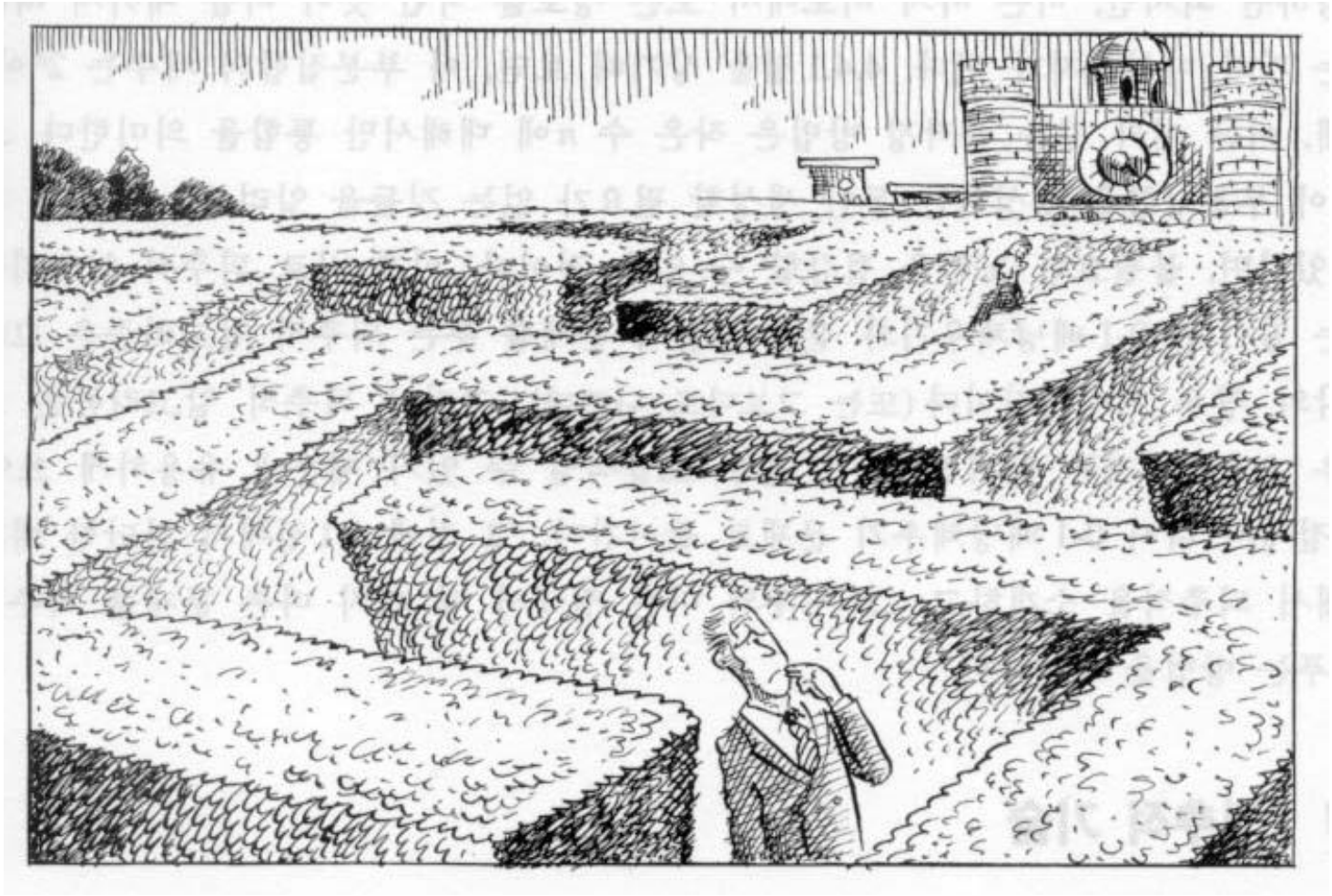
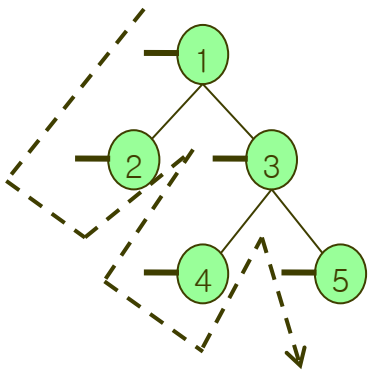
5장 되추적 (Backtracking)



트리 방문(tree traversal)

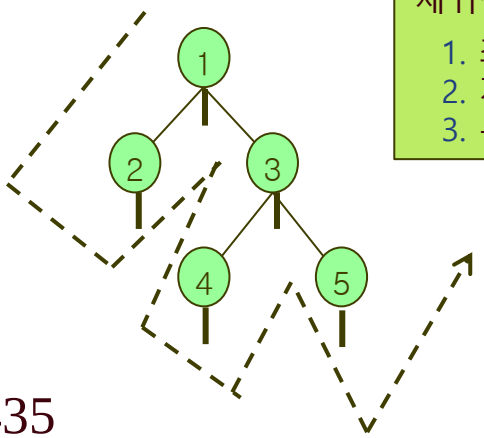
1. preorder



12345

재귀적으로
1. 자신 방문
2. 좌측 방문
3. 우측 방문

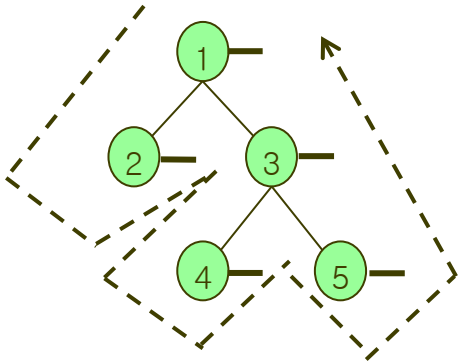
2. inorder



21435

재귀적으로
1. 좌측 방문
2. 자신 방문
3. 우측 방문

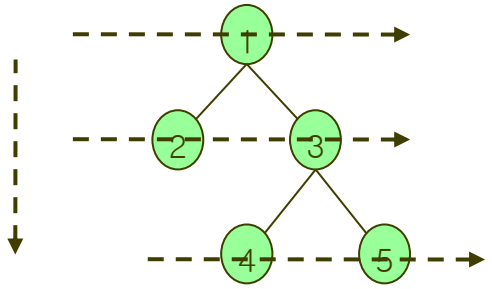
3. postorder



24531

재귀적으로
1. 좌측 방문
2. 우측 방문
3. 자신 방문

4. level order



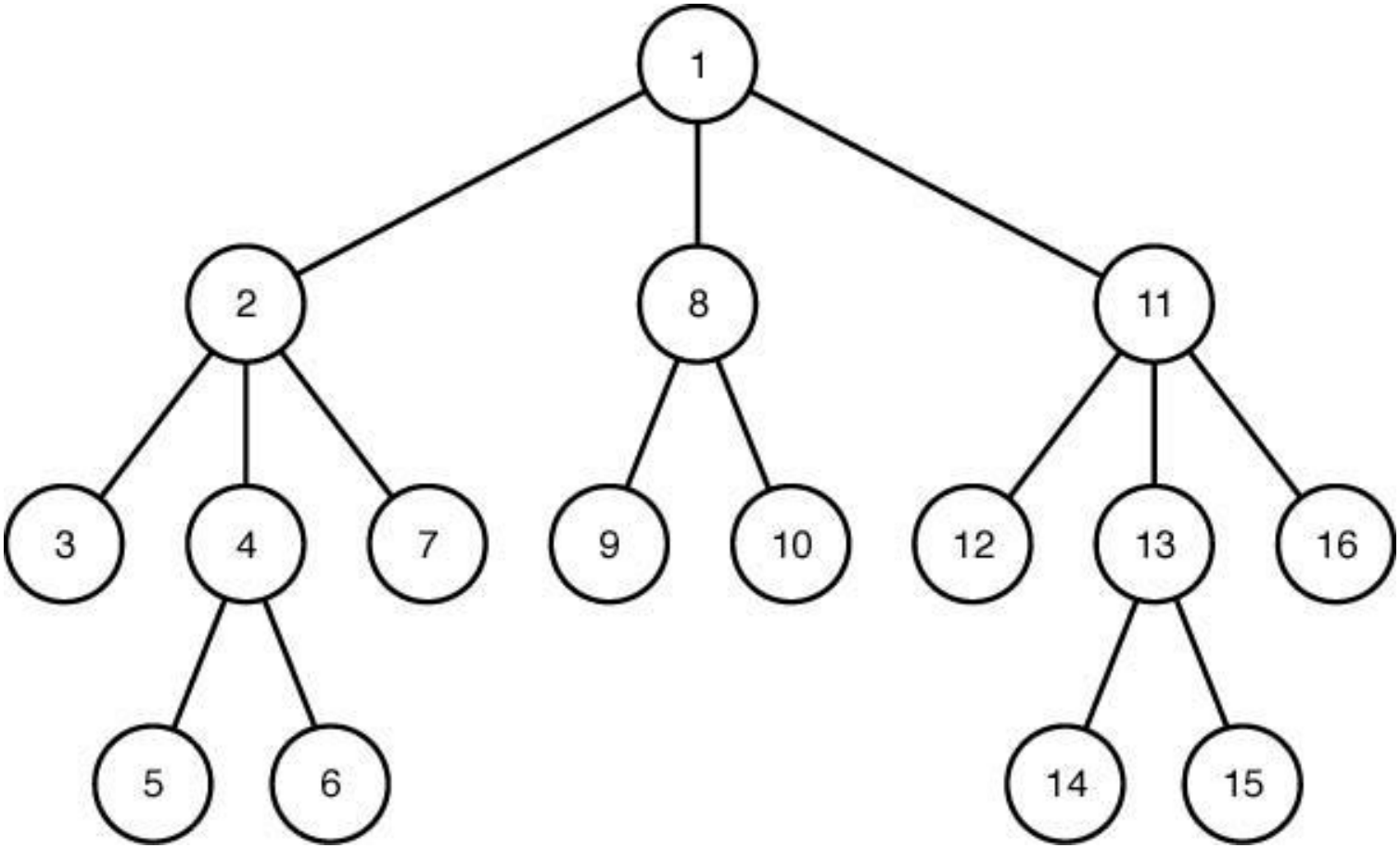
12345

깊이우선검색(depth-first search)

- 뿌리마디(root)가 되는 마디(node)를 먼저 방문한 뒤, 그 마디의 모든 후손마디(descendant)들을 차례로 (보통 왼쪽에서 오른쪽으로) 방문한다.
  (= preorder tree traversal).

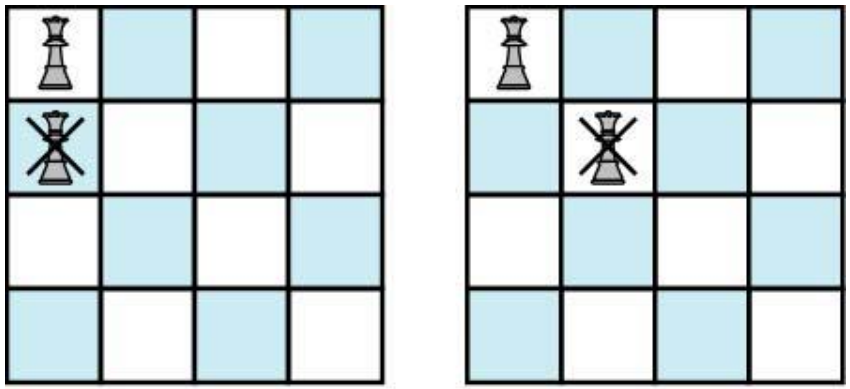
```
void depth_first_tree_search (node v) {
        node u;

        visit v;
        for (each child u of v)
                depth_first_tree_search(u)
}
```

깊이우선검색의 예



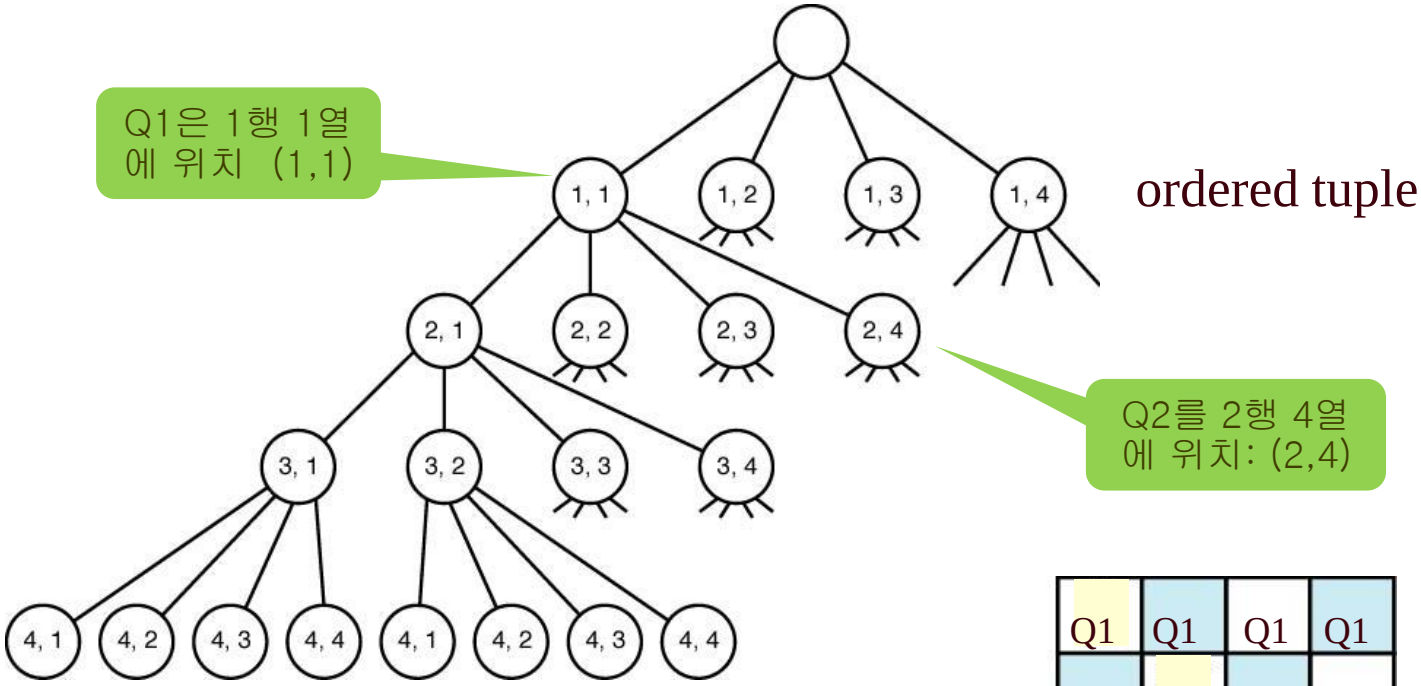
4-Queens 문제

- 4개의 Queen을 서로 상대방을 위협하지 않도록 4×4 서양장기(chess)판에 위치시키는 문제. 서로 상대방을 위협하지 않기 위해서는 같은 행이나, 같은 열이나, 같은 대각선 상에 위치하지 않아야 한다.

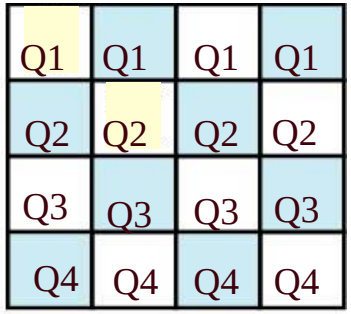


(a) (b)

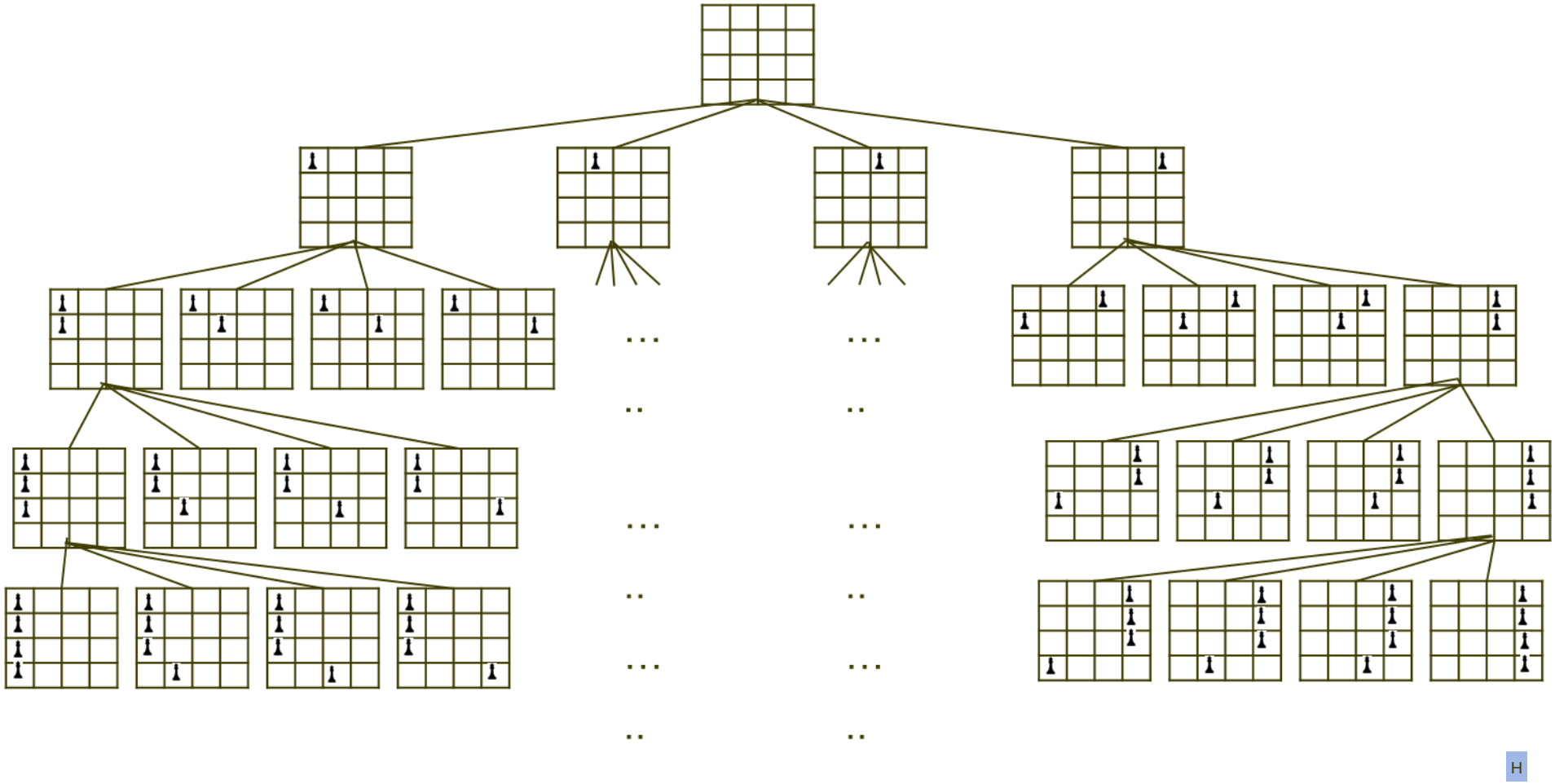
- **무작정 알고리즘**: 각 Queen을 각각 다른 행에 할당한 후에, 어떤 열에 위치하면 해답을 얻을 수 있는지를 차례대로 점검해 보면 된다. 이때, 각 Queen은 4개의 열 중에서 한 열에 위치할 수 있기 때문에, 해답을 얻기 위해서 점검해 보아야 하는 모든 경우의 수는 $4 \times 4 \times 4 \times 4 = 256$가지가 된다.



4-Queens 문제의 상태공간트리

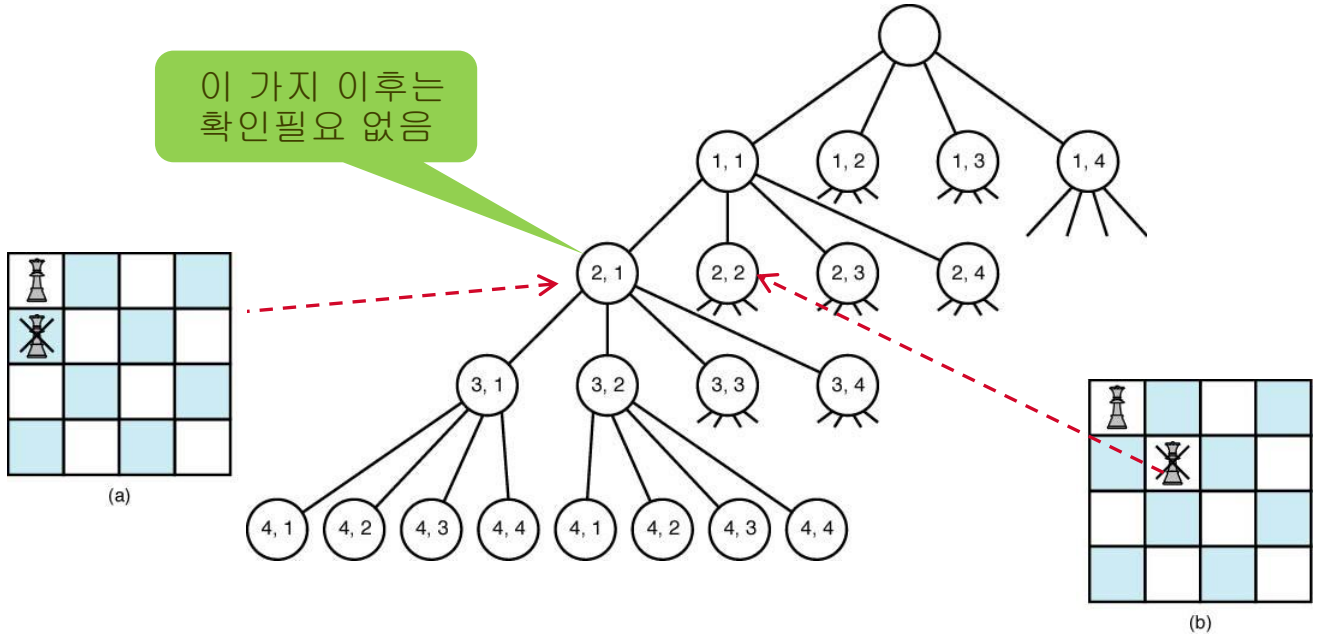


여왕 배치 상태공간을 계층적으로 표시



상태공간트리(state space tree)

- 뿌리마디에서 잎마디(leaf)까지의 경로는 해답후보(candidate solution)가 되는데, 깊이우선검색을 하여 그 해답후보 중에서 해답을 찾을 수 있다.
- 그러나 이 방법을 사용하면 해답이 될 가능성이 전혀 없는 마디의 후손마디(descendant)들도 모두 검색해야 하므로 비효율적이다.



되추적 기술

- **마디의 유망성:**
  - ✓ 전혀 해답이 나올 가능성이 없는 마디는 유망하지 않다(non-promising)
  - ✓ 그렇지 않으면 유망하다(promising).

- 되추적이란?
  - ✓ 어떤 마디의 유망성을 점검한 후, 유망하지 않다고 판정이 되면 그 마디의 부모마디(parent)로 돌아가서("backtrack") 다음 후손마디에 대한 검색을 계속하게 되는 절차.
  - ✓ 부모마디로 돌아가는 것 → 가지치기(pruning)
  - ✓ 이 과정에서 방문한 마디만으로 구성된 부분트리

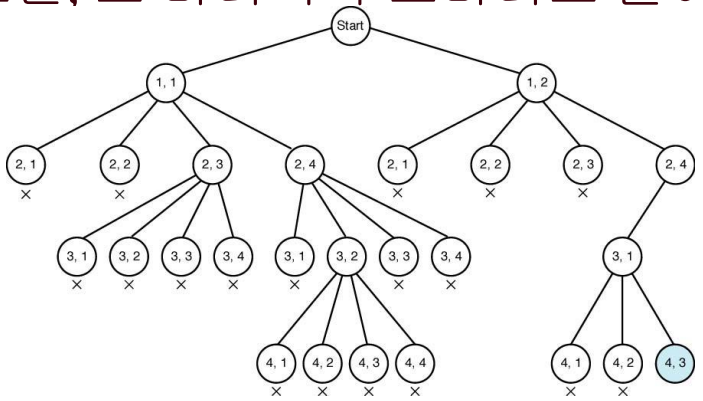
    → 가지친 상태공간 트리(pruned state space tree)

되추적 알고리즘의 개념

- 되추적 알고리즘은 상태공간트리에서 깊이우선검색을 실시하는데,
  - ✓ 유망하지 않은 마디들은 가지쳐서(pruning)) 더 이상 하위 노드들을 검색을 하지 않으며,
  - ✓ 유망한 마디에 대해서만 그 마디의 자식마디(children)를 검색한다.

- 이 알고리즘은 다음과 같은 절차로 진행된다.

  1. 상태공간트리의 깊이우선검색을 실시한다.
  2. 각 마디가 유망한지를 점검한다.
  3. 만일 그 마디가 유망하지 않으면, 그 마디의 부모마디로 돌아가서 검색을 계속한다.



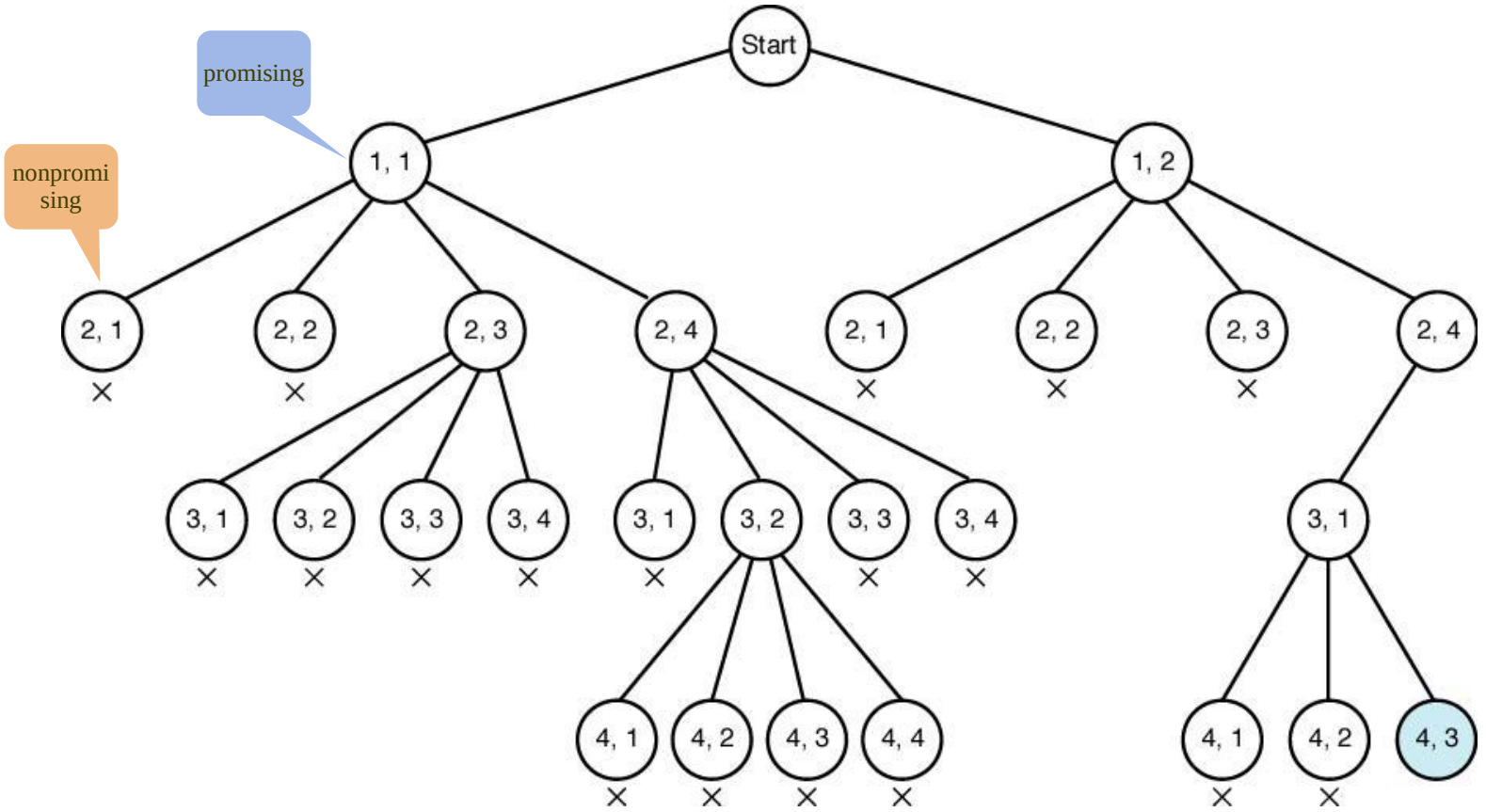
되추적 알고리즘

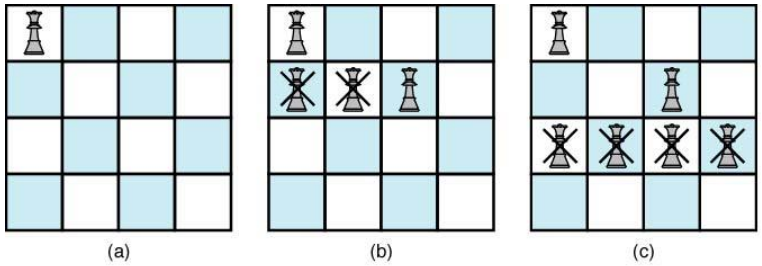
- 일반 되추적 알고리즘:

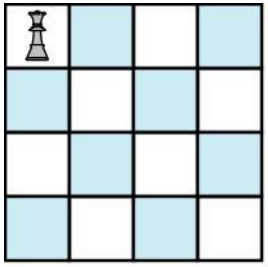
```
void checknode (node v) {
      node u;

  if (promising(v))
      if (there is a solution at v)
          write the solution;
      else
          for (each child u of v)
              checknode(u);
}
```

- 노드를 방문 후 유망성을 검증

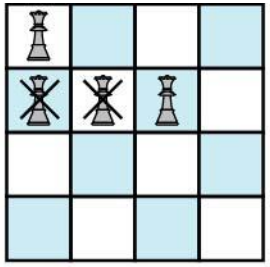
4-Queens 문제의 상태공간트리 (되추적)



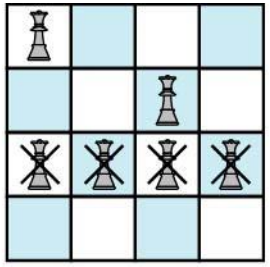




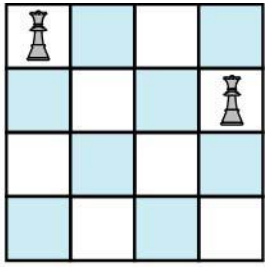
(a)



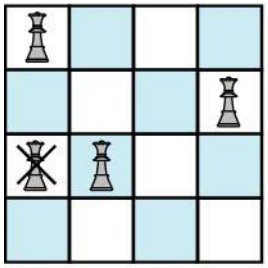
(b)



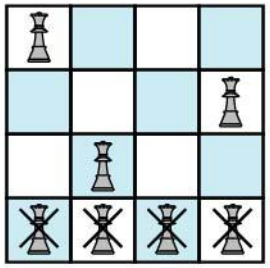
(c)



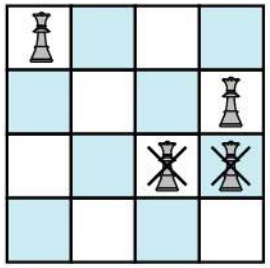
(d)



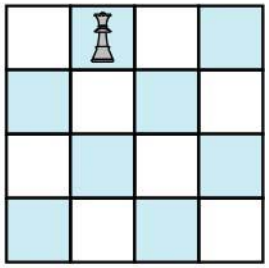
(e)



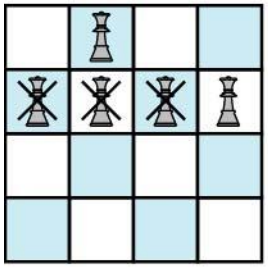
(f)



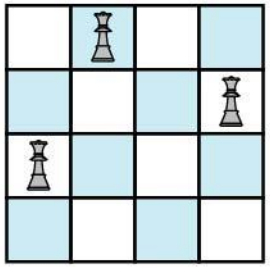
(g)



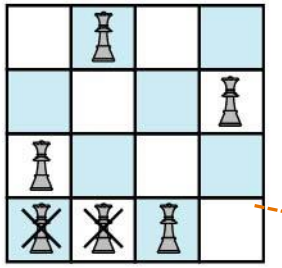
(h)



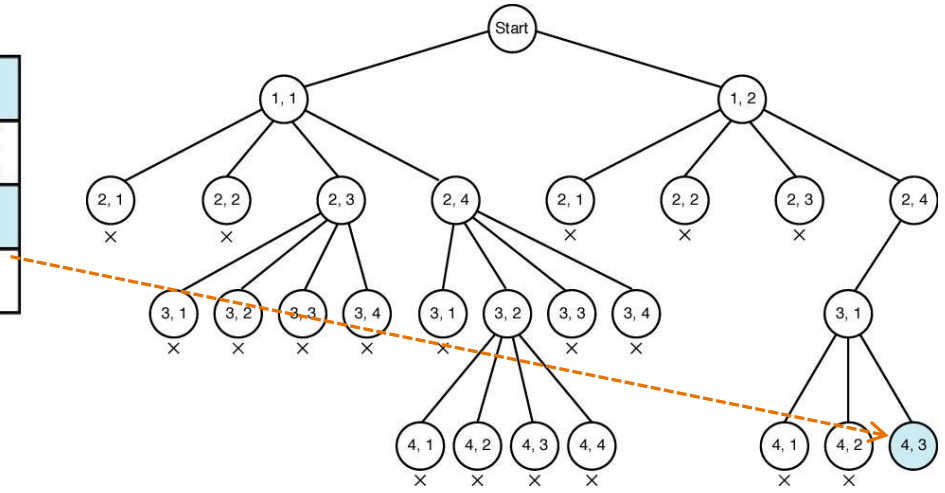
(i)



(j)



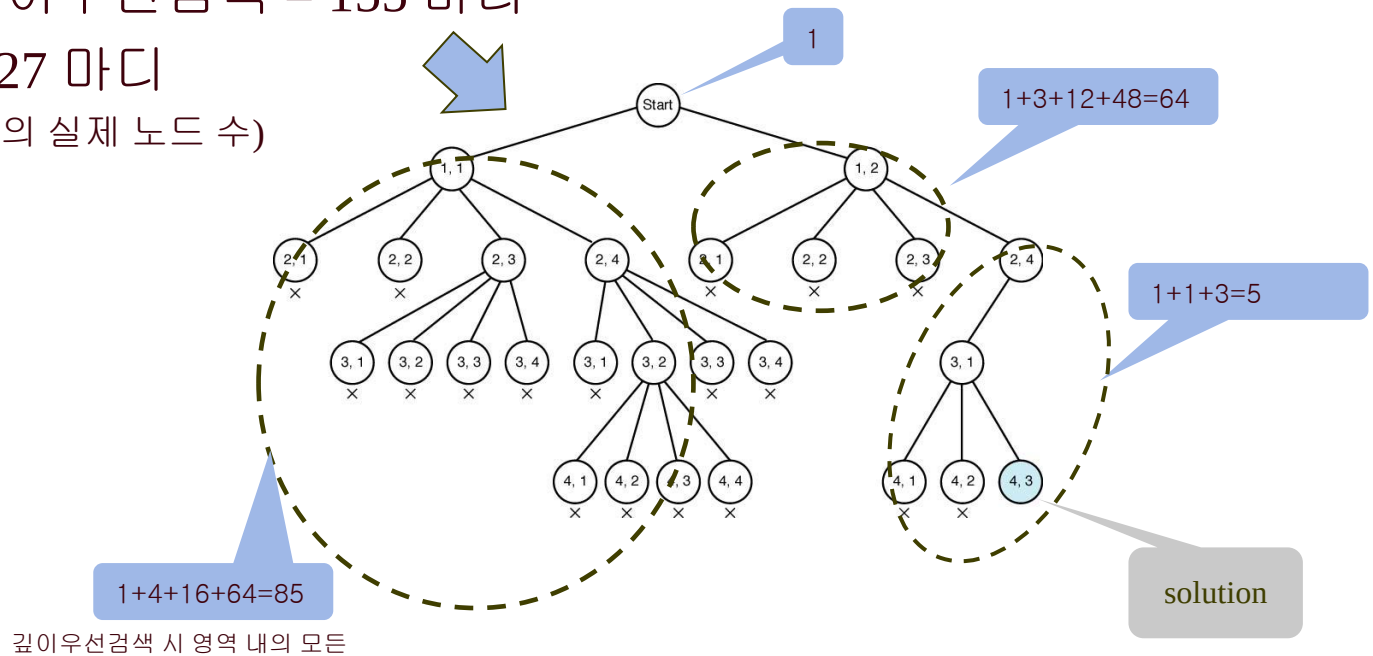
(k)



깊이우선검색 vs. 되추적

- 실제적으로 트리를 생성하지 않고 묵시적으로(implicitly) 존재

- 검색하는 마디 개수의 비교
  - ✓ 순수한 깊이우선검색 = 155 마디
  - ✓ 되추적 = 27 마디
    (그림의 실제 노드 수)



깊이우선검색 시 영역 내의 모든
노드(그림에는 표시되어 있지 않음) 검색

4-Queens 문제: 되추적(개량)

```
void expand (node v) {
    node u;

    for (each child u of v)
      if(promising(u))
          if (there is a solution at u)
              write the solution;
          else
              expand(u);
}
```

```
void checknode (node v) {
   node u;
   if (promising(v))
      if (there is a solution at v)
         write the solution;
      else
        for (each child u of v)
           checknode(u);
}
```

- 이 개량된 알고리즘은 유망성 여부의 점검을 마디를 방문하기 전에 실시하므로, 그만큼 방문할 마디의 수가 적어져서 더 효율적이다.
- 그러나 일반 알고리즘이 이해하기는 더 쉽고, 일반 알고리즘을 개량된 알고리즘으로 변환하기는 간단하므로, 앞으로 이 강의에서의 모든 되추적 알고리즘은 일반 알고리즘과 같은 형태로 표시한다.

n-Queens 문제

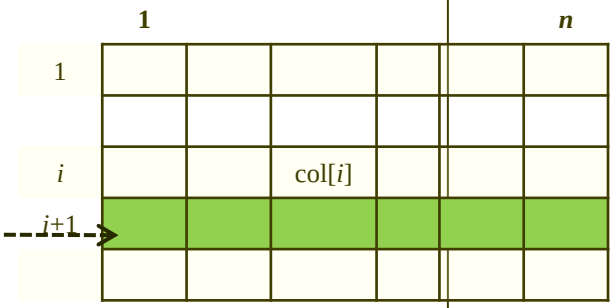
- *n*개의 Queen을 서로 상대방을 위협하지 않도록 $n \times n$ 서양장기(chess) 판에 위치시키는 문제

- 서로 상대방을 위협하지 않기 위해서는 같은 행이나, 같은 열이나, 같은 대각선 상에 위치하지 않아야 한다.

- ***n*-Queens 문제의 되추적 알고리즘**: 4-Queens 문제를 *n*-Queens 문제로 확장 시키면 된다.

col[*i*] = *i*번째 queen이 위치한 column 값

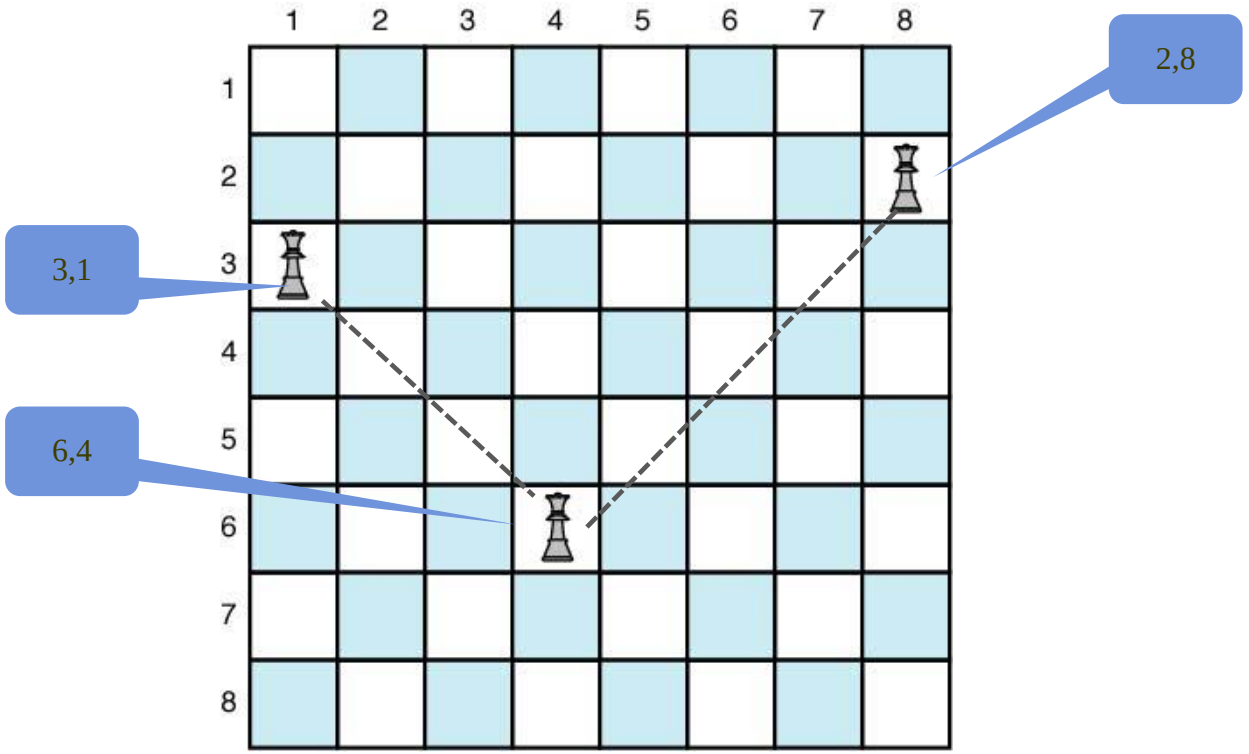
```
void queens(index i){
     index j;
   if(promising(i))
        if (i==n)
            cout << col[1] through col[n];
        else
             for (j=1; j<=n; j++){
                   col[i+1] = j;
                   queens(i+1);
             }
}
bool promising(index i) {
   index k;
   bool switch;

   k=1;
   switch = true;
   while ( k<i  && switch){
         if( col[i]==col[k] || abs(col[i]-col[k]) == i-k)
              switch = false;
         k++;
     }
   return switch;
 }
```



같은 column 인지 확인

같은 대각에 있는지 확인

Initially, queens(0).

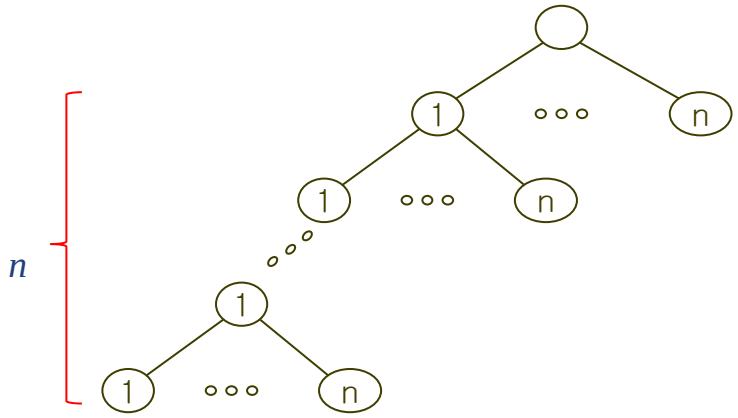

1 2 3 4 5 6 7 8
1
2
3
4
5
6
7
8
2,8
3,1
6,4

n-Queens 문제의 분석 I

- 상태공간트리 전체에 있는 마디의 수를 구함으로서, 가지 친 상태 공간트리의 마디의 개수의 상한을 구한다. 깊이가 i인 마디의 개수는 n^i개 이고, 이 트리의 깊이는 n이므로, 마디의 총 개수는 상한 (upper bound)은:

$$1+n+n^2+n^3+\ldots\cdot+n^n=\frac{n^{n+1}-1}{n-1}$$

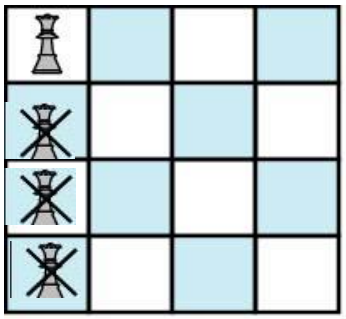
따라서 n = 8일 때, $\frac{8^9-1}{8-1}=19{,}173{,}961$



- 그러나 이 분석은 별 가치가 없다. 왜냐하면 되추적함으로 점검하는 마디 수를 얼마나 줄였는지 상한값을 구해서는 전혀 알 수 없기 때문이다.

n-Queens 문제의 분석 II

- 유망한 마디만 세어서 상한을 구한다. 이 값을 구하기 위해서는 어떤 두 개의 Queen이 같은 열상에 위치할 수 없다는 사실을 이용하면 된다.

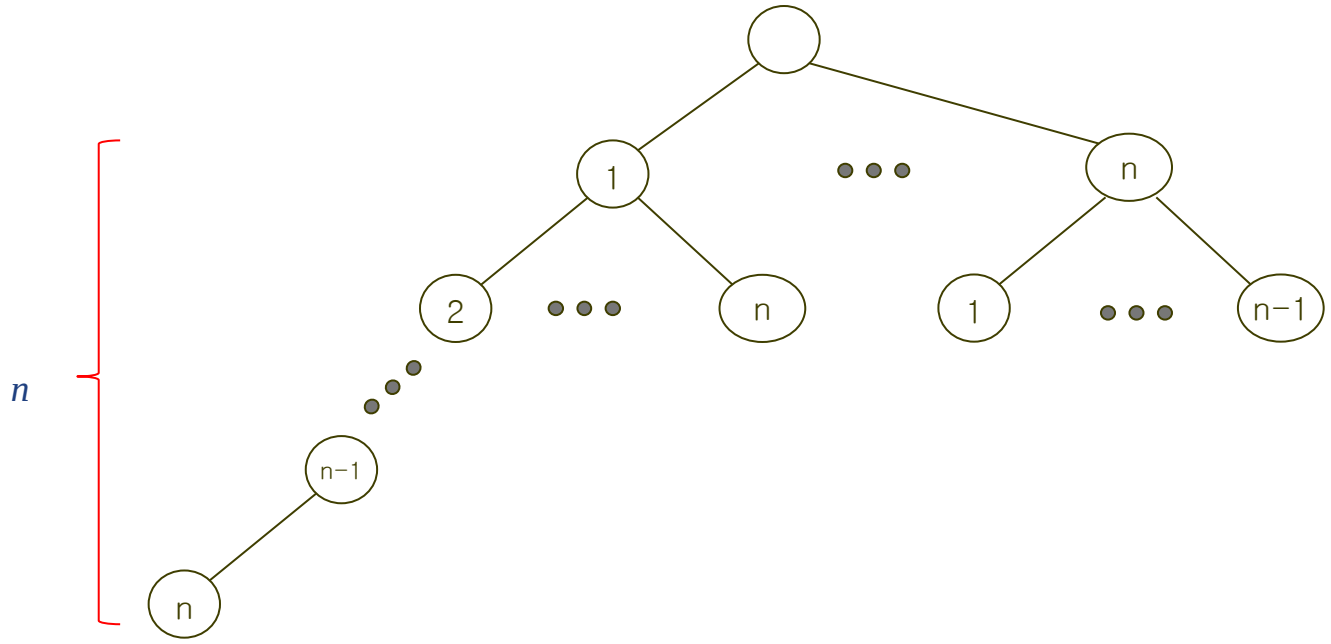


n-Queens 문제의 분석 II

- 예를 들어 $n = 8$. 첫번째 Queen은 어떤 열에도 위치시킬 수 있고, 두 번째는 기껏해야 남은 7열 중에서만 위치시킬 수 있고, 세 번째는 남은 6열 중에서 위치시킬 수 있다. 이런 식으로 계속했을 경우 마디의 수는 $1 + 8 + 8 \times 7 + 8 \times 7 \times 6 + \ldots + 8! = 109{,}601$가 된다. 이 결과를 일반화 하면 유망한 마디의 수는

$$1 + n + n(n-1) + n(n-1)(n-2) + \ldots + n!$$

을 넘지 않는다.



Discussion

- 위 2가지 분석 방법은 알고리즘의 복잡도를 정확히 설명해 주지 못하고 있다.
- 왜냐하면:
  - ✓ 대각선을 점검하는 경우를 고려하지 않았다. 따라서 실제 유망한 마디의 수는 훨씬 더 작을 수 있다.
  - ✓ 유망하지 않은 마디를 포함하고 있는데, 실제로 해석의 결과에 포함된 마디 중에서 유망하지 않은 마디가 훨씬 더 많을 수 있다.

n-Queens 문제의 분석 III

- 유망한 마디의 개수를 정확하게 구하기 위한 유일한 방법은 실제로 알고리즘을 수행하여 구축된 상태공간트리의 마디의 개수를 세어보는 수 밖에 없다.

- 그러나 이 방법은 진정한 분석 방법이 될 수 없다. 왜냐하면 분석은 알고리즘을 실제로 수행하지 않고 이루어져야 하기 때문이다.

All illustration of How Much Checking is Saved by Backtracking in the *n*-Queens Problem

Table 5.1 An illustration of how much checking is saved by backtracking in the *n*-Queens problem*

n	Number of Nodes Checked by Algorithm 1†	Number of Candidate Solutions Checked by Algorithm 2‡	Number of Nodes Checked by Backtracking	Number of Nodes Found Promising by Backtracking
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^{8}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	8.72×10^{10}	3.78×10^{8}	2.74×10^{7}

*Entries indicate numbers of checks required to find all solutions.

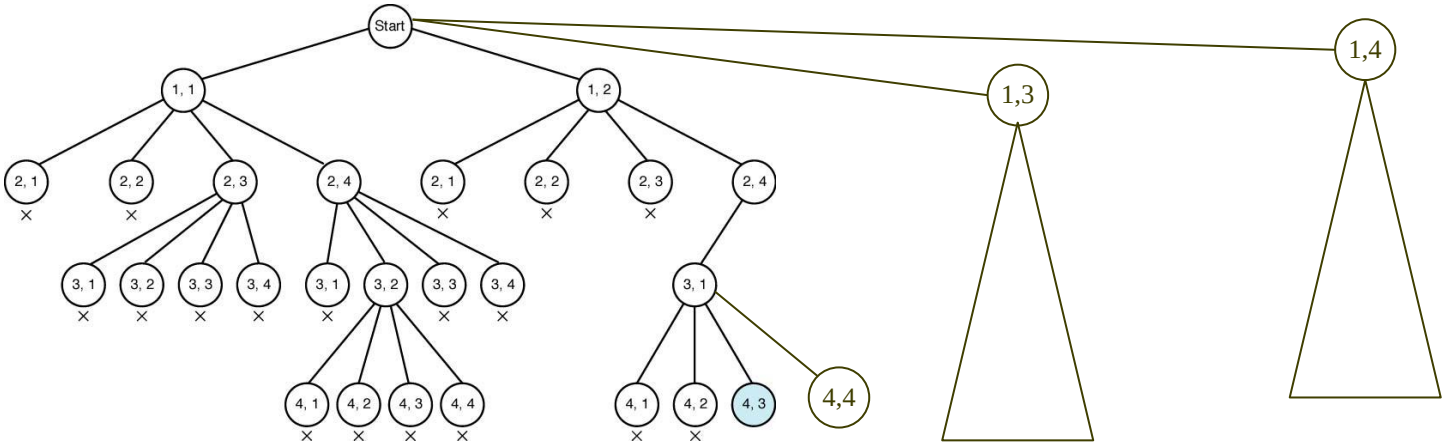
†Algorithm 1 does a depth-first search of the state space tree without backtracking.

‡Algorithm 2 generates the $n!$ candidate solutions that place each queen in a different row and column.

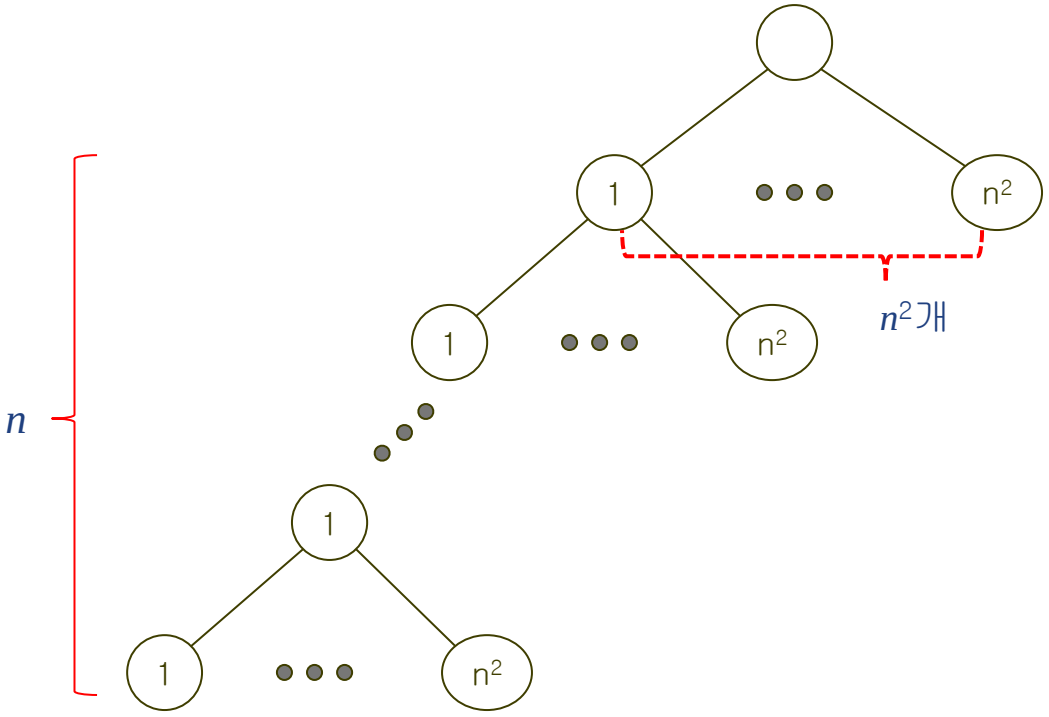
$$1 + n + n^2 + n^3 + \ldots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

$n!$

Find all solutions.



n^2개의 위치를 이용하여 상태공간트리를 만들면



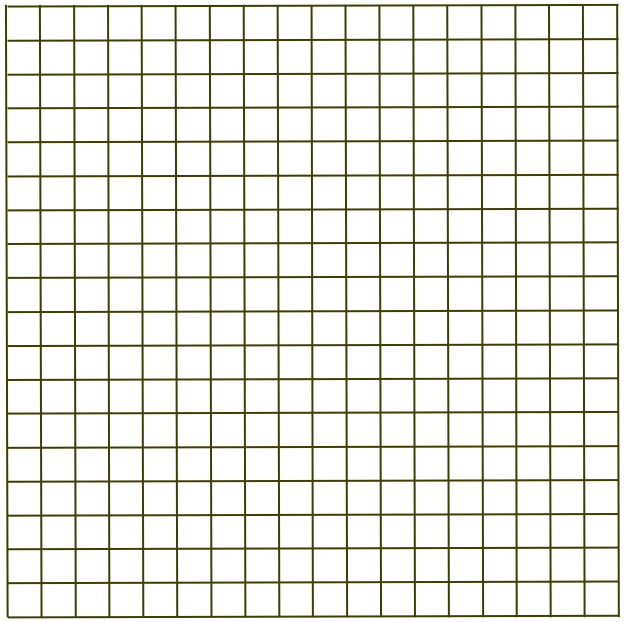
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

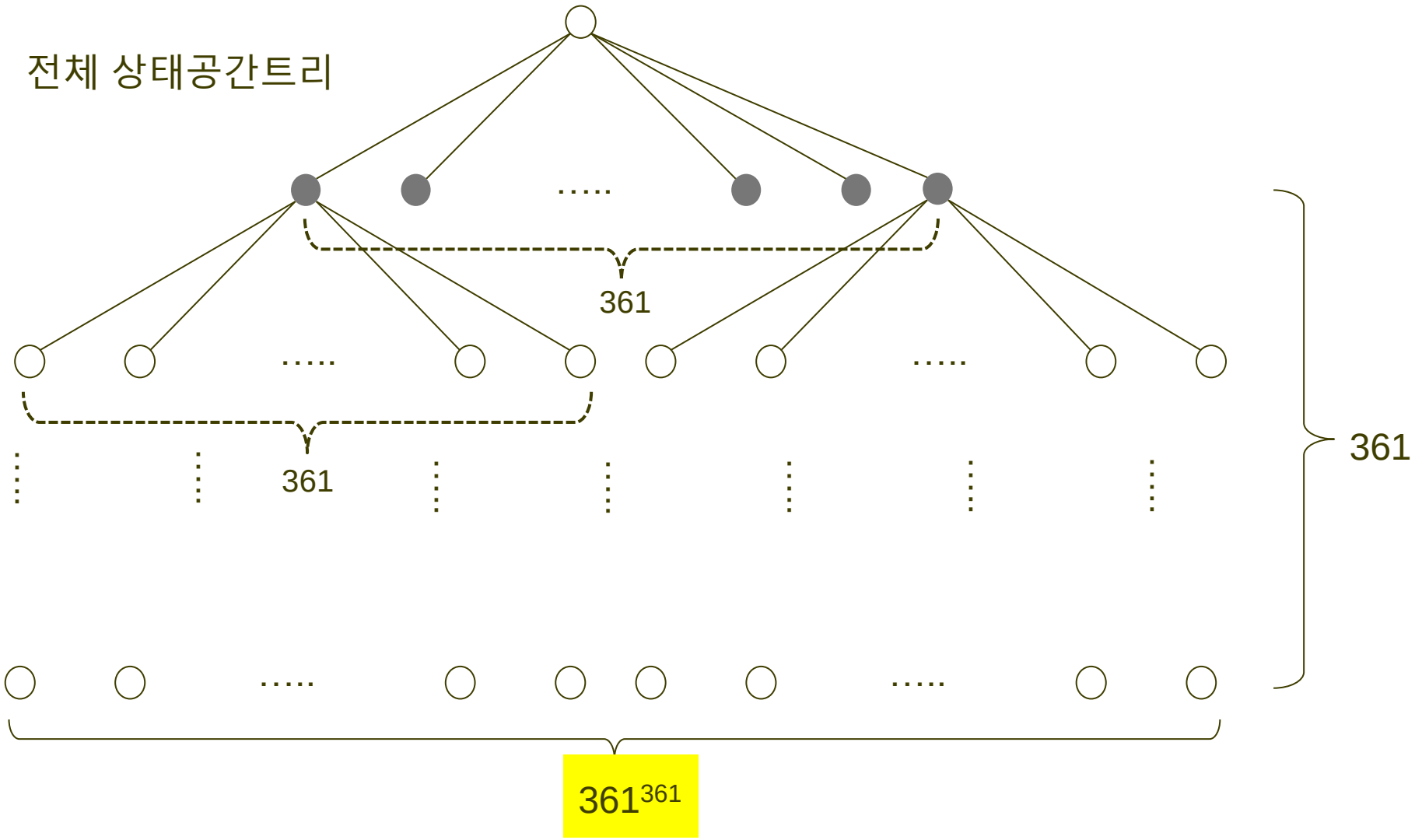
$(n^2)^n$개의 leaf 노드가 존재

바둑 게임에 되추적방법을 적용할 수 있는가?

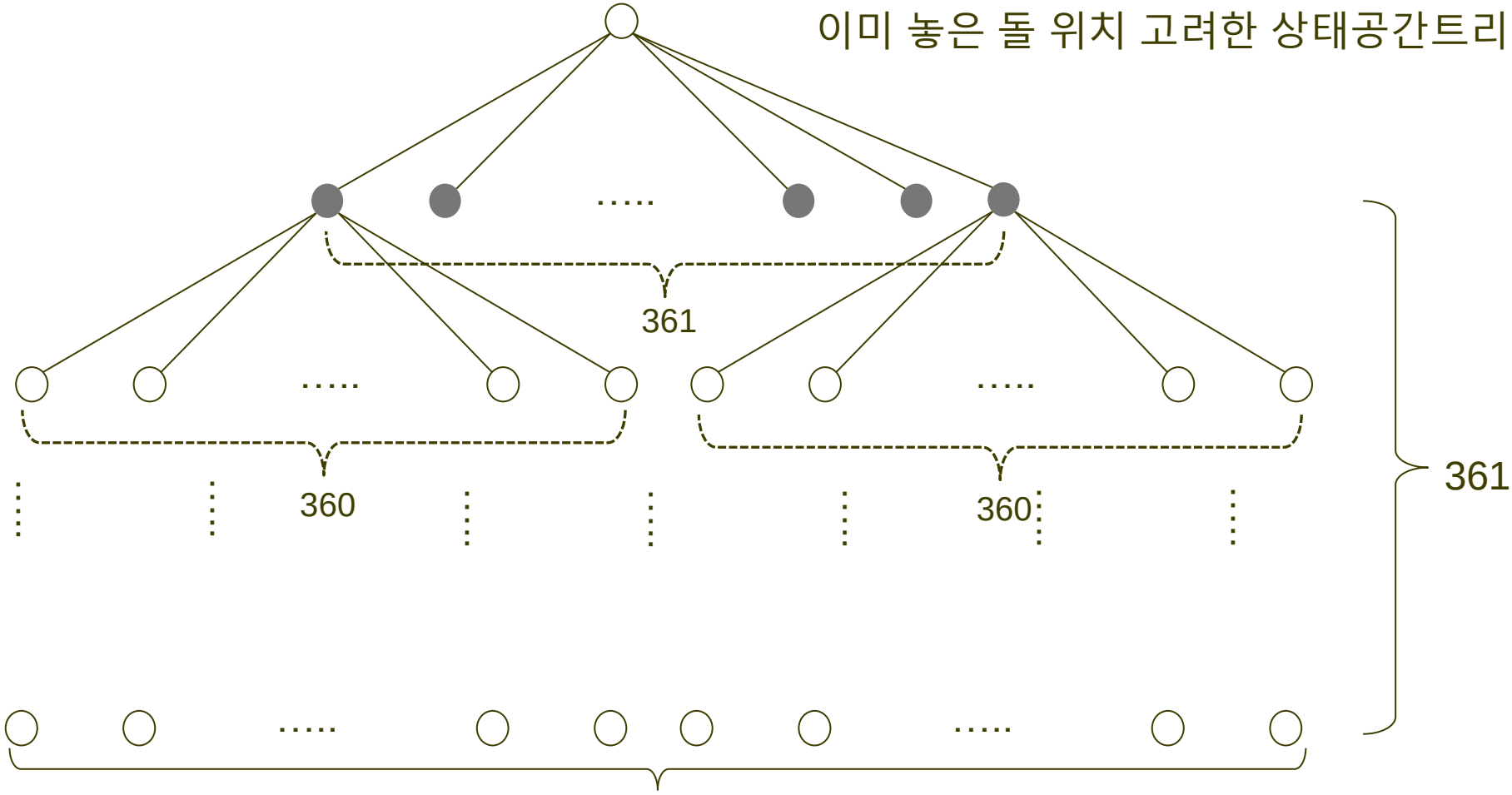
p

돌이 위치할 수 있는 지점의 개수 = 19 X19 = 361




전체 상태공간트리
361
361
361
361^{361}

이미 놓은 돌 위치 고려한 상태공간트리

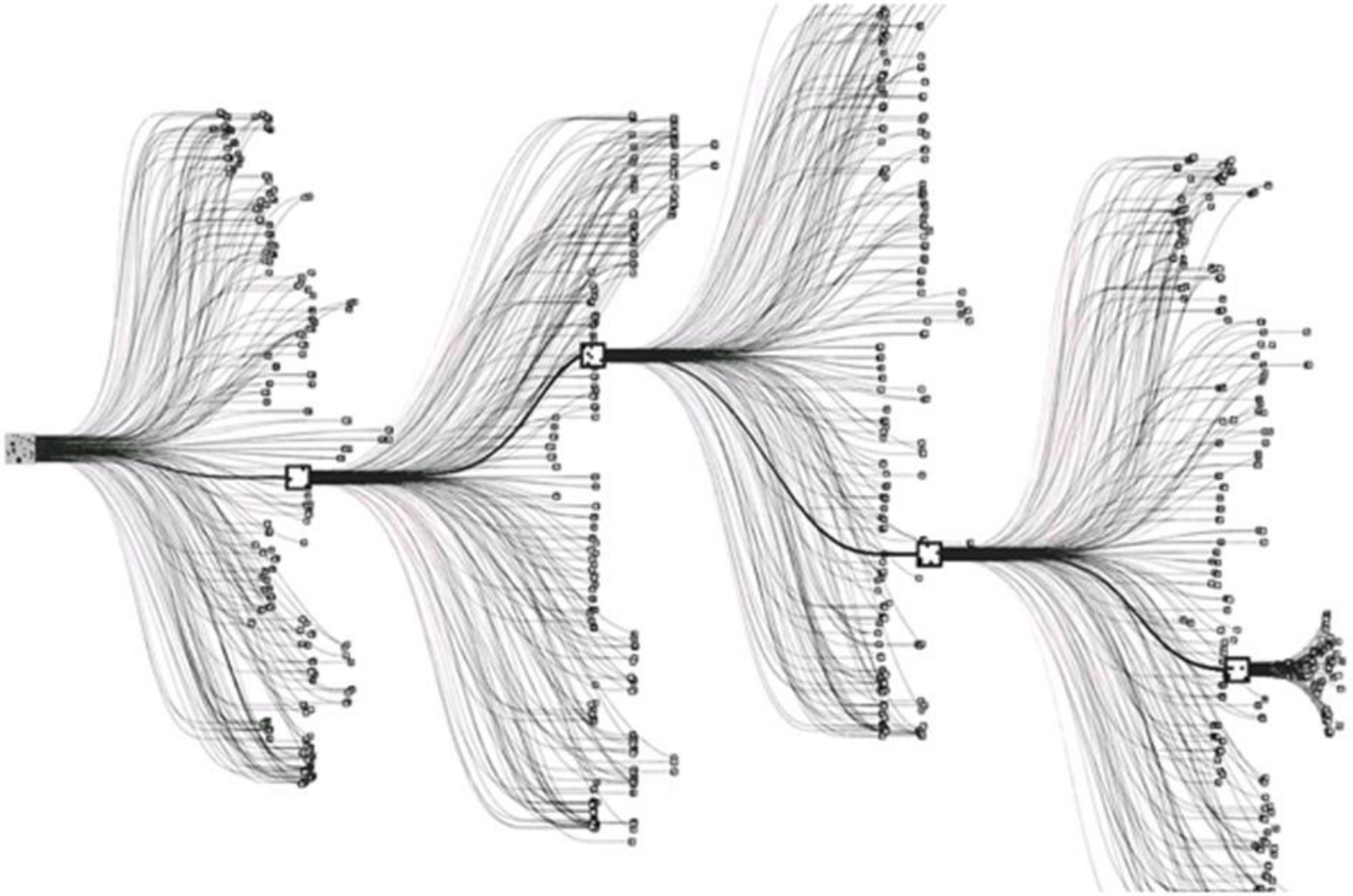


361x360x359x … x2x1= 361!

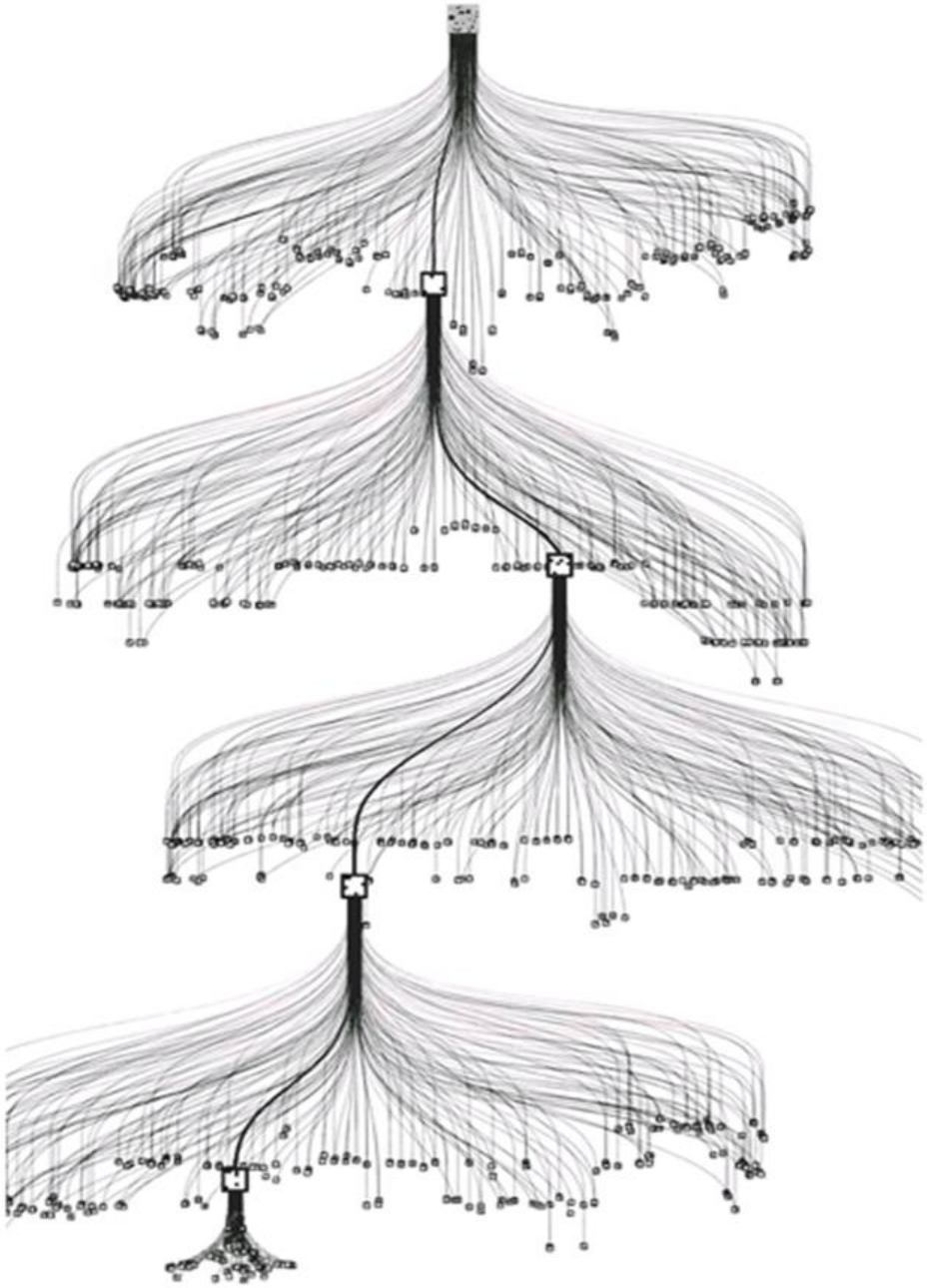
- 바둑에서 돌을 놓을 수 있는 조건을 적용하여 상태공간을 줄인다.
- 그렇다고 하여도 어마어마한 상태공간의 트리가 된다.
- 돌을 놓을 수 있는 수에 대해서 기대값을 계산하여 좋은 안을 채택한다.
- 인공지능: 기존의 수를 학습하여 최적의 수를 찾는다.


DeepMind


AlphaGo



Source : thenewstack.io



Source : thenewstack.io

부분집합의 합 구하기(sum of subsets problem)

- n개의 item을 이용하여 item 들의 무게의 합이 W가 되는 부분집합을 구한다.

$$\sum_{i=1}^{n} w_i x_i = W$$

$$x_i = 0 \quad \text{or} \quad 1, \quad for \;\; i = 1, n$$

- For S={1,4,6,8}, select items so that sum of the subset is 5.

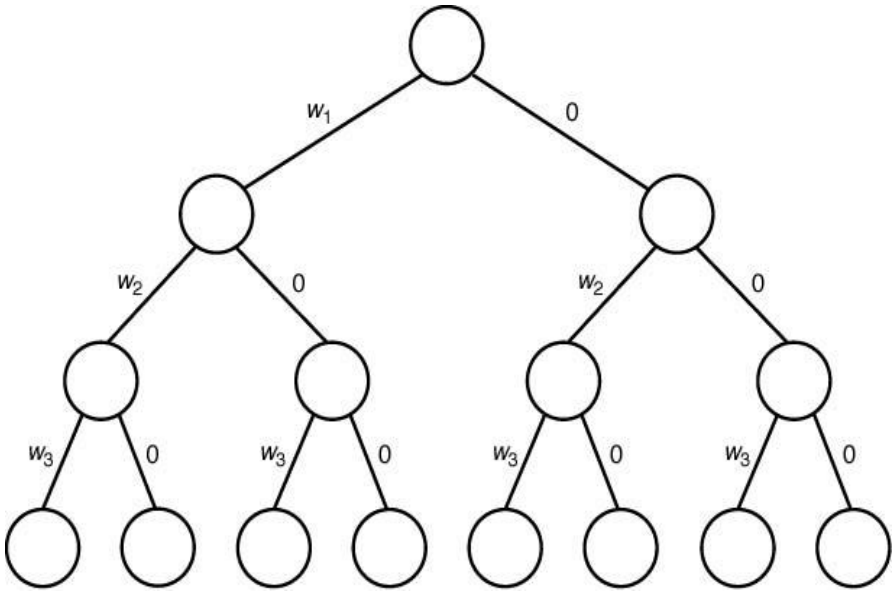
- ✓ $S=\{a_1, a_2, \ldots, a_n\}$인 경우 생성가능한 부분집합

S의 멱집합, power set $= 2^S$

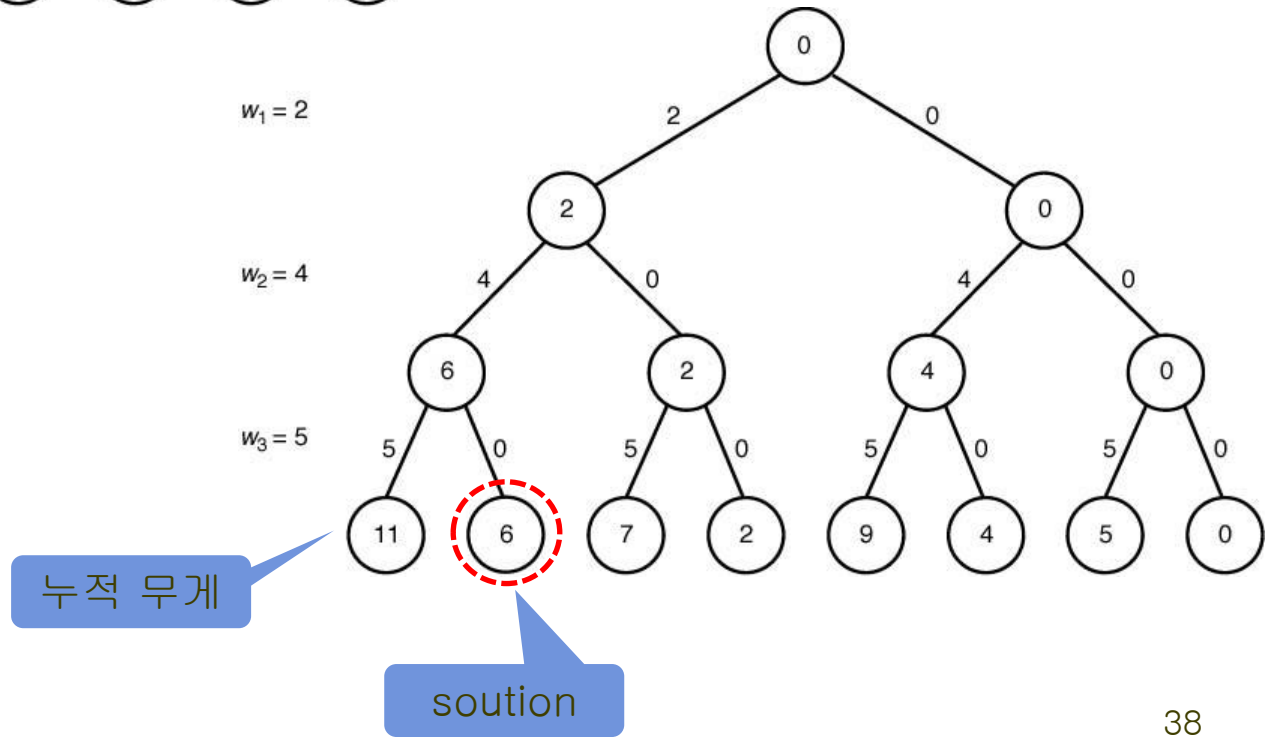
$2^S = \{ \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}, \ldots, \{a_1, a_2, \ldots, a_n\}\}$

- ✓ $S=\{a_1, a_2, \ldots, a_n\}$인 경우 생성가능한 부분집합의 개수

$|2^S| = 2^{|S|} = 2^n$



n=3, w_1=2, w_2=4, w_3=5, W=6



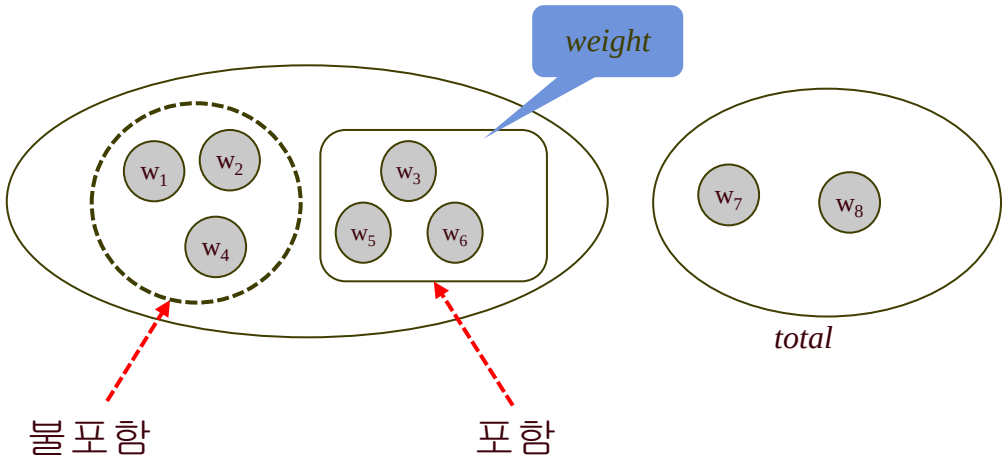
- 무게가 증가하는 순으로 데이터를 정렬→ 유망하지 않은지를 쉽게 판단할 수 있음. w_{i+1} 는 i수준에서 남아있는 가장 가벼운 아이템의 무게. w_{i+1}를 넣을 수 없으면 i+1 이후는 고려할 필요 없음.

- *weight*: 수준 i 의 마디까지 포함된 무게의 합

- *total*: 남아 있는 아이템의 무게의 총 합

- $weight + w_{i+1} > W$ *(if* weight $\neq W$*)* or
  $weight + total < W$ 이면 유망하지 않다.

At level 6,



n=4, W=13, w_1=3, w_2=4, w_3=5, w_4=6

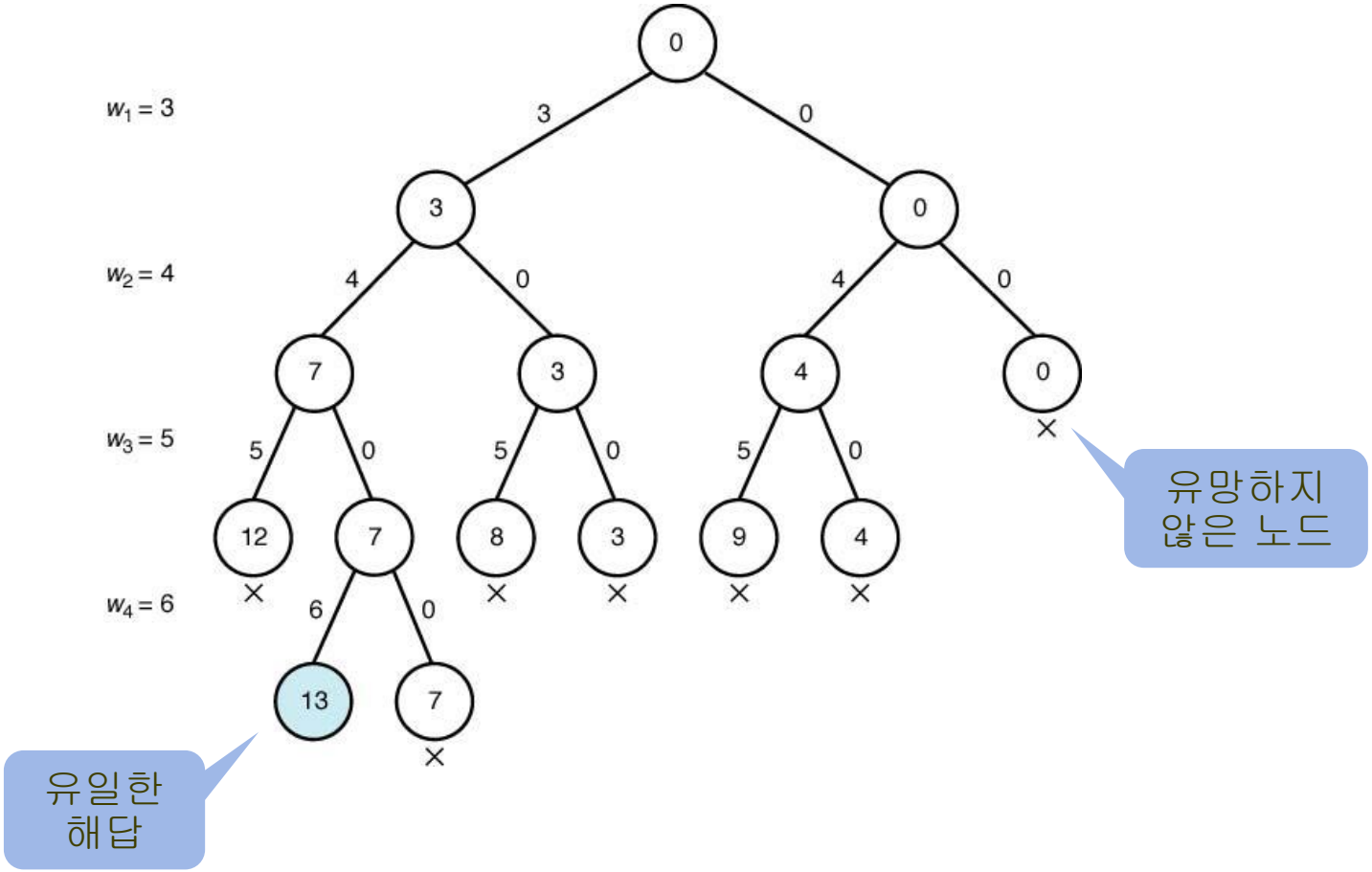


그림 5.9 가지친 상태공간트리. 총 15개의 마디 존재

```
void sum_of_subsets(index i, int weight, int total)
{
   if(promising(i))
        if (weight == W)
            cout << include[1] through include[i];
        else{
             include[i+1]="yes";
             sum_of_subsets(i+1, weight+w[i+1],total-w[i+1]);
             include[i+1]="no";
             sum_of_subsets(i+1, weight,total-w[i+1]);
             }
}
bool promising(index i){
  return(weight+total>=W)&&(weight == W ||weight+w[i+1]<=W);
}
```

w[i+1] 포함

w[i+1] 불포함

not

- $weight + w_{i+1} > W$ (*if* weight $\neq W$) or $weight + total < W$ 이면 유망하지 않다.

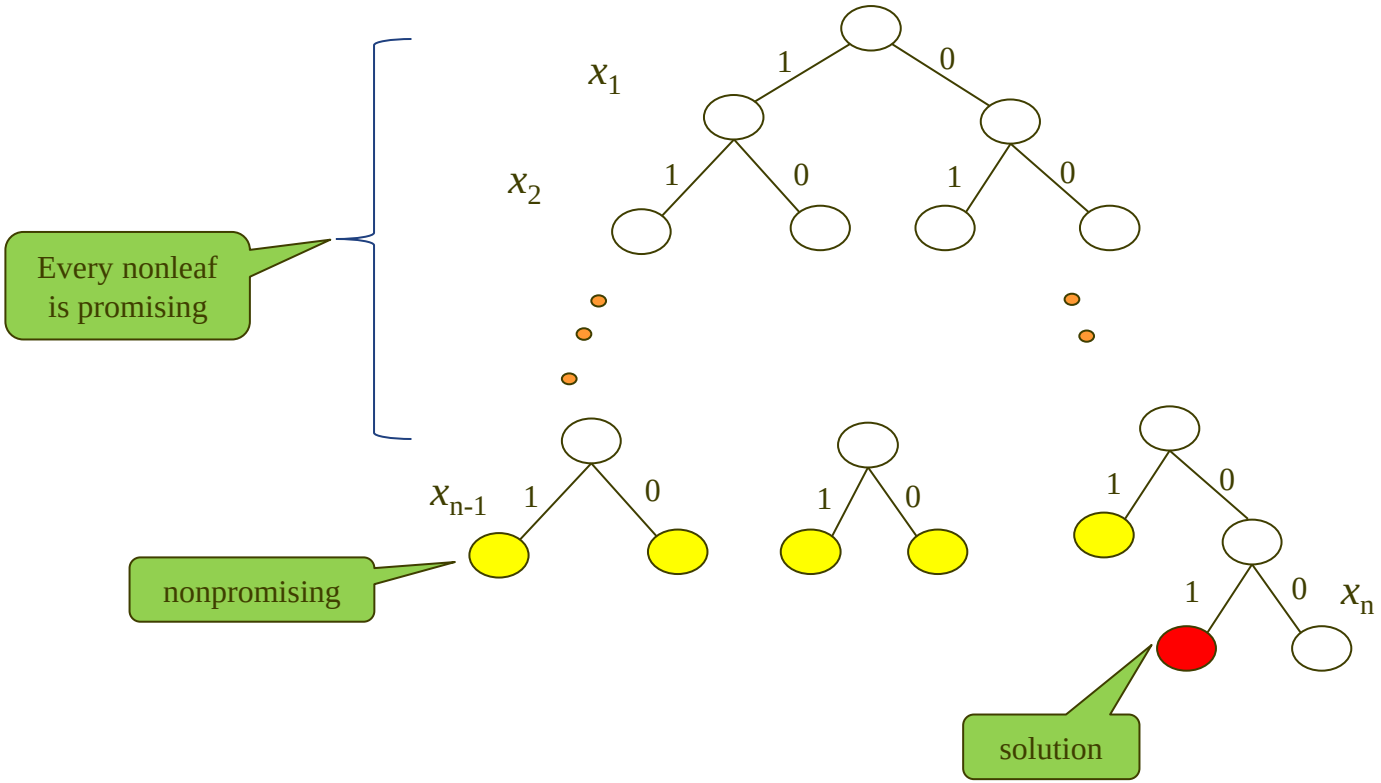
- 최상위 호출: sum_of_subsets(0, 0,total)

1. 검색하는 상태공간 트리에서 마디의 개수

$$1+2+2^2+\ldots+2^n = 2^{n+1}-1$$

2. $If \sum_{i=1}^{n-1} w_i < W, \quad w_n = W,$

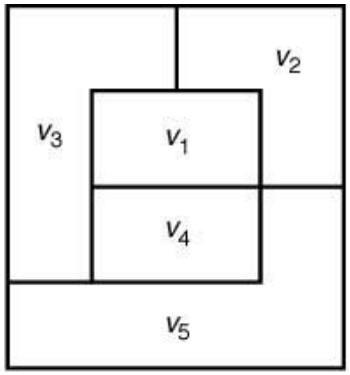
solution = $\{w_n\}$, 지수적으로 많은 개수의 마디를 방문



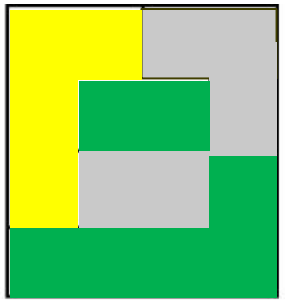
[연습문제] S={1,1,1,4}, W=4, 부분집합의 합 문제의 상태공간트리를 작성하시오.

[연습문제] S={1,2,4,8}, W=6, 부분집합의 합 문제의 상태공간트리를 작성하시오.

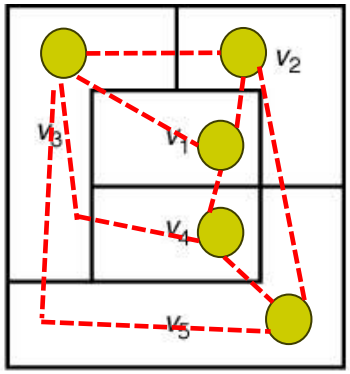
그래프 색칠하기(graph coloring)



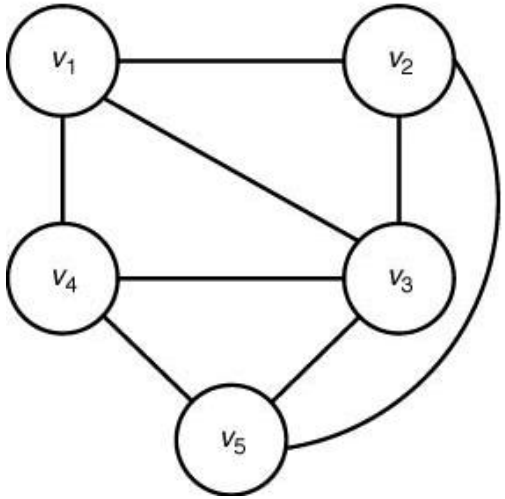
- 지도 칠하기(map coloring): 인접하는 지역을 구분하기 위해 색깔을 할당하는 문제
- 4개의 색깔이면 충분



map coloring

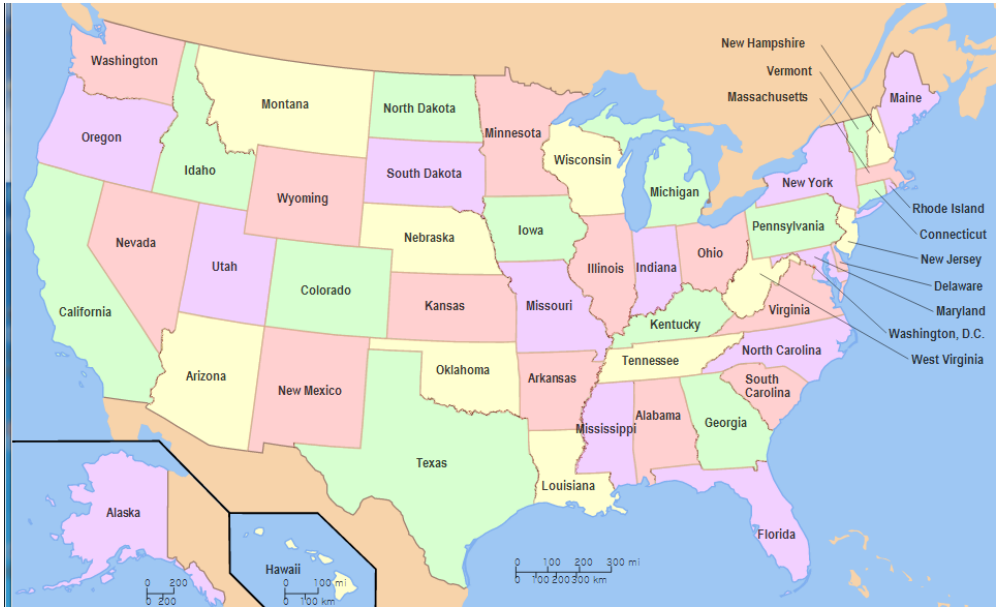


graph coloring



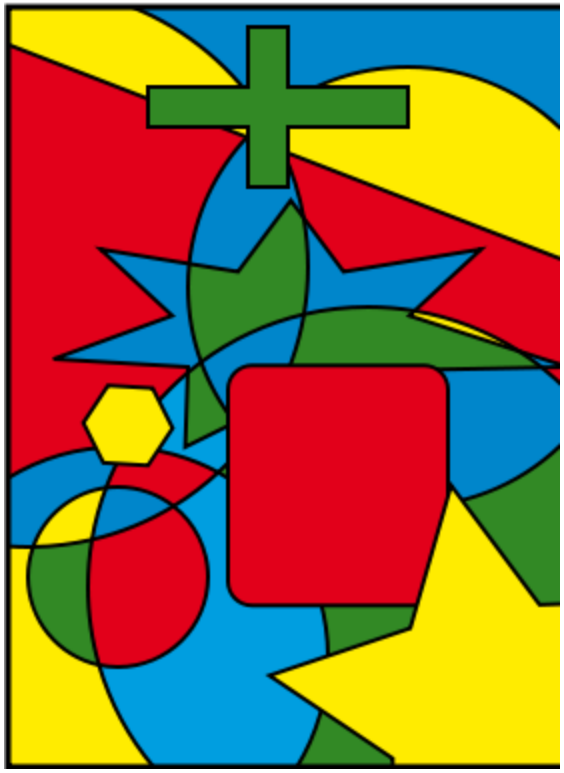
- 그래프 칠하기(graph coloring): 인접하는 노드에 같은 색깔이 할당되지 않도록 색깔을 할당하는 문제
- map의 한 지역(face)에 노드를 할당. 인접하는 face와 연결하는 에지를 설정 – 그래프 coloring
- planar graph인 경우 4개의 색깔이면 충분

✓ map 은 planar graph 로 변환가능



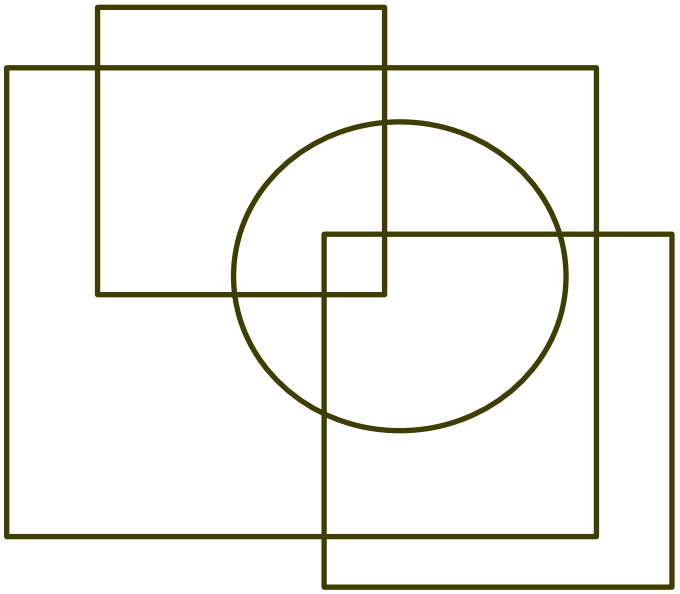
http://cafe.naver.com/solitone4609/798

from wikipedia



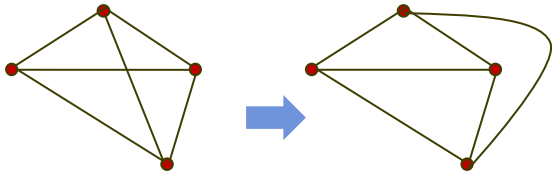
from wikipedia

coloring with 2 colors

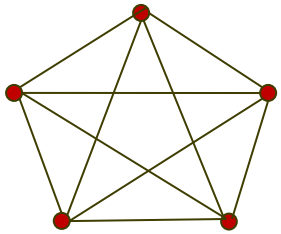


평면그래프(planar graph)

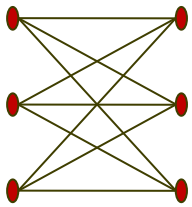
- 평면 상에서 이음선(edge)들이 서로 엇갈리지 않게 그릴 수 있는 그래프.



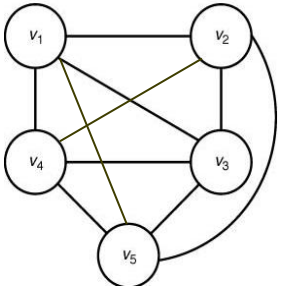
- 지도에서 각 지역을 그래프의 정점으로 하고, 한 지역이 어떤 다른 지역과 인접해 있으면 그 지역들을 나타내는 정점들 사이에 이음선을 그으면, 모든 지도는 그에 상응하는 평면그래프로 표시할 수 있다
- 다음은 대표적인 nonplanar 그래프이다.



완전 (complete) 그래프 K_5

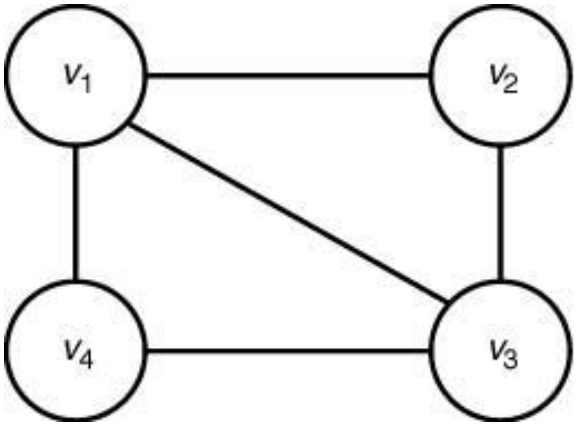


bipartite graph $K_{3,3}$



m coloring problem

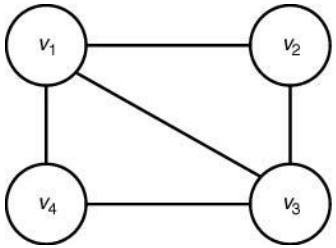
- **지도에 *m*가지 색으로 색칠하는 문제**
  - ✓ *m*개의 색을 가지고, 인접한 지역이 같은 색이 되지 않도록 지도에 색칠하는 문제

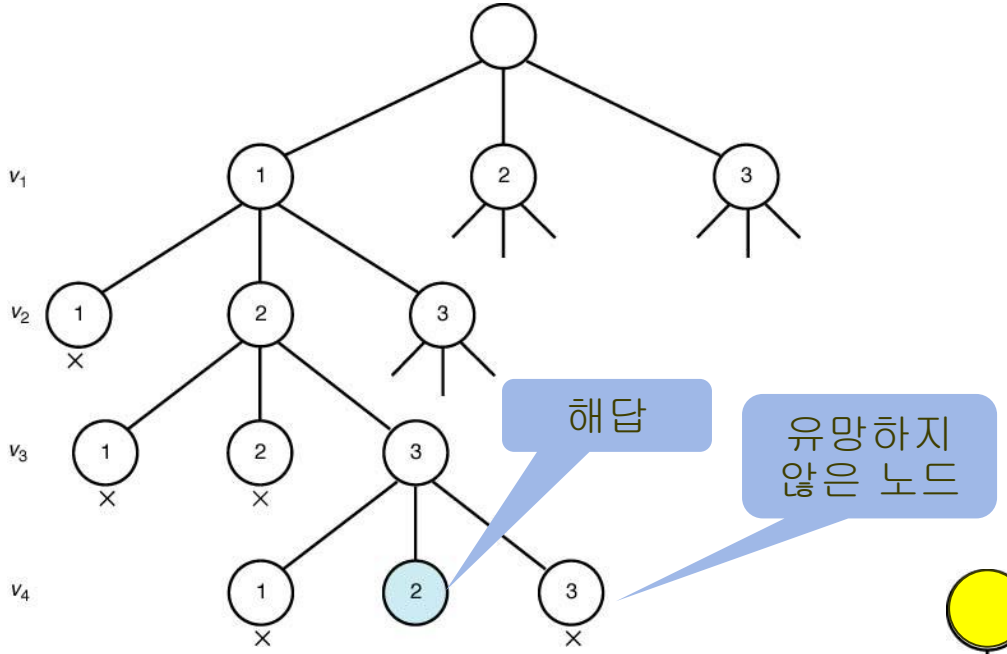


- 이 그래프에서 두 가지 색으로 문제를 풀기는 불가능하다.
- 세 가지 색을 사용하면 총 6가지의 해답을 얻을 수 있다.

그래프 색칠하기 되추적 해법

3종류 색깔 사용할 경우의 해 찾기





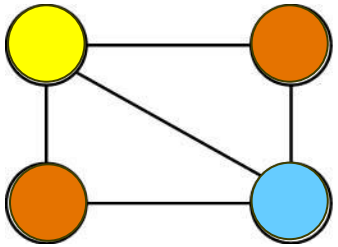


그림 5.12 가지친 상태공간트리

Fig 5.12 A portion of the pruned state space tree produced using backtracking to do a 3-coloring of the graph in Fig.5.10

```
void m_coloring(index i) {
    int color;

   if(promising(i))
       if(i==n)
           cout << vcolor[1] through vcolor[n];
       else
            for (color=1; color<=m; color++){
                  vcolor[i+1] = color;
                  m_coloring(i+1);
            }
}
bool promising(index i) {
   index j;
   bool switch;

    switch = true;
    j=1;
    while ( j<i  && switch){
         if(W[i][j] && vcolor[i]==vcolor[j]) // W[i][j]:연결표시.  T or F
                switch = false;
         j++;
     }
     return switch;
   }
```

vcolor[i]=노드i의 color

서로 인접한 것이 같은 색깔인지 확인.

연결되어 있는지

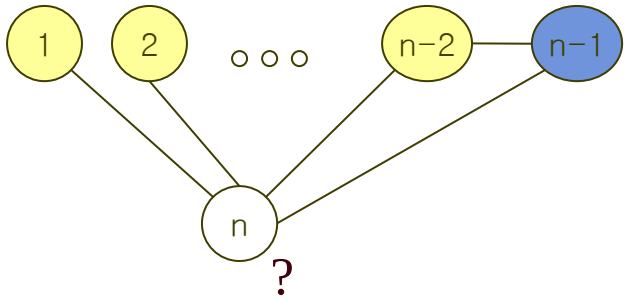
- 최상위 호출: m_coloring(0)

그래프 색칠하기: 분석

- m개의 색. n개의 정점을 가진 비방향그래프
- 상태공간트리 상의 마디의 총수

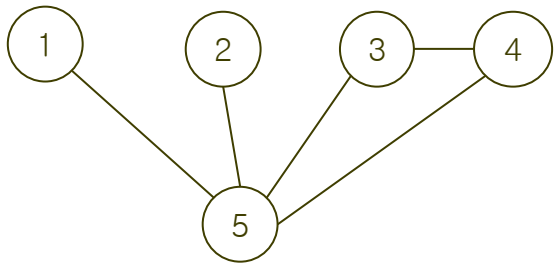
$$1 + m + m^2 + \ldots \cdot + m^n = \frac{m^{n+1} - 1}{m - 1}$$

- 최악의 경우



- m=2일 경우
- 거의 모든 경우를 생성하지 않고는 답이 없다는 것을 알 수 없다.

[연습문제] 다음 그래프의 2-coloring 문제를 backtracking으로 해결하시오. 상태공간트리를 작성하시오.




5장 끝
수고하셨습니다.