Sezgisel Arama (Heuristic Search) 12

Bu metin algoritmaların veri yapıları ile etkileşimine odaklanmıştır. Bu metinde sunulan algoritmaların çoğu arama ve dataları organize etmekle ilgilidir. Böylelikle arama verimli şekilde gerçekleştirilebilir. Pek çok problem bazıları yanlış olan olası bir çok çözümün arasından bir cevap arar. Bazen o kadar fazla olasılık vardır ki tüm olası çözümler arasında doğru olanı verimli şekilde bulacak bir algoritma yazılamaz. Bu durumda arama alanımızdan bazı olasılıkları elemek için bilgisayar biliminde deneme yanılma yöntemi olarak adlandırılan temel kuralları kullanabiliriz. Deneme yanılma yöntemi, olası çözümleri elemese bile olası çözümleri düzenlemeye yardımcı olur. Bu sayede daha iyi olası çözümlere ilk olarak bakabiliriz.
7.bölümde Derin Öncelikli Arama Grafiği sunuldu. Bazen grafikler ve diğer problemler için olan arama alanları o kadar büyük boyutlara ulaşır ki, körü körüne hedef düğüme(node) ulaşmak imkansızdır. Deneme yanılma yönteminin kullanışlı olacağı yer burasıdır. Bu bölüm, aslında bir tür grafik olan bir labirenti aramayı, derinlik öncelikli veya genişlik öncelikli arama ile ilişkili birkaç arama algoritmasını açıklamak için bir örnek olarak kullanır. Bu arama algoritmalarının çeşitli uygulamaları da sunuldu ya da tartışıldı. Deneme yanılma araması genellikle yapay zeka ile ilgili metinlerde ele alınmaktadır[3]. Yapay zekadaki sorunlar daha iyi anlaşıldıkça, zamanla daha yaygın hale gelen algoritmalar ortaya çıkıyor. Bu bölümde sunulan Deneme yanılma algoritmaları yapay zeka metninde daha ayrıntılı ele alındı. Ancak Data boyutları büyüdükçe deneme yanılma aramaları her türlü uygulamada daha gerekli hale gelecektir. Yapay zeka teknikleri birçok arama probleminde faydalı olabilir ve bu nedenle bu bölümde geniş veya sonsuz arama alanlarıyla başa çıkmak üzere tasarlanmış arama algoritmalarına giriş sağlamak amacıyla ele alınmaktadır.

12.1 Bölüm Hedefleri

Bu bölümün sonunda , derinlik öncelikli ve genişlik öncelikli arama örnekleriyle tanıştırılacaksınız. Ayrıca , tepe tırmanışı, en iyi öncelikli arama ve A* (A star olarak telaffuz edilir) algoritması da sunulacaktır. Ek olarak , iki kişilik oyunlarda aramaya yönelik olarak sezgisel yöntemler uygulanacaktır.

Deneme yanılma araması bütün sorunların çözümü olmasa da data boyutları büyüdükçe deneme yanılma daha önemli hale gelir. Bu bölüm performansı arttırmak ve makul bir süre içinde çözülmesi imkansız olan ilginç ve büyük problemleri çözmek için teknikler arasından en azından bazılarını önemli bilgiler için seçmemizi sağlar

12.2 Derin Öncelikli Arama

Derin Öncelikli Arama ile ilk olarak 6.bölümde arama boşluklarını tartışırken ve Derin öncelikli arama ile sudoku bulmacalarının çözümlerini bulurken karşılaştık. Daha sonra 7.

bölümde derinlik öncelikli arama algoritması, döngüleri içeren arama alanlarını ele alacak şekilde biraz genelleştirildi. Bir döngüde sıkışıp kalmamak için, daha önce dikkate alınan köşe noktalarına bakmaktan kaçınmak amacıyla ziyaret edilen bir küme kullanıldı.Grafikler için derinlik öncelikli aramanın biraz değiştirilmiş bir versiyonu 12.2.1`de sunulmaktadır. Bu versiyonda, hedef bulunursa başlangıçtan hedefe giden yol döndürülür. Aksi halde hedefin bulunamadığını belirtmek için boş liste döndürülür.

12.2.1 Yinelemeli Derinlik Bir Grafiğin İlk Araması

(Iterative Depth First Search of a Graph)

```
def graphDFS(G, start, goal):
    # G = (V,E) grafı, düğümler (V) ve kenarlardan (E) oluşur.
    V, E = G
    stack = Stack()
    visited = Set()
    stack.push([start]) # Yığın, yolları içeren bir yığındır.

    while not stack.isEmpty():
        # Yığından bir yol çıkarılır.
        path = stack.pop()
        current = path[0] # Yolun son düğümü alınır.
        if not current in visited:
            # Mevcut düğüm ziyaret edilenler kümesine eklenir.
            visited.add(current)

            # Eğer mevcut düğüm hedef düğümse, arama durdurulur ve hedefe giden yol döndürülür.
            if current == goal:
                return path # Hedefe giden yolu döndür.

            # Aksi takdirde, mevcut düğüme komşu olan her düğüm için:
            # - Eğer düğüm zaten yolda yoksa, yeni yol yığına eklenir.
            # - Eğer düğüm zaten yolda varsa, bu kenar göz ardı edilir.
            for v in adjacent(current, E):
                if not v in path:
                    stack.push([v] + path)

    # Eğer buraya kadar geldiysek, hedef bulunamadı.
    return [] # Boş bir yol döndür.

```

12.2.1 bölümdeki algoritma, başlangıç düğümünden hedef düğüme giden yolu bulan bir while döngüsünden oluşur. Bu yolda bir yön seçimi olduğunda tüm seçimler yığına aktarılır. Tüm seçenekleri zorlayarak, eğer bir yol çıkmaz sokağa çıkıyorsa, algoritma yığına yeni bir şey eklemez. Döngü boyunca bir sonraki sefer, yığından bir sonraki yol açılır ve bu da algoritmanın, gittiği yön konusunda en son karar verdiği noktaya geri dönmesine neden olur.

12.2.2 Labirent Temsili

Labirent nasıl temsil edilmelidir? Veri gösterimi herhangi bir algoritmanın çok önemli bir parçasıdır. Labirent satır ve sütunlardan oluşur. Labirentteki her konumu bir (satır, sütun) tuple'ı olarak düşünebiliriz. Bu tuple'lar, O(1) zamanında arama için bir karma kümesine eklenebilir. Bir karma seti kullanarak, labirent içindeki herhangi bir konum için O(1) zamanında bitişik (satır, sütun) konumları belirleyebiliriz. Bir labirent bir dosyadan

okunduğunda (satır, sütun) çiftleri bir karma kümesine eklenebilir. Daha sonra bitişik fonksiyona bir konum verilmeli ve bitişik konumları belirlemek için labirent karması ayarlanmalıdır.

12.2.3 DFS Örneği

Şekil 12.1'deki labirenti aramayı düşünün. Derinlik öncelikli arama algoritmamızın labirentte arama yaparken mümkünse yukarıya çıkmayı tercih ettiğini varsayalım. Yukarı çıkamıyorsa aşağı inmeyi tercih eder. Bir sonraki tercih labirentte sola gitmek ve ardından son olarak sağa gitmektir. Labirentin tepesinden başladığımızı ve alttan çıkmak istediğimizi varsayalım. Bu bölümde sunulan örneklerde köşegen üzerinde ilerlemenin dikkate alınmadığını unutmayın; aksi halde labirentteki iki köşenin buluştuğu yerde hareket etmek mümkün olacaktır.

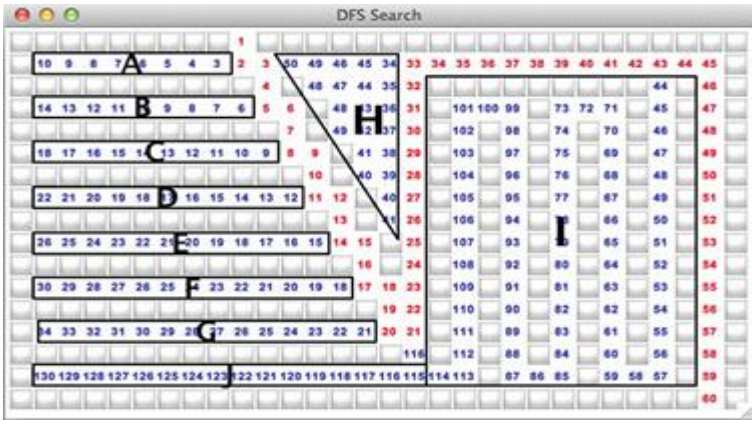


Fig. 12.1 Depth First Search of a Maze (labirentte derinlik öncelikli arama)

Çapraz hareketler, bazı durumlarda labirentteki duvarlara benzeyen şeylerden geçme etkisine sahip olabilir.Yönelim tercihimize göre, algoritma adımları 1 ve 2'yi kırmızı renkte atlayarak ilerler. Daha sonra, Bölge A'ya doğru seyahat etmeye devam eder. Bölge A'da 10'a ulaşıldığında, zaten ziyaret edilmiş olan 10 adımına bitişik olabilecek hiçbir hareket yoktur. Satırlardaki kod 23-25, yığın üzerine itilecek hiçbir şey bulamaz. Ancak, adım 2 ilk düşünüldüğünde, diğer tüm seçenekler, adım 2'nin sağındaki kırmızı üç dahil olmak üzere yığına itilmiştir. Bölge A'daki adım 10'a bakarken yığın üzerine itilecek hiçbir şey olmadığında, yığının üstündeki son değer kırmızı adım 3'tür. Kırmızı adım 3'e bitişik ziyaret edilmemiş düğümler daha sonra yığına itilir. İtinilecek son konum kırmızı adım 4'tür, bu da kırmızı adım 5'in sonraki adım olarak itilmesine ve düşünülmesine yol açar. Ardından, derinlik öncelikli arama tekrar sola devam ederek B bölgesindeki tüm konumları inceler. B bölgesi tükenince, geri izleme tekrar meydana gelir ve bu yol kırmızı adım 6'yı alır. Bu, aramanın bir sonraki adımlarını tüketen D bölgesine girmesine yol açar ve geri izleme adımları, kırmızı adım 12'ye geçmek için gerçekleşir. Benzer şekilde, E, F ve G bölgeleri keşfedilir. Arama kırmızı adım 21'e ulaştığında, derinlik öncelikli arama yukarı gitmeyi tercih

eder ve labirentin en üstüne doğru ilerler ve H bölgesine girer. Labirenti inceleyerek, H bölgesine girilemeyeceğini anlayabiliriz. Ancak derinlik öncelikli arama bunu bilmez veya umursamaz. Sadece hedefe giden bir sonraki olası yolu düşünür ve bu yol hedefe veya tüm olası sonraki adımları tüketene kadar geri izler. H bölgesinden geri izleme, kırmızı adım 34'e ulaşır. Adım 44'e ulaştığımızda, algoritma önce aşağı gitmeyi tercih eder ve bu da I bölgesinden J bölgesine kadar giden bir boşa girişe yol açar, burada olası bir sonraki adım kalmaz ve geri izlemek kırmızı adım 44'e kadar gelir. Sonunda, bu yol hedefe ulaşır.

Bu aramanın dikkat çeken bazı noktaları var. İlk olarak, daha önce de belirtildiği gibi, hedefi sonunda bulmak için geri izleme yaparak kör bir arama yapılmıştır. Bu örnekte derinlik öncelikli arama, labirentteki her konumu inceledi, ancak bu her zaman böyle olmaz. Derinlik öncelikli arama bir çözüm buldu, ancak en iyi çözümü bulmadı. Derinlik öncelikli arama sağa gitmeyi tercih etseydi, bu labirent için çok daha hızlı bir çözüm bulurdu ve en iyi çözümü bulurdu. Ne yazık ki, bu tüm labirentler için işe yaramaz.

Labirent arama alanı sonludur, ancak sonsuz boyutta bir labirenti soldan sağa gitmeye başlarken sağa gitmemiz gerekiyorsa ne olurdu? Algoritma kör bir şekilde sonsuza kadar sola gitmeye devam eder, hiçbir zaman bir çözüm bulmaz. Derinlik öncelikli aramanın dezavantajları aşağıdaki gibidir.

- Derinlik öncelikli arama sonsuz arama alanlarıyla başa çıkamaz, hedefe giden bir yol bulana kadar şanslı olmazsa.
- Her zaman en iyi çözümü bulmaz
- Şanslı olmadıkça, sonlu bir alanda arama sırası, tüm olası yolları denemek için aşırı yıpratıcı olabilir.

Biraz daha iyi bir iş yapabiliriz ya genişlik öncelikli arama ya da sezgisel bir arama kullanarak. Bu algoritmaların nasıl çalıştığını görmek için okumaya devam edin.

12.3 Genişlik Öncelikli Arama

Genişlik öncelikli arama ilk kez 7. Bölümde bahsedildi. Genişlik öncelikli arama kodu, derinlik öncelikli aramadan küçük bir farkla değişir. Bir yığın yerine, alternatif seçenekleri depolamak için bir kuyruk kullanılır. Kod üzerindeki değişiklik küçük olsa da, algoritmanın performansına büyük bir etkisi vardır.

Derinlik öncelikli arama, bir yol hedefe ulaşana veya daha fazla adım atılamayana kadar bir yol boyunca ilerler. Bir yol tükenmiş ve hedefte sona ermiyorsa, geri izleme gerçekleşir. Buna karşılık, genişlik öncelikli arama, başlangıç konumundan tüm yolları aynı anda keşfeder. Bu, her alternatifin sırayla kuyruğa yerleştirilmesi ve ardından her alternatifin sırayla çıkarılması nedeniyledir. Bu, aramanın nasıl ilerlediğini etkiler.

12.3.1 BFS Örneği

Genişlik öncelikli arama, Sektördeki while döngüsünden her geçişte her yol üzerinde bir adım alır. Bu nedenle, Şekil 12.1'deki 2. adımdan sonra iki adet 3. adım gelir. Daha sonra üç adet 4. adım gelir. Sonraki beş adım 6'nın hepsi bir sonraki beş while döngüsü döngüsünde yapılır.

Bu labirentte alternatiflerin sayısının arttığını görebilirsiniz. İlk olarak 2 adet 2. adımdan beş adet 6. adıma kadar arttı. Her adımdaki seçeneklerin sayısı, bir problemdeki dallanma faktörü olarak adlandırılır. Bir dal faktörü bir olursa, bir adımdan diğerine hiçbir seçenek olmadığı anlamına gelir. Bir dal faktörü iki ise, problem her adımda boyut olarak iki katına çıkar.

Çünkü genişlik öncelikli arama her adımda her yöne bir adım atar, iki olan bir dallanma faktörü kötü olurdu. İki olan bir dallanma faktörü, arama alanının boyutunun (tekrar edilen durumlar olmadığını varsayarsak) üssel olarak büyüdüğü anlamına gelir. Genişlik öncelikli arama, hedef düğümün başlangıç düğümünden çok yakın olmadığı sürece bu durumda iyi bir arama değildir.

Şekil 12.2'de gösterilen genişlik öncelikli arama, kör derinlik öncelikli arama tarafından yapılanların neredeyse tamamını kaplar. Sadece birkaç konum ziyaret edilmemiş durumda kalır. Genişlik öncelikli arama, bu labirentin en iyi çözümünü buldu. Aslında, genişlik öncelikli arama, yeterli süre verildiğinde her zaman en iyi çözümü bulur.

Genişlik öncelikli arama ayrıca sonsuz arama alanlarıyla da iyi başa çıkar. Çünkü genişlik öncelikli arama, kaynağı keşfederek aynı anda tüm olası yolları araştırarak dallanır, sonsuza kadar bir yol boyunca sıkışıp kalmaz. Labirente su dökmenin faydalı olabileceğini hayal edebilirsiniz. Su, kaynaktan labirente dolar ve hedefe en kısa yolunu bulur.

Genişlik öncelikli aramanın avantajları ve dezavantajları şöyledir.

- Genişlik öncelikli arama sonsuz arama alanlarıyla başa çıkabilir.
- Genişlik öncelikli arama her zaman en iyi hedefi bulur.
- Problemin dallanma faktörü çok yüksek olduğunda iyi performans göstermeyebilir. Aslında, bazı problemlerde genişlik öncelikli aramayı kullanmak milyonlarca yıl veya daha fazla sürebilir.

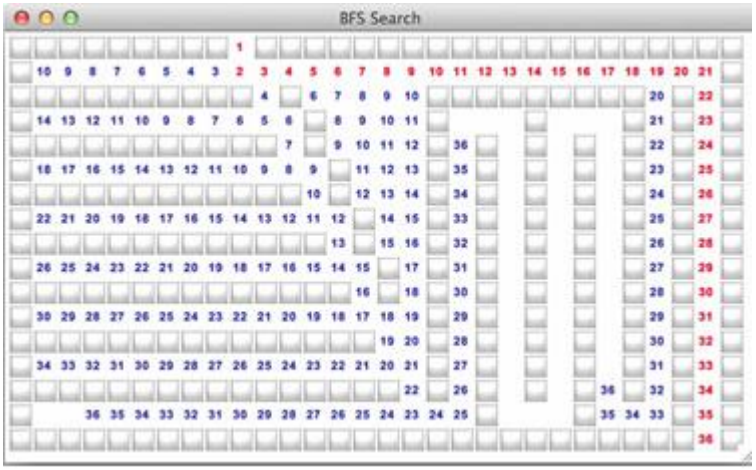


Fig. 12.2 Breadth First Search of a Maze

Problemlere en iyi çözümleri bulmak güzel olabilir ancak genişlik öncelikli arama gerçekte pek pratik değildir. Çoğu ilginç problem yeterince yüksek dallanma faktörlerine sahiptir ki genişlik öncelikli arama pratik değildir.

12.4 Tepe Tırmanışı

Derinlik öncelikli arama, çözümü kör bir şekilde aradığı için pratik değildi. Eğer arama gerçekten körse, bazen şanslı oluruz ve çözümü hızlıca buluruz, diğer zamanlarda ise arama alanının büyüklüğüne bağlı olarak hiçbir zaman çözüm bulamayabiliriz, özellikle sonsuz dallanmalar olduğunda.

Eğer hedefin nerede olduğu hakkında biraz daha bilgi sahibi olsaydık, derinlik öncelikli arama algoritmasını iyileştirebilirdik. Bir dağın zirvesine ulaşmaya çalışmayı düşünün. Dağın zirvesini görebiliyoruz, bu yüzden oraya ulaşmak için izlememiz gereken genel yönü biliyoruz. Tepeye tırmanmak istiyoruz. İşte bu algoritmanın adı buradan gelir.

Dağ tırmanışı yapan herkes, bazen dağa çıkan bir yolun bir çıkmaza götürdüğünü bilir. Bazen, zirveye giden bir yolun sadece yakınlarda bulunan daha küçük bir zirveye götürdüğü görünür. Bu yanıltıcı zirvelere yerelleştirilmiş maksimum denir ve tepe tırmanışı, bir yerelleştirilmiş maksimum bulma ve bunun aranan genel hedef olduğunu düşünme riski taşır.

12.4.1 Tepe Tırmanışı Örneği

Şekil 12.3, aramaya tepe tırmanışı uygulanan aynı labirenti göstermektedir. Tepeyi tırmanmak için bir sezgisel yöntem uygularız. Bir labirenti ararken, çıkış noktasını biliyorsak, her zaman en yakın adıma doğru hareket etmek mantıklı olur. Tepe tırmanışı algoritması da benzer bir şekilde hareket eder. Başlangıç noktasından hedefe doğru ilerlerken, her adımda en yakın komşu düğüme hareket eder. Bu şekilde, tepe tırmanışı, verilen bir başlangıç noktasından hedefe doğru ilerlerken en iyi yol üzerinde tırmanmayı amaçlar.

Ancak, tepe tırmanışı da derinlik öncelikli arama gibi birkaç dezavantaja sahiptir. Özellikle, yerelleştirilmiş maksimumlara yakalanma riski vardır. Eğer başlangıç noktasından hedefe giden en iyi yol, başlangıç noktasının yakınında bulunan bir yerelleştirilmiş maksimumdan geçiyorsa, tepe tırmanışı bu yerelleştirilmiş maksimumda kalabilir ve hedefe ulaşamaz.

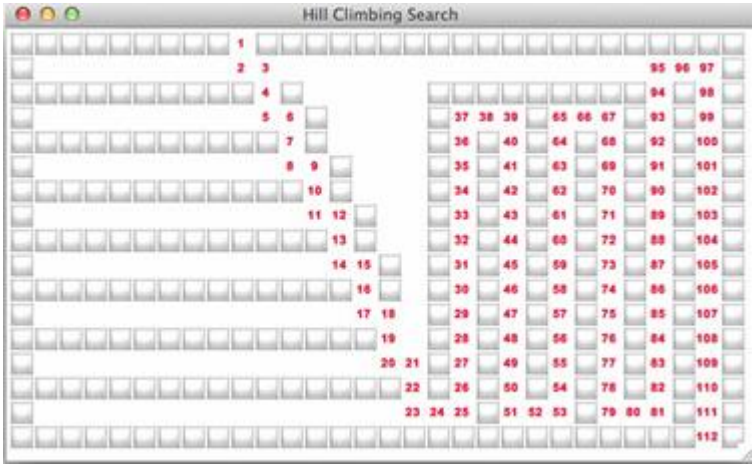


Fig. 12.3 Hill Climbing Search of a Maze

Labirentte hedefe yönlendirilmek için bir sezgisel olarak Manhattan mesafesi kullanabiliriz. Çözüme götürecek yolun uzunluğunu bilmiyoruz çünkü labirentin tüm detaylarını bilmiyoruz, ancak hedefin konumunu ve şu anki konumumuzu biliyorsak, neredeyse hedefe olan mesafeyi tahmin edebiliriz.

Manhattan mesafesi, bir labirent veya harita üzerindeki herhangi iki konumu ayıran satır ve sütun sayısının bir ölçüsüdür. Şekil 12.3'te, başlangıçtan hedefe olan Manhattan mesafesi 36'dır. Bir satır aşağıya inmemiz, ardından 20 sütun sağa gitmemiz ve 15 satır aşağı inmemiz gerekir. Bu mesafe, Manhattan mesafesi olarak adlandırılır çünkü bu, Manhattan'daki binalar arasında veya herhangi bir şehirdeki şehir bloklarında yürümek gibidir.

Manhattan mesafesi ya tam ya da hedefe olan toplam mesafenin bir alt tahmini olacaktır. Şekil 12.3'te bu tam bir tahmin, ancak genel olarak doğrudan bir rota mümkün olmayabilir, bu durumda Manhattan mesafesi bir alt tahmin olacaktır. Bu önemlidir çünkü mesafeyi aşırı tahmin etmek, tepe tırmanışının tekrar derinlik öncelikli arama gibi çalışmasına neden olacaktır. Sezgisel, algoritmanın performansını etkilemeyecektir. Örneğin, mesafemizin her zaman hedeften 100 olduğunu söyleseydik, tepe tırmanışı gerçekleşmeyecektir.

Şekil 12.3'teki örnek, mümkünse önce aşağı gitmeyi tercih ettiğini gösterir. Sonra sağa gider. Hedef konumu bilinir ve minimum Manhattan mesafesi keşfedilecek seçenekleri düzenler. Sol veya yukarı gitmek bir seçenek değildir, başka bir seçenek yoksa. Bu nedenle, algoritma, bu yolu izleyerek 25. adıma kadar aşağı ve sağa doğru ilerler, bu noktada bu yolda başka bir seçeneği olmadığı için yukarı gitmek zorundadır.

Tepe tırmanışı, bir çıkmaza ulaşana kadar bir yoldan vazgeçmeyeceği için derinlik öncelikli arama gibi davranır. Tepe tırmanışı Şekil 12.3'te en iyi çözümü bulmaz, ancak bu durumda genişlik öncelikli veya derinlik öncelikli aramadan çok daha az konumu inceler. Tepe tırmanışının avantajları ve dezavantajları şöyledir.

- Hedefin konumu arama başlamadan önce bilinmelidir.
- Hedefe olan yolun uzunluğunu alt tahmin eden veya tam bir uzunluk sağlayan bir sezgisel olmalıdır. Sezgisel ne kadar iyi olursa, tepe tırmanışı arama algoritması o kadar iyi olur.
- Tepe tırmanışı, büyük arama alanlarında bile iyi performans gösterebilir.
- Sezgisel arama dallanmalarını engelleyebiliyorsa, tepe tırmanışı sonsuz arama dallanmalarıyla başa çıkabilir.
- Tepe tırmanışı yerel maksimumlardan veya zirvelerden etkilenebilir.
- Tepe tırmanışı, genişlik öncelikli arama gibi optimal bir çözüm bulmayabilir.

Tepe tırmanışını uygulamak için her adımdaki alternatif seçenekler, yığına yerleştirilmeden önce sezgisel bir sıraya göre sıralanır. Aksi takdirde, kod, derinlik öncelikli aramanınkiyle tam olarak aynıdır.

12.4.2 Kapalı At Turu

Tepe tırmanışı, kapalı At Turu problemi çözümünde kullanılabilir. Bu problemi çözmek, bir atı satranç oyununda bir satranç tahtası (veya herhangi bir boyutta tahta) etrafında hareket ettirmeyi içerir. At, satranç tahtasında bir L oluşturarak iki kare ilerleyip bir kare dik yönde hareket etmelidir. Kapalı at turu problemi, başlangıç ve bitiş noktası aynı olan ve başlangıç ve bitiş noktası dışında hiçbir karenin iki kez ziyaret edilmediği bir dizi yasal at hareketiyle tahtadaki her konumu ziyaret eden bir yol bulmaktır.

Tahtada bir yol bulmak istediğimizden, çözüm başlangıçtan bitişe bir yol olarak temsil edilebilir. Yol üzerindeki her düğüm, tahtadaki bir hamledir. Bir hamle, tahtada olması ve zaten yolun içinde olmaması durumunda geçerlidir. Bu şekilde, tahta açıkça oluşturulmamış olur.

Bir at için olası hareketleri oluşturmak, tahtanın kenarlarıyla başa çıkmak için kod yazmaya çalışırsanız oldukça karmaşık olabilir. Genel olarak, komşu düğümlerin oluşturulması gerektiğinde ve sınırlar üzerinde özel durumlar ortaya çıktığında, geçerli olmayan hareketlerle birlikte geçerli hareketlerin bir kümesini oluşturmak çok daha kolaydır. Atın etrafında hareket etme durumunda, genel olarak sekiz olası hareket vardır. Tüm olası hareketler oluşturulduktan sonra, geçersiz hareketler açıktır ve filtrelenir. Bu teknik kullanılarak, sınırlar bir kez ele alındığında her bir ayrı olası hareket yerine tutarlı bir şekilde ele alınır. Kod çok daha temiz ve mantık çok daha anlaşılır hale gelir.

Şekil 12.4, 12×12 bir tahta için kapalı at turu problemine bir çözüm sunar. Tur, hesaplanırken turun nerede başladığını ve bittiğini görebileceğiniz şekilde alt sol köşeden başlar. Turu hesaplamak için bir sonraki konum seçeneklerini sıralamadan önce en az kısıtlanmış bir sezgisel uygulandı. En az kısıtlanmış sonraki seçenek, en fazla seçeneğe sahip olan

seçenektir. Bu şekilde, bir sonraki hamle en fazla seçeneğe sahip olduğundan genellikle çıkmazlara götüren yollara bakmamayı sağlar. Başka bir deyişle, en fazla seçeneğe sahip olan yerlere yakın kalmayı

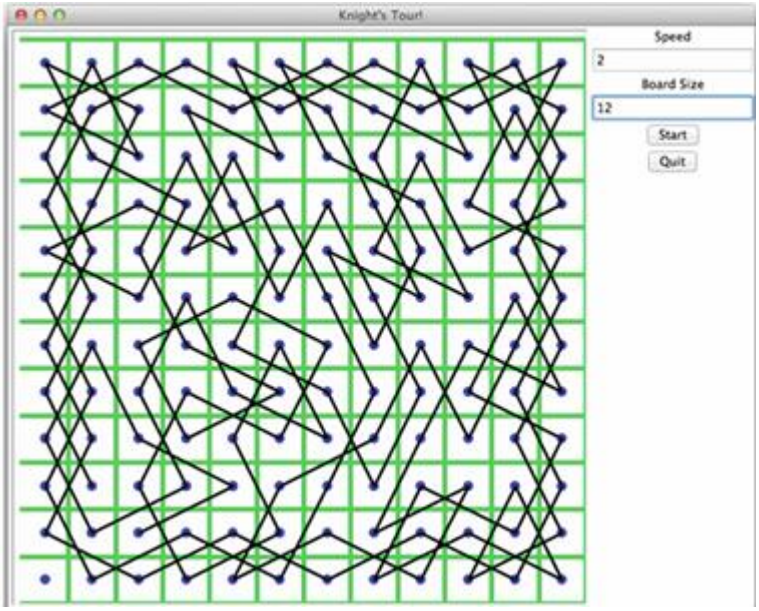


Fig. 12.4 A Closed 12×12 Knight’s Tour

Ve tahtanın ortasında sıkışıp kalmamayı tercih eder. Bu sezgisel mükemmel değil ve çözümü bulmak için hala bazı geri izleme gereklidir. Yine de, bu sezgisel olmadan 12x12 tahta için problemi makul bir sürede çözme umudu olmazdı. Aslında, 8x8 çözümü basit bir derinlik öncelikli arama ile makul bir sürede bulunamaz, her adımda doğru yönde arama şansınız olmadıkça. Sezgisel ve tepe tırmanışı uygulandığında, 8x8 çözümü sadece birkaç saniyede bulunabilir.

12.4.3 Vezirlerin N Problem'i

N-Queens problemi, N vezirin bir N×N satranç tahtasına yerleştirilmesi gerektiğinde, hiçbir iki vezirin aynı sütun, satır veya köşegen üzerinde olmadığı şekilde çözülmelidir. Bu problemi derinlik öncelikli arama kullanarak çözmek çalışmaz. Arama alanı çok büyük olduğundan ve basit kaba kuvvet kullanarak bir çözüm bulmak için çok şanslı olmanız gerekir.

N-Queens problemi, bir vezir tahtaya yerleştirildiğinde, bu vezirin yerleştirildiği sütun, satır veya köşegenlerdeki diğer konumların gelecekteki hamleler için artık mümkün adaylar olmadığı benzersiz bir özelliğe sahiptir. Bu mümkün hamleleri kullanılabilir konumlar listesinden kaldırmak ileri kontrol olarak adlandırılır. Bu ileri kontrol, her adımda arama alanının boyutunu azaltır.

Bir vezirin yerleştirileceği bir sonraki satırın seçimi, N-Queens problemi için başka bir benzersiz özelliktir. Rastgele bir satırın seçilmesi veya sadece sıralı satırlar dizisinden bir sonraki satırın seçilmesi çözümü bulmayı kolaylaştırmaz. Bu nedenle, çözüm sadece bir sonraki veziri hangi sütuna yerleştireceğimizdir.

İleri kontrolde yardımcı olmak için, tahta bir tuple olarak temsil edilebilir: (vezir konumları, kullanılabilir konumlar). Tuple'ın ilk öğesi yerleştirilmiş vezirlerin listesi iken, tuple'ın ikinci öğesi tahtadaki kullanılabilir konumların listesidir. İleri kontrol, bir sonraki satır için kullanılabilir konumlardan birini seçebilir. Bu noktada, tuple'ın ikinci bölümünde, bir sonraki vezir yerleştirmesi seçimiyle çakışan tüm konumlar ortadan kaldırılabilir. Böylece ileri kontrol, bir vezirin yerleştirilmesi için bir seçim yapıldığında artık uygun olmayan tüm olası konumları ortadan kaldırır.

N-Queens problemi'nin çözümünde tepe tırmanışı kısmı, bir vezirin hangi sütuna yerleştirileceği seçildiğinde devreye girer. Seçilen sütun, gelecekteki seçenekleri en az kısıtlayan sütundur. At Turu gibi, N-Queens problemi, bir sonraki seçim yapıldığında daha fazla seçenek bırakan zaman faydalanır. Bu sezgisel kullanılarak, ileri kontrol ve bir sonraki vezirin yerleştirileceği satırın basit seçimi ile, 25-Vezir problemi makul bir sürede çözülebilir. Bir çözüm Şekil 12.5'te gösterilmiştir.

Gözden geçirilecek olursa, tepe tırmanışının uygulanması, her adımdaki alternatif seçeneklerin yığına yerleştirilmeden önce sezgisel bir şekilde sıralanmasını gerektirir. Aksi takdirde, kod, derinlik öncelikli arama ile aynıdır. Bazı durumlarda, At Turu ve N-Queens problemi gibi, herhangi bir çözüm optimal bir çözümdür. Ancak, yukarıda bir labirent ararken belirtildiği gibi, tepe tırmanışı her zaman optimal bir çözüm bulmaz.

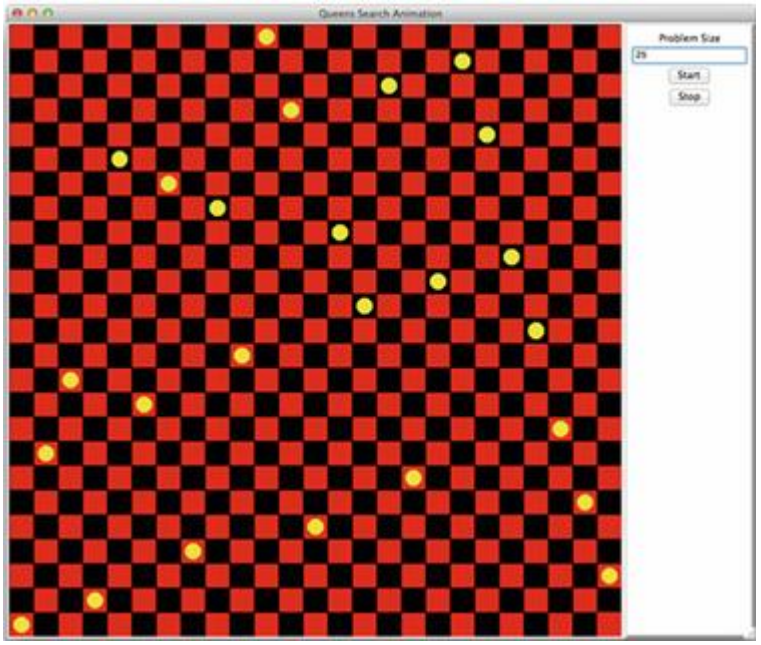


Fig. 12.5 A 25-Queens Solution

12.5 En İyi Öncelikli Arama

Böylece, genişlik öncelikli arama optimal bir çözüm bulabilir ve sonsuz arama alanlarıyla başa çıkabilir, ancak çok verimli değildir ve yalnızca bazı küçük problemlerde kullanılabilir. Tepe tırmanışı daha verimlidir, ancak her zaman optimal bir çözüm bulmayabilir. İkisini birleştirdiğimizde en iyi öncelikli aramayı elde ederiz. En iyi öncelikli aramada, mevcut her düğümün hedefe olan mesafesine göre tüm kuyruğu, tepe tırmanışında kullanılan aynı sezgisel kullanılarak sıralarız.

12.5.1 En İyi Öncelikli Arama Örneği

Şekil 12.6'daki örneği düşünün. Adım 3, bir satır aşağıya, adım 4'e hareket ederek daha da yaklaşıyor. Şimdi, adım 3'ün sağında eşit derecede iyi bir hamle var (aslında optimal çözümü bildiğimiz için daha iyi), ancak adım 5 daha iyi bir adım çünkü nihai hedefe daha yakın. Dolayısıyla, en iyi öncelikli arama, tepe tırmanışı gibi aşağı ve sağa doğru ilerler, ancak kırmızı adıma ulaştığında, hedeften uzaklaşmak zorunda kalır. Bu durumda, kırmızı adım 28, alt kısımdaki mavi adım 24 ile aynı kadar iyi görünüyor. Her ikisinin de Manhattan mesafesi 14'tür. Kırmızı adım 29'a ulaştığımızda, ortadaki labirentteki mavi adım 22 de aynı kadar iyi görünüyor. Hedeften uzaklaşmanın etkisi, tüm yolları aynı anda aramaya başlamaktır. İşte en iyi öncelikli arama böyle çalışır. Hedefe doğru ilerlerken bir yol keşfederken, hedeften uzaklaştıkça birden fazla yol keşfeder.

En iyi öncelikli arama için kod, genişlik öncelikli arama gibi, ancak kuyruktaki olası bir sonraki adımları hedeften tahmini mesafelerine göre sıralamak için bir öncelikli kuyruk kullanılır. En iyi öncelikli aramanın, hedeften uzaklaşırken genişlik öncelikli arama gibi birden fazla yol düşünme avantajı vardır ve hedefe doğru ilerlerken tepe tırmanışı gibi performans gösterir.

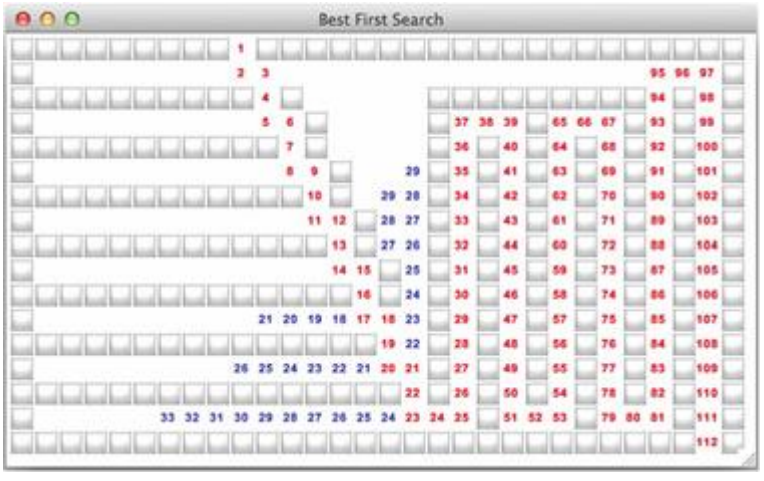


Fig. 12.6 Best First Search of a Maze

Burada gösterilen örnekte, tepe tırmanışından daha iyi bir sonuç elde edemedik. Tabii ki, bu sadece bu örnektir. Genel olarak, tepe tırmanışı, arama alanındaki konumların arama

sırasına bağlı olarak en iyi öncelikli aramadan daha kötü sonuçlar verebilir. Ancak, ne tepe tırmanışı ne de en iyi öncelikli arama, genişlik öncelikli arama gibi optimal bir çözüm bulamadı. Her ikisi de labirentin ortasındaki uzun yola giderken sıkışıp kaldı.

12.6 A* Araması

Bazı yolları, çok uzun gibi görünüyorsa vazgeçebilmek güzel olmaz mıydı? İşte A* algoritmasının temel fikri budur. Bu aramada, bir sonraki seçenekler, hedefe olan mesafenin tahmini (labirent örneklerimizde Manhattan mesafesi) ve şimdiye kadar olan yolun mesafesi tarafından sıralanır. Bu sayede, yollar (bir süre için en azından) hedefe ulaşmak için çok uzun gibi görünüyorsa terk edilir.

12.6.1 A* Örneği

Şekil 12.7'de, aynı yol ilk önce aşağıya ve sağa giderek labirentin alt kısmına, adım 25'e ulaşmaya çalışılır. Ardından, bu yol, kırmızı adımdaki adım 4'teki Manhattan mesafesiyle yolun uzunluğunun toplamından daha iyi olduğu için terkedilir. Tekrar arama aşağıya ve sağa doğru devam eder ve sonunda Şekil 12.1'deki H bölgesini doldurur. Bu noktada, arama, tekrar adım 19'dan üstten devam eder ve tekrar adım 33'e kadar iner, bu noktada kırmızı adım 20, sol taraftaki adım 34'ü almak yerine daha iyi görünür. Arama, mavi yolunu adım 33'te terk eder ve ardından kırmızı adım 20'den hedefe devam eder.

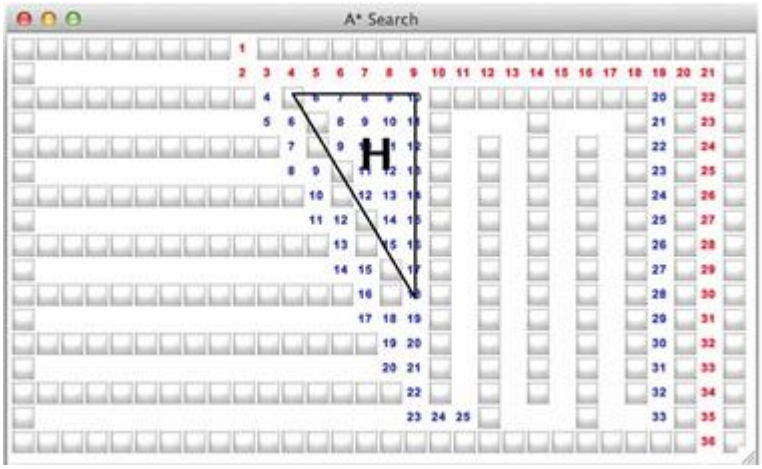


Fig. 12.7 A-Star Search of a Maze

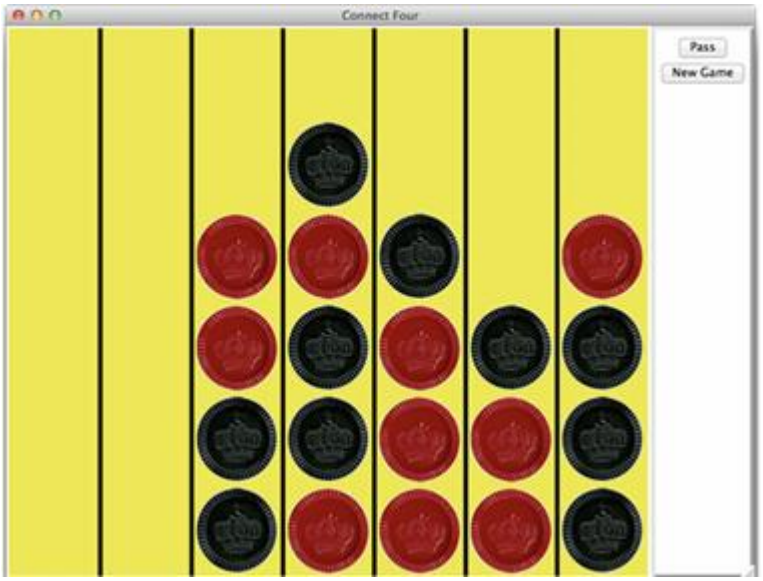
Bu örnekte, A* algoritması, sezgisel ile toplam maliyetin birleşimine dayanarak aşırı derecede uzun olan iki yolunu terk ederek optimal bir çözüm bulur. Ancak, çözümün optimal olması, sezgiselin ve toplam maliyetin doğruluğuna bağlıdır. Özellikle, sezgiselin, mevcut düğümden hedefe ulaşmanın maliyetini doğru bir şekilde yansıtması ve fazla tahminde bulunmaması gerekmektedir.

A* algoritması, Infinite Mario AI yarışması gibi çeşitli alanlarda başarıyla uygulanmıştır. Bu yarışmada, katılımcılara Mario'nun Nintendo oyunu Mario Brothers'ın programlanabilir bir versiyonunda gezinmesi için kod yazma görevi verildi. Makine öğrenimi tekniklerinin yerine, Robin Baumgarten, Mario'nun oyunu yönlendirmek için A* algoritmasını uyguladı. Robin'in çözümünde, Mario, biriktirilen yol uzunluğuna ve hedefe ulaşmanın sezgisel olarak hesaplanan maliyetine dayalı olarak kararlar alır. Bu uygulama sorunu etkili bir şekilde çözdü ve çevrimiçi ortamda popülerlik kazandı. Robin'in çözümü ve geliştirme süreci hakkında daha fazla bilgi edinmek için sağlanan bağlantıyı kullanabilirsiniz: http://aigamedev.com/open/interviews/mario-ai/.

1. Mevcut tahta bilgisayar için bir kazanç durumudur. Bu durumda minimax, bilgisayarın kazanması için 1 döndürür.
2. Mevcut tahta insan için bir kazanç durumudur. Bu durumda minimax, insanın kazanması için -1 döndürür.
3. Mevcut tahta doludur. Bu durumda, ne insan ne de bilgisayar kazanmadığından, minimax 0 döndürür.
4. Maksimum derinlik ulaşılmıştır. Artık arama yapılmadığından tahtayı değerlendirin ve -1.0 ile 1.0 arasında bir sayı raporlayın. Negatif bir değer, bu tahta durumunda insanoğlunun daha olası kazanacağını gösterir. Pozitif bir değer, bilgisayarın daha olası kazanacağını gösterir.

Algoritmanın bu son temel durumunu uygulamak için, minimax algoritmasına yeni bir derinlik parametresi ve muhtemelen başka bazı parametreler de iletilir. Bir oyunda başlangıçta maksimum derinlik çok derin olmayabilir. Ancak, oyun ilerledikçe ve daha az seçenek mevcut olduğunda, maksimum derinlik daha derin olabilir. Arama derinliğini artırmak, bilgisayarın bu tür oyunlarda kazanma yeteneğini artırmanın en iyi yoludur. İyi bir sezgisel de, daha derin arama mümkün olmadığında oyunun erken aşamalarında yardımcı olabilir. İyi bir sezgisel oluşturmak zordur. İşin püf noktası, tahtadaki hareketleri bir şekilde teşvik ederken, hesaplaması nispeten basit tutmaktır.

Bu fikirlere dayalı olarak bir Connect Four uygulaması geliştirdik ve bu uygulama standart donanımda, özel bir çoklu işlem olmadan çalışıyor. Versiyonumuz, ticarette bulunan tüm uygulamaları ve şu anda mevcut olan uygulamaları yeniyor. Eğer ilgilenirseniz, sizin için bir meydan okuma olabilir, daha iyisini yapmak. Bu oyunun ön yüzü, metnin web sitesindeki 20.6. bölümde veya web sitesinde mevcuttur, böylece kendi Connect Four oyununuzu yapabilirsiniz.



12.8 Bölüm Özeti

(heuristik : **tam çözüm bulmak çok zor veya zaman alıcı olduğunda** kullanılan **yaklaşık)**

Bu bölüm, heuristikleri ve arama alanının çok büyüdüğünde kapsamlı bir aramanın yapılamadığı durumlarda nasıl bir rol oynadıklarını ele aldı. Aksi takdirde çözümsüz kalacak birçok problem, heuristikler uygulandığında çözülebilir hale gelir. Ancak, bir heuristiğin doğası gereği, bazen iyi çalışırken diğer zamanlarda iyi çalışmayabilirler. Bir heuristik her zaman çalışıyorsa, bir teknik olarak adlandırılır ve heuristik değildir.
Bir sorunu çözerken gerçekten heuristik aramanın gerekip gerekmediğini dikkatlice düşünmemiz gerekir. Doğru problem temsili, veri yapısı veya algoritmayı seçmek, kaba kuvvet yaklaşımından ve bir heuristiğin uygulanmasından çok daha önemlidir. Çözülmesi gereken çok büyük görünen bir sorunun doğru algoritma ile azaltılabilir hale gelebileceği bir durum olabilir. Böyle bir azaltma mümkün olmadığında, heuristik arama cevap olabilir.
Arama algoritmaları tırmanma, en iyi önce ve A* en iyi derinlik öncelikli arama ve en iyi önce arasındaki farkı akılda tutarak en iyi derinlik öncelikli arama, derinlik öncelikli arama ve genişlik öncelikli arama ile karşılaştırılarak en iyi hatırlanmalıdır. Tırmanma, yeni eklenen düğümleri yığına eklemek için bir heuristik uygulandığından derinlik öncelikli arama gibi. En iyi önce, tüm sıradaki düğümlerin heuristiğe göre düzenlendiği genişlik öncelikli aramaya benzer. Genellikle bir öncelikli kuyrukla uygulanır. A* algoritması, kuyruğun hedefe heuristik olarak tahmin edilen mesafe ile şimdiye kadar yapılan mesafenin toplamına göre düzenlendiği en iyi öncelikli aramaya benzer. Minimax algoritması da arama alanı çok büyük olduğunda bir heuristik kullanır. Etkili bir oyun motoru her zaman mümkün olduğunca derin arayacak, ancak aramanın kesilmesi gerektiğinde, bir hamlenin oyundaki değerini tahmin etmede iyi bir heuristik yardımcı olacaktır.

Tabii, işte metnin Türkçe çevirisi:

12.8 İnceleme Soruları

Bu kısa yanıtlı, çoktan seçmeli ve doğru/yanlış soruları yanıtlayarak bölümü ne kadar iyi anladığınızı test edin.

1. Hangisi daha hızlı, derinlik öncelikli arama mı yoksa genişlik öncelikli arama mı?
2. Hangi arama, derinlik öncelikli mi yoksa genişlik öncelikli mi, bazı durumlarda tamamlanmayabilir? Bu ne zaman olabilir?
3. Tepe tırmanışı sırasında, algoritmanın bir hedef düğüm bulmasını engelleyebilecek şey nedir?
4. En iyi öncelikli arama algoritması bir optimal çözüm bulur mu? Neden veya neden olmasın?
5. A* algoritması bir optimal çözüm bulur mu? Neden veya neden olmasın?
6. Tepe tırmanışı, ne zaman A* algoritmasından daha iyi kullanılabilir?
7. İleri kontrol nedir ve bir sorunu nasıl çözmeye yardımcı olur?

8. Geri izleme gerçekleştiğinde derinlik öncelikli arama algoritmasında ne olur? Geri izlemenin gerçekleştiği noktada algoritmanın davranışı hakkında özel bilgi verin.
9. Poker dışında minimax algoritması kullanılamayan bir oyun adı verin.
10. Minimax'in istenildiği gibi davranmasını sağlamanın en iyi yolu nedir: gerçekten iyi bir sezgi mi yoksa daha derin bir arama mı?

12.10 Programlama Problemleri

1. Bu bölümdeki beş arama algoritmasını kullanarak bir labirent araması yapacak bir program yazın. Örnek labirentler oluşturmak için her boşluk labirentte bir açık konumu ve her boşluk olmayan karakter labirentte bir duvarı temsil eden bir metin dosyası yazın. Labirentin satır ve sütun sayısını dosyanın ilk iki satırında başlatın. Labirenti üstten alta doğru arayarak bir çıkış bulun. Labirentinizde yalnızca bir giriş ve bir çıkış olmalıdır. Farklı algoritmaların ve örnek labirentlerinizdeki performanslarını karşılaştırın ve farklarını ortaya koyun. Sonuçlarınızı görselleştirmek için metnin web sitesinden labirent arama ön ucu indirin. İletişim için ön uç ve arka uç kodlarınız arasındaki mimari, ön uç program dosyasında sağlanmıştır.

2. Şövalyenin Turu problemi çözen bir program yazın. Aramanızda arama alanını daraltmak için bir sezgi kullanmaya özen gösterin. 8x8 bir tahta için turu hızlı bir şekilde çözebildiğinizden emin olun. Çözümünüzü turtle grafikleri kullanarak çizin.

3. N-Vezirler problemi çözen bir program yazın. N-Vezirler problemi için ileri kontrolü ve sezgisel bir çözümü kullanın. Ekstra bir zorluk için bunu 8x8 bir tahta için çözmeyi deneyin. Programın bu biri çözmek için (yarım saat mi?) Bir süre çalışması muhtemeldir. Sonucunuzu görselleştirmek için metnin web sitesinde sağlanan N-Vezirler ön uç kodunu kullanın. Yazdığınız arka uç kodu, ön uç program dosyasının en üstünde sunulan mimariyi izlemelidir.

4. Başka bir öğrencinin Connect Four'ını zorlayacak Connect Four programını yazın. Her ikinizin de bir geçiş düğmesine sahip programlar yazması gerekir. Hangisinin

önce gideceğini belirlemek için bir para atışı yapabilirsiniz. İlk sırayı alan oyuncu, geçiş düğmesine basarak başlamalıdır. Sonra siz ve diğer öğrenci, bilgisayar programlarınız yarışırken birbirinizin arasında gidip gelebilirsiniz. İşlerin akışını sağlamak için, oyununuzun 30 saniye içinde bir hamle yapması gerekir; aksi takdirde kaybedersiniz. Ön uç olarak Sunum 20.6'da sunulan kodu kullanabilirsiniz. Arka uç kodunu yazmalısınız. Ön uç koduyla iletişim kurmak için sunum dosyasının en üstünde sunulan mimariyi izleyin.

5. Ek bir zorluk için, Connect Four programını yazın ve metin web sitesinde yazarlar tarafından sağlanan programı yenin. Yazarların kodunu çalıştırmak için Python sürüm 3 ve Common Lisp kurulu olmalıdır. Yazarın ön uç ve arka uç kodları, yazarın programını çalıştırmak için aynı dizinde veya klasörde olmalıdır. Yazarın programının sürümünü metnin web sitesinden alabilirsiniz.