

# Backtracking

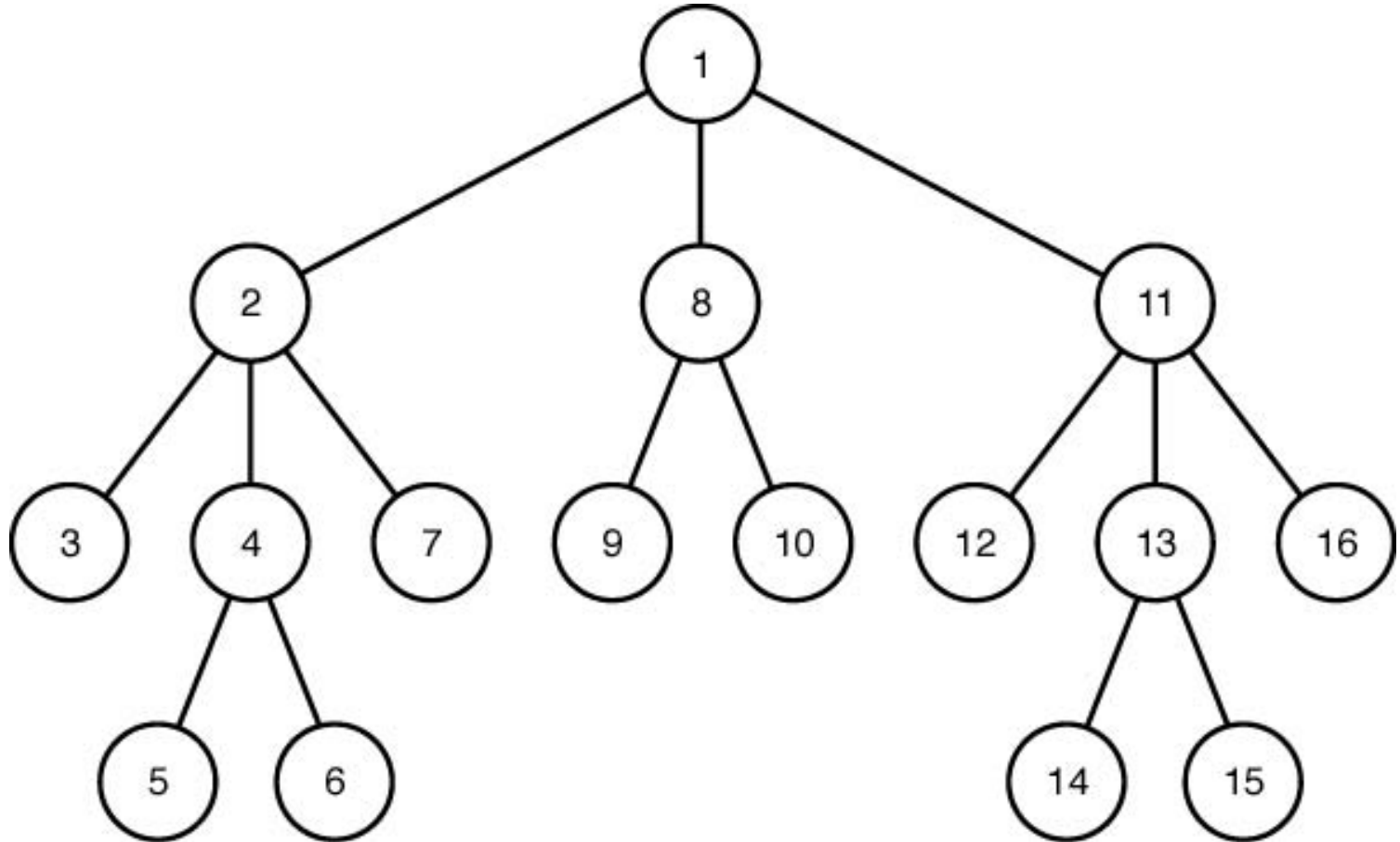
1. The Backtracking Technique
2. The n-Queens Problems
3. Graph Coloring
4. The Hamiltonian Circuits Problem
5. The Sum-of-Subsets Problem
6. The 0-1 Knapsack Problem
7. Using a Monte Carlo Algorithm

# Depth-First Search (깊이우선검색)

- 뿌리노드(**root**)가 되는 노드(**node**)를 먼저 방문한 뒤, 그 노드의 모든 후손노드(**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)  
}
```

# Depth-First Search



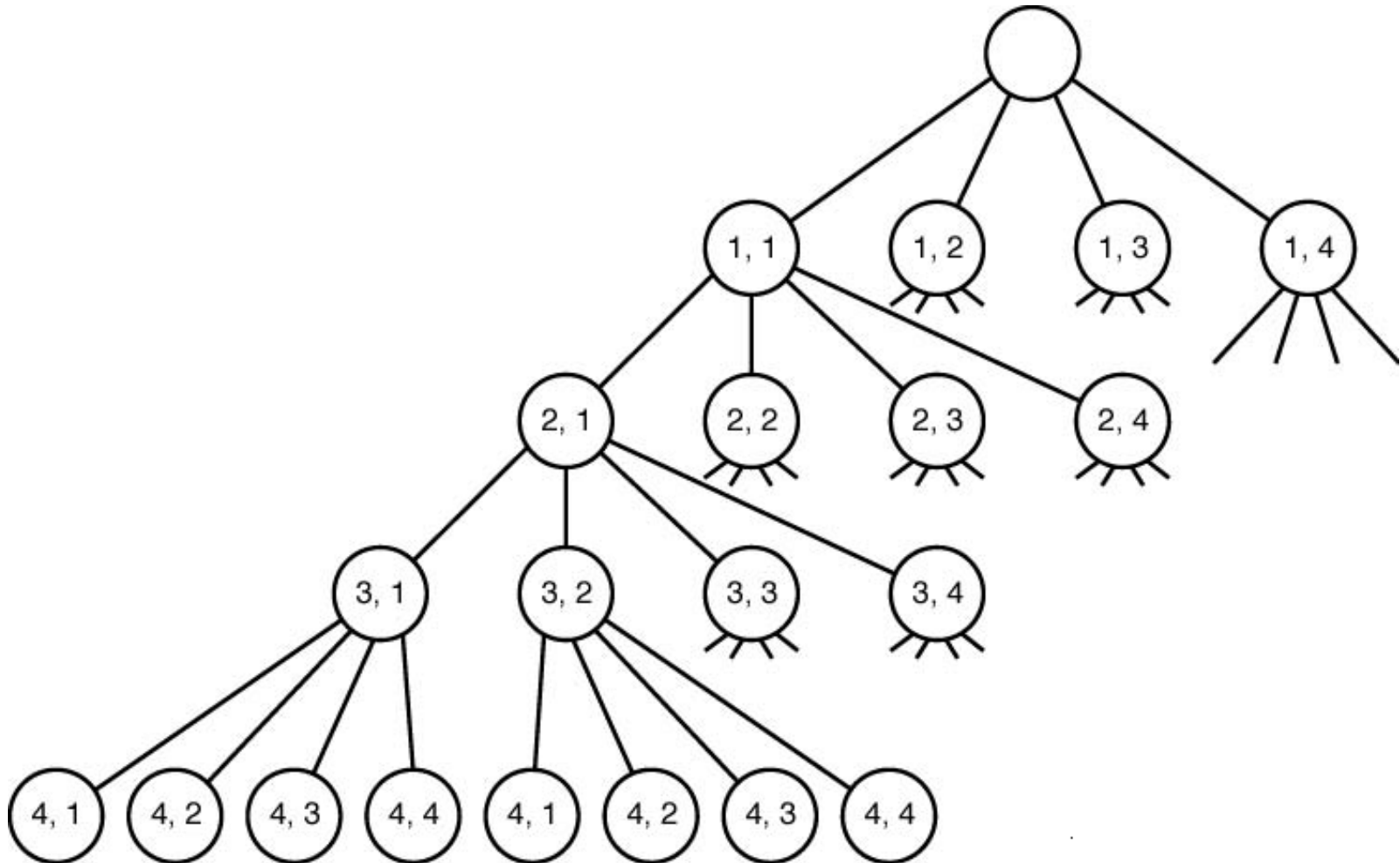
# 4-Queens Problem

## □ 4-Queens Problem

- 4개의 **Queen**을 서로 상대방을 위협하지 않도록  $4 \times 4$  서양장기(chess)판에 위치시키는 문제이다.
- 서로 상대방을 위협하지 않기 위해서는 같은 행이나, 같은 열이나, 같은 대각선 상에 위치하지 않아야 한다.
- 무작정 알고리즘
  - 각 **Queen**을 각각 다른 행에 할당한 후에, 어떤 열에 위치하면 해답은 얻을 수 있는지를 차례대로 점검해 보면 된다.
  - 이때, 각 **Queen**은 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 Problem

## 4-Queens 문제의 상태공간트리



# 4-Queens Problem

## □ State Space Tree (상태 공간 트리)

- 뿌리노드에서 잎노드(**leaf**)까지의 경로가  
해답후보(**candidate solution**)가 되는데,  
깊이우선검색을 하여 그 해답후보 중에서 해답을 찾을 수 있다.
- 이 Tree를 State Space Tree (상태공간트리)라고 한다.
- 그러나 이 방법을 사용하면 해답이 될 가능성이 전혀 없는  
노드의 후손노드(**descendant**)들도 모두 검색해야 하므로  
비효율적이다.

# Back Tracking

## □ 정의: **promising**

- 전혀 해답이 나올 가능성이 없는 노드는 유망하지 않다 (**non-promising**)고 하고,  
● 그렇지 않으면 유망하다(**promising**)고 한다.

## □ 되추적이란?

- 어떤 노드의 유망성을 점검한 후, 유망하지 않다고 판정되면 그 노드의 부모노드(**parent**)로 돌아가서 (“**backtrack**”) 다음 후손노드에 대한 검색을 계속 진행하는 과정이다.

# Back Tracking

## □ 되추적 알고리즘의 개념

- 되추적 알고리즘은 상태공간트리에서 깊이우선검색을 실시하는데,
- 유망하지 않은 노드들은 가지치서(**pruning**) 검색을 하지 않으며,
- 유망한 노드에 대해서만 그 노드의 자식노드(**children**)를 검색한다.

## □ 진행 절차

1. 상태공간트리의 깊이우선검색을 실시한다.
2. 각 노드가 유망한지를 점검한다.
3. 만일 그 노드가 유망하지 않으면,  
    그 노드의 부모노드로 돌아가서 검색을 계속한다.



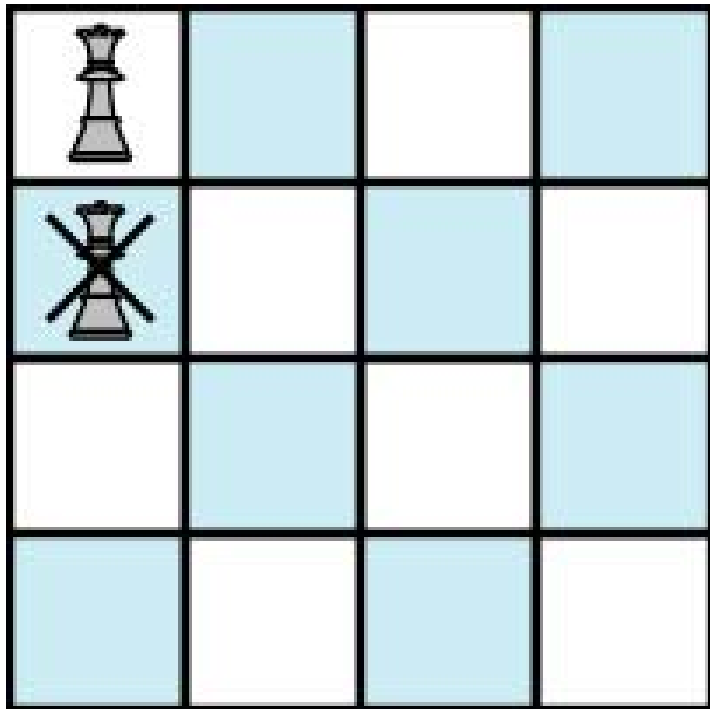
# Back Tracking

- 일반 되추적 알고리즘:

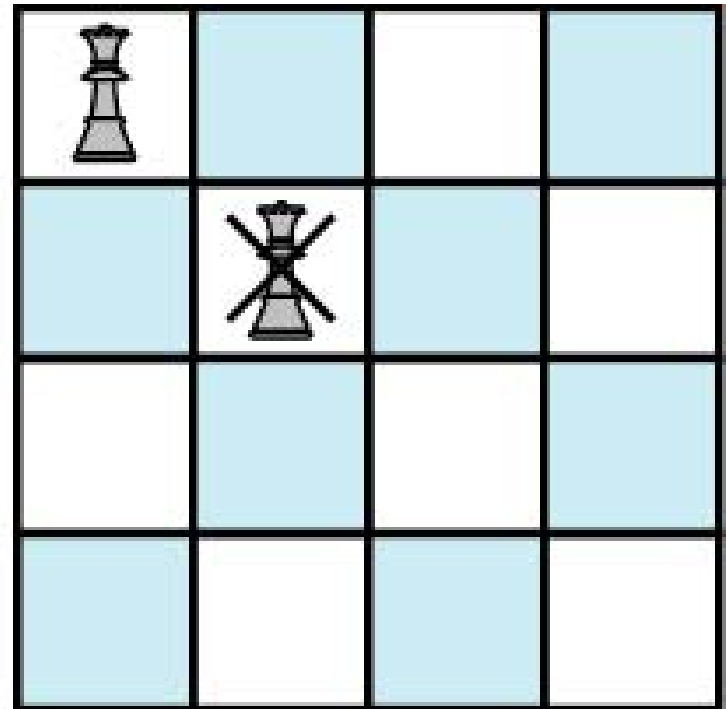
```
void checknode (node v) {  
    node u;  
  
    if (promising(v))  
        if (there is a solution at v)  
            write the solution;  
    else  
        for (each child u of v)  
            checknode(u);  
}
```

- See Example 5.1, page 181-184

# Back Tracking



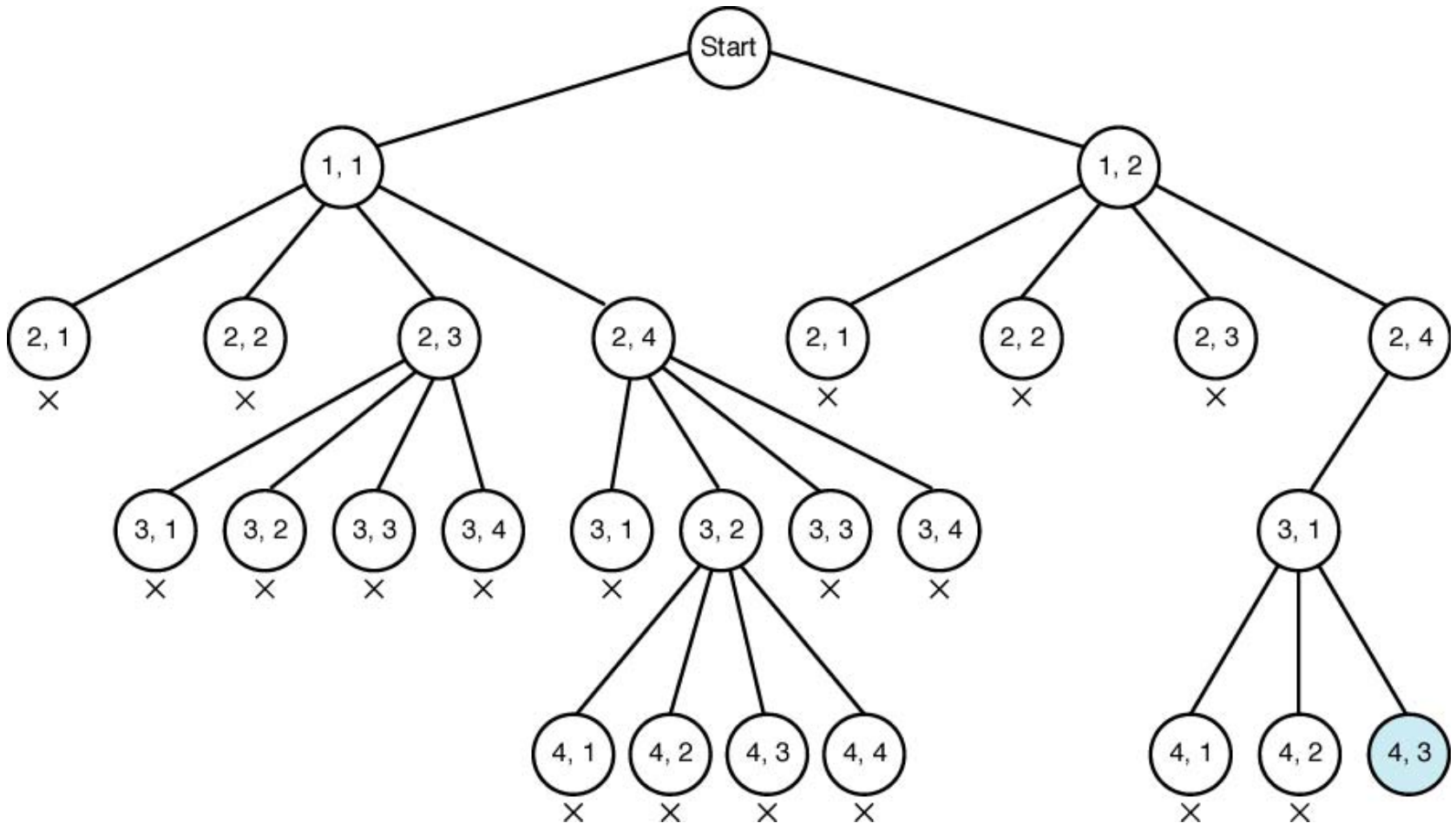
(a)



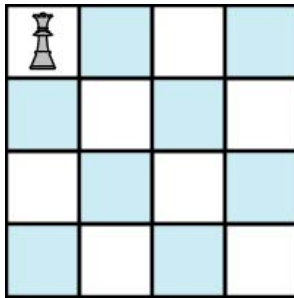
(b)

# Back Tracking

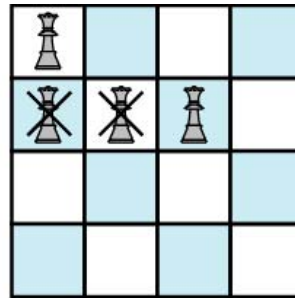
## 4-Queens 문제의 상태 공간 트리 (되추적)



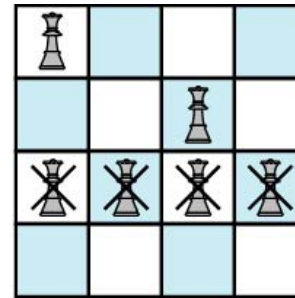
# Back Tracking



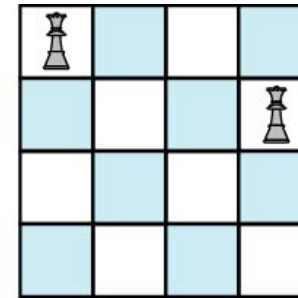
(a)



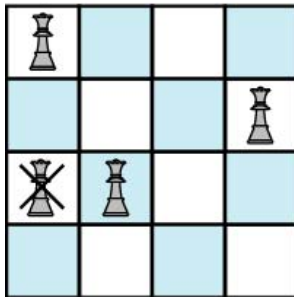
(b)



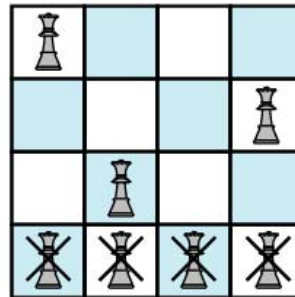
(c)



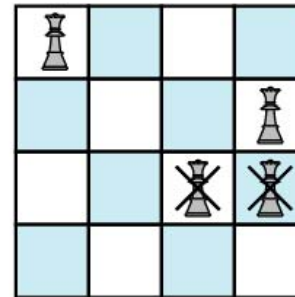
(d)



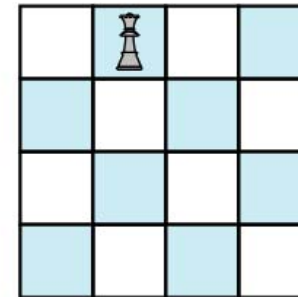
(e)



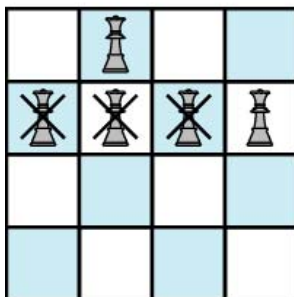
(f)



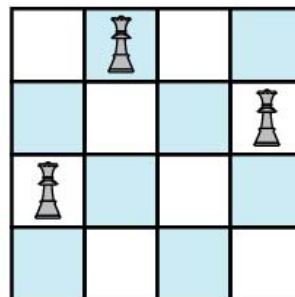
(g)



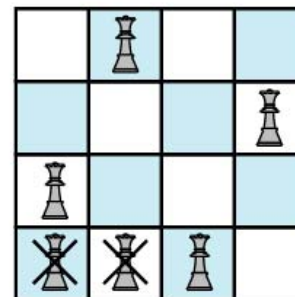
(h)



(i)



(j)



(k)

# Back Tracking

- 깊이우선검색 vs 되추적
  - 검색하는 노드 개수의 비교
    - 순수한 깊이우선검색 = 155 노드
    - 되추적 = 27 노드

# *n*-Queens Problem

## □ *n*-Queens problem

- *n*개의 Queen을 서로 상대방을 위협하지 않도록  $n \times n$  서양장기(chess) 판에 위치시키는 문제이다.  
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 확장시키면 된다.

# *n*-Queens Problem

## □ Promising Function

- must check two queens are in the same column or diagonal
- $\text{col}(i)$  : the column number where the queens in the  $i$ -th row is located.
- How to check whether the queen in  $k$ -th row is in the same column?
  - If  $(\text{col}(i) == \text{col}(k))$  nonpromising;
- How to check whether the queen in  $k$ -th row is in the same diagonal?
  - If  $(\text{abs}(\text{col}(i) - \text{col}(k)) == \text{abs}(i - k))$  nonpromising;
  - Ex.[Ref. Figure 5.6, page 186]
    - $\text{abs}(\text{col}(6) - \text{col}(3)) = \text{abs}(4 - 1) = 3 == \text{abs}(6 - 3) \Rightarrow$  nonpromising
    - $\text{abs}(\text{col}(6) - \text{col}(2)) = \text{abs}(4 - 8) = 4 == \text{abs}(6 - 2) \Rightarrow$  nonpromising

# *n*-Queens Problem

## □ Algorithm 5.1 (I)

- Problem : Position  $n$  queens on a chessboard  
so that no two are in the same row, column, or diagonal.
- Inputs :  $n$
- Outputs : The position number of each queen can be placed  
on an  $n*n$  chessboard
- 최상위호출: queens(0)

```
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);  
            }  
}
```



# *n*-Queens Problem

```
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]) == abs(i-k))  
            switch = FALSE;  
        k++  
    }  
    return switch;  
}
```

# *n*-Queens Problem

## □ *n*-Queens 문제의 분석 I

- 상태공간트리 전체에 있는 노드 (promising test)의 수를 구함으로서, 가지 친 상태공간트리의 노드의 개수의 상한을 구한다.
- 깊이가 *i*인 노드의 개수는  $n^i$ 개 이고, 이 트리의 깊이는 *n*이므로, 노드의 총 개수는 상한(upper bound)은:

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

- 따라서  $n = 8$ 일 때,  $\frac{8^9 - 1}{8 - 1} = 19,173,961$
- 그러나 이 분석은 별 가치가 없다.  
왜냐하면 되추적함으로서 점검하는 노드 수를 얼마나 줄였는지 상한값을 구해서는 전혀 알 수 없기 때문이다.

# *n*-Queens Problem

## □ *n*-Queens 문제의 분석 II

- 유망한 노드만 세어서 상한을 구한다.  
이 값을 구하기 위해서는 어떤 두 개의 **Queen**이 같은 열(**column**)에 위치할 수 없다는 사실을 이용하면 된다.

- 예를 들어  $n = 8$ 일 경우를 생각해 보자.  
첫번째 **Queen**은 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 노드의 수는  
 $1 + 8 + 8 \times 7 + 8 \times 7 \times 6 + \dots + 8! = 109,601$ 가 된다.

이 결과를 일반화하면 유망한 노드의 수는

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

을 넘지 않는다.

# *n*-Queens Problem

## □ 사색

- 위 2가지 분석 방법은 알고리즘의 복잡도를 정확히 얘기해주지 못하고 있다.
- 왜냐하면:
  - 대각선을 점검하는 경우를 고려하지 않았다.  
따라서 실제 유망한 노드의 수는 훨씬 더 작을 수 있다.
  - 유망하지 않은 노드를 포함하고 있는데,  
실제로 해석의 결과에 포함된 노드 중에서  
유망하지 않은 노드가 훨씬 더 많을 수 있다.

# *n*-Queens Problem

## □ *n*-Queens 문제의 분석 III

- 유망한 노드의 개수를 정확하게 구하기 위한 유일한 방법은 실제로 알고리즘을 수행하여 구축된 상태공간트리의 노드의 개수를 세어보는 수 밖에 없다.
- 그러나 이 방법은 진정한 분석 방법이 될 수 없다.  
왜냐하면 분석은 알고리즘을 실제로 수행하지 않고 이루어져야 하기 때문이다.
- **Monte Carlo** 방법으로 추정하는 것이 일반적이다. (5장 마지막에 배움)

# *n*-Queens Problem

## 알고리즘의 수행시간 비교

$n$	알고리즘 1 <sup>†</sup> 로 검사한 마디의 개수	알고리즘 2 <sup>‡</sup> 로 검사한 해답후보의 개수	되추적으로 검사한 마디의 개수	되추적으로 유망함을 알아낸 마디의 개수
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	$9.73 \times 10^{12}$	$4.79 \times 10^8$	$1.01 \times 10^7$	$8.56 \times 10^5$
14	$1.20 \times 10^{16}$	$8.72 \times 10^{10}$	$3.78 \times 10^8$	$2.74 \times 10^7$

\* 해답을 모두 찾는 데 필요한 검사횟수를 나타냄.

† 알고리즘 1은 되추적 없이 상태공간 트리를 깊이우선 검색함.

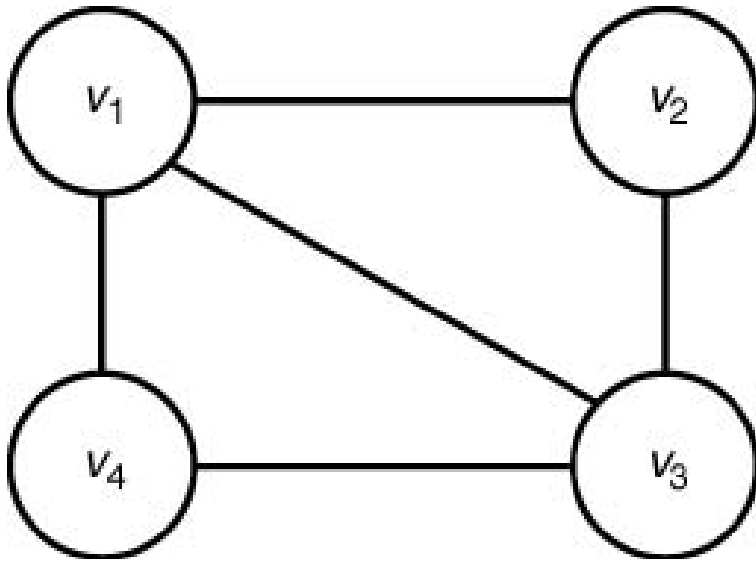
‡ 알고리즘 2는 각 여왕말을 다른 행과 열에 위치하는  $n!$ 개의 해답후보를 생성함.

# Graph Coloring

## □ Graph Coloring Problem

- $m$ -coloring – 지도에  $m$ 가지 색으로 색칠하는 문제
- $m$ 개의 색을 가지고, 인접한 지역이 같은 색이 되지 않도록 지도에 색칠하는 문제

### Ex 5.5



이 그래프에서 두가지 색으로 문제를 풀기는 불가능하다.

세 가지 색을 사용하면 총 6가지의 해답을 얻을 수 있다.

# Graph Coloring

## □ Coloring of maps

- An important application of graph coloring
- A graph is called *planar* (평면) if it can be drawn in a plane in such a way that no two edges cross each other

## □ Planar Graph (평면그래프)

- To every map, there corresponds a planar graph
- 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.

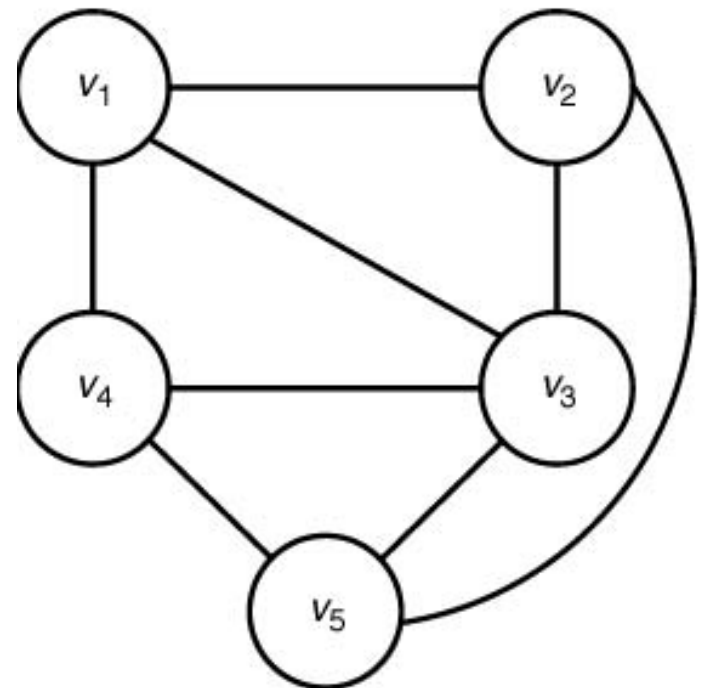
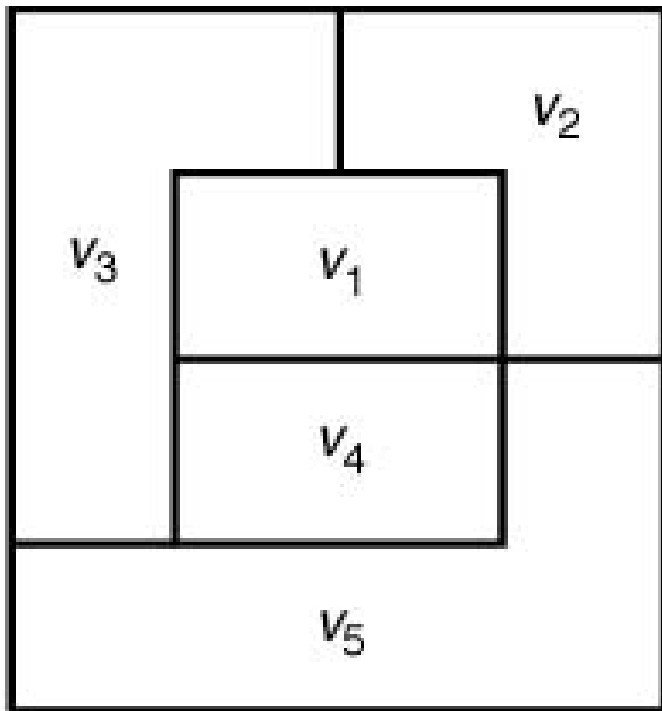
## □ $m$ -coloring problem for planar graphs

- Determine how many ways the map can be colored, using at most  $m$  colors, so that no 2 adjacent regions are the same color



# Graph Coloring

## □ 지도와 평면 그래프



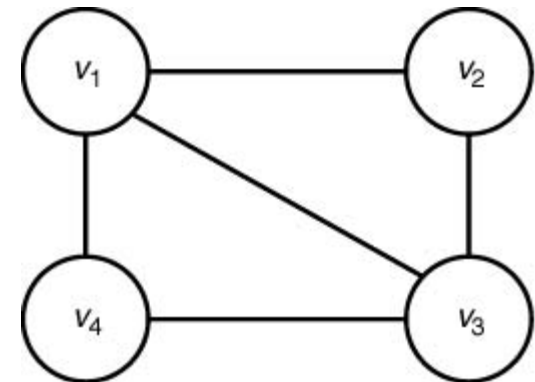
# Graph Coloring

## □ Strategy of $m$ -coloring problem

- Use state space tree

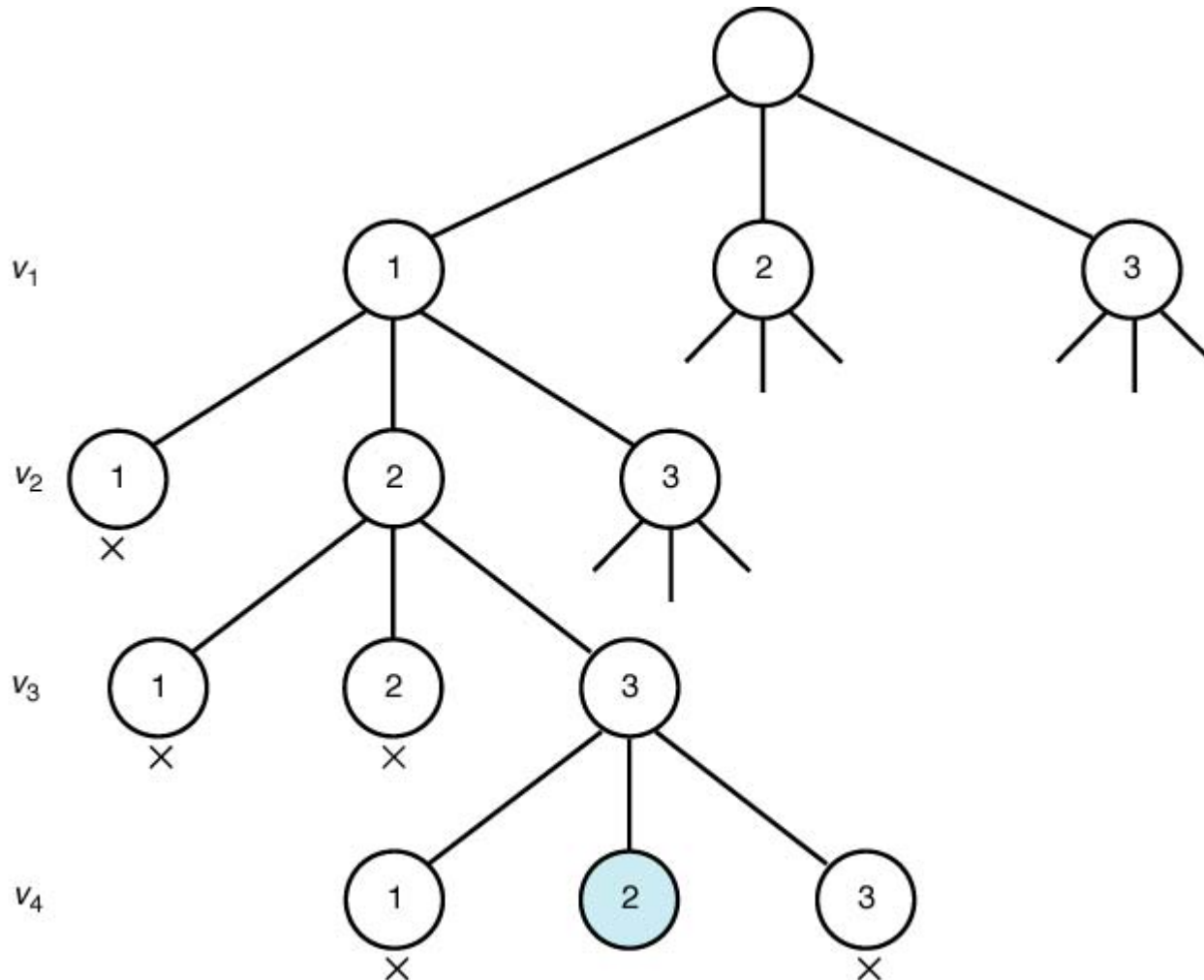
- Procedure

- Each possible color is tried for vertex  $v_1$  at level 1
- Each possible color is tried for vertex  $v_2$  at level 2
- .....
- Each possible color is tried for vertex  $v_n$  at level  $n$
- Each path from the root to a leaf is a candidate solution
  - Check whether a candidate solution is a solution by determining whether any two adjacent vertices are the same color



# Graph Coloring

- 그래프 색칠하기 되추적 해법



# Graph Coloring

## □ Algorithm 5.5 (I)

- Problem :  $m$ 개의 색을 사용하여, 그래프의 정점의 색을 할당할 수 있는 모든 방법을 결정하라. 단, 인접한 정점은 서로 같은 색을 가질 수 없다.
- Inputs :  $n$ (정점의 수),  $m$ (색의 수),  
 $W[1..n][1..n]$  ( $W[i][j]=1$  or  $0$  (True or False))
- Outputs :  $vcolor[1..n]$
- 최상위호출:  **$m\_coloring(0)$**

# Graph Coloring

```
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) {
    int j;    bool switch;
    switch = TRUE;
    j = 1;
    while (j < i && switch) {
        if (W[i][j] && vcolor[i] == vcolor[j]) switch = FALSE;
        j++;
    }
    return switch;
}
```

# Graph Coloring

## □ 그래프 색칠하기: 분석

- 상태공간트리 상의 노드의 총수는

$$1 + m + m^2 + \cdots + m^m = \frac{m^{m+1} - 1}{m - 1}$$

- 여기서도 **Monte Carlo** 기법을 사용하여 수행시간을 추정할 수 있다.

# The Hamiltonian Circuits Problem

## □ Traveling Salesperson Problem (Dynamic Programming)

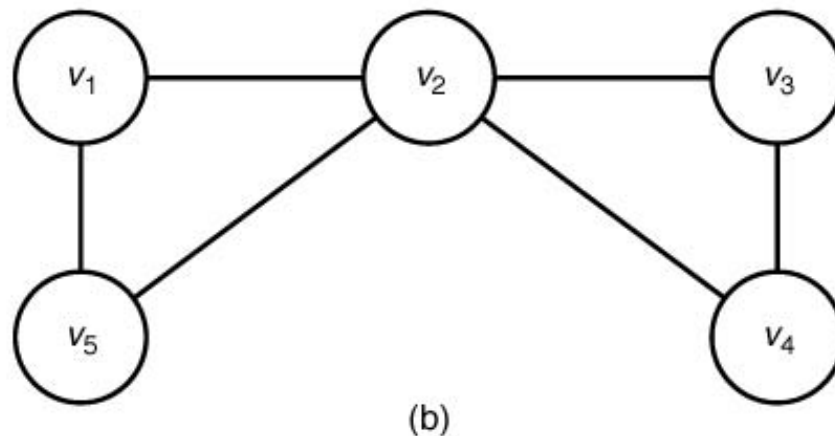
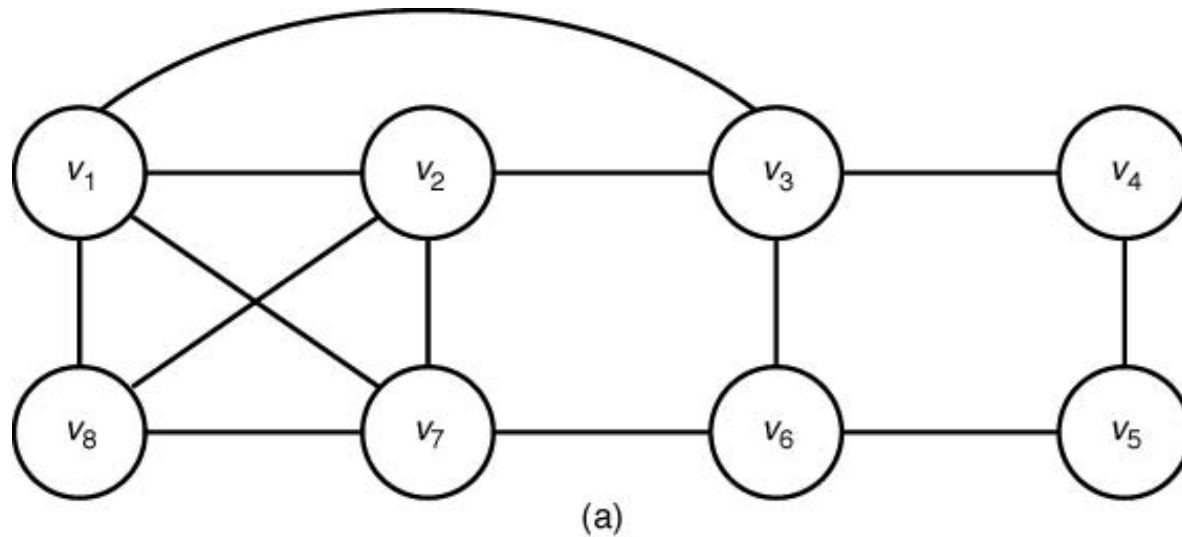
$$T(n) = (n-1) (n-2) \cdot 2^{n-3} \quad \text{cf) Brute-force } T(n) = (n-1)!$$

- if  $n=20$ ,  $T(20) = 45$  secs
- if  $n=40$ ,  $T(40) = 39 \cdot 38 \cdot 2^{37} = 6.46$  yrs

## □ Hamiltonian Circuits Problem (mostly for undirected graph)

- A path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex
- So, TSP without optimizing the sum of weights.
- in complete graph, Hamiltonian circuits always exist.

# The Hamiltonian Circuits Problem





# The Hamiltonian Circuits Problem

## □ Hamiltonian Circuits Problem

- 연결된 비방향성 그래프에서 해밀토니안 회로를 결정하는 문제
- 되추적 방법을 적용하기 위해서 다음 사항을 고려해야 한다.
  - 경로 상의  $i$ 번째 정점은 그 경로 상의  $(i-1)$ 번째 정점과 반드시 이웃해야 한다.
  - $(n-1)$ 번째 정점은 반드시 0번째 정점(출발점)과 이웃해야 한다.
  - $i$ 번째 정점은 처음  $i-1$ 개의 정점이 될 수 없다.
- 상태공간트리 상의 노드 수는

$$1 + (n-1) + (n-1)^2 + \cdots + (n-1)^{(n-1)} = \frac{(n-1)^n - 1}{n-2}$$

Cf) dynamic programming –  $(n-1)(n-2)2^{n-3}$

# The Hamiltonian Circuits Problem

## □ Algorithm 5.6

- Problem : 주어진 그래프에서 **Hamiltonian** 경로를 찾아라.
- Inputs :  $n$ (정점의 수),  $W[1..n][1..n]$  ( $W[i][j] = 1$  or  $0$  (T or F) )
- Outputs :  $vindex[1..n]$
- $vindex[0] = 1$ ; **hamiltonian(0);**

```
void hamiltonian(index i) {  
    index j;  
  
    if (promising(i))  
        if (i == n-1)  
            cout << vindex[1] through vindex[n-1];  
        else  
            for (j = 2; j <= n; j++) {           // Try all vertices as  
                vindex[i+1] = j;                  // next one.  
                hamiltonian (i+1);  
            }  
}
```

# The Hamiltonian Circuits Problem

```
bool promising(index i) {  
    int j;    bool switch;  
  
    if (i== n-1 && !W[vindex[n-1]][vindex[0]])  
        switch = FALSE;  
    else if (i>0 && !W[vindex[i-1]][vindex[i]])  
        switch = FALSE;  
    else {  
        switch = TRUE;  
        j= 1;  
        while (j<i && switch) {  
            if (vindex[i] == vindex[j]) switch = FALSE;  
            j++;  
        }  
    }  
    return switch;  
}
```

# Sum-of-Subsets Problem

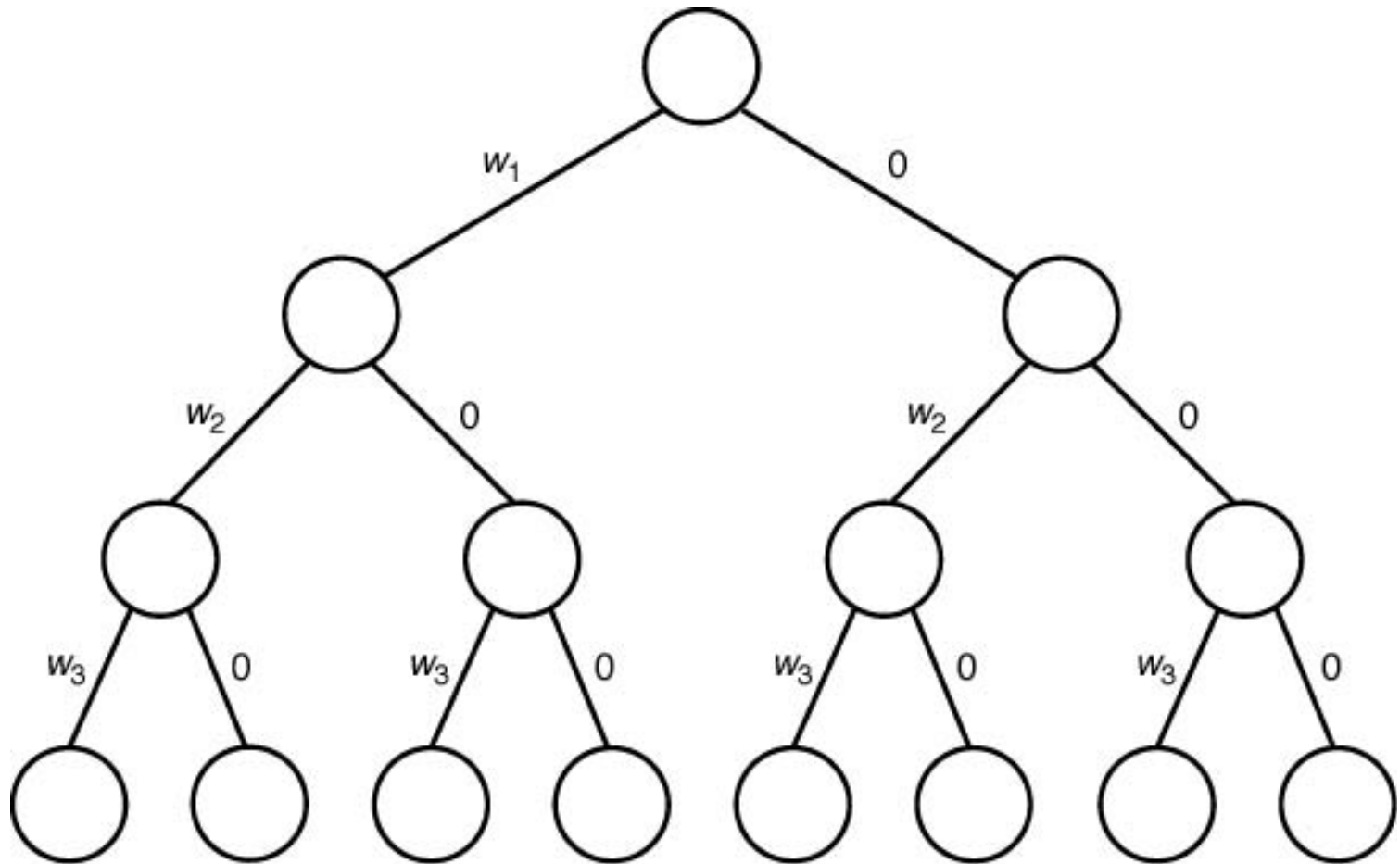
## □ Sum-of-Subsets Problem

- There are  $n$  positive integers  $w_i$  and a positive integer  $W$ .
- The goal is to find all subsets of integers that sum of  $W$ .

Ex 5.2

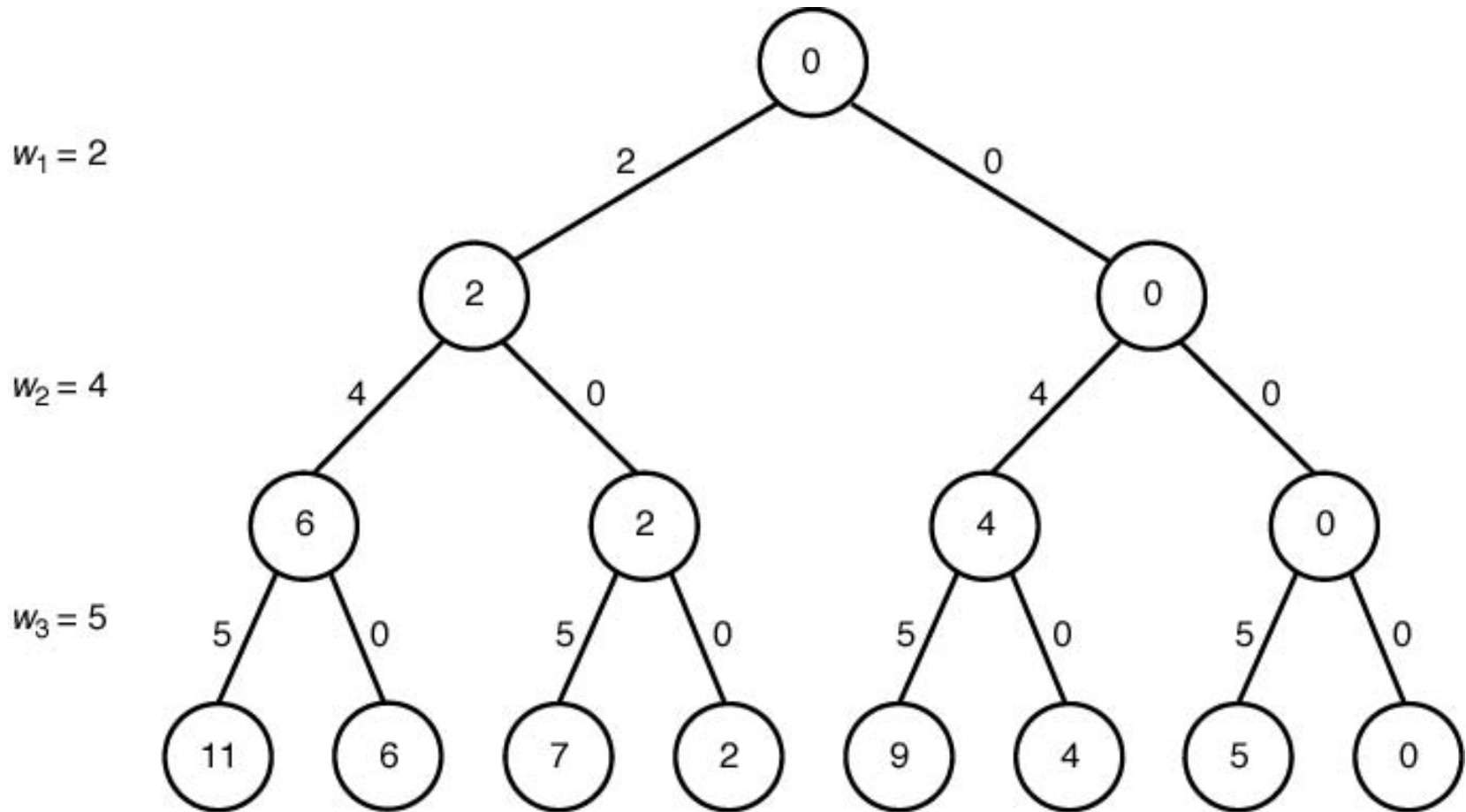
- Suppose  $n = 5$ ,  $W = 21$
- $w_1 = 5$ ;  $w_2 = 6$ ;  $w_3 = 10$ ;  $w_4 = 11$ ;  $w_5 = 16$ ;
- Because  $w_1 + w_2 + w_3 = 5 + 6 + 10 = 21$   
 $w_1 + w_5 = 5 + 16 = 21$   
 $w_3 + w_4 = 10 + 11 = 21$
- Solutions are  $\{w_1, w_2, w_3\}$ ,  $\{w_1, w_5\}$ ,  $\{w_3, w_4\}$

# Sum-of-Subsets Problem



A state space tree for Sum-of-Subsets ( $n=3$ )

# Sum-of-Subsets Problem



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

# Sum-of-Subsets Problem

- Backtracking Strategy

- If we sort the weights in nondecreasing order,  
there are obvious signs that a node is nonpromising

1. *weight* – the sum of the weights that have been included  
up to a node at level  $i$

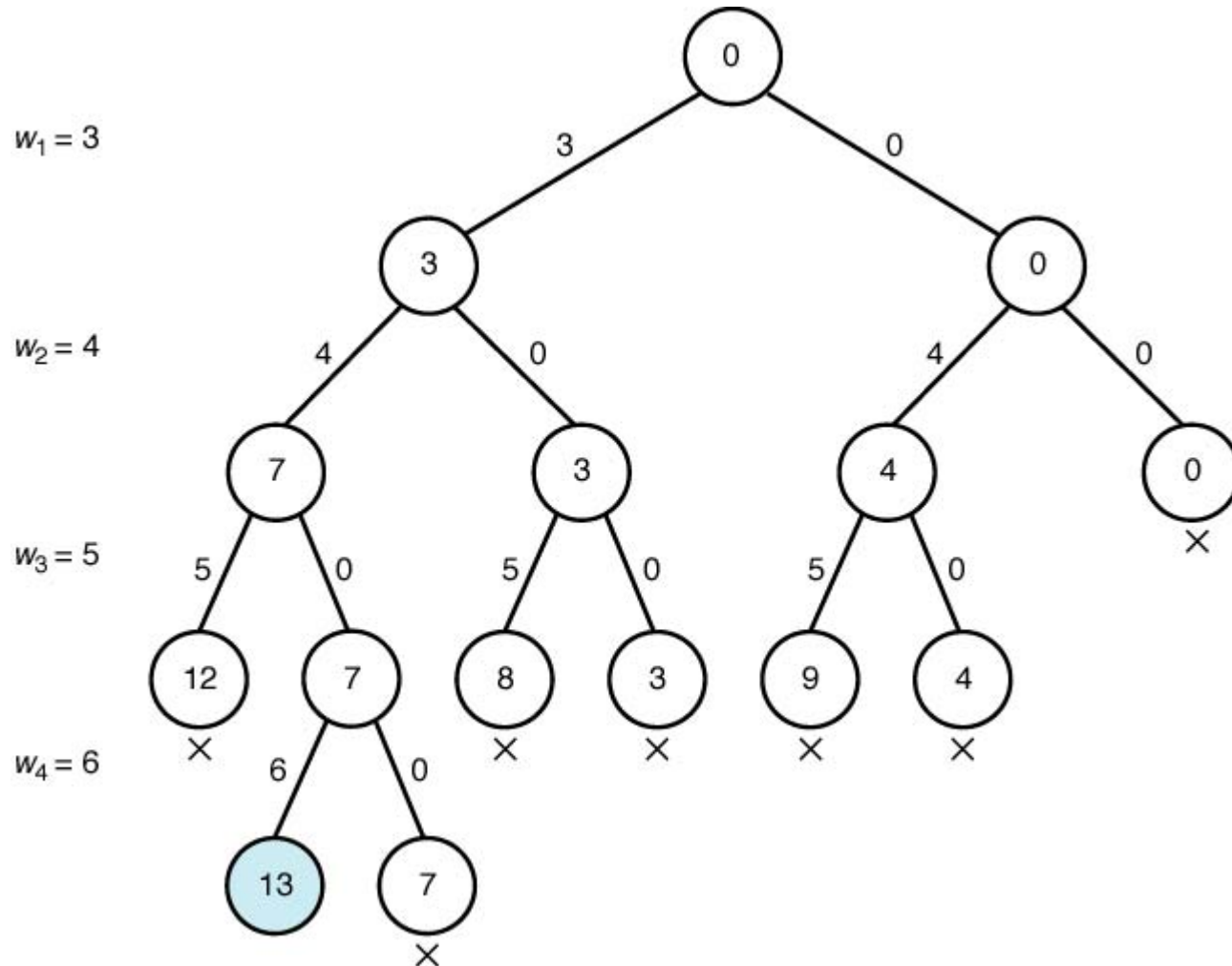
if  $\textit{weight} + w_{i+1} > W \rightarrow$  a node is non-promising

2. *total* – the total weight of the remaining weights

if  $\textit{weight} + \textit{total} < W \rightarrow$  a node is non-promising

# Sum-of-Subsets Problem

Ex 5.4  $n=4$ ,  $W=13$  and  $w_1 = 3$ ;  $w_2 = 4$ ;  $w_3 = 5$ ;  $w_4 = 6$





# Sum-of-Subsets Problem

- 최상위호출: `sum_of_subsets(0, 0, total)`

```
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 ) );
}
```

# The 0-1 Knapsack Problem

## □ Backtracking algorithm for the 0-1 Knapsack Problem

- 상태공간트리를 구축하여 되추적 기법으로 문제를 푼다.
- 뿌리마디에서 왼쪽으로 가면 첫번째 아이템을 배낭에 넣는 경우이고, 오른쪽으로 가면 첫번째 아이템을 배낭에 넣지 않는 경우이다.
- 동일한 방법으로 두 번째 아이템을 넣으면 수준 1의 노드에서 왼쪽으로 가고, 빼면 오른쪽으로 가고 .....
- 이런 식으로 계속하여 상태공간트리를 구축하면, 뿌리마디로부터 잎마디까지의 모든 경로는 해답후보가 된다.
- 이 문제는 최적의 해를 찾는 문제 (**optimization problem**)이므로 검색이 완전히 끝나기 전에는 해답을 알 수가 없다.  
따라서 검색하는 과정 동안 항상 그 때까지 찾은 최적의 해를 기억해 두어야 한다.

# The 0-1 Knapsack Problem

## □ A general algorithm

```
void checknode(node v) {  
    node u;  
  
    if (value(v) is better than best)  
        best = value(v);  
    if (promising(v))  
        for (each child u of v)  
            checknode(u);  
}
```

- **best** : 지금까지 찾은 제일 좋은 해답치
- **value(v)** : v 마디에서의 해답치

# The 0-1 Knapsack Problem

## □ Strategy for the 0-1 Knapsack problem

- $w_i$  and  $p_i$  are the weight and profit of the  $i$ -th item, respectively
- Sort the items in nonincreasing order according to the values of  $p_i / w_i$   
(일종의 탐욕적인 방법이 되는 셈이지만,  
알고리즘 자체는 탐욕적인 알고리즘은 아니다.)
- Let:
  - *profit* – the sum of the profits of the items included up to the node
  - *weight* – the sum of the weights of those items
  - *totweight* – the sum of the weights could be obtained  
by expanding beyond that node (절대  $W$ 를 초과 못 함)
  - *bound* – the upper bound on the profit that could be obtained  
by expanding beyond that node

# The 0-1 Knapsack Problem

- 알고리즘 스케치

- $bound$ 와  $totweight$ 를  $profit$ 과  $weight$  값으로 초기화
- 그 다음에 탐욕적으로 아이템을 취함
- 이 과정을  $totweight$ 이  $W$ 를 초과하게 되는 아이템을 잡을 때까지 반복
- 남은 공간이 허용하는 무게만큼 마지막 아이템의 일부분을 취하고, 그 일부분에 해당하는  $profit$ 을  $bound$ 에 더함
- 마디가 수준  $i$ 에 있다고 하고, 수준  $k$ 에 있는 마디에서 총무게가  $W$ 를 넘는다면,

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = \left( profit + \sum_{j=i+1}^{k-1} p_j \right) + (W - totweight) \times \frac{p_k}{w_k}$$

- $maxprofit$ : 지금까지 찾은 최선의 해답이 주는 값어치  
if  $bound \leq maxprofit \rightarrow$  a node at level  $i$  is nonpromising

# The 0-1 Knapsack Problem

- $maxprofit := \$0; profit := \$0; weight := 0$
- 깊이우선순위로 각 마디를 방문하여 다음을 수행한다:
  1. 그 마디의 *profit*와 *weight*를 계산한다.
  2. 그 마디의 *bound*를 계산한다.
  3.  $weight < W$ 이고,  $bound > maxprofit$ 이면, 검색을 계속한다; 그렇지 않으면, 되추적.
- 고찰: 최선이라고 여겼던 마디를 선택했다고 해서 실제로 그 마디로부터 최적해가 항상 나온다는 보장은 없다.

# The 0-1 Knapsack Problem

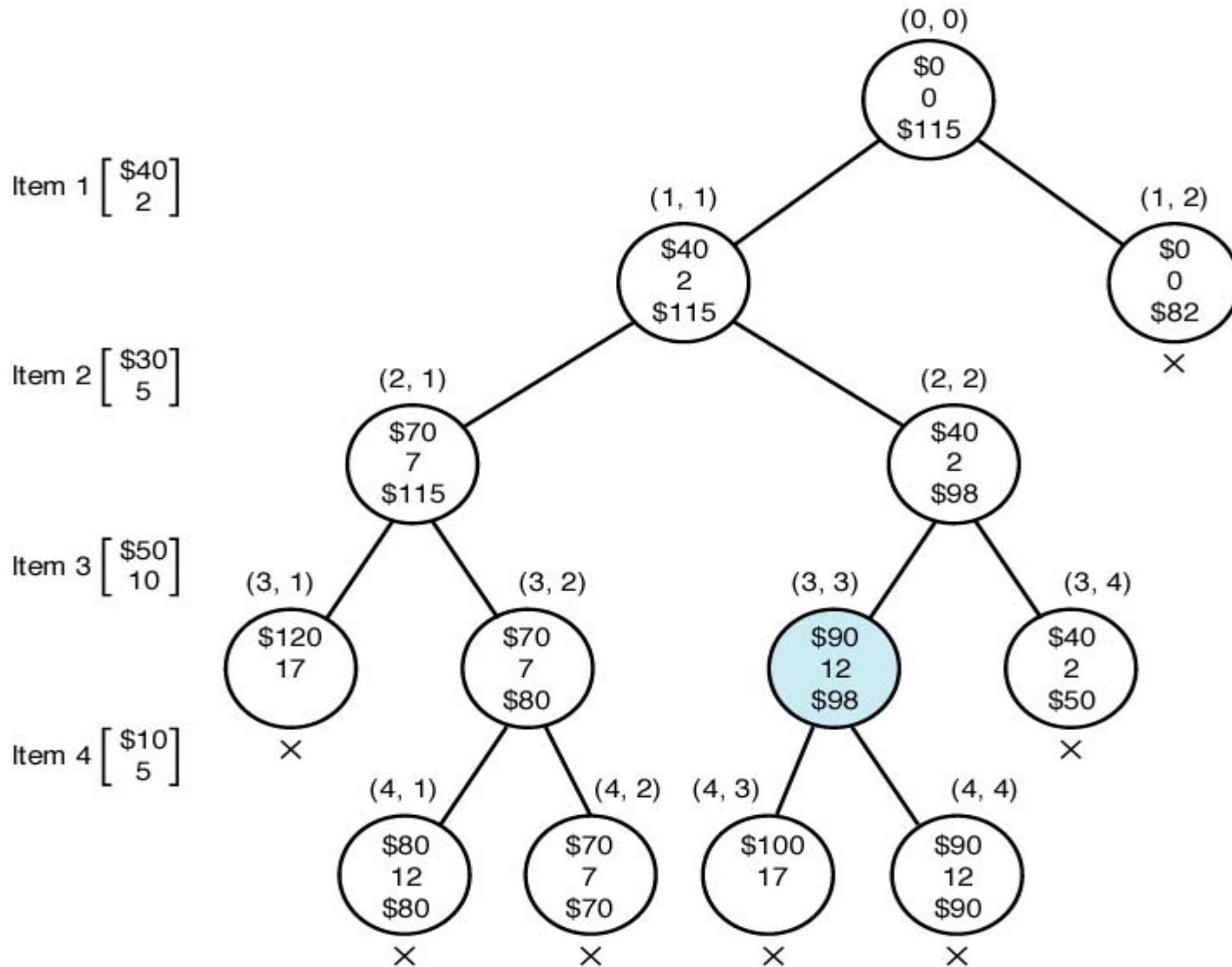
## □ Ex 5.6

$n = 4, W = 160$ 이고,

$i$	$p_i$	$w_i$	$\frac{p_i}{w_i}$	일 때,
1	\$40	2	\$20	
2	\$30	5	\$6	
3	\$50	10	\$5	
4	\$10	5	\$2	

되추적을 사용하여 구축되는 가지친 상태공간트리를 그려 보시오.

# The 0-1 Knapsack Problem





# The 0-1 Knapsack Problem

## □ Algorithm

- Problem : Let  $n$  items be given, where each item has a *weight* and a *profit*.  
Let  $W$  be given. Determine a set of items with maximum total profit, under the constraint that the sum of their weights cannot exceed  $W$ .
- Inputs :  $n, W, w[1..n], p[1..n]$ .  $w$  and  $p$  arrays are containing positive integers sorted in nonincreasing order according to the values of  $p[i]/w[i]$ .
- Outputs :  $bestset[1..n]$ .  
$$\left( \begin{array}{ll} bestset[i]=YES; & \text{If the } i\text{-th item is included} \\ bestset[i]=NO; & \text{Otherwise.} \end{array} \right)$$

# The 0-1 Knapsack Problem

```
void knapsack (index i, int profit, int weight) {  
    if (weight <= W && profit > maxprofit) {    // best so far  
        maxprofit = profit;  
        numset = i;  
        bestset = include;  
    }  
  
    if (promising(i)) {  
        include[i+1] = "YES";                    // Include w[i+1]  
        knapsack(i+1, profit+p[i+1], weight+w[i+1]);  
        include[i+1] = "NO";                     // Not include w[i+1]  
        knapsack(i+1, profit, weight);  
    }  
}
```

# The 0-1 Knapsack Problem

```
bool promising(index i) {  
    index j, k;  
    int totweight;  
    float bound;  
    if (weight >= W) return FALSE;  
    else {  
        j = i+1;  
        bound = profit;  
        totweight = weight;  
        while ((j <= n) && (totweight + w[j] <= W)) {  
            totweight = totweight + w[j];  
            bound = bound + p[j];  
            j++;  
        }  
        k=j;  
        if (k <= n)  
            bound = bound + (W-totweight)*p[k]/w[k];  
        return bound > maxprofit;  
    }  
}
```

# The 0-1 Knapsack Problem

## □ 분석

- 이 알고리즘이 점검하는 마디의 수는  $\Theta(2^n)$ 이다.
- 위 보기의 경우의 분석: 점검한 마디는 13개이다.
  - 이 알고리즘이 동적계획법으로 설계한 알고리즘 보다 좋은가?
  - 확실하게 대답하기 불가능
  - Dynamic Programming – 최악의 경우  $O(\min(2^n, nW))$
  - Backtracking – 최악의 경우  $\Theta(2^n)$
- Horowitz와 Sahni(1978)는 Monte Carlo 기법을 사용하여  
되추적 알고리즘이 동적계획법 알고리즘보다  
일반적으로 더 빠르다는 것을 입증하였다.
- Horowitz와 Sahni(1974)가 분할정복과 동적계획법을  
적절히 조화하여 개발한 알고리즘은  $O(2^{n/2})$ 의 시간복잡도를 가지는데,  
이 알고리즘은 되추적 알고리즘보다 일반적으로 빠르다고 한다.

# A Monte Carlo Algorithm

- **Monte Carlo** 기법을 사용한 백트래킹 알고리즘의 수행시간 추정
  - **Monte Carlo** 기법은 어떤 입력이 주어졌을 때 점검하게 되는 상태공간트리의 “전형적인” 경로를 무작위(**random**)로 생성하여 그 경로 상에 있는 노드의 수를 센다.  
이 과정을 여러 번 반복하여 나오는 결과의 평균치를 추정치로 한다.
  - **Monte Carlo** 기법은 확률적 (**probabilistic**) 알고리즘
    - 표본 공간의 무작위 표본의 추정치를 가지고  
무작위 변수의 기대치를 추정
  - Cf) 결정적 (**deterministic**) 알고리즘
  - 이 기법을 적용하기 위해서는 다음 두 조건을 반드시 만족하여야 한다.
    - 상태공간트리에서 같은 수준(**level**)에 있는 모든 노드에서는 같은 유망함수를 사용해야 한다.
    - 상태 공간트리에서 같은 수준에 있는 모든 노드들은 같은 수의 자식노드들을 가지고 있어야 한다.
  - **n-Queens** 문제는 이 두 조건을 만족한다.

# A Monte Carlo Algorithm

- Monte Carlo 기법을 사용한 백트래킹 알고리즘의 수행시간 추정 방법
  1. 뿌리노드의 유망한 자식노드의 개수를  $m_0$ 이라고 한다.
  2. 상태공간트리의 수준 1에서 유망한 노드 하나를 무작위로 정하고, 그 노드의 유망한 자식노드의 개수를  $m_1$ 이라고 한다.
  3. 위에서 정한 노드의 유망한 노드 하나를 다시 무작위로 정하고, 그 노드의 유망한 자식노드의 개수를  $m_2$ 라고 한다.
  4. 더 이상 유망한 자식노드가 없을 때까지 이 과정을 반복한다.
- 되추적 알고리즘에 의해서 점검한 노드의 총 개수의 추정치
  - $m_i$ 는 수준  $i$ 에 있는 노드의 유망한 자식노드의 개수의 평균의 추정치
  - $t_i$ 는 수준  $i$ 에 있는 한 노드의 자식노드의 총 개수
  - 따라서 노드의 총 개수의 추정치는

$$1 + t_0 + m_0 t_1 + m_0 m_1 t_2 + \cdots m_0 m_1 \cdots m_{i-1} t_i + \cdots$$

# A Monte Carlo Algorithm

## □ Algorithm 5.2

- **Problem : Monte-Carlo** 알고리즘을 사용하여,  
되추적 알고리즘의 효율성을 평가하라.
- **Inputs** : 되추적 알고리즘이 해결하여야 할 문제의 예시
- **Outputs** : 주어진 예시에 대한 모든 해를 찾기 위하여 주어진  
알고리즘이 검사해야 하는 노드의 수

# A Monte Carlo Algorithm

```
int estimate() {  
    node v;  
    int m, mprod, t, numnodes;  
  
    v = root of state space tree;  
    numnodes = 1;  
    m=1;  
    mprod = 1;  
  
    while (m != 0) {  
        t = number of children of v;  
        mprod = mprod*m;  
        numnodes = numnodes + mprod*t;  
        m = number of promising children of v;  
        if (m != 0)  
            v = randomly selected promising child of v;  
    }  
    return numnodes;  
}
```



# A Monte Carlo Algorithm

## Monte-Carlo Estimate for Algorithm 5.1

```
int estimate_n_queens (int n) {
    index i, j, col[1..n];
    int m, mprod, numnodes;
    set_of_index prom_children;

    i = 0; numnodes = 1; m=1; mprod = 1;
    while (m != 0 && i!=n) {
        mprod = mprod*m;
        numnodes = numnodes + mprod*n;
        i++; m = 0; prom_children =  $\Phi$ ;
        for (j=1; j<=n; j++) {
            col[i] = j;
            if (promising(i)) {
                m++;
                prom_children = prom_children  $\cup$  {j};
            }
        }
        if (m != 0) {
            j = random selection from prom_children;
            col[i] = j;
        }
    }
    return numnodes;
}
```