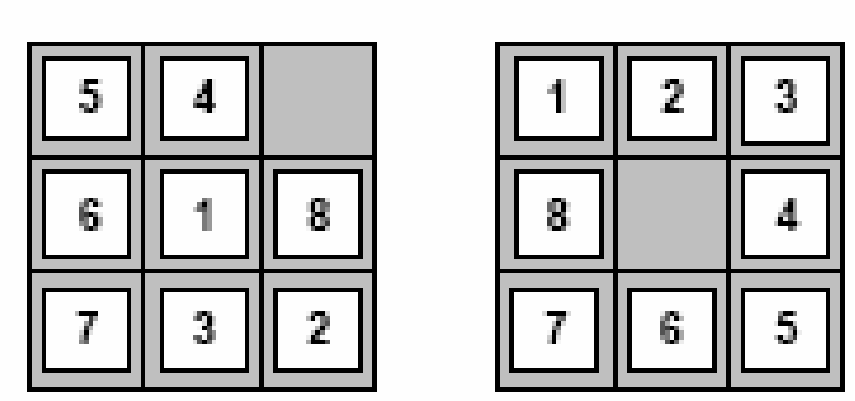
SMA nın genel özellikleri

- Aramayı yapmak için mevcut belleğin hepsini kullanır.
- Tekrar eden durumları belleğin izin vermesine göre önler.
- Eğer herhangi bir erişilebilir bir çözüm varsa tamdır (Complete)
- Eğer erişilebilir en uygun çözüm varsa optimaldir. Aksi taktirde en iyi erişilebilir çözümü getirir.
- Uygun çözümü bulmak için ileri ve geri hareketler yapması ekstra zaman gerektirmektedir.
- Space: sınırlıdır [4].

4.2.4. Sezgisel bulma fonksiyonları (heuristic functions)

Sezgisel bulma fonksiyonları çözüm gereken hesaplama miktarlarını indirgemek için kullanılır [31].

Bir önceki örnekte iki şehir arasındaki direkt mesafe bulma (heuristic) fonksiyonu olarak kullanılmıştı. Bu bölümde 8-puzzle problemi incelenecektir.



Şekil 4.19. 8-puzzle problemi

Problemin tipik çözümü 20 adımdır. Bu başlangıç durumuna göre değişebilir. Dallanma faktörü 3 kadardır (boşluk ortada olduğunda 4 olası hareket, köşelerde olduğunda 2 olası hareket ve kenarlarda 3 olası hareket vardır). Bunun anlamı 20 derinlikli aramada yaklaşık $3^{20} = 3,5 \times 10^9$ durumdur. Tekrarlanan durumlar dikkate alınır ise 9 karenin 9! = 362 880 farklı düzenlemesi olacaktır [19].

Ama bunların yalnız yarası çözülebilir durumlardır. Durum oldukça azalmasına rağmen hala çok fazladır. Bu nedenle iyi bir bulma fonksiyonu gerekmektedir. Eğer en kısa çözümü bulmak istiyor isek bulma fonksiyonu hedef durum için gerekli adımları fazla tahmin etmemesi gerekir. Aşağıdaki fonksiyonlar bulma fonksiyonu olarak kullanılabilir:

- *h1*= Yanlış pozisyondaki /Yerinde olmayan taşların sayısı. Yukarıdaki şekilde başlangıç durumunda 8 taşın hiçbiri doğru yerde olmadığı için h1=8 dir. Yerinde olmayan taşlar en az bir kez hareket ettirileceği için h1 kabul edilebilir bir bulma fonksiyonudur.

- *h2*= Taşların hedef pozisyonundan uzaklıklarının toplamı. Taşlar çapraz hareket edemeyeceği için uzaklık yatay ve düşey mesafelerin toplamı

olacaktır. Bu uzaklık bazen Manhattan uzaklığı olarak isimlendirilir. H2 kabul edilebilir bir bulma fonksiyonudur. Çünkü herhangi bir hareket bir taşı amaca bir adım daha yaklaştırır. 8 taşın başlangıç durumundaki Manhattan mesafesi:

h2= 2+3+2+1+2+2+1+2 = 15 dir.

Bulma Fonksiyonunun Performans Üzerindeki Etkisi

Bulma kalitesini belirtmenin bir yolu *etkin dallanma faktörüdür (b*).* Eğer bir problem için A* ile açılan düğümlerin sayısı N ve çözüm derinliği d ise b* d derinliğindeki dengeli (uniform) bir ağacın N+1 düğüm içermesi için gerekli dallanma faktörüdür:

$N+1 = 1 + b^* + (b^*)^2 + ... + (b^*)^d$

Örneğin A* 5 derinliğindeki bir çözümü 52 düğüm kullanarak buluyor ise etkin dallanma faktörü 1,92'dir. İyi tasarlanmış bulma fonksiyonu ile b* 1'e yaklaşır.

8-bulmacası için yukarıda verilen h1 ve h2 bulma fonksiyonlarını karşılaştırmak için çözüm uzunluğu 2 ... 16 aralığında rasgele 100'er problem üretilerek A* ve iteratif derinleşme (IDS) ile çözülmüştür. Çözümler aşağıdaki tabloda görülmektedir [4].

Çizelge 4.4. *h1, h2* `li IDA ve A* algoritmaları için arama maliyetlerinin ve etkili dallanma faktörlerinin karşılaştırılması. Veriler 100 8-puzzle örneğinin ortalamasıdır [4].

	Arama Maliyeti			Etkin Dallanma Faktörü		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2,45	1,79	1,79
4	112	13	12	2,87	1,48	1,45
6	680	20	18	2,73	1,34	1,30
8	6384	39	25	2,80	1,33	1,24
10	47127	93	39	2,79	1,38	1,22
12	374404	227	73	2,78	1,42	1,24
14	3473941	539	113	2,83	1,44	1,23
16	-	1301	211	-	1,45	1,25

h2 daima h1'den daha iyidir. Çünkü daha az düğüm açılmaktadır.

4.2.4.1. Bulma fonksiyonlarının çıkartılması

h1 ve *h2* 8-puzzle için iyi birer bulma fonksiyonudur ve h2 daha iyidir. Herhangi bir problem için bu fonksiyonlar nasıl bulunabilir? Bilgisayar bulabilir mi?

h1, *h2* 8 puzzle için tahmini yol uzunluk değerleridir. Eğer 8 bulmacasında kurallar, taşlar yalnız bitişikteki boş kareye değil dolulara da geçebilecek şekilde gevşetilir ise h1 ve h2 en kısa çözüm için gerekli adımı doğru bir şekilde verir. İşlemlerdeki kısıtlamaların azaltıldığı problemlere *gevşek problem (relaxed problem)* adı verilir. Gevşek problemin çözümü orijinal problemin çözümü için iyi bir bulma fonksiyonu verir.

Eğer problemin tanımı formal bir dille yazılır ise gevşek problemler otomatik olarak oluşturulabilir. 8-puzzle için aşağıdaki tanımı yapabiliriz:

Eğer A B'ye bitişik ise ve B boş ise taş A'dan B'ye hareket edebilir.

Bir veya daha fazla koşulu gözardı ederek aşağıdaki gevşek problemi yazabiliriz:

a. Eğer A B'ye bitişik ise taş A'dan B'ye hareket edebilir.
b. Eğer B boş ise taş A'dan B'ye hareket edebilir.
c. Taş A'dan B'ye hareket edebilir.

(a) `dan h2 *(Manhattan mesafe*) elde edilir. (c)`den h1 (yanlış yere konmuş taşlar) elde edilir.

Bu yöntemi kullanan ABSOLVER programı (1993) problem tanımından bulma fonksiyonları çıkartabilmektedir. Bu program 8-bulmacası için mevcut fonksiyonlardan daha iyi bir fonksiyon üretmiştir. Rubik küp için de ilk faydalı fonksiyonu bulmuştur [4].

Eğer kabul edilebilir fonksiyonlar birden fazla ise maksimum değer vereni seçilir:

$h(n) = max(h_1(n), ... , h_m(n))$.

4.2.5. Informed aramalar özet

- Sezgisel arama yöntemleri, problem hakkındaki bilgiden yararlanırlar.
- Sezgi (Heuristic), hedefe ulaşmak için kalan maliyetin tahminidir.
- İyi bir sezgi, arama süresini, üstelden doğrusala indirir.
- Best-first Search tam da, optimal de değildir.
- A*, AI'da anahtar teknolojidir. f=g+h, (g, harcanan miktar ve h, hedefe ulaşmada harcanması beklenen miktar)

5. MATERYAL VE METOT

Bu çalışmada sıklıkla kullanılan arama algoritmalardan Breadht-first, depth-first, A* (h1-Yerinde olmayan taşlar) ve A* (h2-Manhattan uzaklığı) birbirleriyle karşılaştırılması incelenmiştir.

5.1. Materyal

Örnek bir problem üzerinde arama algoritmaları karşılaştırılması yapılmak istenmektedir. Karşılaştırma işlemi için problem olarak 8-puzzle örneği ele alınmıştır.

8-puzzle örneği durum uzayı ve zaman karmaşıklığı açısından fazla büyük olmadığından tercih edilmiştir. Arama algoritmalarının performanslarının ölçülmesi için yaygın olarak kullanılmaktadır [19].

8-puzzle probleminde başlangıç durumu ve hedef durumu bulunmaktadır.

?	?	?
?	?	?
?	?	?

Başlangıç durumu

1	2	3
8	0	4
7	6	5

Hedef durumu

Algoritmalar 8- puzzle oyununda operatörler (sağ, sol, yukarı, aşağı) aracılığıyla kurallara bağlı olarak başlangıç durumdan hedef duruma ulaşmaya çalışmaktadırlar.

Kullanılan algoritmalar;

BFS (Breadth-first Search): Başlangıç durumdan itibaren mümkün operatörlerin hepsi kullanılarak o derinlikteki tüm düğümler açılmakta ve belleğe kaydedilmektedir. Sonuç bulunmayınca bir sonraki derinliğe inilerek

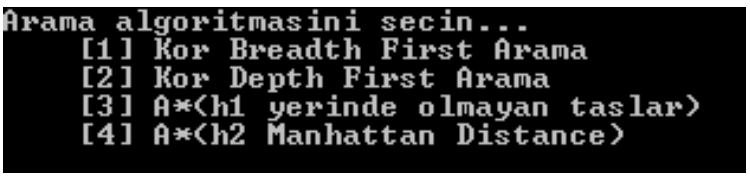
daha önce açılmış olan düğümlere mümkün operatörler kullanılarak yeni düğümler açılmaktadır. Bu işlem sonuç buluncaya kadar devam etmektedir.

DFS (Depth-first search): BFS nin tersine açılması mümkün olan düğümler içerisinden kök düğüme en uzak olan düğüm önce açılır. Bu belirli bir derinliğe kadar devam eder. Sonuç bulunamayınca geri dönülür ve bir sonraki düğümün belirlenen derinliğine kadar iner. Bu işlem sonuç bulununcaya kadar veya belirtilen derinlikteki tüm düğümler açılıncaya kadar devam eder.

A (Yerinde olmayan taşlar-h1):* Değer fonksiyonu olarak açılan düğümdeki yerinde olmayan taşların sayısı kullanılmaktadır. Verilen başlangıç düğümden başlayarak operatörler yardımıyla yeni düğümler açılır. Bu düğümler içerisinde hedef duruma göre yerinde olmayan taşlar her bir düğüm için hesaplanır. Ve en küçük değere sahip düğüm seçilerek ona bağlı yeni düğümler açılır. Ve yine oluşturulan düğümde en düşük değerdeki düğüm açılır. Eğer yeni düğüm yoksa bir üst seviyeye çıkılarak bir üst düşük değer seçilir.

A(Manhattan uzaklığı-h2):* Değer fonksiyonu olarak taşların hedefteki yerlerine uzaklıkları toplamı alınmaktadır. A* (Yerinde olmayan taşlar) kısmında anlatılanlar burada da geçerlidir.

Algoritmaların kodlanmasında Visual C++ 6.0 programlama dili kullanılmıştır. Ayrıca rasgele başlangıç durumları oluşturmak için yine C++ programı kullanılmıştır.



Şekil 5.1. Programın genel görüntüsü

8-puzzle problemini çözmek için P IV 2.8 Ghz, 1 Gigabyte RAM, Windows XP SP2 işletim sistemine sahip bilgisayar kullanılmıştır.

Ayrıca elde edilen sonuçların değerlendirilmesinde istatistiksel veri analizi programı olan MINITAB 14 paket programı kullanılmıştır.

5.2. Metot

Bu çalışmada arama algoritmalarını karşılaştırmak için çözüme ulaşmak için açtıklar *durum sayıları* kriter olarak kullanılmıştır. Çözüm süresini sabitlemek için 2 ve 5 dakikalık limitler kullanılmıştır. Bu limitlerin kullanılmasının önemi BFS ve DFS algoritmalarının bellek ihtiyaçlarının üssel olarak artması ve belirli bir süreden sonra bilgisayarın bellek hatası vermesidir. Ayrıca DFS algoritmasında derinlik sınırı olarak 26 belirtilmiştir.

8-puzzle örneğinde hedef durum sabit tutulmuş olup birbirinden farklı 1000 adet başlangıç durumları oluşturulup her bir başlangıç durumu dört algoritmayla da çözülmeye çalışılmış ve çıkan sonuçlar txt dosyasına kayıt edilmiştir. 8-puzzle örneğinde hedef durumu aşağıda belirtilmiştir.

?	?	?
?	?	?
?	?	?

Başlangıç durumu

1	2	3
8	0	4
7	6	5

Hedef durumu

Başlangıç durumu oluşturmada iki farklı yöntem izlenmiştir.

1. Tam rasgele yöntemi
2. Mantıksal rasgele yöntemi

Tam rasgele yöntemi

Tam rasgele yönteminde başlangıç durumu 9 boyutlu bir dizi gibi düşünülmüş ve C++`ta kullanılan Time kütüphanesi ve Rand komutu aracılığıyla rasgele sayılar bu diziye yerleştirilmiştir. Bu yöntemin sakıncası çözümsüz pek çok başlangıç durumu oluşturmasıdır. Bu yöntem ile her algoritmada 1000 defa işlem yapılarak çözümü bulmak için açılan durum sayıları ve 5 dakika içerisinde çözümsüz adetler saptanmıştır. Algoritmalardaki adım sayılarının ortalamaları arasındaki farkların tesadüften ileri gelip gelmediği tek yönlü varyans analizi tekniği ile saptanmıştır. Farklı grupların tespitinde ise Tukay testi kullanılmıştır. Oranların birbirleriyle karşılaştırılmasında ise Z testi kullanılmıştır.

Bu çalışmada kullanılan ve istatistik analiz yöntemlerinden biri olan, varyans analizi k bağımsız ya da k bağımlı gruptan elde edilen verilerin grup ortalamalarının ya da işlem ortalamalarının farklılığını test etmek için yararlanılan bir yöntemdir [32].

Tek yönlü varyans analizi, k bağımsız grup denemelerinden elde edilen nicel verilerin analizinde yararlanılan bir yöntemdir. Normal dağılım gösteren k bağımsız grup ortalamalarının birbirlerine eşitliğini test etmek için tek yönlü varyans analizi uygulanır [32].

Örnek Deneme-1

6	4	2
3	0	1
7	5	8

Başlangıç durumu

1	2	3
8	0	4
7	6	5

Hedef durumu

Çözüm için algoritmaların açtıkları durum sayıları

BFS: 599 265

DFS: 544 517

A*(h1): 40 703

A*(h2) 3 612

Örnek Deneme-2

0	3	6
1	7	2
8	4	5

Başlangıç durumu

1	2	3
8	0	4
7	6	5

Hedef durumu

Çözüm için algoritmaların açtıkları durum sayıları

BFS: 198 276

DFS: 80 159

A*(h1): 12 523

A*(h2) 3 466

Örnek Deneme-3

8	7	3
1	6	0
2	5	4

Başlangıç durumu

1	2	3
8	0	4
7	6	5

Hedef durumu

Çözüm için algoritmaların açtıkları durum sayıları

BFS: 102 900

DFS: 89 924

A*(h1): 8 128

A*(h2) 3 457

Mantıksal rasgele yöntemi

Mantıksal rasgele yönteminde ise hedef durum baz alınarak boşluk (“0”) rasgele ve kurallara bağlı olarak defalarca sağa, sola, yukarı ve aşağı kaydırılmış ve böylece birbirinden farklı başlangıç durumları oluşturulmuştur. Bu yöntemin avantajı çözümsüz başlangıç durumlarının olmayışıdır. Bu yöntem ile her algoritmada 1000 defa işlem yapılarak çözümü bulmak için açılan durum sayıları ve 5 dakika içerisinde çözümsüz adetler saptanmıştır. Algoritmalardaki adım sayılarının ortalamaları arasındaki farkların tesadüften ileri gelip gelmediği tek yönlü varyans analizi tekniği ile saptanmıştır. Farklı grupların tespitinde ise Tukay testi kullanılmıştır. Oranların birbirleriyle karşılaştırılmasında ise Z testi kullanılmıştır.

Örnek Deneme-1

0	2	3
1	4	5
8	7	6

Başlangıç durumu

1	2	3
8	0	4
7	6	5

Hedef durumu

Çözüm için algoritmaların açtıkları durum sayıları

BFS: 90

DFS: 10

A*(h1): 16

A*(h2) 21

Örnek Deneme-2

2	4	3
0	7	5
1	8	6

Başlangıç durumu

1	2	3
8	0	4
7	6	5

Hedef durumu

Çözüm için algoritmaların açtıkları durum sayıları

BFS: 1 604

DFS: 68 803

A*(h1): 136

A*(h2) 154

Örnek Deneme-3

8	3	5
7	2	1
0	4	6

Başlangıç durumu

1	2	3
8	0	4
7	6	5

Hedef durumu

Çözüm için algoritmaların açtıkları durum sayıları

BFS: 5 400

DFS: 365 282

A*(h1): 431

A*(h2) 107

Alt gruplarda hangi ortalamalar arasında farklılığın olduğu, belirtici istatistikler üzerinde aşağıdaki kurallara uygun olarak gösterilmiş ve yorumlanmıştır.

Ortalama değerlerinin yanında yer alabilecek harfler:

- a harfi: En az duruma bakarak çözümün bulunduğunu ifade etmektedir.
- b harfi: Daha fazla duruma bakarak çözümün bulunduğunu ifade etmektedir.

Her ortalama değerinin yanında yer alan aynı harf, ortalamalar arasında istatistik olarak önemli bir farklılığın olmadığını göstermektedir. Farklı harfler ise ortalamalar arasında istatistik olarak önemli bir farklılığın olduğunu göstermektedir.

6. BULGULAR VE TARTIŞMA

Algoritmaların çözüm için açtıkları durumlar 5 dakikalık zaman diliminde varyans analizi tekniği ile incelenmiş ve aşağıdaki bulgular elde edilmiştir.

6.1. Tam Rasgele Yöntemi ile Elde Edilen Sonuçlar

Algoritmalara ilişkin belirtici istatistikler ve her bir algoritmanın çözülebilir olan problem adetleri Çizelge 6.1 deki gibidir.

Çizelge 6.1. Tam rasgele yöntemi ile elde edilen arama algoritmaları faktörüne göre oluşan belirtici istatistikler

Algoritma Türü	Tekrar Adedi	Çözümlü	Çözümsüz	Ortalama	Ortalama. Standart Hatası	Standart Sapma	En Düşük	En Yüksek
BFS	1000	512	488	988529 b	63391	1434367	413	9198347
DFS	1000	503	497	820303 b	50462	1131756	164	6039339
A* (h1)	1000	207	793	46826 a	2678	38530	207	139399
A* (h2)	1000	506	494	9062 a	459	10333	49	50185

Tek yönlü varyans analizi tekniğine ilişkin hesaplamalardan sonra elde edilen sonuçlar EK-4`te bir çizelge şeklinde gösterilmiştir. Varyans analizi tekniğine ilişkin hesaplamalardan sonra Tukay testi sonunda sadece A* (h1) ve A* (h2) algoritmalarının ortalamaları arasındaki farkın istatistik olarak önemli olmadığı anlaşılmıştır. BFS-A*(h1), BFS-A*(h2), DFS-A*(h1), DFS-A*(h2) algoritmaların ortalamaları arasındaki farkların hepsi istatistik olarak önemlidir ($p<0,01$).

Çizelge 6.1 den de anlaşılacağı gibi yapılan denemelerin sonucunda çözümsüz değerlerinin oldukça yüksektir. Bunun nedeni tam rasgele yöntemi ile çözümsüz durumların oluşturulduğu veya beş dakikalık sürenin çözüm için

yeterli olmadığı denilebilir. Ayrıca en düşük ve en yüksek değerlerden arasındaki farkın oldukça yüksek olduğu gözlenmiştir.

Çizelge 6.2. 4 algoritmadan tam rasgele yöntemi ile elde edilen, çözüme ulaşanların oranlarının ayrı ayrı birbirleri ile karşılaştırılması (Z Testi)

	Çözümlü	Deneme sayısı	Oranlar	önemlilik
BFS	512	1000	0,512000	NS
DFS	503	1000	0,503000	
BFS	512	1000	0,512000	**
A*(h1)	207	1000	0,207000	
BFS	512	1000	0,512000	NS
A*(h2)	506	1000	0,506000	
DFS	503	1000	0,503000	NS
A*(h1)	207	1000	0,207000	
DFS	503	1000	0,503000	NS
A*(h2)	506	1000	0,506000	
A*(h1)	207	1000	0,207000	**
A*(h2)	506	1000	0,506000	

*) $p<0.05$
**) $p<0.01$
NS) önemli değil

Çizelge 6.2 den anlaşılacağı gibi her bir algoritmada çözümlü olanların oranları birbirleriyle karşılaştırılmıştır. Yapılan hesaplamalar sonunda BFS-DFS, BFS-A*(h2), DFS-A*(h2) algoritmalarının çözebilme oranlarının

arasındaki farklar istatistik olarak önemli değildir. Diğer algoritmaların oranları arasındaki farklar ise istatistik olarak önemli bulunmuştur (p<0,01).

6.2. Mantıksal Rasgele Yöntemi ile Elde Edilen Sonuçlar

Algoritmalara ilişkin belirtici istatistikler ve her bir algoritmanın çözülebilir olan problem adetleri Çizelge 6.3 deki gibidir.

Çizelge 6.3. Mantıksal rasgele yöntemi elde edilen arama algoritmaları faktörüne göre oluşan belirtici istatistikler

Algoritma Türü	Tekrar Adedi	Çözümlü	Çözümsüz	Ortalama	Ortalama. Standart Hatası	Standart Sapma	En Düşük	En Yüksek
BFS	1268	1267	1	945847 b	34565	1230332	3	8311627
DFS	1268	1264	4	852261 b	31600	1123459	3	6373566
A* (h1)	1268	542	726	48694 a	1999	46544	3	171390
A* (h2)	1268	1256	12	9063 a	296	10508	3	54042

Tek yönlü varyans analizi tekniğine ilişkin hesaplamalardan sonra elde edilen sonuçlar EK-5`te bir çizelge şeklinde gösterilmiştir. Varyans analizi tekniğine ilişkin hesaplamalardan sonra Tukay testi sonunda sadece A* (h1) ve A* (h2) algoritmalarının ortalamaları arasındaki farkın istatistik olarak önemli olmadığı anlaşılmıştır. BFS-A*(h1), BFS-A*(h2), DFS-A*(h1), DFS-A*(h2) algoritmaların ortalamaları arasındaki farkların hepsi istatistik olarak önemlidir (p<0,01).

Çizelge 6.3 den de anlaşılacağı gibi yapılan denemelerin sonucunda çözümsüz değerleri oldukça düşüktür. Bunun nedeni mantıksal rasgele yöntemi ile çözümsüz durumların oluşturulma ihtimalinin düşük olması veya beş dakikalık sürenin çözüm için yeterli olduğu denilebilir.

Çizelge 6.4. 4 algoritmadan mantıksal rasgele yöntemi ile elde edilen, çözüme ulaşanların oranlarının ayrı ayrı birbirleri ile karşılaştırılması (Z Testi)

	Çözümlü	Deneme sayısı	Oranlar	önemlilik
BFS	1267	1268	0.999211	NS
DFS	1264	1268	0.996845	
BFS	1267	1268	0.999211	**
A*(h1)	542	1268	0.427445	
BFS	1267	1268	0.999211	**
A*(h2)	1256	1268	0.990536	
DFS	1264	1268	0.996845	**
A*(h1)	542	1268	0.427445	
DFS	1264	1268	0.996845	*
A*(h2)	1256	1268	0.990536	
A*(h1)	542	1268	0.427445	**
A*(h2)	1256	1268	0.990536	

*) $p<0.05$
**) $p<0.01$
NS) önemli değil

Çizelge 6.4 den anlaşılacağı gibi yapılan hesaplamalar sonucunda BFS ile DFS algoritmaların çözülebilir oranları arasındaki fark istatistik olarak önemli değildir. Diğer algoritmaların çözülebilme oranları arasındaki farklar önemlidir ($p<0,05$) ve ($p<0,01$).

7. SONUÇ VE ÖNERİLER

Elde edilen bulgular ve yapılan istatistik değerlendirmeler sonucunda aşağıdaki sonuçlar elde edilmiştir;

Algoritmaların bellek gereksinimi ve işlem süresinin fazla oluşundan dolayı denemeler için en düşük özellikleriyle P IV işlemciye sahip 512 MB RAM belleği olan bilgisayar kullanılması gerekmektedir. Daha düşük konfigürasyona sahip bilgisayarlar işlemi daha uzun sürede yapmakta veya bazen kilitlenmektedir.

BFS ve DFS algoritmaları kör arama grubunda olduklarından benzer özellikler göstermektedirler. Sonuç bulma hususunda BFS`nin daha avantajlı olduğu söylenebilir. Eğer çözüm derinlerde değilse BFS algoritması avantajlıdır. Aksi taktirde DFS algoritması daha avantajlı hale gelmektedir. Ayrıca BFS ve DFS algoritmaları çözüm süresi ve açtıkları durumlar arttıkça bellek ihtiyacı da fazlalaşacağından bilgisayarda bellek hatası verme ihtimali oldukça yüksektir.

A* (h1) ve A* (h2) algoritmaları sezgisel arama grubunda olduğundan benzer özellikler göstermektedirler. Bu algoritmalar özellikle işlemciyi oldukça çok kullandığı görülmüştür. Bellek ihtiyacı BFS ve DFS ye göre oldukça düşüktür. Bunun nedeni çözümü daha az durum açarak bulmalarıdır. Algoritmaları içerisinde A* (h2) algoritması en az durum açarak problemleri çözmeyi başarmıştır. BFS ve DFS algoritmaları bazı problemleri çözerken bilgisayarı kilitlemektedir.

Tam rasgele ve mantıksal rasgele yöntemleriyle başlangıç durumları oluşturma bakımından karşılaştırmak gerekirse;

- Tam rasgele yönteminde çözümsüz pek çok deneme oluşmuştur. Bu oran hemen hemen yarı yarıya yakındır. Ve çözümsüz olan bu denemeler hiçbir algoritma tarafından da çözülememiştir.
- Mantıksal rasgele yönteminde ise çözümsüz denemeler sayısı oldukça düşük ve problem en az bir algoritma tarafından çözümlenmiştir.

Bu Konuda Çalışacak Araştırmacılar İçin Öneriler

- Farklı algoritmalar geliştirilerek anlatılan algoritmalarla karşılaştırılabilir.
- Sezgisel algoritmasının farklı yapay zeka problemleri üzerinde çalışma performansları incelenebilir.
- A* algoritmasıyla farklı sezgisel fonksiyonlar bulma üzerinde araştırma yapabilir. Ve daha etkin algoritmalar ortaya çıkartabilir.

KAYNAKLAR

1. İnternet: Department of Computing Science - University of Alberta "Artificial Intelligence Search Algorithms" http://www.cs.ualberta.ca/courses/cmput675.php (2006).

2. İnternet: University of Alberta "Heuristic Search" http://www.cs.ualberta.ca/~jonathan/Courses/657/Notes/1.Introduction.pdf (2006).

3. İnternet: Ege Üniversitesi "Yapay Zekanın Tanımı" http://bornova.ege.edu.tr/~ugur/05_06_Fall/AI/Lecture1_2004.pdf (2006).

4. Russell S. and Norvig P., "Artificial Intelligence: Modern Approach", ***Prentice Hall***, USA, 59-80,95-109 (2003).

5. Zhou, R. and Hansen, A., "Memory-Bounded A* Graph Search", ***15th International FLAIRS Conference***, Florida, 203 - 209 (2002).

6. Sağıroğlu, S., Beşkok, E. ve Erler, M., "Mühendislikte Yapay Zeka Uygulamaları-I", ***Ufuk Yayıncılık***, Kayseri, 1-5 (2003).

7. Uğur, A. ve Kınacı, A.C., "Yapay Zeka Teknikleri ve Yapay Sinir Ağları Kullanılarak Web Sayfalarının Sınıflandırılması", ***XI. Türkiye'de İnternet Konferansı***, Ankara, 78 (2006).

8. Benzer, R., "Zeki karar sistemleri ve bazı askeri uygulamaları", Yüksek Lisans Tezi, ***Sakarya Üniversitesi Fen Bilimleri Enstitüsü***, Sakarya, 1-2 (1998).

9. Uğur, A. ve Binici, E., "Java ile Yapay Zeka Yazılımları Geliştirme", ***II. Ulusal Meslek Yüksekokulları Sempozyumu***, İzmir, 22 (2003).

10. Kocabaş, Ş. ve Öztemel, E., "Harp Oyunlarında Yapay Zeka Uygulamaları", ***Modelleme ve Simülasyon Konferansı***, Ankara, 1-4 (1998).

11. Nabiyev, V.V., "Yapay Zeka", ***Seçkin Yayıncılık***, Ankara, 97-98,105-106,113,127-140 (2003).

12. Haris, S. and Ross, J., "Beginning Algorithms", ***Wiley Publishing***, USA,1-2 (2005).

13. Weixiong, Z., "State-Space Search", ***Springer Publisher***, USA, 1-3 (1999).

14. Pearl, J.,"Heuristics: Intelligent Search Stratagies for Computer Problem Solving", ***Addison-Wesley Publishing Company***, USA, 30-45 (1984).

15. Winston, H.P., "Artificial Intelligence", ***Addison-Wesley Publishing Company***, USA, 63-79 (1992).

16. Coppin, B., "Artificial intelligence illuminated", ***Jones and Bartlett Publishers***, USA, 70,75-81,90-91,103-110 (2004).

17. Kalaycı, T. E., "Yapay zeka teknikleri kullanan üç boyutlu grafik yazılımları için "extensıble 3d" (x3d) ile bir altyapı oluşturulması ve gerçekleştirimi", Yüksek Lisans Tezi, ***Ege Üniversitesi Fen Bilimleri Enstitüsü***, İzmir, 69-70 (2006).

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

19. Reinefeld, A., "Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in IDA*", ***IJCAI-93. Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence***, USA, 248-253 (1993).

20. İnternet: Arizona State University "Arama Metotları" http://cips.eas.asu.edu/fagelgi/studies/olimp/mufredat/algor/e1-kor.htm (2006).

21. Brackeen, D., Baker, B. and Vanhelswue, L., "Developing Games in Java", ***New Riders Publisher***, USA, 655-658 (2003).

22. Tucker, A., "The Computer Science and Engineering Handbook", ***CRC Press***, Florida, 676 - 696 (1996).

23. İnternet: University of California "Problem Solving and Search" http://www.cogsci.ucsd.edu/ ~batali/ 108b/lectures/search.html (2000).

24. LaValle, S. M., "Planning Algorithms", ***Cambridge University Press***, New York, 35-36 (2006).

25. Ruml, W., "Real-time Heuristic Search for Combinatorial Optimization and Constraint Satisfaction", ***AAAI/KDD/UAI-2002 Joint Workshop on Real-Time Decision Support and Diagnosis Systems***, Edmonton - Kanada, 85-90 (2002).

26. İnternet: The University of Colorado "Assignment 3" http://carbon.cudenver.edu/~bstilman /teaching /AIpdfs/05.pdf (2007).

27. Ghosh, S., Mahanti, A. and Nau, D. S., "Improving the efficiency of limited-memory heuristic search", ***Tech. Rep. UMIACS-TR-95-23, USA***, 1-2 (1995).

28. Zhou, R. and Hansen, A., "Breadth-First Heuristic Search", ***Artificial Intelligence***, Volume 170 : 385 - 408 (2006).

29. İnternet: Northeastern University "Solving the 8 Puzzle in a Minimum Number of Moves: An Application of the A* Algorithm" http://www.ccs.neu.edu/home/kunkle/docs/EightPuzzle.pdf (2001).

30. Zhou, R. and Hansen, A., "A Breadth-First Approach to Memory-Efficient Graph Search", ***21st National Conference on Artificial Intelligence (AAAI-06)***, Boston- USA, 1-2 (2006).

31. Korf, E., "Heuristic evaluation functions in artificial intelligence search algorithms", ***Minds and Machines***, Volume 5(4): 489-498 (1995).

32. Özdamar, K., "Paket Programlar ile İstatistiksel Veri Analizi", ***Kaan Kitabevi***, Eskişehir, 381-382 (2002).

EKLER

EK – 1 BAŞLANGIÇ DURUMUNU TAM RASGELE YÖNTEMİ İLE OLUŞTURMAK İLE İLGİLİ KOD DÜZENİ

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <conio.h>
#include <string.h>

int main(void)
{
   char komut[30];
   int Grid[10];
   double sayi[1000];
   char yol1[1],yol2[1],yol3[1],yol4[1],yol5[1],yol6[1],yol7[1],yol8[1],yol9[1];
   char aynimi[10];
   int dizi[10],dizi1[10],i,gecici,j,k;

 //BFS k□tugu ile ilgili degiskenler
 char birkelime[10];
 int c;
 long cozumu;
 FILE *fpbfs;
 time_t t;
srand((unsigned) time(&t));
clrscr();

   for (i=0;i<=1000;i++) sayi[i]=0;

   for (j=1; j<=1000; j++) {
  bas:  dizi[1]=1;dizi[2]=2;dizi[3]=3;dizi[4]=4;dizi[5]=5;dizi[6]=6;dizi[7]=7;dizi[8]=8;dizi[9]=0;


   for (i=1; i<=9; i++) {
   tekrar:  gecici=rand() % 10;
   if (gecici==0) goto tekrar;
   if (dizi[gecici]!=-1)
             {
             dizi1[i]=dizi[gecici];
             dizi[gecici]=-1;
             }
   else
   goto tekrar;
     }


    //Rastgele se‡ilen say□lar□n diziye aktar□lmas□
 Grid[1]=dizi1[1]; itoa(Grid[1],yol1,10);
 Grid[2]=dizi1[2]; itoa(Grid[2],yol2,10);
 Grid[3]=dizi1[3]; itoa(Grid[3],yol3,10);
 Grid[4]=dizi1[4]; itoa(Grid[4],yol4,10);
 Grid[5]=dizi1[5]; itoa(Grid[5],yol5,10);
 Grid[6]=dizi1[6]; itoa(Grid[6],yol6,10);
 Grid[7]=dizi1[7]; itoa(Grid[7],yol7,10);
 Grid[8]=dizi1[8]; itoa(Grid[8],yol8,10);
```

EK – 1 (Devam) BAŞLANGIÇ DURUMUNU TAM RASGELE YÖNTEMİ İLE OLUŞTURMAK İLE İLGİLİ KOD DÜZENİ

```
Grid[9]=dizi1[9]; itoa(Grid[9],yol9,10);

 //Ayn m kontrol
 aynimi[0]=yol1[0]; aynimi[1]=yol2[0]; aynimi[2]=yol3[0];
  aynimi[3]=yol4[0]; aynimi[4]=yol5[0]; aynimi[5]=yol6[0];
   aynimi[6]=yol7[0]; aynimi[7]=yol8[0]; aynimi[8]=yol9[0];
    sayi[j] = atof(aynimi);
   for (k=1;k<j;k++)
   if (sayi[j]==sayi[k]) goto bas;//eger daha ”nce varsa basa git

 komut[0]='A';
 komut[1]='l';
 komut[2]='A';
 komut[3]='l';
 komut[4]='.';
 komut[5]='e';
 komut[6]='x';
 komut[7]='e';
 komut[8]=' ';
 komut[9]=yol1[0]; komut[10]=' ';
 komut[11]=yol2[0]; komut[12]=' ';
 komut[13]=yol3[0]; komut[14]=' ';
 komut[15]=yol4[0]; komut[16]=' ';
 komut[17]=yol5[0]; komut[18]=' ';
 komut[19]=yol6[0]; komut[20]=' ';
 komut[21]=yol7[0]; komut[22]=' ';
 komut[23]=yol8[0];komut[24]=' ';
 komut[25]=yol9[0];komut[26]=' ';
komut[27]='1'; system(komut);
 komut[27]='2'; system(komut);
 komut[27]='3'; system(komut);
 komut[27]='4'; system(komut);
   }
    return 0;
}
```

EK – 2 BAŞLANGIÇ DURUMUNU MANTIKSAL RASGELE YÖNTEMİ İLE OLUŞTURMAK İLE İLGİLİ KOD DÜZENİ

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>

main ()
{
 char komut[30];
 int Grid[10];
 double sayi[1000];
 char yol1[1],yol2[1],yol3[1],yol4[1],yol5[1],yol6[1],yol7[1],yol8[1],yol9[1];
 char aynimi[10];
 int dizi[10],dizi2[10],i,gecici,j,t,k;

 //BFS k□tugu ile ilgili degiskenler
 char birkelime[10];
 int c;
 long cozumu;
 FILE *fpbfs;

 //Ayn□ dizinin tekrar oluŸturulmas□n□ ”nlemek
 //i‡in kulland□g□m sayi degiskeninin icini s□f□rlad□m
 for (i=0;i<=1000;i++) sayi[i]=0;

 t=2;  //Rastgele icin ilk islem sekli 2 yani up

 dizi[1]=1;dizi[2]=2;dizi[3]=3;dizi[4]=8;dizi[5]=4;dizi[6]=5;dizi[7]=7;dizi[8]=6;dizi[9]=0;
 //  dizi[1]=1;dizi[2]=2;dizi[3]=3;dizi[4]=4;dizi[5]=5;dizi[6]=6;dizi[7]=7;dizi[8]=8;dizi[9]=0;
   for (j=1;j<1000;j++) {    //1 -1000`E kadar başlangış matris dizisi olustur.

   bas:
   if (t==0) goto rigth;//Saga g”t□r
   if (t==1) goto left;//Sola g”t□r
   if (t==2) goto up;//Yukar□ g”t□r
   if (t==3) goto down;//Asagi g”t□r

   rigth:if ((dizi[7]!=0) & (dizi[4]!=0) & (dizi[1]!=0))
           { for (i=1; i<10; i++)  {if (dizi[i]==0) gecici=i;}
             dizi[gecici]=dizi[gecici-1];dizi[gecici-1]=0;goto tekrar;}
             else goto up;

   left:if ((dizi[3]!=0) & (dizi[6]!=0) & (dizi[9]!=0))
           { for (i=1; i<10; i++)  {if (dizi[i]==0) gecici=i;}
             dizi[gecici]=dizi[gecici+1];dizi[gecici+1]=0;goto tekrar;}
             else goto down;

   up:if ((dizi[1]!=0) & (dizi[2]!=0) & (dizi[3]!=0))
           { for (i=1; i<10; i++)  {if (dizi[i]==0) gecici=i;}
             dizi[gecici]=dizi[gecici-3];dizi[gecici-3]=0;goto tekrar;}
             else goto left;

   down:if ((dizi[7]!=0) & (dizi[8]!=0) & (dizi[9]!=0))
```

EK – 2 (Devam) BAŞLANGIÇ DURUMUNU MANTIKSAL RASGELE YÖNTEMİ İLE OLUŞTURMAK İLE İLGİLİ KOD DÜZENİ

```
{ for (i=1; i<10; i++)  {if (dizi[i]==0) gecici=i;}
            dizi[gecici]=dizi[gecici+3];dizi[gecici+3]=0;goto tekrar;}
            else goto rigth;
// bas:
 tekrar:
 t=rand() % 4;//Yapilacak islemi rasgele se‡
 //Rastgele se‡ilen say□lar□n diziye aktar□lmas□
 Grid[1]=dizi[1]; itoa(Grid[1],yol1,10);
 Grid[2]=dizi[2]; itoa(Grid[2],yol2,10);
 Grid[3]=dizi[3]; itoa(Grid[3],yol3,10);
 Grid[4]=dizi[4]; itoa(Grid[4],yol4,10);
 Grid[5]=dizi[5]; itoa(Grid[5],yol5,10);
 Grid[6]=dizi[6]; itoa(Grid[6],yol6,10);
 Grid[7]=dizi[7]; itoa(Grid[7],yol7,10);
 Grid[8]=dizi[8]; itoa(Grid[8],yol8,10);
 Grid[9]=dizi[9]; itoa(Grid[9],yol9,10);

 //Ayn□ m□ kontrol□
 aynimi[0]=yol1[0]; aynimi[1]=yol2[0]; aynimi[2]=yol3[0];
  aynimi[3]=yol4[0]; aynimi[4]=yol5[0]; aynimi[5]=yol6[0];
   aynimi[6]=yol7[0]; aynimi[7]=yol8[0]; aynimi[8]=yol9[0];
    sayi[j] = atof(aynimi);
   for (k=1;k<j;k++)
   if (sayi[j]==sayi[k]) goto bas;//eger daha ”nce varsa basa git
 komut[0]='A';
 komut[1]='l';
 komut[2]='A';
 komut[3]='l';
 komut[4]='.';
 komut[5]='e';
 komut[6]='x';
 komut[7]='e';
 komut[8]=' ';
 komut[9]=yol1[0]; komut[10]=' ';
 komut[11]=yol2[0]; komut[12]=' ';
 komut[13]=yol3[0]; komut[14]=' ';
 komut[15]=yol4[0]; komut[16]=' ';
 komut[17]=yol5[0]; komut[18]=' ';
 komut[19]=yol6[0]; komut[20]=' ';
 komut[21]=yol7[0]; komut[22]=' ';
 komut[23]=yol8[0];komut[24]=' ';
 komut[25]=yol9[0];komut[26]=' ';
 komut[27]='1'; system(komut);
 komut[27]='2'; system(komut);
 komut[27]='3'; system(komut);
 komut[27]='4'; system(komut);
} //j li for d”ng□s□n□n sonu

 return 0;
}
```

EK – 3 8-PUZZLE PROGRAMININ KOD DÜZENİ

Eightpuzz.CPP

```
#include"Eightpuzz.h"
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>


int main(int argc, char* argv[]) {
FILE *fp0;
char denesay[12];
int bosluk,i,uzunluk;
//int durum;
Grid1=atoi(argv[1]);
Grid2=atoi(argv[2]);
Grid3=atoi(argv[3]);
Grid4=atoi(argv[4]);
Grid5=atoi(argv[5]);
Grid6=atoi(argv[6]);
Grid7=atoi(argv[7]);
Grid8=atoi(argv[8]);
Grid9=atoi(argv[9]);
durum=atoi(argv[10]);
printf("%d %d %d %d %d %d %d %d %d
\n",Grid1,Grid2,Grid3,Grid4,Grid5,Grid6,Grid7,Grid8,Grid9);

        deneme1=0; // Kontrol yapıyorum. Yazıyı süreyi geçince sadece bir defa yapsın diye
        dfskontrol=0; // Kontrol yapıyorum. Yazıyı süreyi geçince sadece bir defa yapsın diye
        cozuldu=0; // Eğer çözülmüşse zamanı dikkkate almasın.
        char Choice=0;

        first = time(NULL);
        fp0 = fopen("ONSATIR.TXT","a");
        if (durum==1) fprintf(fp0,"Matris: %d %d %d %d %d %d %d %d %d
",Grid1,Grid2,Grid3,Grid4,Grid5,Grid6,Grid7,Grid8,Grid9);
        if (durum==1) fprintf(fp0,"BFS = ");
        if (durum==2) fprintf(fp0,"DFS = ");
        if (durum==3) fprintf(fp0,"A*  = ");
        if (durum==4) fprintf(fp0,"A** = ");
        fclose(fp0);


        if (durum==1) GeneralSearch(BREADTH_FIRST, NOT_USED);
        if (durum==2) GeneralSearch(DEPTH_FIRST, NOT_USED);
        if (durum==3) GeneralSearch(DEPTH_FIRST, WRONG_TILES);
        if (durum==4) GeneralSearch(DEPTH_FIRST, MANHATTAN_DISTANCE);




        ltoa(deneme,denesay,10);

   fp0 = fopen("ONSATIR.TXT","a");
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
                if (durum<3) {
                if (deneme1==2) fprintf(fp0,"---      ");
                if (deneme1==3) fprintf(fp0,"---      ---      ");
                if (dfskontrol) fprintf(fp0,"---      ---      ---      ");
                }

        if (durum>2) {
                if (deneme1==2) fprintf(fp0,"---     ");
                if (deneme1==3) fprintf(fp0,"---     ---     ");
                }
        // Adım sayısını dosyaya yazar
        fprintf(fp0,"%d ",deneme);

        // bfs ve dfs dosyaya yazı yazarken boşluk bırakmak için
        if ((durum<3) & (deneme1<2)) bosluk=26;
        if ((durum<3) & (deneme1==2)) bosluk=18;
        if ((durum<3) & (deneme1==3)) bosluk=9;

        // a* a** için dosya yazı yazarken boşluk bırakmak için
        if ((durum>2) & (deneme1<2)) bosluk=23;
        if ((durum>2) & (deneme1==2)) bosluk=15;
        if ((durum>2) & (deneme1==3)) bosluk=8;
        uzunluk=strlen(denesay);
        if (uzunluk<bosluk)
                for (i=uzunluk;i<bosluk;i++) fprintf(fp0," ");

        if (durum==4) fprintf(fp0,"\n");
         fclose(fp0);
        return 0;

}
```

Eightpuzz.H

```
Void gotoxy(int x, int y);

enum TYPE      {BREADTH_FIRST, DEPTH_FIRST };
enum HEURISTIC {NOT_USED, MANHATTAN_DISTANCE, WRONG_TILES};

#include<iostream.h>
#include<windows.h>
#include"State.h"
#include"Llist.h"

long int deneme;
int dfskontrol; // Bir defa ekrana mesajı göstersin.
CLList PrevStates;
CLList MainQue;
SIDE PushSide;
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
SIDE PopSide;
CState* GeneralExpand(int DepthLimit, HEURISTIC heuristic);

double Expanded = 0;

void ShowSolution(CState* Solution) {

		CLList Reverse;
	bool EndLoop=false;

	while(!EndLoop) {
		Reverse.Push(LEFT, Solution);
		if(Solution->GetPrevState() != 0) {
			Solution = Solution->GetPrevState();
		}
		else {
			EndLoop = true;
		}
	}

	int NewLine = 0;

	CState *Temp;

	cout << "\n";

	while(!Reverse.IsListEmpty()) {
		Temp = Reverse.Pop(LEFT);
		NewLine++;

		if(NewLine > 10) { cout << "\n"; NewLine=0;}

		switch(Temp->GetLastOperator()) {
		case NORTH:
			cout << "North, ";
			break;
		case EAST:
			cout << "East, ";
			break;
		case SOUTH:
			cout << "South, ";
			break;
		case WEST:
			cout << "West, ";
			break;
		}
	}
	deneme=Expanded;
	cozuldu=1;
	cout << "\n\n" << "Expanded: " << Expanded << endl;
}

void gotoxy(int x, int y) {
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
	COORD  coord  = {x,y};
	HANDLE handle = GetStdHandle(STD_OUTPUT_HANDLE);

	SetConsoleCursorPosition(handle,coord);
}

void GeneralSearch(TYPE Type, HEURISTIC heuristic) {

	CState *Solution;
	int DepthLimit=0;

	switch(heuristic) {
	case NOT_USED:

		if(Type == BREADTH_FIRST) {
			cout << "BREADTH FIRST Search :";
			PushSide = RIGHT;
			PopSide  = LEFT;
		}
		else {
			// Depth First Search
			cout << "Depth Limit Search :";
			//cin >> DepthLimit;
			DepthLimit=26;
			PushSide = RIGHT;
			PopSide  = RIGHT;
		}
		break;
	case MANHATTAN_DISTANCE:
		cout << "A** Search :";
		PushSide = RIGHT;
		PopSide  = RIGHT;
		break;
	case WRONG_TILES:
		cout << "A* Search :";
		PushSide = RIGHT;
		PopSide  = RIGHT;
		break;
	}
	MainQue.Push(PushSide, new CState());

	Solution = GeneralExpand(DepthLimit, heuristic);

	if(Solution != 0) {
		cout << "FOUND SOLUTION!" << endl;
		ShowSolution(Solution);
		cout << "DONE" << endl;
	}
	else {dfskontrol=1;

		cout << "FAIL" << endl;
	}

}
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
CState* GeneralExpand(int DepthLimit, HEURISTIC heuristic) {

	CState *CurrentState = 0;
	CState *TempState = 0;

	int LastDepth=-1;

	if(PushSide == PopSide) {cout << "THINKING...please wait." << endl;}

	while(!MainQue.IsListEmpty()) {

		if(heuristic == NOT_USED) {
			CurrentState = MainQue.Pop(PopSide);
		}
		else {
			CurrentState = MainQue.PopBestByHeuristics(heuristic);
		}

		PrevStates.Push(RIGHT, CurrentState);
			if(LastDepth < CurrentState->GetDepth() && PushSide != PopSide) {
			LastDepth = CurrentState->GetDepth();
			cout << "EXPANDING LEVEL " << LastDepth << endl;
		}


		if((CurrentState->GetDepth() < DepthLimit) || (DepthLimit == 0 )) {

			if((CurrentState->CanBlankMove(NORTH)) && (CurrentState-
>GetLastOperator() != SOUTH)) {

				TempState = new CState(CurrentState, NORTH);
				MainQue.Push(PushSide, TempState);
				Expanded++;

				if(TempState->Solution()) {
					return TempState;
				}

			}

			if((CurrentState->CanBlankMove(EAST)) && (CurrentState-
>GetLastOperator() != WEST)) {

				TempState = new CState(CurrentState, EAST);
				MainQue.Push(PushSide, TempState);
				Expanded++;


				if(TempState->Solution()) {
					return TempState;
				}
			}
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
                        if((CurrentState->CanBlankMove(SOUTH)) && (CurrentState-
>GetLastOperator() != NORTH)) {

                                TempState = new CState(CurrentState, SOUTH);
                                MainQue.Push(PushSide, TempState);
                                Expanded++;


                                if(TempState->Solution()) {
                                        return TempState;
                                }
                        }

                        if((CurrentState->CanBlankMove(WEST)) && (CurrentState-
>GetLastOperator() != EAST)) {

                                TempState = new CState(CurrentState, WEST);
                                MainQue.Push(PushSide, TempState);
                                Expanded++;

                                if(TempState->Solution()) {
                                        return TempState;
                                }
                        }
                }
        }

        return 0;
}
```

List.H

```
#include<iostream.h>
#include<windows.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

time_t first, second;
int deneme1,durum,cozuldu;
FILE *fp;
enum SIDE { LEFT, RIGHT };

struct Link {
        Link   *LeftLink;
        Link   *RightLink;
        CState *Data;
};

class CLList {
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
private:
        Link* LeftPointer;
        Link* RightPointer;
        double ListLen;

        void EmptyUsedMemory() {
                CState *temp;
                while(!IsListEmpty()) {
                        temp = Pop(LEFT);
                        delete temp;
                }
        }

public:
        class ERROR_CANT_POP_EMPTY_LIST{};

        CLList() {

                LeftPointer  = new Link;
                RightPointer = new Link;
                ListLen      = 0;

                LeftPointer->LeftLink = 0;
                LeftPointer->RightLink = RightPointer;
                RightPointer->RightLink = 0;
                RightPointer->LeftLink = LeftPointer;
        }

        ~CLList() {
                EmptyUsedMemory();
        }

        inline double GetListLen() {
                return ListLen;
        }

        inline bool IsListEmpty() {
                return (LeftPointer->RightLink == RightPointer);
        }

        CState* Pop(SIDE Side) {

                Link  ForReturn;
                Link* ForDelete;
                if (!IsListEmpty()) {

                        ListLen--;
                        if (Side == LEFT) {

                                ForReturn                  = *(LeftPointer->RightLink);
                                ForDelete                  = LeftPointer->RightLink;
                                LeftPointer->RightLink     = ForReturn.RightLink;
                                ForReturn.RightLink->LeftLink  = LeftPointer;
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
                        }
                        else {

                                ForReturn                   = *(RightPointer->LeftLink);
                                ForDelete                   = RightPointer->LeftLink;
                                RightPointer->LeftLink      = ForReturn.LeftLink;
                                ForReturn.LeftLink->RightLink  = RightPointer;
                        }
// başla Burası BFS ve DFS için zaman ayarının yapıldığı yer
second = time(NULL);
if ((second-first>301) & (cozuldu==0))exit(1);


else if ((second-first>300) &(cozuldu==0)){
        if (deneme1==3) {
                printf("310 saniye gecti\n");
                                fp = fopen("ONSATIR.TXT","a");

                                fprintf(fp,"---     ---     ---     ");
                                fclose(fp);
                                }
                                deneme1=4;
                                }

else if ((second-first>295) &(cozuldu==0)){
        if (deneme1==2) {
                printf("300 saniye gecti\n");

                                }

                                deneme1=3;
                                }
else if ((second-first>120)&(cozuldu==0)) {
        if (deneme1==1) {
                printf("120 saniye gecti\n");

                                }

                                deneme1=2;
                                }

else if ((second-first>5)&(cozuldu==0)) {
        if (deneme1==0) {
                //printf("20 saniye geçti\n");

                                }

                                deneme1=1;
                                }
// Bitti.
                delete ForDelete;
                        return ForReturn.Data;
                }
                return 0;
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
}
void Push(SIDE Side, CState* What) {

        Link* NewLink = new Link;
        NewLink->Data = What;
        ListLen++;
        if (Side == LEFT) {

                NewLink->RightLink          = LeftPointer->RightLink;
                NewLink->LeftLink           = LeftPointer;
                LeftPointer->RightLink      = NewLink;
                NewLink->RightLink->LeftLink = NewLink;
        }
        else {

                NewLink->RightLink          = RightPointer;
                NewLink->LeftLink           = RightPointer->LeftLink;
                RightPointer->LeftLink      = NewLink;
                NewLink->LeftLink->RightLink = NewLink;
        }
}

CState* PopBestByHeuristics(HEURISTIC heuristic) {

        int   BestValue=9999;
        int   Temp=0;
        Link* BestState = 0;
        Link* Current = LeftPointer;
        CState* ForReturn = 0;

        if(!IsListEmpty()) {


                while(Current->RightLink != RightPointer) {

                        Current = Current->RightLink;

                        if(heuristic == MANHATTAN_DISTANCE) {
                                Temp = Current->Data->GetManhattanDistance();
                        }
                        else {
                                Temp = Current->Data->GetWrongTiles();
                        }

                        if(Temp < BestValue) {
                                BestValue = Temp;
                                BestState = Current;
                        }
                }


                BestState->RightLink->LeftLink = BestState->LeftLink;
                BestState->LeftLink->RightLink = BestState->RightLink;
```

```
                        ForReturn = BestState->Data;

// başla Burası A* Wrong ve A* Manhattan için zaman ayarı
                second = time(NULL);
                if ((second-first>301) &(cozuldu==0))

                        exit(1);

                else if ((second-first>300) &(cozuldu==0)) {
                                if (deneme1==3) {
                                        printf("310 sn gecti\n");
                                        fp = fopen("ONSATIR.TXT","a");

                                        fprintf(fp,"---     ---     ---     ");
                                        if (durum==4) fprintf(fp,"\n");


                                        fclose(fp);

                                }
                                deneme1=4;
                                }
                else if ((second-first>295) &(cozuldu==0)){
                                if (deneme1==2) {
                                        printf("300 sn gecti\n");


                                        }
                                deneme1=3;
                                }
                else if ((second-first>120) &(cozuldu==0)){
                                if (deneme1==1) {
                                        printf("120 sn gecti\n");


                                        }
                                deneme1=2;
                                }
                else if ((second-first>5) &(cozuldu==0)){
                                if (deneme1==0) {


                                }

                                deneme1=1;
                                }
// Bitti.

                        delete BestState;

                        return ForReturn;
                }
                return 0;
        }

        CState* PeekByBestHueristics(HEURISTIC heuristic) {
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
		int   BestValue=9999;
		int   Temp=0;
		Link* BestState = 0;
		Link* Current = LeftPointer;
		CState* ForReturn = 0;

		if(!IsListEmpty()) {


			while(Current->RightLink != RightPointer) {

				Current = Current->RightLink;

				if(heuristic == MANHATTAN_DISTANCE) {
					Temp = Current->Data->GetManhattanDistance();
				}
				else {
					Temp = Current->Data->GetWrongTiles();
				}

				if(Temp < BestValue) {
					BestValue = Temp;
					BestState = Current;
				}
			}

			ForReturn = BestState->Data;

			return ForReturn;
		}
		return 0;
	}

};
```

State.H

```
int Grid1,Grid2,Grid3,Grid4,Grid5,Grid6,Grid7,Grid8,Grid9;
enum DIRECTION { NONE, NORTH, EAST, SOUTH, WEST };

class CState {

private:

	char Grid[10];
	char Depth;
	DIRECTION OperatorApplyed;

	CState *PrevState;


	inline int FindBlank(char Search=0) {
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
		int Blank=1;
		while (Grid[Blank] != Search) {
			Blank++;
		}
		return Blank;
}

void MoveBlank(DIRECTION Direction) {

		if (!CanBlankMove(Direction)) {
			throw ERROR_ILLEGAL_MOVE();
		}
		else {

			int Blank = FindBlank();

			OperatorApplyed = Direction;

			switch (Direction) {
			case NORTH:
				Grid[Blank] = Grid[Blank - 3];
				Grid[Blank - 3] = 0;
				break;

			case EAST:
				Grid[Blank] = Grid[Blank + 1];
				Grid[Blank + 1] = 0;
				break;

			case SOUTH:
				Grid[Blank] = Grid[Blank + 3];
				Grid[Blank + 3] = 0;
				break;

			case WEST:
				Grid[Blank] = Grid[Blank - 1];
				Grid[Blank - 1] = 0;
				break;
			}
		}
}

int GetDistanceBetween(int Tile1, int Tile2) {

		int X1, X2;
		int Y1, Y2;
		int temp=0;


		Y1 = 1;
		Y2 = 2;
		X1 = Tile1;
		X2 = Tile2;
		if(Tile1 > 3) { Y1 = 2; X1 = Tile1 - 3; }
```

```
                if(Tile1 > 6) { Y1 = 3; X1 = Tile1 - 6; }
                if(Tile2 > 3) { Y2 = 2; X2 = Tile2 - 3; }
                if(Tile2 > 6) { Y2 = 3; X2 = Tile2 - 6; }

                        if(Y1 - Y2 < 0) {
                        temp = Y1;
                        Y1 = Y2;
                        Y2 = temp;
                }
                if(X1 - X2 < 0) {
                        temp = X1;
                        X1 = X2;
                        X2 = temp;
                }

                return ((Y1 - Y2) + (X1 - X2));

        }

public:

        class ERROR_ILLEGAL_MOVE{};
        class ERROR_NO_MORE_DIRECTIONS{};
        class ERROR_OUT_OF_BOUNDS{};

        int GetDepth() {
                return Depth;
        }

        CState() {
                Depth   = 0;

                Grid[1] = Grid1;
                Grid[2] = Grid2;
                Grid[3] = Grid3;
                Grid[4] = Grid4;
                Grid[5] = Grid5;
                Grid[6] = Grid6;
                Grid[7] = Grid7;
                Grid[8] = Grid8;
                Grid[9] = Grid9;

                PrevState = 0;
                OperatorApplyed = NONE;
        }

        void SetPrevState(CState *Set) {
                PrevState = Set;
        }

        CState* GetPrevState() {
                return PrevState;
        }

        bool CanBlankMove(DIRECTION Direction) {
                int Blank = FindBlank();
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
		switch (Direction) {
		case NORTH:
			if (Blank > 3) {
				return true;
			}
			else {
				return false;
			}
			break;

		case EAST:
			if (Blank != 3 && Blank != 6 && Blank != 9) {
				return true;
			}
			else {
				return false;
			}
			break;

		case SOUTH:
			if (Blank < 7) {
				return true;
			}
			else {
				return false;
			}
			break;

		case WEST:
			if (Blank != 1 && Blank != 4 && Blank != 7) {
				return true;
			}
			else {
				return false;
			}
			break;

		default:
			return false;
			break;
		}
}

void setgrid(int index, int value) {
		Grid[index] = value;
}

bool Solution() {

if (Grid[1] == 1 && Grid[2] == 2 && Grid[3] == 3 && Grid[4] == 8 && Grid[5] == 0 &&
			Grid[6] == 4 && Grid[7] == 7 && Grid[8] == 6 && Grid[9] == 5) {

			return true;
		}
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
		else {
			return false;
		}
	}

	char GetGridValue(int Index) {

		if (Index < 1 || Index > 9) {
			throw ERROR_OUT_OF_BOUNDS();
		}
		else {
			return Grid[Index];
		}
	}

	CState(CState* Init) {

		Depth = (Init->GetDepth());

		OperatorApplyed = Init->GetLastOperator();
		PrevState = Init->GetPrevState();

		for (int n=1; n<=9; n++) {
			Grid[n] = Init->GetGridValue(n);
		}
	}

	DIRECTION GetLastOperator() {
		return OperatorApplyed;
	}

	CState(CState* Init, DIRECTION Direction) {

		int n;

		PrevState = Init;

		Depth = (Init->GetDepth() + 1);

		for (n=1; n<=9; n++) {
			Grid[n] = Init->GetGridValue(n);
		}

		MoveBlank(Direction);
	}

	void Print(int x, int y) {

		gotoxy(x,y);   cout << Grid[1] + 0 << Grid[2] + 0 << Grid[3] + 0 << endl;
		gotoxy(x,y+1); cout << Grid[4] + 0 << Grid[5] + 0 << Grid[6] + 0 << endl;
		gotoxy(x,y+2); cout << Grid[7] + 0 << Grid[8] + 0 << Grid[9] + 0 << endl;
	}

	int GetWrongTiles() {
```

EK – 3 (Devam) 8-PUZZLE PROGRAMININ KOD DÜZENİ

```
                return ((Grid[1] != 1) +
                        (Grid[2] != 2) +
                                (Grid[3] != 3) +
                                (Grid[4] != 8) +
                                (Grid[5] != 0) +
                                (Grid[6] != 4) +
                                (Grid[7] != 7) +
                                (Grid[8] != 6) +
                                (Grid[9] != 5) +
                                (Depth     ));
        }

        int GetManhattanDistance() {

                int Result=0;

                Result =        GetDistanceBetween(1, FindBlank(1));
                Result = Result + GetDistanceBetween(2, FindBlank(2));
                Result = Result + GetDistanceBetween(3, FindBlank(3));
                Result = Result + GetDistanceBetween(4, FindBlank(8));
                Result = Result + GetDistanceBetween(5, FindBlank(0));
                Result = Result + GetDistanceBetween(6, FindBlank(4));
                Result = Result + GetDistanceBetween(7, FindBlank(7));
                Result = Result + GetDistanceBetween(8, FindBlank(6));
                Result = Result + GetDistanceBetween(9, FindBlank(5));

                Result = Result + Depth;

                return Result;
        }
};
```

EK – 4 TAM RASGELE YÖNTEMİYLE ELDE EDİLEN TEK YÖNLÜ VARYANS ANALİZİ SONUÇLARI

Variyasyon Kaynağı	Serbestlik Derecesi	Kareler Toplamı	Kareler Ortalaması	F Değeri	P
Gruplar Arası	3	3,35509E+14	1,11836E+14	113,77	0,000**
Hata	1724	1,69469E+15	9,83000E+11	-	-
Toplam	1727	2,03020E+15	-	-	-

p<0,01**

EK – 5 MANTIKSAL RASGELE YÖNTEMİYLE ELDE EDİLEN TEK YÖNLÜ VARYANS ANALİZİ SONUÇLARI

Variyasyon Kaynağı	Serbestlik Derecesi	Kareler Toplamı	Kareler Ortalaması	F Değeri	P
Gruplar Arası	3	8,16690E+14	2,72230E+14	335,27	0,000**
Hata	4325	3,51178E+15	8,11973E+11	-	-
Toplam	4328	4,32847E+15	-	-	-

p<0,01**

ÖZGEÇMİŞ

Kişisel Bilgiler

Soyadı, adı	: BENZER, Ali ihsan
Uyruğu	: T.C.
Doğum tarihi ve yeri	: 08.02.1982 Antakya
Medeni hali	: Evli
Telefon	: 0 (326) 344 25 89
Faks	:
e-mail	: aibenzer@gmail.com.

Eğitim

Derece	Eğitim Birimi	Mezuniyet tarihi
Lisans	Gazi Üniversitesi/ Bilgisayar Öğrt.	2003
Lise	Kırıkhan And. Tic. Mes. Lisesi	1999

İş Deneyimi

Yıl	Yer	Görev
2003-2007	Kırıkhan Teknik ve E.M.L	Öğretmen
2003-2007	Kırıkhan Teknik ve E.M.L	Atölye Şefi

Yabancı Dil

İngilizce

Yayınlar

Hobiler

Bilgisayar teknolojileri, Satranç