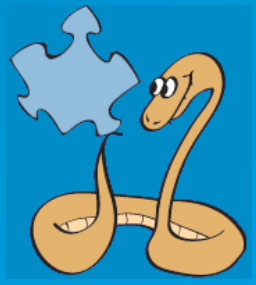
파이썬 알고리즘

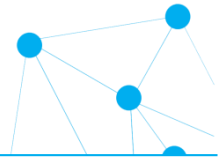
ALGORITHMS IN PYTHON 최영규 지음



CHAPTER 09

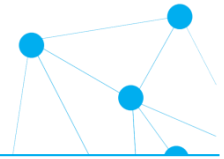
백트래킹과 분기한정

알고리즘의 설계 기법들



- 억지(brute-force) 기법과 완전 탐색: 3장
  - 문제 정의를 가장 직접 사용, 원하는 답 구할때까지 모든 경우 테스트
- 축소 정복(decrease-and-conquer): 4장
  - 주어진 문제를 하나의 좀 더 작은 문제로 축소하여 해결
- 분할 정복(divide-and-conquer): 5장
  - 주어진 문제를 여러 개의 더 작은 문제로 반복적으로 분할하여 해결 가능한 충분히 작은 문제로 분할 후 해결
- 공간을 이용해 시간을 버는 전략: 6장
  - 추가적인 공간을 사용하여 처리시간 줄이는 전략
- 동적 계획법(divide-and-conquer): 7장
  - 더 작은 문제로 나누는 분할정복과 유사하지만, 작은분제 먼저해결 저장하고 다음에 더 큰 문제 해결
- 탐욕적(greedy) 기법: 8장
  - 단순하고 직관적인 방법으로 모든 경우 고려하여 가장 좋은 답을 찾는 것이 아니라 "그 순간에 최적"이라고 생각되는 것을 선택
- 백트래킹과 분기 한정 기법: 9장
  - 상태공간에서 단계적 해 찾기, 현재의 최종 해가 않된다면 더 이상 탐색하지 않고 백트래킹(되돌아가서)해서 다른 후보 해 탐색

학습 내용



9.1 백트래킹을 이용한 간단한 문제의 해결

9.2 미로탐색

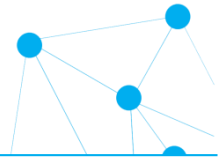
9.3 N-Queen

9.4 그래프 색칠

9.5 0-1 배낭 채우기와 분기 한정

9.6 일 배정 문제와 최적우선 분기 한정(심화)

백트래킹과 분기 한정 기법

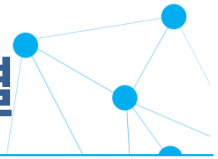


- 좀 더 어려운 문제들
  - NP-완전 (10장)

- 백트래킹(backtracking)
  - 예: 미로탐색 문제
  - 1950년대의 미국 수학자 레머(D.H. Lehmer)
  - 상태공간트리를 보다 효율적으로 탐색하는 방법을 제공
    - 기본 골격은 깊이 우선 탐색(DFS)
    - 어떤 노드(후보해)가 문제에서 요구하는 해가 아니라고 판단되면 더 이상 탐색하지 않고 이전 노드로 되돌아 나옴(backtracking)

- 분기 한정 기법 (branch and bound)
  - 백트래킹의 문제를 보완
  - 다 많은 가지치기(pruning)를 유도

9.1 백트래킹을 이용한 간단한 문제 해결



- 순열 생성

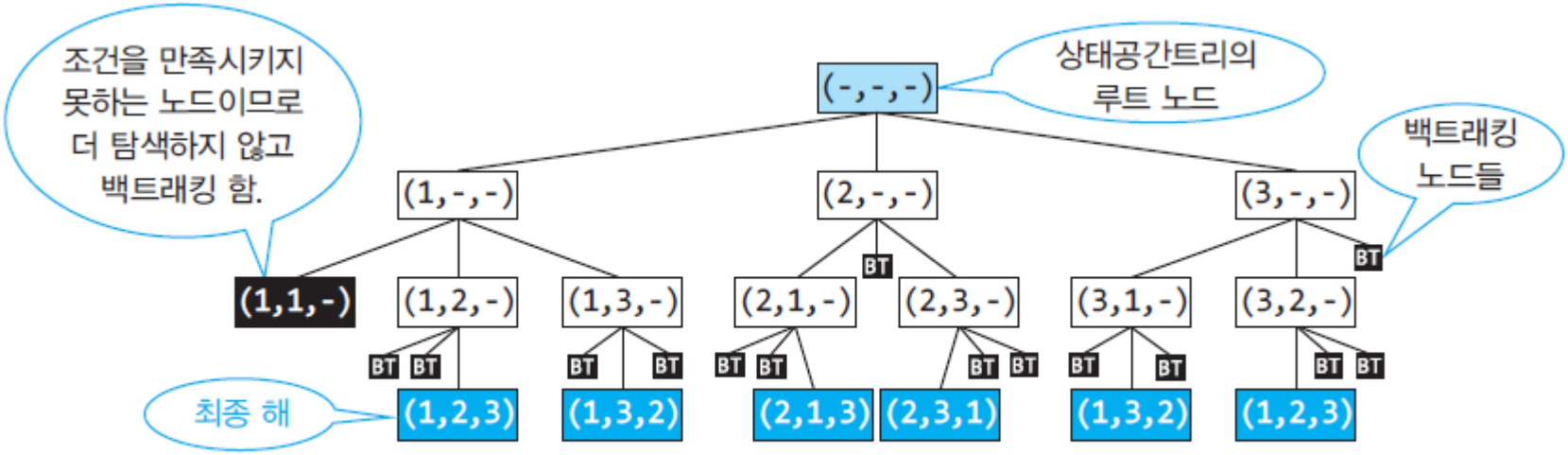
> n개의 원소를 가진 집합이 주어졌다. 모든 가능한 원소의 순서를 나열하라.

  - 예: {1, 2, 3}의 순열
    - (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)
  - 사용 분야:
    - 새로운 단어를 만드는 단어 구성(Anagrams) 문제
    - 조합론, 그룹 이론 등 거의 모든 수학과 과학 분야에서 중요

- 상태공간트리에서 백트래킹 조건

> - **노드에서 같은 원소가 중복되는 경우:** 가능한 순열이 아니므로 백트래킹
> - **노드에서 하나의 순열을 완성한 경우:** 하나의 해를 찾음. 백트래킹

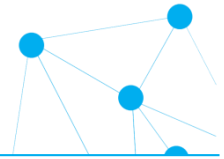
순열 생성문제



[그림 9.2] 순열 생성 문제에서의 상태공간트리: 검은 노드에서 백트래킹 발생

- 구현
  - 스택 사용
  - 순환 호출 사용

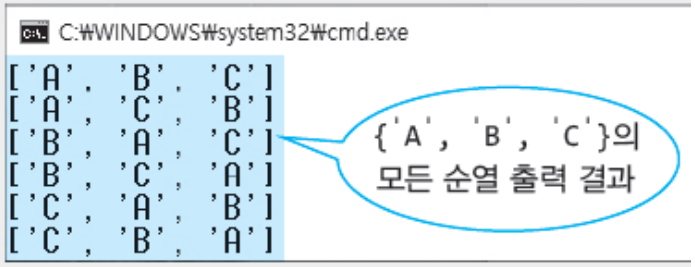
알고리즘



알고리즘 9.1 순열 생성 알고리즘(백트래킹)

```
01  def all_permutations(data):
02      bUsed = [False] * len(data)
03      DFS_permutation (data, [], 0, bUsed)
04
05  def DFS_permutation (data, sol, level, bUsed):
06      if level == len(data):
07          print(sol)
08          return
09
10      for i in range(len(data)):
11          if not bUsed[i]:
12              sol.append(data[i])
13              bUsed[i] = True
14              DFS_permutation (data, sol, level+1, bUsed)
15              sol.pop()
16              bUsed[i] = False
```

```
all_permutations(['A', 'B', 'C'])
```

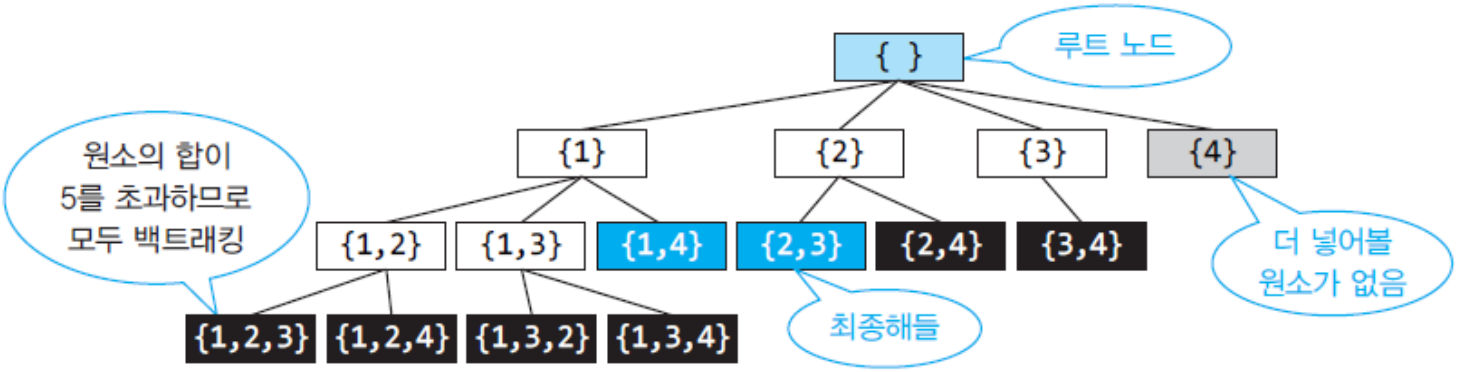


- 시간 복잡도: $O(n * n!)$

합이 M인 모든 부분 집합을 찾는 문제

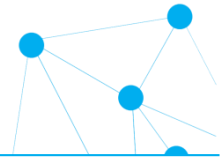
양수로 이루어진 집합 S와 어떤 수 M이 주어졌다. 이 S의 부분집합 중에 원소의 합이 M이 되는 모든 가능한 부분집합을 찾아라.

- S={1, 2, 3, 4}, M=5
  - 합이 5인 모든 가능한 부분 집합: {1,4}, {2,3}
- 집합의 모든 부분 집합(2^n 가지)을 만들고 조건을 검사
- 백트래킹 조건
  - 노드(후보 해)의 원소의 합이 M인 경우
  - 노드의 원소의 합이 M을 초과하는 경우



- 추가 조건: 남은 원소의 숫자 합 이용 → 더 많은 백트래킹 유도

알고리즘



알고리즘 9.2 합이 M인 모든 부분집합 찾기(백트래킹)

```
01  def all_sum_of_subsets(S, M):
02      DFS_sum_of_subsets(S, M, 0, [], sum(S))
03
04  def DFS_sum_of_subsets(S, M, level, sol, remaining):
05      if sum(sol) == M:
06          print(sol)
07          return
08      if sum(sol)>M or (remaining+sum(sol))<M:
09          return
10
11      for i in range(level, len(S)):
12          sol.append(S[i])
13          DFS_sum_of_subsets(S, M, i+1, sol, remaining-S[i])
14          sol.pop()
```

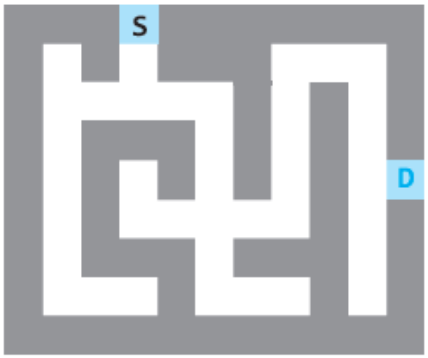
```
nums = [3, 34, 4, 12, 5, 2]
M = 9
solution = all_sum_of_subsets(nums, M)
print("입력 집합: ", nums, "M=", M )
```

```
C:₩WINDOWS₩system32₩cmd.exe
입력 집합:  [3, 34, 4, 12, 5, 2] M= 9
[3, 4, 2]
[4, 5]
```

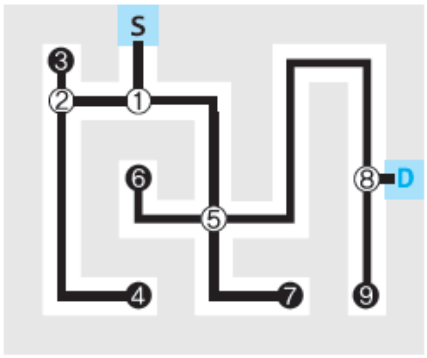
합이 9인 모든 부분집합들

- 시간 복잡도: $O(2^n)$

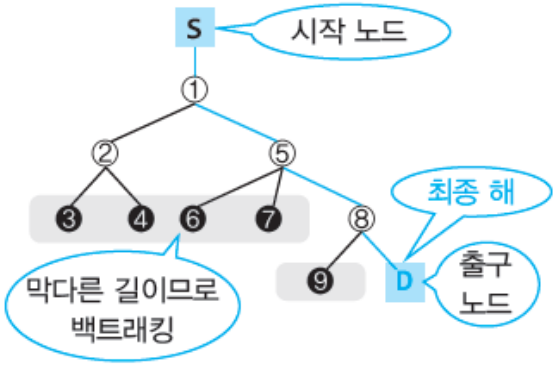
9.2 미로 탐색



(a) 2차원 배열로 표현된 미로
(S는 출발점, D는 출구)



(b) 그래프로 변환된 미로

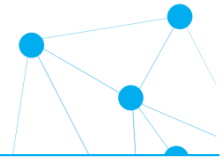


(C) 미로탐색의 상태공간트리

- 2차원 배열로 표현된 미로 탐색 문제
  - 백트래킹 조건
    - 위치 (x, y)가 유효하지 않거나,
    - 더 이상 갈 수 있는 칸이 없는 경우

```
maze = [ [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
         [0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0],
         [0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0],
         [0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0],
         [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 2],
         [0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0],
         [0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0],
         [0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

미로 탐색 알고리즘



- 미로내 위치 (x, y)의 유효성
  - (1) (x, y)가 미로 내의 위치
  - (2) 벽이 아님
  - (3) 지나갔던 위치가 아님

알고리즘 9.3 미로탐색에서 위치의 유효성 검사

```
01  def isSafe( maze, x, y, mark ):
02      W, H = len(maze[0]), len(maze)
03      if x>=0 and x<W and y>=0 and y<H :
04              if maze[y][x]!=0 and mark[y][x]==0:
05                  return True
06          return False
```

- 미로 탐색 알고리즘

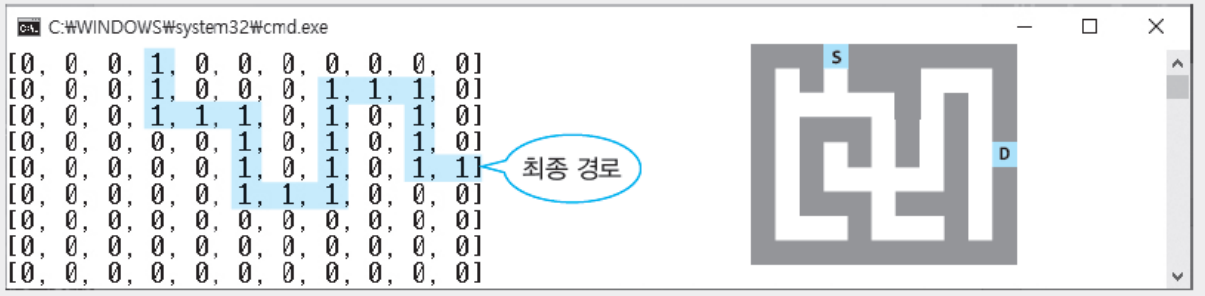
알고리즘 9.4 백트래킹을 이용한 미로탐색

```
01  def solve_maze( maze, x, y ):
02      W, H = len(maze[0]), len(maze)
03      sol = [[0 for j in range(W)] for i in range(H)]
04      mark= [[0 for j in range(W)] for i in range(H)]
05
06      if DFS_maze(maze, x, y, sol, mark) == False:
07          print("출구를 찾을 수 없음")
08      else :
09          for i in sol: print(i)
```

알고리즘

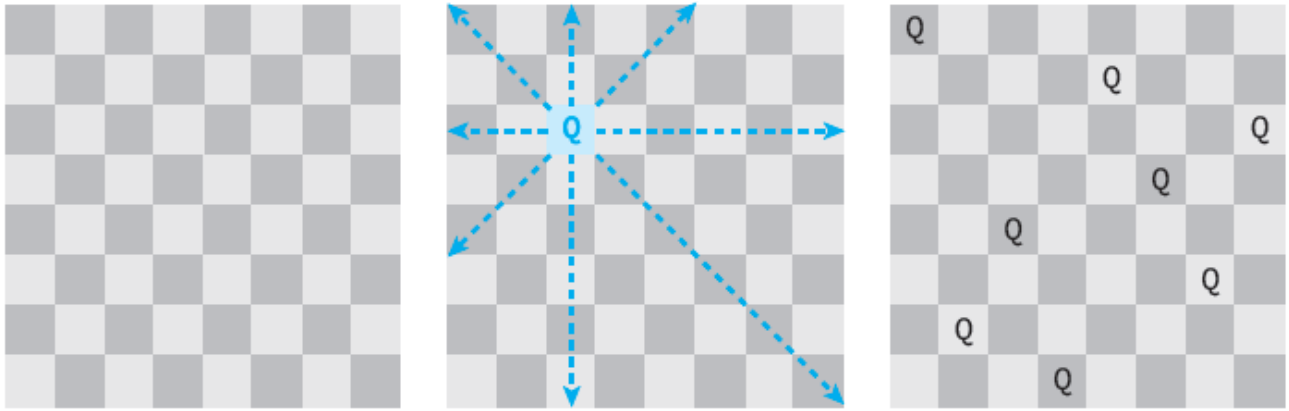
```
11  def DFS_maze(maze, x, y, sol, mark):
12      W, H = len(maze[0]), len(maze)
13
14      if not isSafe(maze, x,y, mark):
15          return False
16
17      mark[y][x] = 1
18      sol[y][x] = 1
19      if maze[y][x] == 2 :
20          return True
21
22      if DFS_maze(maze, x+1, y, sol, mark)==True: return True
23      if DFS_maze(maze, x, y+1, sol, mark)==True: return True
24      if DFS_maze(maze, x-1, y, sol, mark)==True: return True
25      if DFS_maze(maze, x, y-1, sol, mark)==True: return True
26
27      sol[y][x] = 0       # (x,y)는 이제 해의 일부가 아님 → sol에서 제거
28      return False
```

solve_maze(maze, 3, 0)



9.3 N-Queen

- N×N의 체스보드에 N개의 퀸을 놓는 문제
  - 어떤 퀸도 다른 퀸을 공격할 수 없어야 함

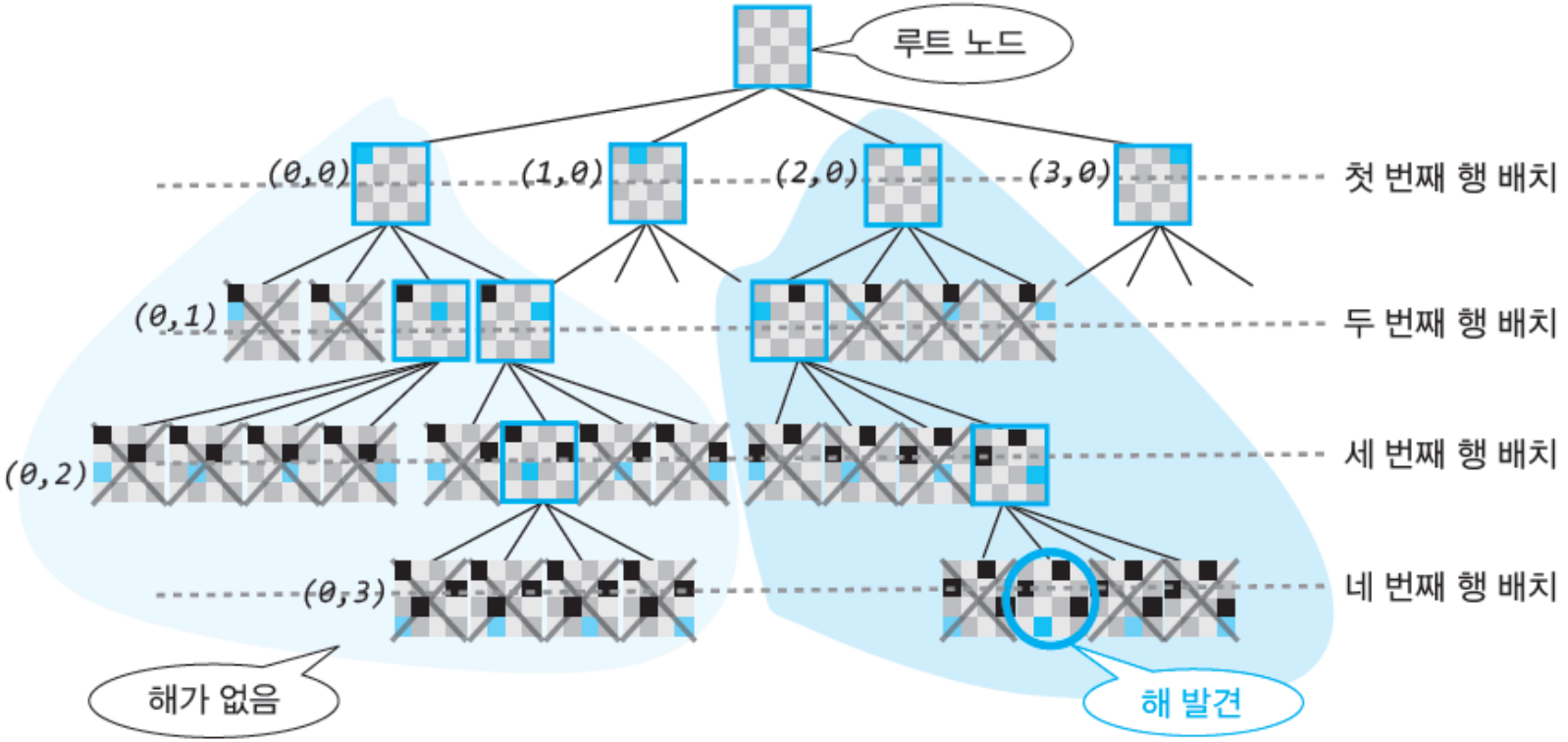


(a) 8x8 체스판 (b) 퀸의 공격 범위 (c) 8–Queen의 하나의 해

  - 8-Queen 문제: 총 92개의 가능한 해
  - 7-Queen은 40가지, 6-Queen은 4가지, 5-Queen은 10가지, 4-Queen은 2가지의 해가 있음
- N-Queen 문제의 가능한 조합의 수
  - $C(N^2, N)$

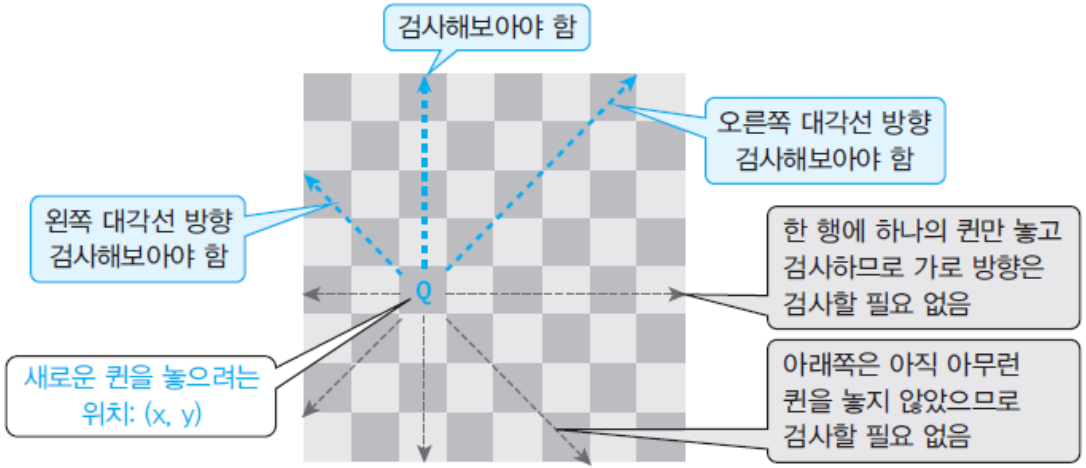
백트래킹 전략

- 4-Queen 문제의 상태공간트리



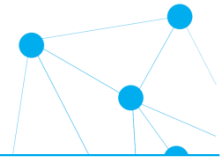
[그림 9.7] 4-Queen 문제의 상태공간트리의 일부

위치의 유효성 검사




```
01  def isSafe(board, x, y):
02      N = len(board)
03
04      for i in range(y):
05          if board[i][x] == 1: return False
06      for i, j in zip(range(y-1, -1, -1), range(x-1, -1, -1)):
07          if board[i][j] == 1: return False
08      for i, j in zip(range(y-1, -1, -1), range(x+1, N)):
09          if board[i][j] == 1: return False
10      return True
```

알고리즘



알고리즘 9.6 N-Queen

```
01  def solve_N_Queen(board, y):
02      N = len(board)
03      if y == N:
04          printBoard(board)
05          return
06
07      for x in range(N):
08          if isSafe(board, x, y):
09              board[y][x] = 1
10              solve_N_Queen(board, y+1)
11              board[y][x] = 0
```

```
N = 4
board = [[0 for i in range(N)] for j in range(N)]
solve_N_Queen(board, 0)
```

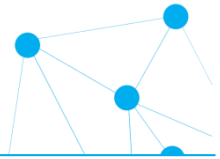
선택 C:₩WINDOWS₩system32₩cmd.exe

```
. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .
```

4-Queen 문제의 두 개의 해

9.4 그래프 색칠

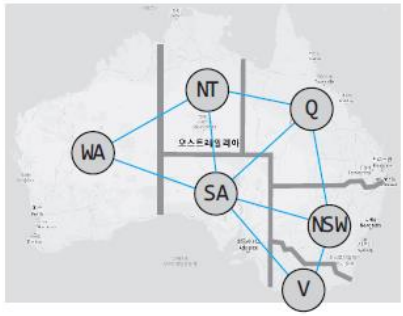


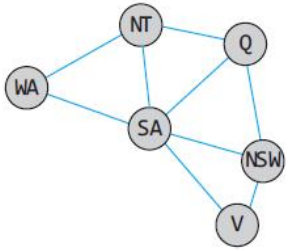
- 호주의 각 주들과 색칠 문제

(a) 호주 대륙의 주들 (Tasmania 제외)

(b) 인접한 주가 다른 색이 되도록 색칠하기

- 지도 색칠 → 그래프 색칠(graph coloring) 문제

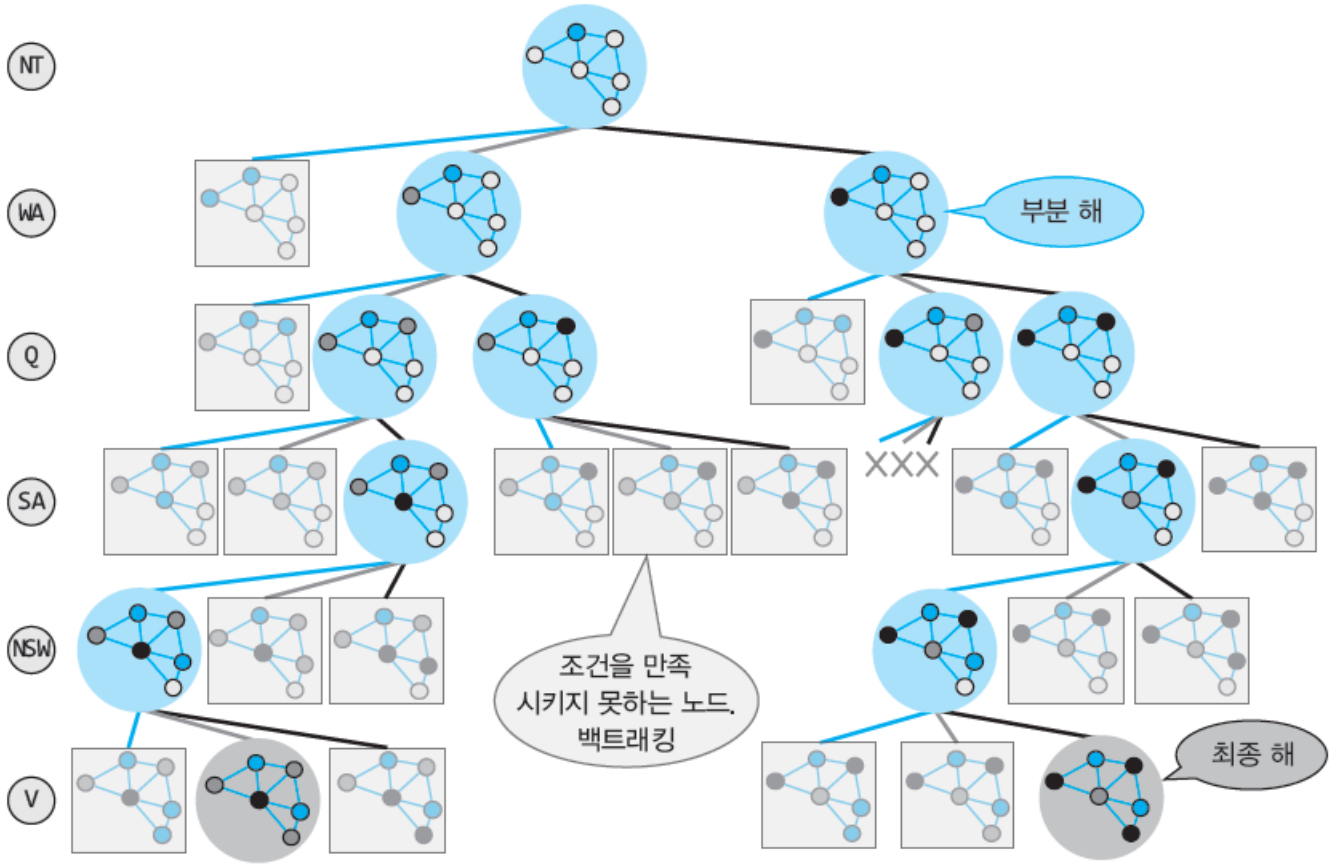




- n 개의 노드, k 개의 색칠
  - k^n가지 조합이 가능

백트래킹 전략

- 백트래킹 조건
  - v의 인접 정점 중에 색이 c인 정점이 있는 경우 → backtracking

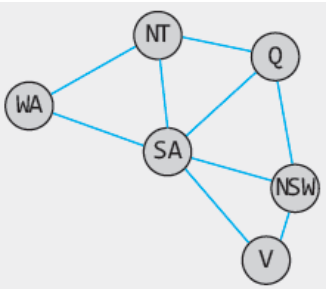


알고리즘

알고리즘 9.8 그래프 색칠

```
01  def DFS_graph_coloring(graph, k, v, color):
02      if v == len(graph):
03          return True
04
05      for c in range(1, k+1):
06          if isSafe(graph, v, c, color) :
07              color[v] = c
08              if DFS_graph_coloring(graph, k, v+1, color):
09                  return True
10              color[v] = 0
11
12      return False
13
14  def graphColouring(graph, k, states):
15      color = [0] * len(graph)
16      ret = DFS_graph_coloring(graph, k, 0, color)
17      if ret :
18          for i in range(len(graph)) :
19              print("%3s = "%states[i], color_name[color[i]])
20      else :
21          print("그래프를 색칠할 수 없음!")
```

```
def isSafe(g, v, c, color):
    for i in range(len(g)):
        if g[v][i] == 1 and color[i] == c:
            return False
    return True
```



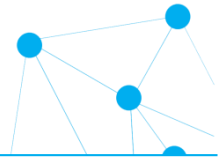

```
print("색상 3개:")
graphColouring(g, 3, states)
print("색상 2개:")
graphColouring(g, 2, states)
```

```
C:₩WINDOWS₩system32₩cmd.exe
색상 3개:
 NT =  빨강
 WA =  초록
  Q =  초록
 SA =  파랑
NSW =  빨강
  V =  초록
색상 2개:
그래프를 색칠할 수 없음!
```

3색 칠하기. 주별로 배정된 색

2색으로는 칠할 수 없음

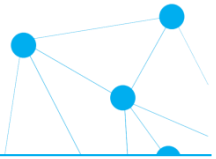
9.5 0-1 배낭 채우기와 분기 한정



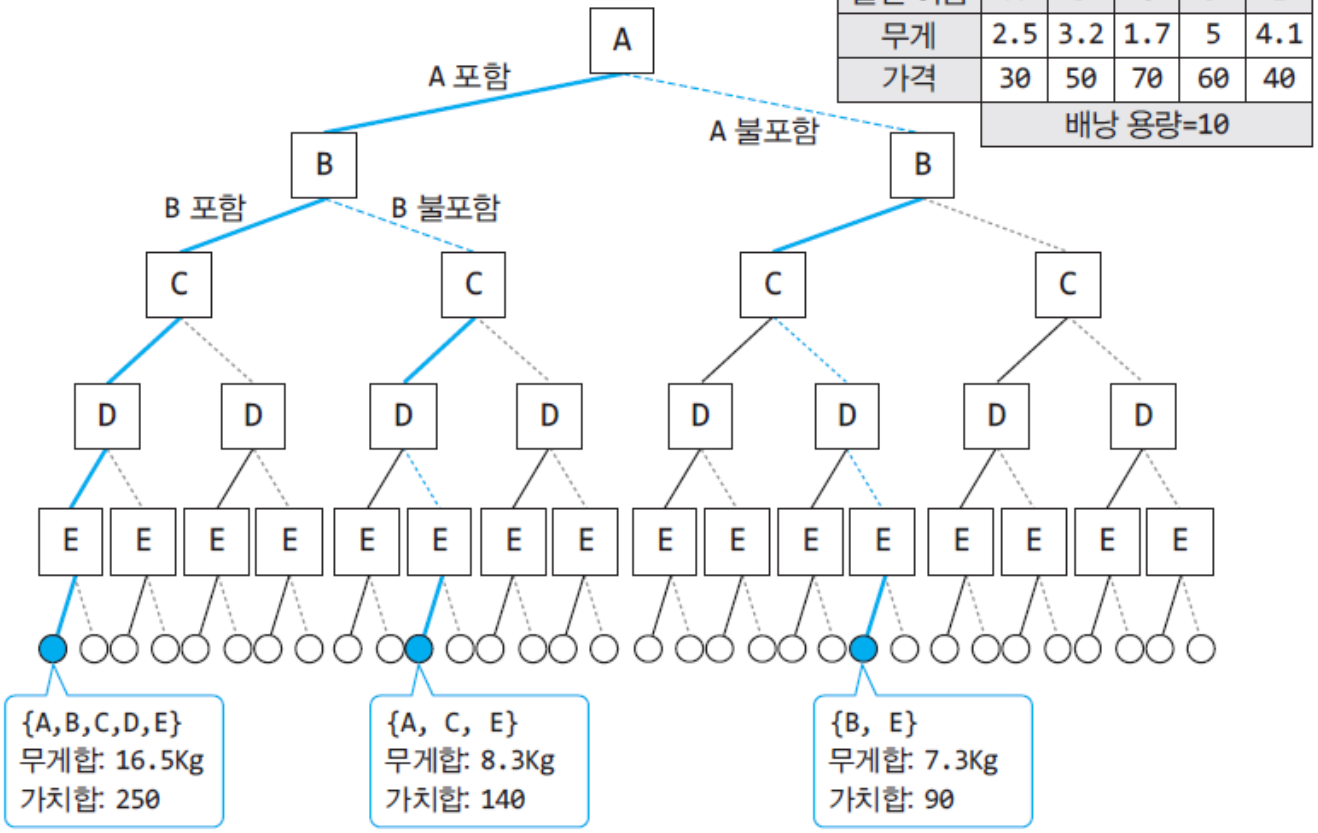
- 0-1 배낭 문제 해법

0-1 배낭 문제 해법	특징
탐욕적 기법(8장)	• 무게 당 가격이 높은 것부터 탐욕적으로 선택하여 배낭의 용량을 초과하지 않을 때 까지 넣는 방법 • 부분적인 배낭 채우기 문제에서만 최적해를 보장
동적 계획법(7장)	• 2차원 테이블을 이용하는데, 0-1 배낭 채우기 문제의 최적해를 구할 수 있었다. • 실수 무게를 허용하면 이 방법을 적용할 수 없다.
완전 탐색(3장)	• 최적해를 보장하지만 시간이 많이 걸림(그림 9.13)
백트래킹(9장)	• 완전 탐색의 상태공간트리에서 많은 노드들을 가지치기함 • 탐색의 효율을 높임(그림 9.14)
분기 한정(9장)	• 가지치기의 효율을 더욱 높이는 방법(그림 9.15)

완전 탐색과 상태공간트리



물건 이름	A	B	C	D	E
무게	2.5	3.2	1.7	5	4.1
가격	30	50	70	60	40
배낭 용량=10					

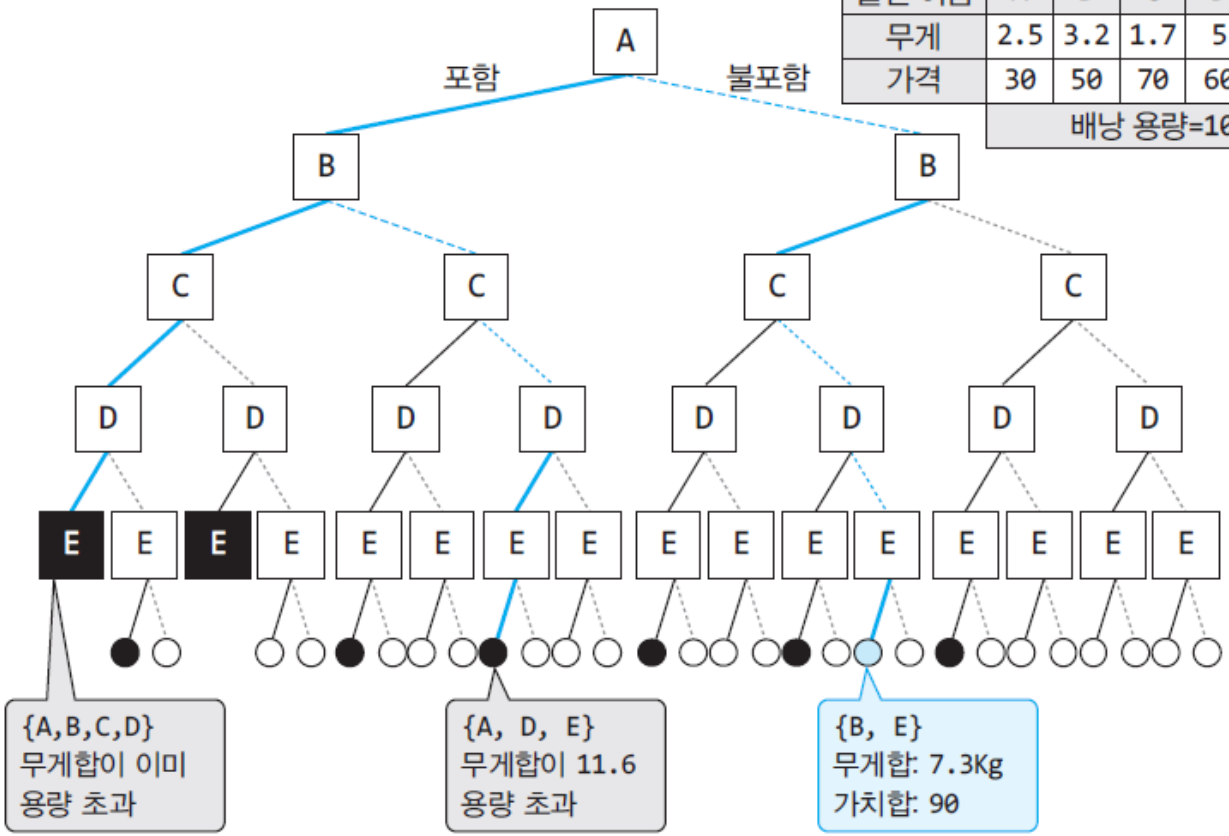


- 단말 노드의 수: 2^n가지

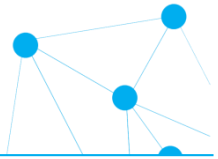
백트래킹을 이용한 0-1 배낭 채우기

- 백트래킹 조건
  - 노드의 무게 합이 이미 배낭 용량을 초과한 경우

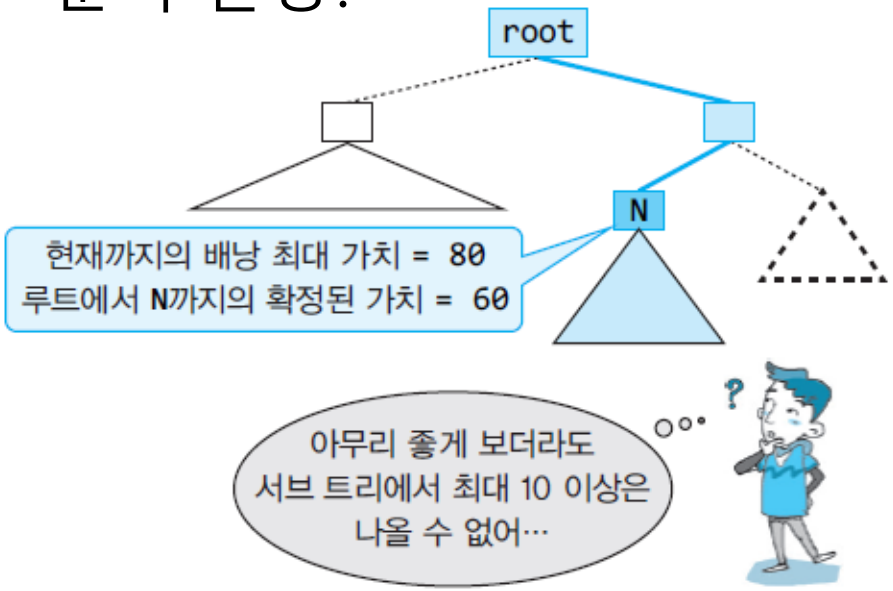
물건 이름	A	B	C	D	E
무게	2.5	3.2	1.7	5	4.1
가격	30	50	70	60	40
배낭 용량=10					



분기 한정 아이디어



- 분기 한정?



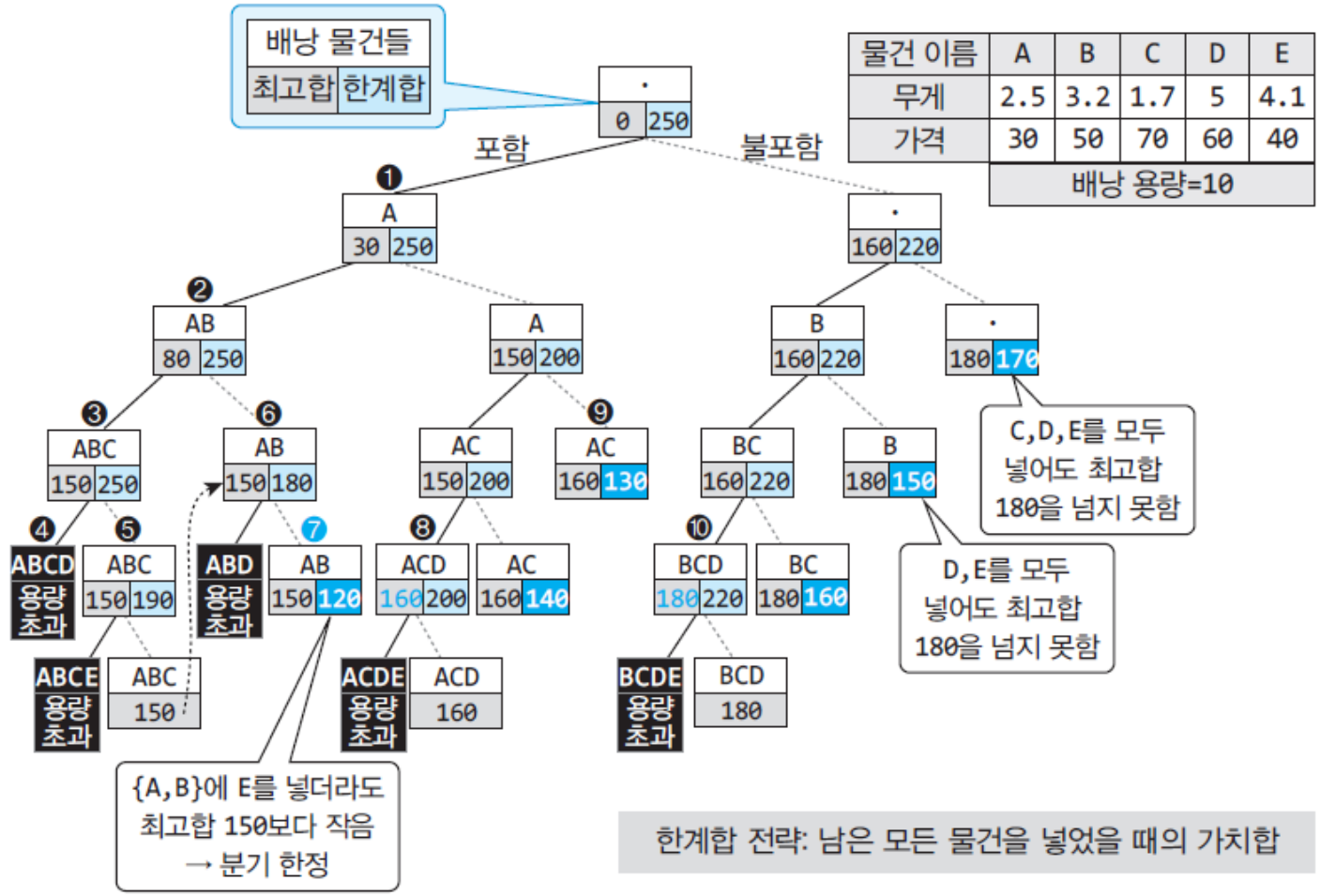
백트래킹
N의 서브 트리에서 가치가 더 추가될 수 있음.
→ N의 서브 트리 계속 탐색

분기 한정
서브 트리에서 기대할 수 있는 최대 가치의 합 10
(N까지의 확정가치 + 서브 트리 최대 기대가치)
= (60 + 10) < 80 (현재 알고있는 최대 가치)

→ N에서 백트래킹

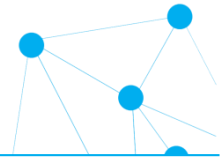
  - 최고 가치(maximum profit) 또는 최고합
  - 한계 가치(bounded profit) 또는 한계합
- 만약 N의 한계합이 최고합보다 작다면 ?
  - 서브 트리를 탐색할 필요가 없음

분기 한정의 한계합 계산



N의 한계합 = N의 확정된 가치합 + 아직 결정하지 않은 모든 물건의 가치합

알고리즘



알고리즘 9.9 분기 한정 기법을 이용한 01-배낭 채우기

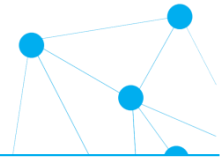
```
01  def knapSack_bnb(obj, W, level, weight, profit, maxProfit) :
02      if (level == len(obj)) :
03          return maxProfit
04
05      if weight + obj[level][0] <= W :
06          newWeight = weight + obj[level][0]
07          newProfit = profit + obj[level][1]
08          if newProfit > maxProfit :
09              maxProfit = newProfit
10
11          newBound  = bound(obj, W, level, newWeight, newProfit)
12          if newBound >= maxProfit :
13              maxProfit = knapSack_bnb(obj, W, level+1, newWeight,
14                                                  newProfit, maxProfit)
15
16      newWeight = weight
17      newProfit = profit
18      newBound  = bound(obj, W, level, newWeight, newProfit)
19      if newBound >= maxProfit :
20          maxProfit = knapSack_bnb(obj, W, level+1, newWeight,
21                                                  newProfit, maxProfit)
22
23      return maxProfit
```

```
def bound(obj, W, level, weight, profit) :
    if weight > W :
        return 0

    pBound = profit
    for j in range(level+1, len(obj)) :
        pBound += obj[j][1]

    return pBound
```

테스트



알고리즘 테스트 분기 한정 기법을 이용한 01-배낭 채우기

```
obj = [(2.5,30,"A"), (3.2,50,"B"), (1.7,70,"C"), (5,60,"D"), (4.1,40,"E") ]
print(obj)
print("0-1배낭문제(분기 한정): ", knapSack_bnb(obj, W))
```

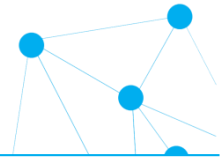
```
C:₩WINDOWS₩system32₩cmd.exe
[(2.5, 30, 'A'), (3.2, 50, 'B'), (1.7, 70, 'C'), (5, 60, 'D'), (4.1, 40, 'E')]
0-1배낭문제(분기 한정):  180
```

최적해

```
C:₩WINDOWS₩system32₩cmd.exe
[(2.5, 30, 'A'), (3.2, 50, 'B'), (1.7, 70, 'C'), (5, 60, 'D'), (4.1, 40, 'E')]
0 []                :  0.0Kg 가치/한계합 =   0.0 /   0.0 >   0.0(최고합)
1 ['A']             :  2.5Kg 가치/한계합 =  30.0 / 250.0 >  30.0(최고합)
2 ['A', 'B']        :  5.7Kg 가치/한계합 =  80.0 / 250.0 >  80.0(최고합)
3 ['A', 'B', 'C']   :  7.4Kg 가치/한계합 = 150.0 / 250.0 > 150.0(최고합)
4 ['A', 'B', 'C']   :  7.4Kg 가치/한계합 = 150.0 / 190.0 > 150.0(최고합)
5 ['A', 'B', 'C']   :  7.4Kg 가치/한계합 = 150.0 / 150.0 > 150.0(최고합)
3 ['A', 'B']        :  5.7Kg 가치/한계합 =  80.0 / 180.0 > 150.0(최고합)
2 ['A']             :  2.5Kg 가치/한계합 =  30.0 / 200.0 > 150.0(최고합)
3 ['A', 'C']        :  4.2Kg 가치/한계합 = 100.0 / 200.0 > 150.0(최고합)
4 ['A', 'C', 'D']   :  9.2Kg 가치/한계합 = 160.0 / 200.0 > 160.0(최고합)
5 ['A', 'C', 'D']   :  9.2Kg 가치/한계합 = 160.0 / 160.0 > 160.0(최고합)
1 []                :  0.0Kg 가치/한계합 =   0.0 / 220.0 > 160.0(최고합)
2 ['B']             :  3.2Kg 가치/한계합 =  50.0 / 220.0 > 160.0(최고합)
3 ['B', 'C']        :  4.9Kg 가치/한계합 = 120.0 / 220.0 > 160.0(최고합)
4 ['B', 'C', 'D']   :  9.9Kg 가치/한계합 = 180.0 / 220.0 > 180.0(최고합)
5 ['B', 'C', 'D']   :  9.9Kg 가치/한계합 = 180.0 / 180.0 > 180.0(최고합)
0-1배낭문제(분기 한정):  180
```

레벨 / 노드의 가치합 / 한계합 / 최고합 갱신

한계합 계산방법의 개선(심화)



남은 물건들을 분할 가능한 배낭 문제로 생각하고, 무게당 가격이 높은 물건부터 배낭의 용량을 꽉 채워서 넣고, 이때의 가치합을 한계가치로 사용한다.

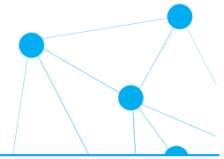
```
01  def bound2(obj, W, level, weight, profit) :
02      if weight > W :
03          return 0
04
05      pBound = profit
06      tWeight= weight
07
08      j = level+1
09      n = len(arr)
10      while j < n and (tWeight+obj[j][0] <= W) :
11          tWeight += arr[j][0]
12          pBound  += arr[j][1]
13          j += 1
14
15      # 분할 가능한 배낭 채우기 문제로 남은 용량을 채움
16      if (j < n) :
17          pBound += (W - tWeight) * (arr[j][1]/arr[j][0])
18
19      return pBound
```

```
C:₩WINDOWS₩system32₩cmd.exe
[(1.7, 70, 'C'), (3.2, 50, 'B'), (2.5, 30, 'A'), (5, 60, 'D'), (4.1, 40, 'E')]
0 []                :   0.0Kg 가치/한계합 =   0.0 /   0.0 >   0.0(최고합)
1 ['C']             :   1.7Kg 가치/한계합 =  70.0 / 181.2 >  70.0(최고합)
2 ['C', 'B']        :   4.9Kg 가치/한계합 = 120.0 / 181.2 > 120.0(최고합)
3 ['C', 'B', 'A']   :   7.4Kg 가치/한계합 = 150.0 / 181.2 > 150.0(최고합)
4 ['C', 'B', 'A']   :   7.4Kg 가치/한계합 = 150.0 / 175.4 > 150.0(최고합)
5 ['C', 'B', 'A']   :   7.4Kg 가치/한계합 = 150.0 / 150.0 > 150.0(최고합)
3 ['C', 'B']        :   4.9Kg 가치/한계합 = 120.0 / 181.0 > 150.0(최고합)
4 ['C', 'B', 'D']   :   9.9Kg 가치/한계합 = 180.0 / 181.0 > 180.0(최고합)
5 ['C', 'B', 'D']   :   9.9Kg 가치/한계합 = 180.0 / 180.0 > 180.0(최고합)
0-1배낭문제(분기 한정):  180
```

물건의 정렬

한계합

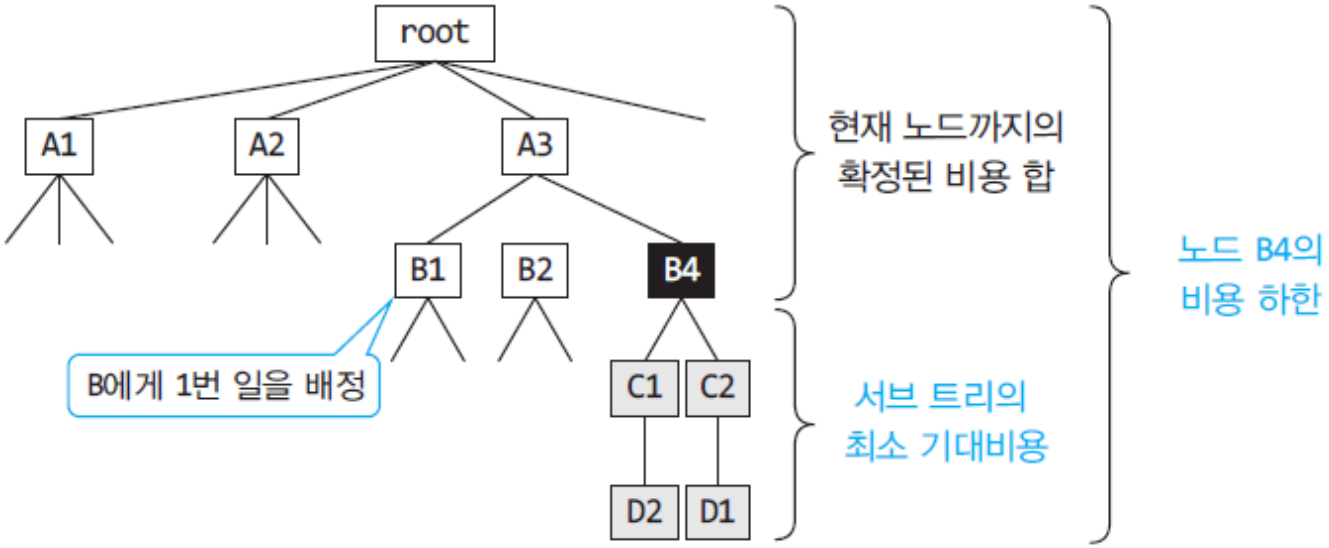
9.6 일 배정 문제와 최적우선 분기 한정



- 일 배정 문제

	Job 1	Job 2	Job 3	Job 4
A	9	2	6	8
B	6	4	3	7
C	5	7	1	9
D	7	6	8	4

- 일 배정 문제의 한계값



일 배정 문제의 한계값

- 한계값 전략

> 아직 결정되지 않은 근로자들이 현재 선택할 수 있는 남은 일들 중에서 가장 비용이 적은 일을 선택하고, 이 비용을 모두 더하는 것이다. 어떤 배정도 이 방법보다 비용이 적을 수는 없다.

	Job 1	Job 2	Job 3	Job 4
A	~~9~~	~~2~~	6	~~8~~
B	~~6~~	~~4~~	~~3~~	7
C	5	7	~~1~~	~~9~~
D	7	6	~~8~~	~~4~~

노드 B4의 하한(6+7+5+6)

	Job 1	Job 2	Job 3	Job 4
A	9	2	6	8
B	6	4	3	7
C	5	7	1	9
D	7	6	8	4

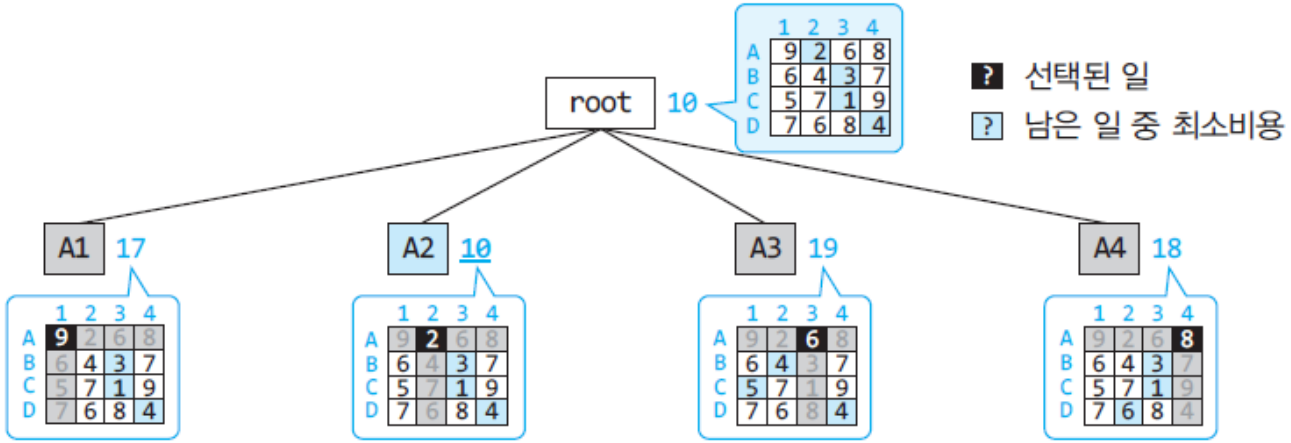
루트 노드의 하한(2+3+1+4)

[그림 9.21] 일 배정 문제에서 노드의 한계비용을 구하는 예

최적우선(best-first) 분기 한정

분기 한정 기법에서는 반드시 DFS를 사용할 필요가 없다. 모든 노드를 순회할 수 있는 방법이라면 BFS나 다른 방법으로 탐색해도 된다.

최적우선탐색(best-first-search)은 여러 가능한 노드 중에 가장 **유망한** 노드를 선택해 먼저 탐색하는 방법이다. 이러한 탐색 전략을 사용하는 분기 한정 기법을 **최적우선(best-first) 분기 한정**이라고 한다.

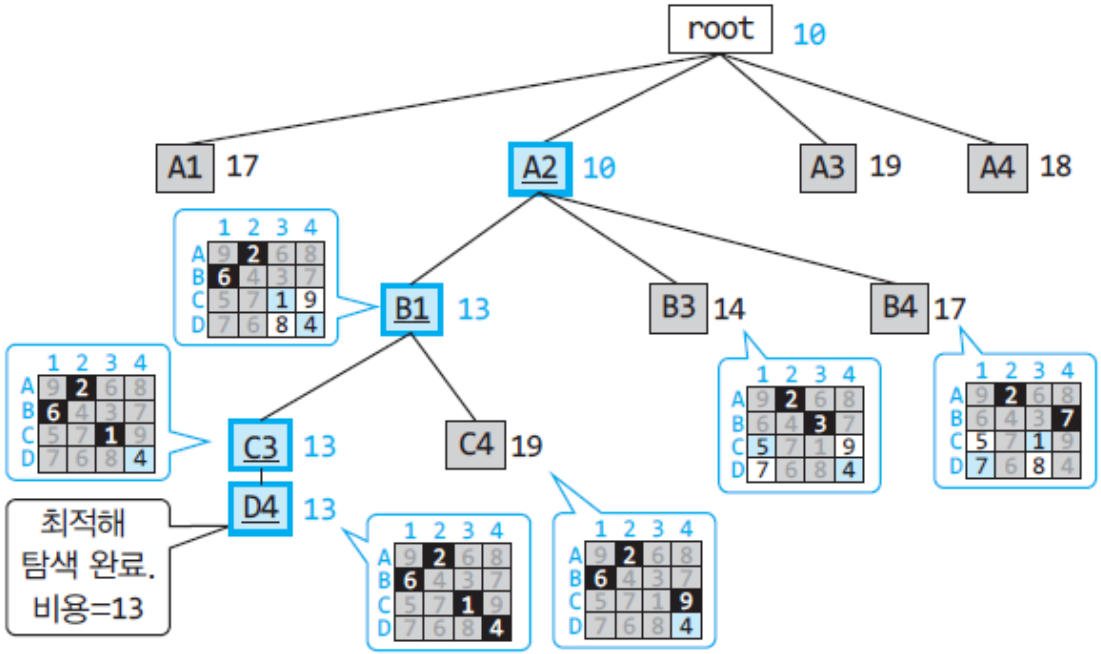


루트의 4개의 자식 노드의 한계 비용: 17, 10, 19, 18
다음으로 어느 노드의 서브 트리를 먼저 탐색하고 싶을까? A2

최적우선 분기 한정 알고리즘

알고리즘 9.12 우선순위 큐를 이용한 최적우선 탐색 알고리즘

1. 공백상태의 우선순위 큐 Q에 루트 노드를 삽입한다.
2. Q에서 한계값이 가장 작은 노드 n을 꺼낸다.
3. 만약 n이 단말 노드이면 최적해가 나온 것이므로 이를 반환한다.
4. 단말 노드가 아니면 가능한 모든 자식 노드를 생성하여 Q에 삽입한다.
5. Step2로 돌아간다.



Q=[root]

Q=[A1,A2,A3,A4]

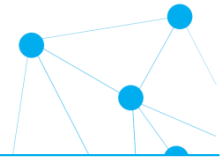
Q=[A1,A3,A4,B1,B3,B4]

Q=[A1,A3,A4,B3,B4,C3]

Q=[A1,A3,A4,B3,B4,D4]

우선순위 큐

알고리즘과 테스트



- 하한값 계산: 알고리즘 9.14(409쪽)
- 최적우선 분기한정: 알고리즘 9.13(408~409쪽)

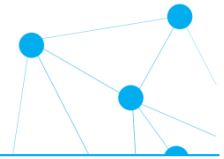
알고리즘 테스트 최적우선 분기 한정을 이용한 일 배정

```
Man2Job = [ [ 9, 2, 6, 8],
            [ 6, 4, 3, 7],
            [ 5, 8, 1, 8],
            [ 7, 6, 9, 4] ]
total, jobs = job_assign_BFBnB(Man2Job)
print("배정 결과: ", jobs)
print("전체 비용: ", total)
```

```
C:₩WINDOWS₩system32₩cmd.exe
현재 노드:  10 []
현재 노드:  10 [1]
현재 노드:  13 [1, 0]
현재 노드:  13 [1, 0, 2]
현재 노드:  13 [1, 0, 2, 3]
배정 결과:  [1, 0, 2, 3]
전체 비용:  13
```

root
A2
B1
C3
D4
최적해

실습 과제



감사합니다!