6.4.2 DFS: depth first search

todo

6.4.2.1 at tur problemi

- → at turu: knights tour (KT)
- → KT yapbozu, oyun tahtasında tek bir atla oynanır
- → atın gidebileceği potansiyel yerler neresidir?
- → oyun tahtasındaki her bir kutucuk yalnızca bir kez ziyaret edilir
- → her bir ziyaret ardışıllığı “tur” adlanır

tur olasılıkları

- → 8x8 oyun alanında geçerli tur sayısının üst sınırı 1305 x 10^35'dir
- → bunun dışında bir sürü geçersiz turcuk vardır
- → bir sürü çözümü olsa da en kolay ifade edilebileni/çözülebileni
- → programlanabileni çizge aramadır

algoritmik aşamalar

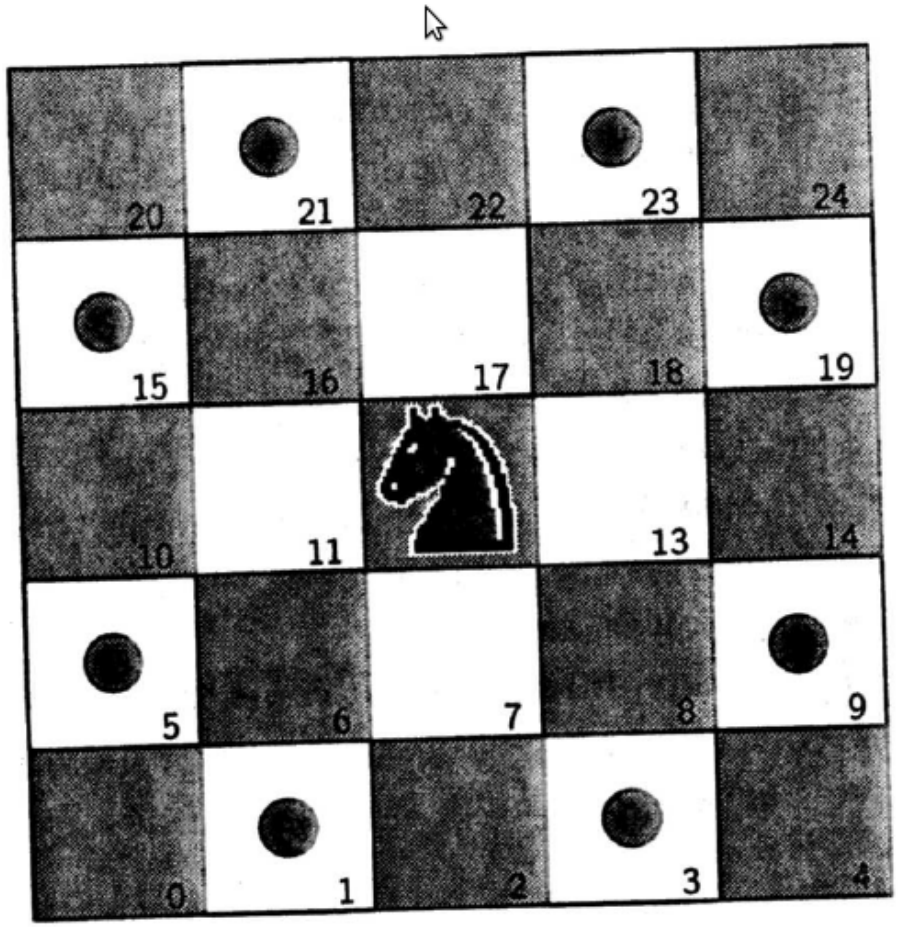
- → oyun tahtasında geçerli hareketlerin çizgeyle temsili
- → satırxsütun boyutlarındaki çizgede geçerli yolları bulmak için çizge algoritması
- → bu algoritma her bir düğümü yalnızca bir kez ziyaret etmelidir

temsil

→ oyun alanındaki her bir kare çizgede bir düğümle temsil edilir

→ her bir gerçerli hareket çizgede bir kirişle temsil edilir

temsil



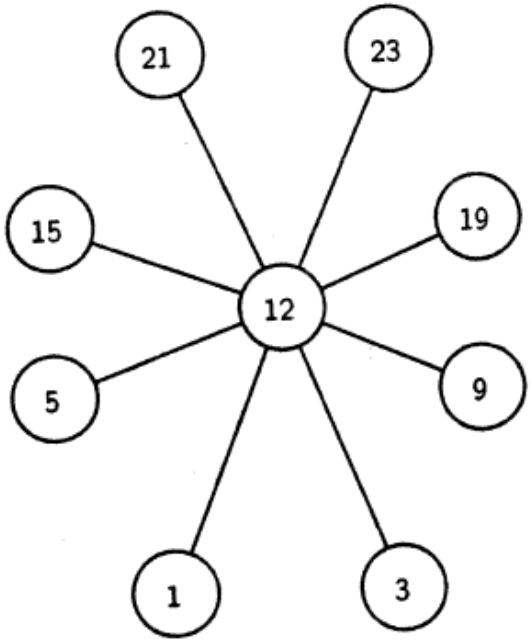


Figure 6.9: Legal Moves for a Knight on Square 12, and the Corresponding Graph

→ kare: düğüm, geçerli hareket: kiriş

gerçekleme

```
1       def knightGraph(bdSize):
2           ktGraph = Graph()
3           # Build the graph
4           for row in range(bdSize):
5               for col in range(bdSize):
6                   nodeId = posToNodeId(row,col,bdSize)
7                   newPositions = genLegalMoves(row,col,bdSize)
8                   for e in newPositions:
9                       nid = posToNodeId(e[0],e[1])
10                      ktGraph.addEdge(nodeId,nid)
11          return ktGraph
```

açıklamalar

- → s4-5: satır x sütün boyutlarında bir oyun alanı, burada satır=sütun=dbSize
- → s6: posToNodeId, alt satırdan başlayarak, sola doğru artarak ilerleyen ve
- → sıfırla başlayan Id'ler (en alt sol:0, hemen sağı 1 gibi)

```
def posToNodeId(x, y, bdSize):
    return x + y * bdSize
```

- → s7: genLegalMoves işlevi, şu anki düğümden gidilebilecek potansiyel düğümler listesini döndürür
- → s8-10: nodeId ile olası tüm geçerli nid'ler arasında kiriş oluşturur

gerçekleme

gerçekleme

```
1       def genLegalMoves(x,y,bdSize):
2           newMoves = []
3           moveOffsets = [(-1,-2),(-1,2),(-2,-1),(-2,1),
4                          ( 1,-2),( 1,2),( 2,-1),( 2,1)]:
5           for i in moveOffsets:
6               newX = x + i[0]
7               newY = y + i[1]
8               if legalCoord(newX,bdSize) and \
9                                   legalCoord(newY,bdSize):
10                  newMoves.append((newX,newY))
11          return newMoves
12
13      def legalCoord(x,bdSize):
14          if x >= 0 and x < bdSize:
15              return True
16          else:
17              return False
```

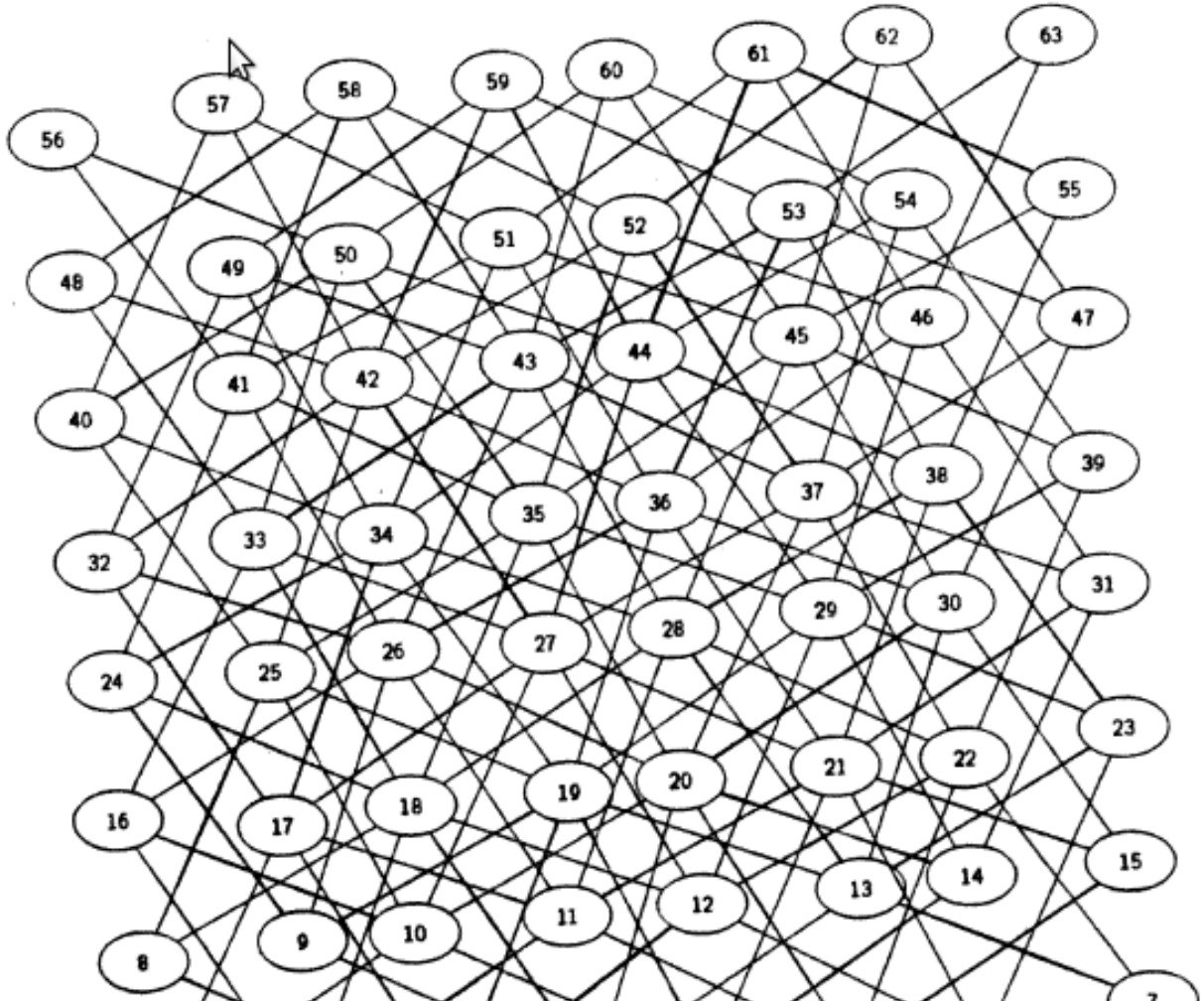
açıklamalar

açıklamalar

- → s2: geçerli hareketler bir listede tutulacak
- → s3-4: bulunulan noktadan (x, y), geçerli hareket ofsetleri
- → eksen olarak x'in pozitif değerleri sağda y'nin pozitif değerleri de üstte
- → s7-8: sorgulanan koordinat oyun alanının dışına çıkmadıysa

örnek

- → bdSize = 8 iken knightGraph() ile üretilen çizge
- → 336 kiriş
- → olası kiriş sayısı: 4096
- → doluluk oranı: 336/4096 * 100 = %8.2



geçerli yolun bulunması

→ arama algoritması olarak DFS-depth first search

→ BFS, arama ağacının bir seviyesini inşa ediyordu

→ DFS ise olası en derin noktaya inmeye zorlar

→ DFS için iki farklı algoritmayla ilgileneceğiz

1. her düğümün yalnızca bir kez ziyaret edildiği
2. daha genel ve birden fazla ziyarete izin var

geçerli yolun bulunması

→ derinlemesine arama, tam olarak 64 düğümlü yolu bulmamızı sağlayacak

→ DFS, ölü uçlu yolları da belirleyebilmeli -> elemeli

→ *ölü uçlu yol, çıkmaz yol; 64 düğümden daha kısa yol*

→ girilen yol çıkmaz yol ise/ölü uçluysa

→ geriye dönüp diğer derinlemesine arama denemesi yapılmalıdır

gerçekleme

```
1       def knightTour(n,path,u,limit):
2               u.setColor('gray')
3               path.append(u)
4               if n < limit:
5                   nbrList = orderByAvail(u)                i = 0
6                   done = False
7                   while i < len(nbrList) and not done:
8                       if nbrList[i].getColor() == 'white':
9                           done = knightTour(n+1,
10                                            nbrList[i],
11                                            limit)
12                  if not done:  # prepare to backtrack
13                      path.remove(u)
14                      u.setColor('white')
15              else:
16                  done = True
17              return done
```

açıklamalar

→ s1: arama ağacındaki şimdiki derinlik

→ TODO: defter-s42'den devam

demo

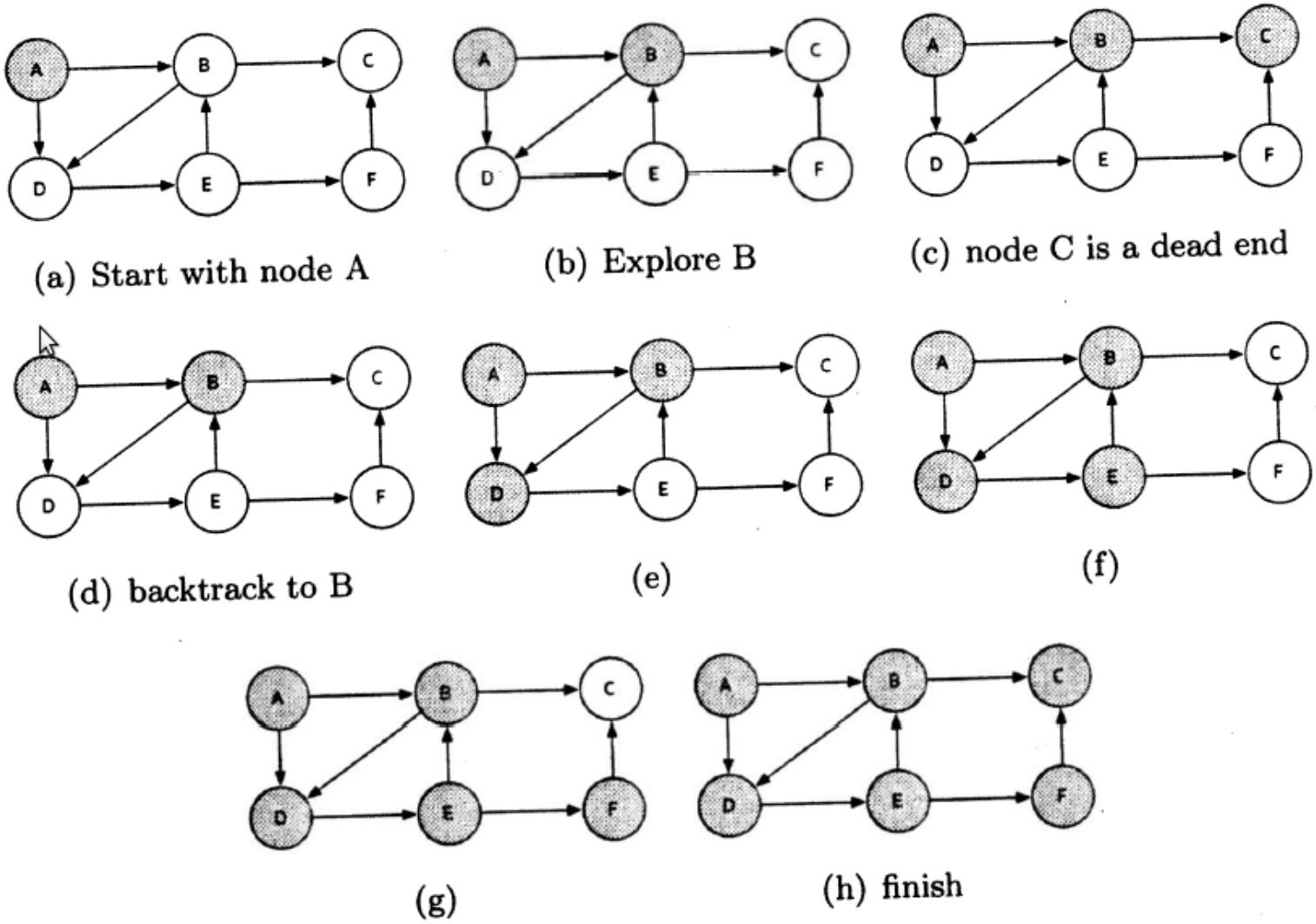
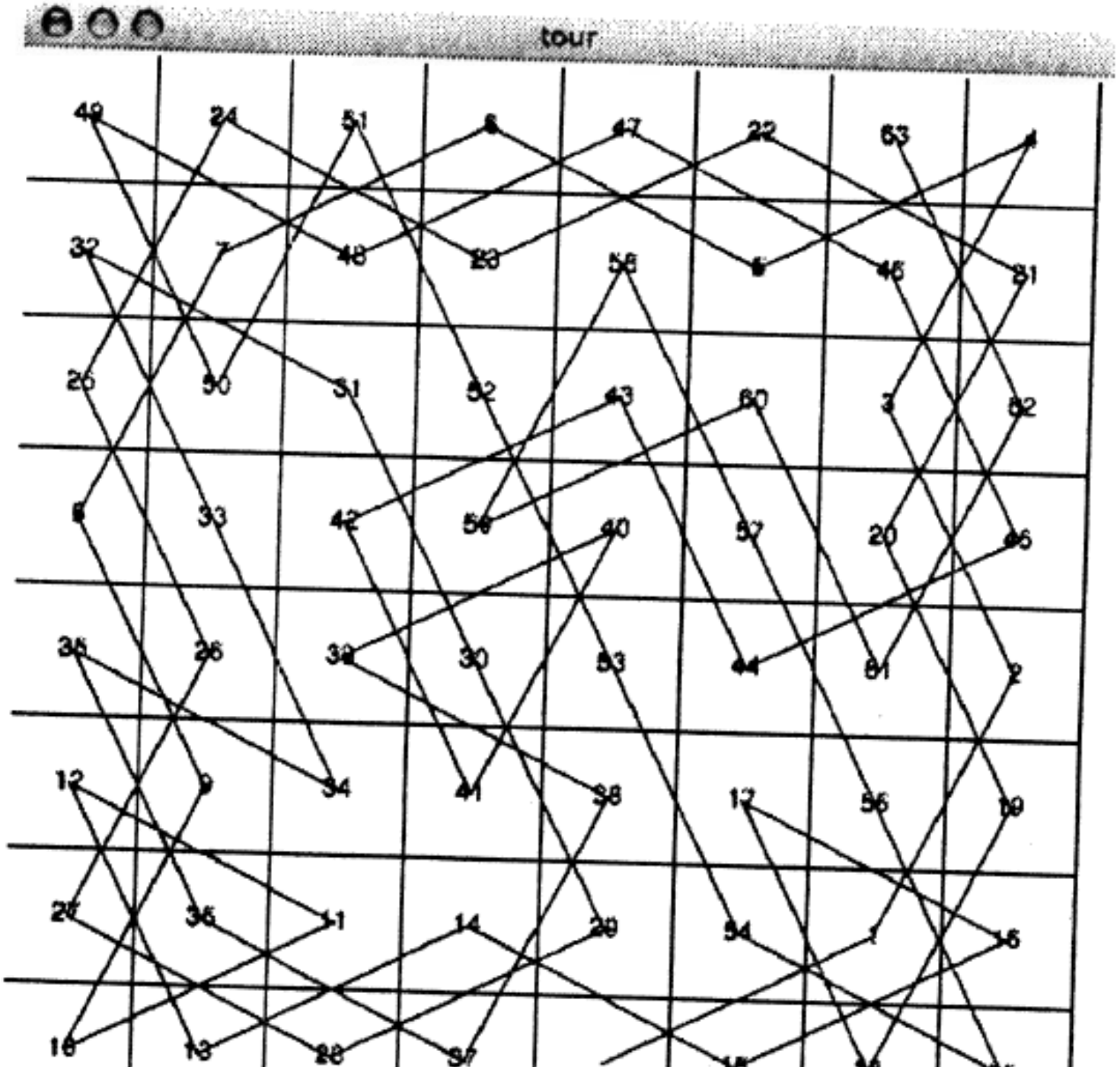


Figure 6.11: Finding a Path Through a Graph with `knightTour`

→ "dead end", "backtrack" ve "finish" e dikkat!

tam bir tur



genelleştirilmiş DFS

- → konu=başarım
- → sonraki düğümün ziyareti KRİTİKTİR
- → komşuluk listesindekilerin sırayla denenmesi ilk akla gelen, basit yöntemdir
- → bu şekilde yapıldığında 5x5'lik oyun alanında hızlı bir PC 1.5 sn bekler
- → 8x8'lik oyun alanı içinse **yarım saat** bekler
- → **1 sn**'den kısa sürede sonuç verecek bir yol var

gerçekleme

1 **sn**'den kısa sürede sonuç verecek bir yol var

```
1       def orderByAvail(n):
2           resList = []
3           for v in n.getAdj():
4               if v.getColor() == 'white':
5                   c = 0
6                   for w in v.getAdj():
7                       if w.getColor() == 'white':
8                           c = c + 1
9                   resList.append((c,v))
10          resList.sort()          return [y[1] for y in resList]
```

açıklamalar

→ TODO: defter-s43'den devam

6.4.2.2 Genelleştirilmiş DFS

FINISH