ANADOLU ÜNİVERSİTESİ

Bilim ve Teknoloji Dergisi A-Uygulamlı Bilimler ve Mühendislik
Cilt: 15 Sayı: 1 2014
Sayfa: 69-80

ARAŞTIRMA MAKALESİ / **RESEARCH ARTICLE**

Muzaffer DOĞAN [1]

SOLO TEST OYUNU ÜZERİNDE KUYRUK LİSTESİ İLE BİR PARALEL ÖNCE-DERİNE ARAMA ALGORİTMASI

ÖZ

Solo Test oyununda tablada tek taşın kaldığı duruma ulaşmak için takip edilmesi gereken hamleler, DFS algoritmasıyla günümüz bilgisayarlarında kısa sürede bulunabilmektedir. Bu sürenin kısaltılması amacıyla bu makalede paralel işlemenin avantajları kullanılmaya çalışılmıştır. Paralel DFS algoritmalarında karşılaşılan, ortak kaynaklara erişimde kullanılan kilitlerin kapatılıp açılması ve bir düğümün çocuklarının işlenmeden kardeşlerinin işlenmesi durumlarında ortaya çıkan problemler, *Kuyruk Listesi* adı verilen veri yapısının kullanımı ile aşılmaya çalışılmıştır ve paralel olmayan DFS çözümüne oranla daha kısa sürelerde çözüme ulaşılmıştır. Deneylerde Solo Test oyununun İngiliz versiyonu kullanılmıştır.

Anahtar Kelimeler: Kuyruk Listesi, Paralel Önce-Derine Arama, Oyun Programlama, Solo Test.

A PARALLEL DEPTH-FIRST SEARCH ALGORITHM USING LIST OF QUEUES ON PEG SOLITAIRE GAME

ABSTRACT

For the Peg Solitaire game, existing computers can easily compute the moves required to obtain the solution board with one peg in a short time by applying the DFS algorithm. In order to shorten the solution time, advantages of parallel processing has been used in this paper. The problems that occur during locking and unlocking while accessing shared resources and processing sibling nodes before child nodes are solved by introducing *List of Queues* data structure and shorter execution times has been obtained compared to non-parallel DFS solution. English version of the Peg Solitaire game has been used in the experiments.

Keywords: List of Queues, Parallel Depth-First Search, Game Programming, Peg Solitaire.

[1] Anadolu Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği, İki Eylül Kampüsü, 26555, Eskişehir.
E-mail: E-posta: muzafferd@anadolu.edu.tr

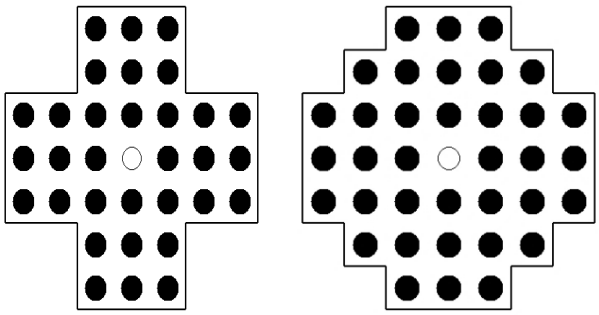
Geliş: 21 Mayıs 2014 **Düzeltme:** 21 Temmuz 2014 **Kabul:** 31 Temmuz 2014

1. GİRİŞ

Solo Test, $n \times n$ boyutlu delikli bir tabla üzerinde tek oyuncu tarafından oynanan bir oyundur. Taşlar tablaya genellikle + şeklinde dizilir ve merkezdeki delik boş bırakılır. Taşlar belli bir başlangıç yerleşimiyle tablaya dizildikten sonra her hamlede bir taş, arkasında delik olan, yatay veya dikeydeki komşu bir taşın üzerinden atlatılarak deliğe konur ve üzerinden atlanan taş tabladan alınır. Tabla üzerinde tek taş kalana kadar aynı şekilde oynanmaya devam edilir (Kendall ve ark., 2008).

Solo Test hakkındaki ilk kayıt, 1697 yılında Fransa kralı XIV. Louis'nin sarayında Claude Auguste Berey tarafından Soubise prensesi Anne de Rohan-Chabot'yu Solo Test oynarken resmeden gravüre kadar uzanır. *Mercure Galant* isimli Fransız edebiyat dergisinin Ağustos 1697 sayısında Solo Test oyunu ve kuralları hakkında bilgiler verilmiştir. Bu, Solo Test hakkında tarihte rastlanan ilk basılı kaynaktır. (Peg Solitaire, 2014).

En bilinen Solo Test tablaları İngiliz ve Avrupa (veya Fransız) tablalarıdır (Peg Solitaire, 2014). Bu tablaların başlangıç dizilişleri Şekil 1'de gösterilmiştir.



(a) (b)

Şekil 1. En bilinen Solo Test tablaları: (a) İngiliz tablası (b) Avrupa (Fransız) tablası.

Beasley (1985), Solo Test tablası üzerindeki 200 problemin çözümünü 550 diyagram üzerinde gösteren geniş kapsamlı bir kitap yazmıştır. Uehara ve Iwata (1990) Solo Test probleminin çözülebilir ve NP-Tam olduğunu göstermiştir. Moore ve Eppstein (2002), tek satırla sınırlandırılmış bir Solo Test probleminin polinom zamanda çözülebileceğini göstermiştir. Ravikumar (2004) ispatı genişleterek $k \times n$ boyutuyla sınırlanan Solo Test probleminin doğrusal zamanda çözülebileceğini göstermiştir. Kiyomu ve Matsui (2001) tamsayı programlama algoritmalarını ve Matos (1998) Önce-Derine Arama (Depth-First Search = DFS) algoritmasını Solo Test'e uygulamıştır. Aynı taşın arka arkaya diğer taşlar üzerinden atlatılması tek hamle olarak sayıldığında, Solo Test'i en az hamle ile çözen ilk kişi 1912 yılında Ernest Bergholt olmuştur (Peg Solitaire, 2014) ve 18 hamleli bu çözümün en kısa çözüm olduğunu 1964 yılında Beasley ispatlamıştır (Beasley, 1985).

Solo Test oyununda tablada tek taş kalması durumuna ulaşma problemi, DFS yöntemiyle çözülebilir (Matos, 1998). Problemin daha kısa sürede çözülmesi için DFS yöntemi paralelleştirilebilir (Saad ve ark., 2012). Fakat üzerinde DFS yönteminin kullanılacağı durum ağacının en başta belli olmaması ve ağaç düğümleri ziyaret edildikçe sonraki düğümlerin hesaplanıyor olması, DFS yönteminin paralelleştirilmesini zorlaştırmaktadır. Paralel DFS algoritmasında genel olarak kullanılan yöntem, her işlemcinin ağacın bir düğümünü işleyerek alt düğümleri hesaplaması ve bu işlem bittikten sonra diğer işlemcilerden DFS sıralamasında ilk gelen ve henüz işlenmemiş bir düğümü işleyerek çözüme ulaşmaya çalışmasıdır (Rao ve Kumar, 1987).

Solo Test'in paralel olmayan DFS çözümünün paralelleştirilmesi için Saad ve ark. (2012), Hesaplama Ağacı Mantığı (Computation Tree Logic = CTL) üzerinde bir model geliştirmiştir. Bu modele göre bütün işlemciler aynı fonksiyonu çalıştırırlar ve çalıştırılan fonksiyon ziyaret edilen durumları bir S listesinde, ziyaret edilecek durumları ise bir W yığıtında tutarak ziyaret edilmemiş durumları işleyerek bu durumları takip eden olası durumları hesaplar ve çözüme ulaşmaya çalışır.

Rao ve Kumar (1987) yönteminde, henüz işlenmemiş bir düğüm bir işlemci tarafından istendiğinde ve DFS sıralamasında o düğümden daha önce gelen bir düğüm o sırada başka bir işlemci tarafından işlenmekteyse, DFS sıralamasından sapma ve çözüme geç ulaşma olasılığı ortaya çıkmaktadır. Bu durumda ya başka bir çözüme daha kısa sürede ulaşılır, ya da DFS sıralamasıyla hedeflenen ilk gelen çözüme daha geç sürede ulaşılır. Saad ve ark. (2012) yönteminde ise bütün işlemciler tarafından paylaşılan S listesi ve W yığıtının çakışmayı engellemek amacıyla kilitlenmesi ve açılması işlemci sayısı arttıkça çözüm süresini uzatabilecektir.

Bu makalede, Solo Test probleminin Paralel DFS yöntemiyle çözümünde DFS sıralamasına riayet edilerek ve bir Kuyruk Listesi (List of Queues) veri yapısı kullanılarak paylaşılan kaynakların kilitlenmesi ve açılması sırasında kaybedilen zamanın azaltılmasına çalışılacaktır. Yöntemin etkinliğinin gösterilmesi için paralel olmayan DFS çözümü ve yığıt kullanan Paralel DFS çözümü ile karşılaştırmalar yapılacaktır. Deneylerde kullanılan programlar MS Visual Studio 2012 ortamında C# 5.0 dili ile nesne yönelimli programlama ile geliştirilecektir.

2. PARALEL OLMAYAN DFS ÇÖZÜMÜ

Solo Test probleminin paralel olmayan DFS yöntemiyle çözümü için durum ağacının düğümleri bir yığıt üzerinde tutulur. Bir düğüm ziyaret edilirken o düğüme ait Solo Test durumunu takip eden olası durumlar hesaplanır ve yığıta eklenir. Her seferinde yığıtın en tepesindeki durum alınarak işlenir ve taş sayısı 1 olan durum bulununcaya veya yığıttaki eleman sayısı sıfır oluncaya kadar devam edilir. Paralel olmayan DFS çözümünün algoritması Şekil 2'de gösterilmiştir.

Paralel olmayan DFS algoritmasında kullanılan **ComputeChildBoards** fonksiyonu, Solo Test tablası üzerindeki bütün hücreler üzerinde dolaşır ve dolu bir hücredeki taşı sırasıyla kuzeye, batıya, güneye ve doğuya doğru hareket ettirmeye çalışır. Geçerli bir hamle yapılabilmesi için denenen yöndeki bir sonraki hücrenin dolu, ondan sonraki hücrenin ise boş olması gerekir. Geçerli bir hamle bulunursa bu hamle sonunda oluşan tabla nesnesi oluşturularak bir listeye eklenir ve bütün tablanın dolaşılması sonucunda oluşan ve tabladan sonraki olası hamleleri içeren liste geri döndürülür. Bahsedilen işlemi yapan **ComputeChildBoards** fonksiyonunun algoritması Şekil 3'te listelenmiştir.

```
1  function Stack NonParallelSolver(S : Stack)
2      while (S is not empty) do
3          b ← S.Pop() ;
4          if (b.PegCount = 1) then // Çözüm bulundu
5              return CreateSolutionStack(b)
6          children ← ComputeChildBoards(b) ;
7          forall child in children do
8              S.Push(child)
9      return NULL
```

Şekil 2. Paralel olmayan DFS çözümünün algoritması.

```
1  function List ComputeChildBoards(b : PegSolitaireBoard)
2      directions ← new List(NORTH, WEST, SOUTH, EAST) ;
3      children ← new List() ;
4      for row ← 0 to 6 do
5          for col ← 0 to 6 do
6              if (b.CellAt(row, col) = FILLED) then
7                  forall dir in directions do
8                      child ← b.CreateChildByMove(row, col, dir) ;
9                      if (child is not NULL) then
10                         children.Add(child)
11     return children
```

Şekil 3. Bir Solo Test tablasında yapılabilecek hamleleri liste halinde hazırlayan ComputeChildBoards isimli fonksiyonun algoritması.

Çözüme hangi hamleler yapılarak ulaşıldığını takip edebilmek için çözüme varana kadar işlenen tablalar bir yığıt içerisine kaydedilerek **NonParallelSolver** fonksiyonu tarafından geri döndürülmektedir. Bulunun çözümden geri gidebilmek için Solo Test tablasını temsil eden **PegSolitaireBoard** sınıfının içerisine **Parent** isimli bir üye değişken eklenmiştir. Başlangıç durumu en altta, bulunan çözüm en üstte olacak şekilde bir yığıtın oluşturulması için kullanılan öz-yinelemeli **CreateSolutionStack** isimli fonksiyonun algoritması Şekil 4'te listelenmiştir.

Matos (1998), bir taşı hareket ettirmek için denenen yönlerin sıralamasının değiştirilmesiyle farklı çözümlere ulaşılacağını ve bu çözümlere ulaşmak için 20.278 veya 7.667.769 hamle yapılması gerektiğini hesaplamıştır. Hamleler Şekil 3'te verilen yön sıralamasıyla hesaplandığında 7.667.769 hamle sonunda çözüme ulaşılacaktır.

3. KUYRUK İLE PARALEL DFS ÇÖZÜMÜ

Rao ve Kumar'ın (1987) önerdiği Paralel DFS yöntemini gerçeklemek için, henüz test edilmemiş durumları tutan bir yığıt oluşturulmuştur. İşlemciler işleyecekleri durumu bu yığıttan alırlar ve aldıkları duruma ait bütün hamleleri hesaplayıp yığıta atarlar. Taş sayısı az olan tablalar yığıtın hep en tepesinde olacağı için, işlemciler durum ağacının o ana kadar tespit edilen halinin en derin seviyesindeki durumunu işlemiş olurlar. Böylece Önce-Derine Arama felsefesine uygun bir arama yapılmış olur. Bahsedilen arama yöntemini gerçekleyen algoritma Şekil 5'te listelenmiştir.

```
1  function Stack CreateSolutionStack(b : PegSolitaireBoard)
2      if (b.Parent = NULL) then
3          S0 ← new Stack() ;
4          S0.Push(b) ;
5          return S0
6      else
7          S ← CreateSolutionStack(b.Parent) ;
8          S.Push(b) ;
9          return S
```

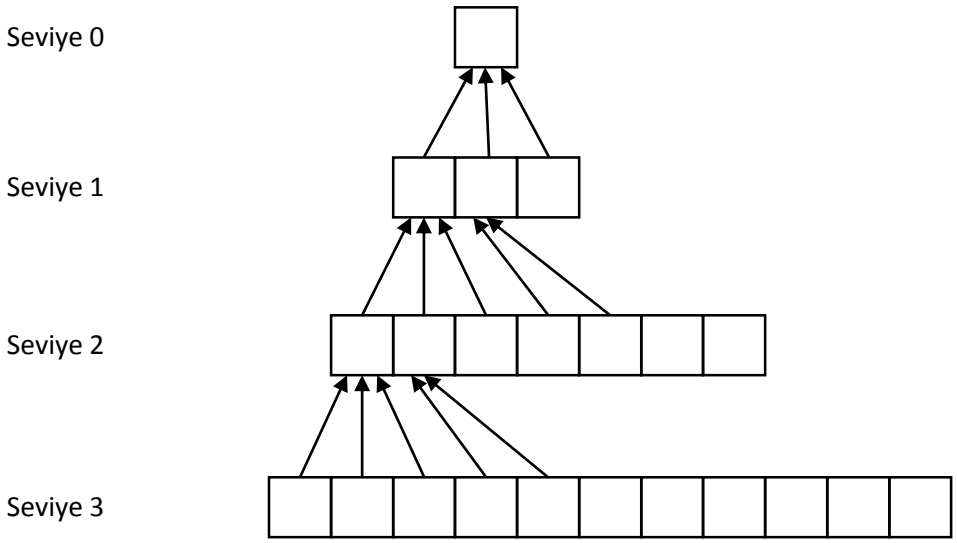
Şekil 4. Bulunan çözüme varana kadar takip edilmesi gereken hamleleri saklayan yığıtı oluşturan algoritma.

```
1  S ← new Stack() ;
2  solution_found ← FALSE ;
3  soln ← new PegSolitaireBoard()
4  function Stack ParallelSolver(b0 : PegSolitaireBoard)
5      S.Push(b0) ;
6      forall processor in ProcessorList do
7          processor.StartThread(ProcessorWork)
8      ProcessorList.WaitAll ;
9      return CreateSolutionStack(soln)
10
11 function void ProcessorWork()
12     while (solution_found = FALSE) do
13         lock(S) ;
14         if (S.Count > 0) then
15             b ← S.Pop()
16         unlock(S) ;
17         if (b.PegCount = 1) // Çözüm bulundu
18             solution_found ← TRUE ;
19             soln ← b ;
20             return
21         else
22             ComputeAndPushChildBoards(b)
23     return
24
25 function void ComputeAndPushChildBoards(b : PegSolitaireBoard)
26     directions ← new List(NORTH, WEST, SOUTH, EAST) ;
27     for row ← 0 to 6 do
28         for col ← 0 to 6 do
29             if (b.CellAt(row, col) = FILLED) then
30                 forall dir in directions do
31                     child ← b.CreateChildByMove(row, col, dir) ;
32                     if (child is not NULL) then
33                         lock(S) ;
34                         S.Push(child) ;
35                         unlock(S)
36     return
```

Şekil 5. Kuyruk ile Paralel DFS algoritması.

4. KUYRUK LİSTESİ İLE PARALEL DFS ÇÖZÜMÜ

Şekil 5'te verilen ve Kuyruk ile Paralel DFS algoritması ilk bakışta Solo Test problemini kısa zamanda çözecek gibi görünse de, işlemci sayısı arttıkça Önce-Enine Arama (Breadth-First Search = BFS) yöntemine yakınsayacaktır. Çünkü işlemcilerden biri durum ağacının en derindeki elemanını işleyip çocuklarını hesaplarken, başka bir işlemci kuyruğun en derinindeki elemanını isteyecek ve önceki işlemcinin işlediği düğümün kardeşi olan bir durumu işlemeye başlayacaktır. Sonuç olarak program, çok fazla sayıda durumun kontrol edildiği ve paralel olmayan DFS çözümünden kat kat uzun süren bir sürece girecektir. Bu problemin çözümü için, durum ağacındaki bir durumun bütün çocukları işlenmeden bir sonraki kardeşinin işlenmesinin engellenmesi gerekir. Bunu sağlamak için bu makalede bir *Kuyruk Listesi* (List of Queues = LoQ) veri yapısı kullanımı önerilecektir.



Şekil 6. Her seviyenin ayrı bir kuyrukta tutulduğu Kuyruk Listesi veri yapısı.

Solo Test problemine ait durum ağacı, bellekte bir ağaç şeklinde tutulmak yerine, Şekil 6'da gösterildiği şekilde, durum ağacının aynı seviyesindeki elemanları aynı kuyrukta tutulacaktır. Şekildeki kutular durumları, oklar ise durumlar arasındaki ebeveyn ilişkisini göstermektedir. Seviye 0'daki kuyruğun tek elemanı, Solo Test oyunundaki başlangıç tablası olacaktır. Başlangıç tablasının çocukları bir işlemci tarafından hesaplanacak ve Seviye 1'deki kuyruğa eklenecektir. Seviye numaraları ile Solo Test tablasındaki taş sayısı arasında bir ilişki kurulabilir. Başlangıç tablasında 32 taş vardır. Herhangi bir seviyedeki tablada bir hamle yapıldığında taş sayısı 1 azalır ve bir alt seviyeye düşülür. Dolayısıyla, herhangi bir seviyedeki seviye numarasıyla taş sayısının toplamı 32 yapacaktır. Çözüme ulaşıldığında tablada 1 taş kalacağından, çözümün 31 numaralı seviyede bulunacağı hesaplanabilir.

İşlemciler, işleyecekleri durumu almak için Kuyruk Listesinin en alt seviye kuyruğundan başlayacak ve eleman sayısı sıfırdan büyük olan en alt seviyeli kuyruğun sıradaki elemanını "kuyruktan çıkarmadan" işlemeye başlayacaktır. Kuyruktan çıkarmama hususu, çocuklar bitmeden kardeşlerin işlenmemesi kısıtı bakımından önemli bir husustur. Kardeşlerin işlenmesini engellemek için kuyruğun başındaki durum kuyruktan çıkarılmamalı ve bu durumun işleniyor olduğuna dair bir işaret konulmalıdır. Bu amaçla Solo Test tablasını ifade etmekte kullanılan **PegSolitaireBoard** sınıfına tablanın işlenme durumunu gösteren yeni bir üye değişken eklenmelidir. Bu üye değişkenin 3 olası değeri olacaktır: **İşlenmedi** (NotProcessed), **İşleniyor** (Processing) ve **Bitti** (Finished).

Bir işlemci, kuyruk listesinden işleme durumu **İşlenmedi** olan bir tablayı alır almaz tablanın işleme durumunu **İşleniyor** yapmalı, tablanın bütün çocuklarını teker teker oluşturup bir alt seviyedeki kuyruğa ekledikten sonra ise işleme durumunu **Bitti** yapmalıdır.

İşlemcilerden biri, Kuyruk Listesinden alması gereken tablanın işleme durumunun **İşleniyor** olduğunu gördüğünde hiçbir şey yapmamalı ve en alt seviyeli kuyruktan tekrar başlayarak daha alt seviyeli tablaların hazırlanmasını beklemelidir. İşlenmekte olan tablanın çocukları, sorumlu işlemci tarafından teker teker hazırlanıp bir alt seviyeli kuyruğa **İşlenmedi** durumuyla, peyderpey eklenecektir.

İşleme durumu **Bitti** olan bir tablaya rastlandığında ise, tablanın işlenmeyi bekleyen çocukları olup olmadığı kontrol edilmeli ve bütün çocuklarının işlenmesi tamamlandıysa tabla kuyruktan çıkarılmalıdır. Çocuk sayısı sıfır olduğunda, tablanın bütün çocukları ve torunları kontrol edilmiş ve çözüme ulaşılamamış demektir ve kardeş düğümlerden aramaya devam edilmelidir.

Her seviyenin ayrı bir kuyrukta tutulmasının avantajlarından biri de, her birinin ayrı ayrı kilitlenebilmesi sayesinde farklı kuyruklar üzerinde işlemcilerin aynı anda işlem yapabilecek olmasıdır. Yani işlemcilerden biri ikinci seviyedeki kuyruğa tabla eklerken, aynı anda başka bir işlemci, üçüncü seviyedeki kuyruğa hiçbir çakışma olmadan eleman ekleyip çıkarabilecektir. Paylaşılan kaynakların birbirinden ayrılması, işlemcilerin paralel çalışmalarını kolaylaştıracaktır.

Paylaşılan bir kaynak bir işlemci tarafından kilitlendiğinde, aynı kaynak üzerinde işlem yapmak isteyen başka bir işlemci, önceki işlemci kaynağı serbest bırakana kadar bloke olur. Bu yüzden Kuyruk Listesinden alınacak bir sonraki tabla aranırken, kuyrukların "kilitlenmesi" yerine "kilitli olup olmadığının kontrol edilmesi" performansı arttıracaktır. Eğer P_i işlemcisi, işleyeceği bir sonraki tablayı aramak amacıyla en alt seviyeli kuyruktan başlayarak, içerisinde en az 1 eleman bulunan en alt seviyeli kuyruğu ararken L seviyesindeki kuyruğun kilitli olduğunu gördüyse, başka bir işlemci bu kuyrukta başka bir işlemci işlem yapıyor demektir ve P_i işlemcisinin bu kuyruktaki işlemin tamamlanmasını beklemesine gerek yoktur. L seviyesinde işlem yapmakta olan bir $P_j, j \neq i$ işlemcisi 3 işlemden birini yapıyor olabilir: 1) P_i işlemcisi gibi, işleyeceği bir sonraki tablayı aramak, 2) Kuyruğa yeni bir eleman eklemek, 3) Kuyruktan sıradaki elemanı çıkarmak. Birinci durum söz konusuysa, P_i işlemcisi $L-1$ seviyesine geçerek aramasına devam etmelidir. İkinci ve üçüncü durumlarda ise en alt seviyeye dönerek aramaya yeniden başlamalıdır. Fakat bu durumlarda $L-1$ seviyesine geçerse, henüz işlenmekte olan bir tablaya rastlayacaktır ve arama işlemi başarısız olacağı için en alt seviyeden aramaya yeniden başlayacaktır. Her halükârda, P_i işlemcisi $L-1$ seviyesine geçerek aramaya devam etmelidir.

C# dilinde bir kaynağın kilitlenmesi için **lock** bloğu veya dengi olan **Monitor.Enter()** metodu kullanılmaktadır. **Monitor.TryEnter()** metodu ise kaynağın kilitli olup olmadığını test edebilmektedir. **Monitor.TryEnter()** metodu, kaynak kilitli değilse kaynağın kullanım iznini alıp kilitlemekte, kaynak kilitliyse içinde bulunulan izleği (Thread) bloke etmeyip parametre olarak aldığı **bool** türündeki bir değişkeninin değerini **false** yapmaktadır. Bu yüzden paylaşılan bir kaynağa hangi amaçla erişilmeye çalışıldığı göz önünde bulundurularak uygun kilit metodu kullanılmalıdır. C# dilindeki **lock** ve **Monitor.TryEnter()** kullanımına dair bir örnek Şekil 7'de gösterilmiştir.

```
object queueLocker = new object();
lock (queueLocker)
{
    Board b = Q.Dequeue();
}
```

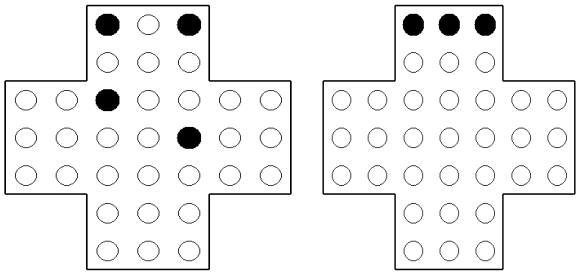
(a)

```
object queueLocker = new object();
bool locked = false;
Monitor.TryEnter(queueLocker, ref locked);
if (locked)
{
    Board b = Q.Dequeue();
    Monitor.Exit(queueLocker);
}
```

(b)

Şekil 7. C# dilinde lock bloğu ve Monitor.TryEnter() metodunun kullanımı. (a) Kilitleme başarısız olursa izlek bloklanır. (b) Kilitleme başarısız olursa izlek bloklanmaz ve çalışmaya devam eder.

Solo Test durum ağacının en alt seviyelerinde artık daha fazla hamle yapılamayacak durumlar sıklıkla ortaya çıkmaktadır. Örneğin Şekil 8'deki gibi birbiriyle komşu olmayan dolu hücreler veya tablanın bir kenarına sıkışmış 3 yan yana taş durumlarında başka hamle yapılamaz ve oyun sona erer. Bu tür durumlarla sıklıkla karşılaşılan seviyelerde, kardeş düğümlerin işlenmesi için çocuk düğümlerin işlenmesini beklemek, programın çalışma süresini uzatacaktır. Bu yüzden Kuyruk Listesi ile Paralel DFS çözümünün içerisine, hangi seviyeye kadar kardeşlerin işlenmesi için çocukların işlenmesinin bekleneceği, **maxDFSDepth** isimli bir parametre olarak eklenmiştir. **maxDFSDepth** seviyesinin altında kalan seviyelerdeki kuyruklardaki tablalar, işlenme durumlarına bakılmaksızın kuyruktan çıkartılıp işlenecektir.



(a) (b)

Şekil 8. Başka hamle yapılamayacak durumlara örnekler. (a) Dolu hücreler komşu değil. (b) Birbirine komşu 3 taş, tablanın en üst satırına sıkışmış.

```
1  LOQ ← new List<Queue<PegSolitaireBoard>>() ;
2  lockerLevels ← new object[32] ;
3  maxLevel ← 0
4  solution_found ← FALSE ;
5  soln ← new PegSolitaireBoard()
6
7  function Stack LOQ_Solver(b0 : PegSolitaireBoard, maxDFSDepth : int)
8      maxLevel ← maxDFSDepth ;
9      for i ← 0 to 31 do
10         LOQ.Add(new Queue<PegSolitaireBoard>()) ;
11         lockerLevels[i] = new object()
12     AddIntoLOQ(b0) ;
13     forall processor in ProcessorList do
14         processor.StartThread(ProcessorWork)
15     ProcessorList.WaitAll ;
16     return CreateSolutionStack(soln)
17
18 function void ProcessorWork()
19     while (solution_found = FALSE) do
20         b ← DequeueFromLOQ() ;
21         if (b is not NULL) then
22             if (b.Level = 31) then
23                 solution_found ← TRUE ;
24                 soln ← b ;
25                 return
26             else
27                 ComputeAndEnqueueChildBoards(b) ;
28                 b.State ← FINISHED
29                 // If necessary, remove b from children of its parent:
30                 while (b.Level >= maxLevel AND b.ChildCount = 0) do
31                     b.Parent.RemoveChild(b) ;
32                     b ← b.Parent
33     return
34
35 function void ComputeAndEnqueueChildBoards(b : PegSolitaireBoard)
36     directions ← new List(NORTH, WEST, SOUTH, EAST) ;
37     for row ← 0 to 6 do
38         for col ← 0 to 6 do
39             if (b.CellAt(row, col) = FILLED) then
40                 forall dir in directions do
41                     child ← b.CreateChildByMove(row, col, dir) ;
42                     if (child is not NULL) then
43                         AddIntoLOQ(child)
44     return
```

Şekil 9. Kuyruk Listesi ile Paralel DFS algoritması.

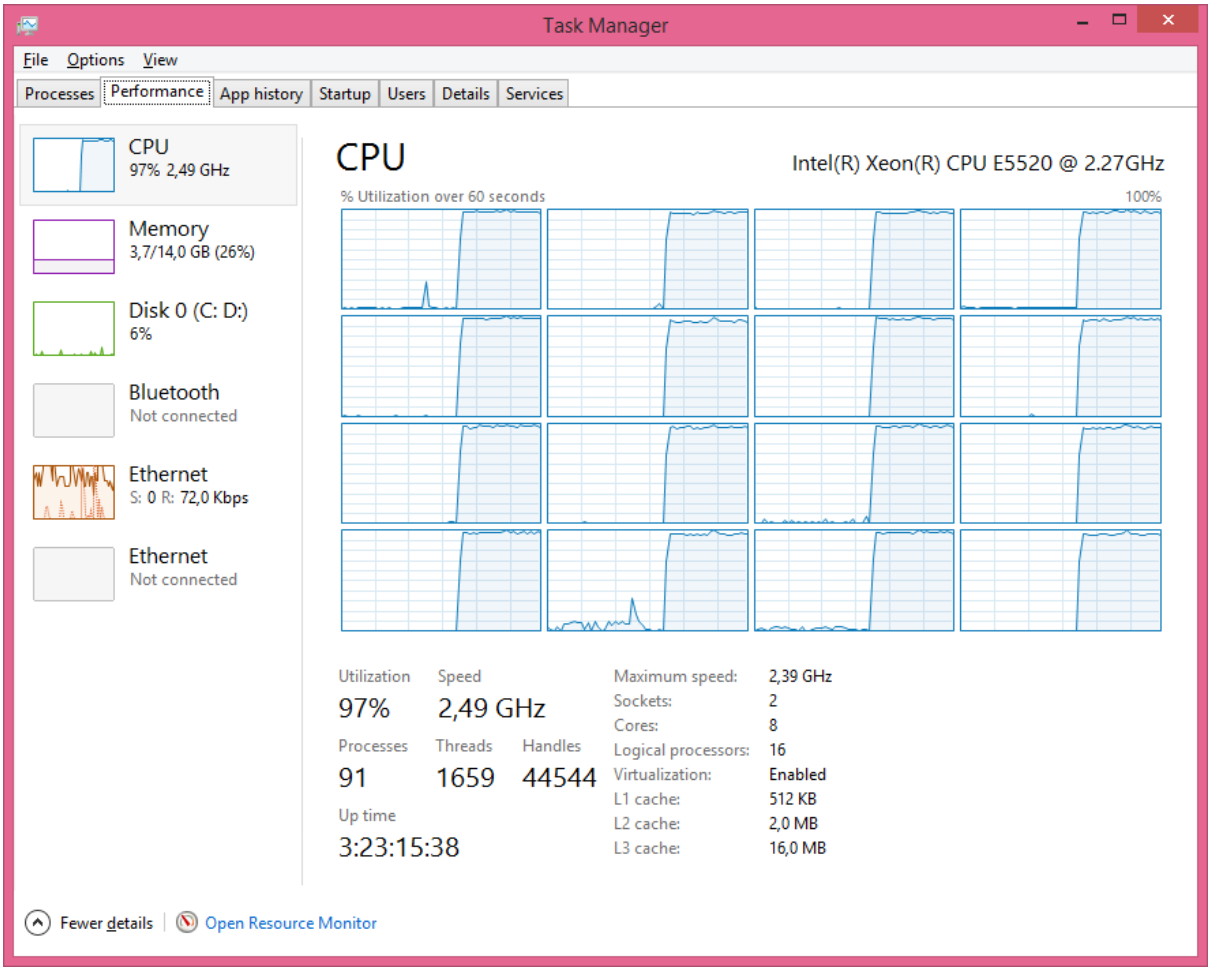
Şekil 9'da verilen algoritmanın en önemli kısmı olan ve paylaşılan kaynakların yönetildiği, Kuyruk Listesine eleman ekleme (**AddIntoLOQ**) ve Kuyruk Listesinden eleman çıkarma (**DequeueFromLOQ**) fonksiyonları ise Şekil 10'da listelenmiştir.

```
1  function void AddIntoLOQ(b : PegSolitaireBoard)
2      lock(lockerLevels[b.Level]) ;
3      q ← LOQ[b.Level] ;
4      q.Enqueue(b) ;
5      unlock(lockerLevels[b.Level]) ;
6      return
7
8  function PegSolitaireBoard DequeueFromLOQ()
9      board ← NULL ;
10     maxLevelInLOQ ← 31 ;
11     flag ← true ;
12     while (flag) do
13         lockTaken ← FALSE ;
14         takenLockNdx ← -1 ;
15         Monitor.TryEnter(lockerLevels[maxLevelInLOQ], ref lockTaken) ;
16         if (lockTaken) then
17             takenLockNdx ← maxLevelInLOQ ;
18             q ← LOQ[maxLevelInLOQ] ;
19             if (q.Count > 0) then
20                 board ← q.Peek() ; // Take the board without removing it
21                 if (board.Level >= maxDFSDepth) then
22                     board ← q.Dequeue() // Dequeue the board now
23                 else
24                     if (board.State = NOTPROCESSED) then
25                         board.State ← PROCESSING
26                     elseif (board.State = PROCESSING) then
27                         board ← NULL
28                     elseif (board.State = FINISHED) then
29                         if (board.ChildCount = 0) then
30                             q.Dequeue() ;
31                             board.Parent.RemoveChild(board) ;
32                             board ← NULL
33                 flag ← FALSE ;
34                 // Release previously taken lock:
35                 Monitor.Exit(lockerLevels[takenLockNdx])
36             elseif (maxLevelInLOQ > 0) then // Continue with previous level:
37                 maxLevelInLOQ ← maxLevelInLOQ – 1
38             else // LOQ is empty, wait for it to be filled by other processors
39                 flag ← FALSE
40         else // Lock can't be taken, continue with previous level:
41             if (maxLevelInLOQ > 0) then
42                 maxLevelInLOQ ← maxLevelInLOQ – 1
43             else // LOQ is empty, wait for it to be filled by other processors
44                 flag ← FALSE
45     return board
```

Şekil 10. Kuyruk Listesine eleman ekleme (AddIntoLOQ) ve Kuyruk Listesinden eleman çıkarma (DequeueFromLOQ) fonksiyonları.

5. DENEYLER

Önerilen algoritmanın test edilmesi için 2,27 GHz hızında, 2 fiziksel işlemcili ve her işlemcide 8 çekirdek (core) bulunan, toplam 16 çekirdekli Intel Xeon E5520 işlemcili bir bilgisayar kullanılmıştır. 16 çekirdeğin tamamının kullanıldığı bir andaki Görev Yöneticisi ekranı Şekil 11'de verilmiştir.



Şekil 11. Deneylerin yapıldığı bilgisayara ait Görev Yöneticisi görüntüsü.

Kullanılan bilgisayarda, paralel olmayan DFS çözümü ortalama 12.145 ms. sürmüştür ve bu süre içerisinde yığıttaki toplam 7.667.770 durum işlenmiştir. Belirtilen durum sayısı, yığıta ilk eklenen başlangıç durumu hariç tutulduğunda Matos'un (1998) hamle sayısına karşılık gelmektedir. Bir milisaniyede işlenen durum sayısı 631 civarındadır.

Kuyruk ile Paralel DFS çözümünde tek işlemci kullanıldığında ortalama 12.970 ms. sonunda çözüme ulaşılmıştır. Paralel olmayan DFS çözümüyle arasındaki fark olan 825 ms. ise, paylaşılan kaynaklara erişimi kontrol altında tutmak için kullanılan kilitlerin açılıp kapanması için harcanan süredir. İşlemci sayısı arttığında çözüme ulaşamama olasılığı artmaktadır. Çünkü durum ağacındaki bir düğümün çocukları henüz hesaplanmadan, kardeşleri başka işlemciler tarafından işlenebilmektedir. Bu yüzden işlenen durum sayısı 60 milyon ile sınırlandırılarak deneyler yapılmıştır. Kuyruk ile Paralel DFS yöntemine ait bazı deney sonuçları Tablo 1'de verilmiştir.

Tablo 1. Kuyruk ile Paralel DFS çözümüne ait deney sonuçları.

İşlemci Sayısı	Deney No	Durum Sayısı	Süre (ms.)	Durum/Süre (adet/ms)
1	1	**6.667.770**	**12.970**	**514**
2	1	**34.615.623**	**37.495**	**923**
	2	60.000.000	65.671	913
	3	60.000.000	106.475	563
4	1	**42.210.055**	**74.615**	**565**
	2	60.000.000	100.279	598
	3	**3.244.493**	**5.717**	**567**
8	1	60.000.000	112.995	530
	2	**16.092.203**	**28.672**	**561**
	3	60.000.000	107.299	559
16	1	60.000.000	130.459	460
	2	60.000.000	129.669	462
	3	60.000.000	131.380	456

Tablo 1'deki sonuçlara göre işlemci sayısı arttıkça birim zamanda işlenen durum sayısının da arttığı görülmektedir. Birim zamanda işlenen durum sayısı 2 işlemcide maksimuma ulaşmakta, işlemci sayısı 2'nin üzerine çıktığında ise azalmaktadır. Tek işlemci ile 6.667.770 durumun test edilmesi sonucunda çözüme ulaşılabilirken, işlemci sayısı arttığında daha az sayıda veya daha fazla sayıda durumun test edilmesiyle de çözüme ulaşılabilmektedir. Farklı deneylerde ulaşılan çözümler aynı olmak zorunda değildir. Matos'un (1998) bahsettiği, takip edilen yönlerin değiştirilmesi ile 20.278 hamle sonucunda elde edilen çözüme ulaşılabileceği gibi, şans eseri uygun yönlerin takip edilmesiyle 6.667.769 hamle sonucunda ulaşılan çözüme de ulaşılabilmektedir. İşlemci sayısı 16'ya çıktığında ise 60 milyon hamle içerisinde sonuca ulaşmak neredeyse mümkün olmamaktadır, çünkü bu durum BFS yöntemine yaklaşmaktadır. Ayrıca 16 işlemci kullanıldığında, paylaşılan alanların kilitlenmesi ve açılması için geçen süre arttığından ve işlemciler uzun süre boş kaldığından, birim zamanda işlenen durum sayısı tek işlemcili durumun bile altına inmektedir.

Kuyruk Listesi ile Paralel DFS çözümü için farklı sayıda işlemci ve maksimum DFS derinliği kullanılarak yapılan deneyler sonucunda 6.667.770 civarında durumun test edilmesi sonucunda makul sürelerde sonuca ulaşılmıştır. Her bir durum için arka arkaya yapılan 3 deneyin ortalama süreleri Tablo 2'de gösterilmiştir.

Tablo 2. Kuyruk Listesi ile Paralel DFS Çözümüne ait çözüme ulaşma süreleri (ms.).

		İşlemci Sayısı					
		16	**12**	**8**	**4**	**3**	**2**
Max. DFS derinliği	**31**	50.442	38.760	31.510	24.400	20.801	20.489
	28	47.062	31.582	31.775	21.802	15.478	18.103
	24	19.031	14.801	11.489	8.140	7.207	9.270
	20	8.504	7.723	7.354	7.541	8.583	9.999
	19	8.510	8.025	7.682	7.708	8.609	9.994
	18	8.359	7.671	7.697	7.166	**6.067**	9.000
	17	8.385	8.181	7.806	7.750	8.572	9.994

Tablo 2'nin ilk satırı, yani maksimum DFS derinliğinin 31 olduğu durum, hiçbir kardeş düğümün çocuk düğümden önce işlenmediği durumdur ve işlemci sayısı arttıkça paylaşılan kaynakların yönetimi için kullanılan kilitler sebebiyle süre uzamaktadır.

Deneylerde maksimum DFS derinliği 17'ye kadar azaltılmıştır, çünkü durum ağacının 16'ncı seviyedeki ilk düğümü altında ilk çözüme ulaşılmaktadır. Maksimum DFS derinliğinin 16 ve aşağısına ayarlanması, başka çözümlere ulaşılmasına veya ilk çözüme daha uzun sürede ulaşılmasına sebep olmaktadır.

Maksimum DFS derinliğinin azalmasıyla çözüme genellikle daha kısa sürelerde ulaşıldığı görülmektedir. Bunun sebebi, bu derinliğin altındaki düğümlerin, işlenmekte olan bir düğümün kardeşlerinin işlenmesinin beklenmemesi sebebiyle hiçbir işlemcinin boşta kalmamasıdır. 20'nin altındaki maksimum DFS derinliklerinde, paralel olmayan DFS çözümünden daha kısa sürelerde çözüme ulaşılmıştır.

İşlemci sayısı arttığında, kilitleme işlemleri sebebiyle çözüme ulaşma süresi de uzamaktadır. Optimum süre ise maksimum DFS derinliğinin 18, işlemci sayısının 3 olduğu durumda elde edilmiştir. Bu durumda, paralel olmayan DFS çözümündeki sonucun yaklaşık yarısı kadar bir sürede çözüme ulaşılmıştır.

6. SONUÇLAR

Solo Test oyununda tablada tek taşın kaldığı çözüme ulaştıran hamleleri hesaplama problemi, DFS algoritması ile günümüz bilgisayarlarında kısa sürede çözülebilmektedir. Fakat bu yöntemin paralelleştirilmesine çalışıldığında, ortak kaynakların senkronize bir şekilde kullanılması için kullanılan kilitlerin açılıp kapanması ve bir duruma ait olası hamlelerle oluşan çocuk düğümlerin hesaplanması için geçen zaman içerisinden işlemcilerin boş kalması sebebiyle etkin bir iyileşme sağlanamamaktadır.

Bir düğümün çocukları işlenmeden kardeşlerinin işlenebildiği **Kuyruk ile Paralel DFS** algoritmasında çözüme en fazla 4 işlemci ile en hızlı şekilde ulaşılabilmektedir, fakat işlemci sayısı arttıkça çözüme kısa sürede ulaşma olasılığı azalmaktadır. Bunun sebebi, bir düğümün çocukları bir işlemci tarafından işlenmekte iken, düğümün kardeşlerinin diğer işlemciler tarafından işlenmesi sonucu, DFS sıralamasında daha önce işlenmesi gereken alt-ağaçların işlenmesinin gecikmesidir.

Bir düğümün çocuklarından önce kardeşlerinin işlenmesini engellemek amacıyla bu makalede **Kuyruk Listesi ile Paralel DFS** adı verilen bir yöntem önerilmiştir. Durum ağacının her seviyedeki düğümlerinin ayrı birer kuyrukta tutulması esasına dayanan bu yöntemde, her kuyruk için farklı bir kilit kullanıldığı için paylaşılan kaynaklara erişim de hızlanmıştır. Maksimum DFS derinliği parametresi ile belirlenen bir seviyenin altındaki düğümler için kardeş önceliğinin iptal edilmesiyle işlemcilerin boşta kalma süreleri azaltılmış ve maksimum DFS derinliği 20 ve altı olduğunda, paralel olmayan DFS çözümünden daha kısa sürelerde çözüme ulaşılabilmiştir. Optimum çözüm ise maksimum DFS derinliğinin 18, işlemci sayısının 3 olduğu durumda elde edilmiştir.

İleriki zamanlarda Kuyruk Listesi veri yapısı bir kütüphane haline dönüştürülerek DFS ile çözülebilen diğer problemlerin paralelleştirilmesi için kullanılabilecektir.

KAYNAKLAR

Beasley, J. D. (1985). The Ins and Outs of Peg Solitaire. *AMC*, *10*, 12.

Kendall, G., Parkes, A. J., & Spoerer, K. (2008). A Survey of NP-Complete Puzzles. *ICGA Journal*, 31(1), 13-34.

Kiyomi, M., & Matsui, T. (2001). Integer Programming Based Algorithms for Peg Solitaire Problems. In *Computers and Games* (229-240). Springer Berlin Heidelberg.

Matos, A. (1998). Depth-first Search Solves Peg Solitaire. Technical Report DCC-98-10, Universidade do Porto. Available from http://www.dcc.fc.up.pt/Pubs/treports.html, last accessed 16 July 2014.

Moore, C. and Eppstein, D. (2002). One-Dimensional Peg Solitaire, and Duotaire. *More Games of No Chance* (ed. R. Nowakowski), number 42, 341–350. Cambridge University Press, New York, NY.

Peg Solitaire. http://en.wikipedia.org/wiki/Peg_solitaire [last accessed 16 July 2014].

Rao, V. N., & Kumar, V. (1987). Parallel Depth First Search. Part I. Implementation. *International Journal of Parallel Programming*, *16*(6), 479-499.

Ravikumar, B. (2004). Peg-solitaire, String Rewriting Systems and Finite Automata. *Theoretical Computer Science*, *321*(2), 383-394.

Saad, R. T., Dal Zilio, S., & Berthomieu, B. (2012). An Experiment on Parallel Model Checking of A CTL Fragment. In *Automated Technology for Verification and Analysis* (284-299). Springer Berlin Heidelberg.

Uehara, R., & Iwata, S. (1990). Generalized Hi-Q is NP-complete. *IEICE TRANSACTIONS (1976-1990)*, *73*(2), 270-273.