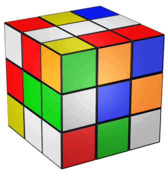
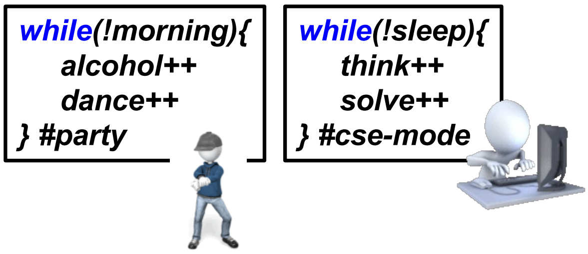
되추적

한국기술교육대학교 컴퓨터공학부 김상진





교육목표

- 되추적(backtracking)
  - 전수조사를 효과적으로 하는 방법
    - 기본 생각. 배제할 것은 빠르게 배제하자
  - 되추적은 보통 주어진 집합에서 어떤 조건이 충족되도록 일련의 객체를 선택하는 문제에 적합
  - 되추적을 사용하여도 여전히 최악의 경우에는 지수 비용이 필요할 수 있지만, 문제에 따라 많은 경우 입력이 매우 크더라도 효율적으로 답을 구해 줌
- 문제
  - n-Queens
  - 부분집합의 합 (cansum, howsum 등)
  - 그래프 색칠하기
  - 0-1 배낭 채우기
  - 해밀톤 회로 문제

되추적

- 되추적(backtracking): 특정 집합에서 어떤 조건이 충족되도록 일련의 요소를 선택하는 문제를 해결할 때 사용함
- 되추적은 깊이 우선 검색(DFS, Depth-First-Search)을 변형한 기술
  - 실제 트리를 만들어 검색하는 것은 아님 (재귀 호출 트리에 대한 DFS)
- 깊이 우선 검색을 통해 방문해야 하는 전체 트리를 상태 공간 트리라 함
- 방법. 상태 공간 트리에 있는 모든 경우를 방문하지 않고 해를 찾자
- 되추적 기법은 어떤 노드의 유망성(promising)을 점검한 후에 유망하지 않다고 결정되면 그 노드의 부모로 되돌아가(backtracking) 다음 자식 노드를 이용하여 문제의 답을 찾는 기법을 말함
  - 유망 여부: 해답이 될 수 없으면 유망하지 않은 것임
  - 유망하지 않은 노드를 만나면 그 아래로는 더 이상 탐색하지 않으며, 이 과정을 가지치기(pruning)라 함
- 결국 상태 공간 트리를 가지치기하여 만들어진 가지를 친 상태 공간 트리에 있는 노드만 방문하는 형태가 되추적 기법임

일반적인 되추적 알고리즘

- 알고리즘

```
checknode(node v)
if promising(v) then
    if solution found at v then
        write the solution
    else
        for all child u of v do
            checknode(u)
```

```
visited[s] := true
for all edge (s, w) do
    if not visited[w] then
        DFS(G,w)
```

노드를 많이 배제하는 것도 중요하지만 promising 함수를 효과적으로 작성하는 것도 매우 중요함

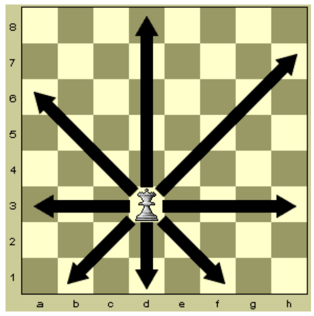
  - 해결하고자 하는 문제에 따라 promising 함수가 달라짐
  - 실제 트리를 만들어 수행하는 것은 아님
- 개선된 되추적 알고리즘
  - 함수 스택 사용 측면
    - promising 함수 호출 위치

```
expand(node v)
for all child u of v do
    if promising(u) then
        if solution found at u then
            write the solution
        else expand(u)
```

재귀 호출 전 검사 vs. 재귀 호출 후 검사

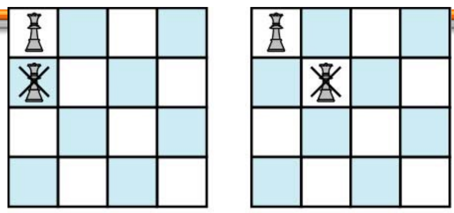
n-Queens 문제

- $n \times n$ 크기의 chess 판에 n개의 Queen들이 서로 위협하지 않도록 배치하는 문제
- 입력. chess판의 크기 n (배치하는 Queen의 수)
- 출력. n개의 chess판 위치
- Queen의 이동 가능 방향: 대각선, 상하좌우
- 관찰
  - 경우의 수: $C(n^2, n)$
  - 같은 행에 위치할 수 없음 $\Rightarrow$ 경우의 수 n^n
  - 같은 열에 위치할 수 없음 $\Rightarrow$ 경우의 수 $n!$
- 되추적은 보통 주어진 집합에서 어떤 조건을 충족하도록 일련의 요소를 선택하는 문제에 적합
  - 주어진 집합: $n \times n$ 크기의 chess판 위치
  - 일련의 요소: n개의 chess판 위치

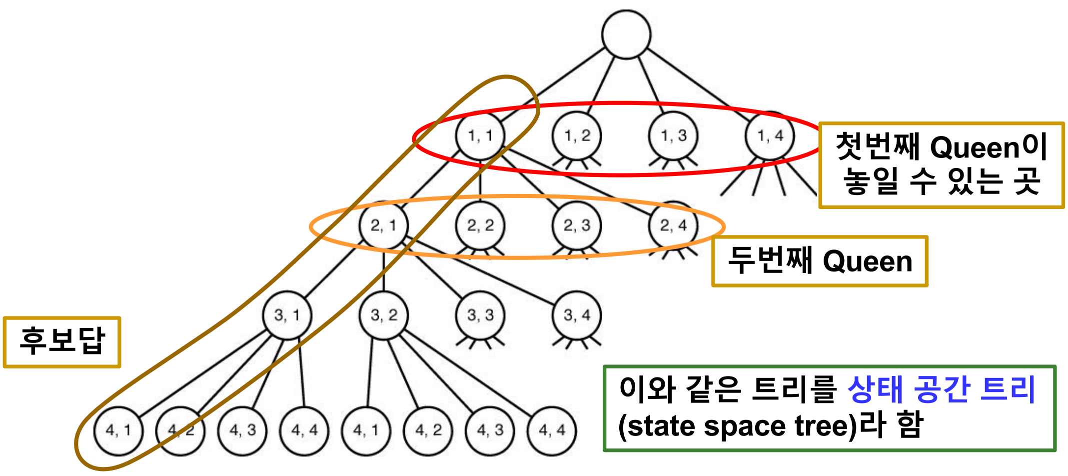


4-Queens 문제 (1/3)

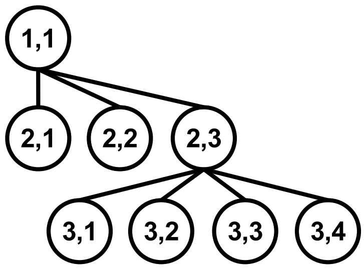
- 총 후보의 수는 $4^4 = 256$개임

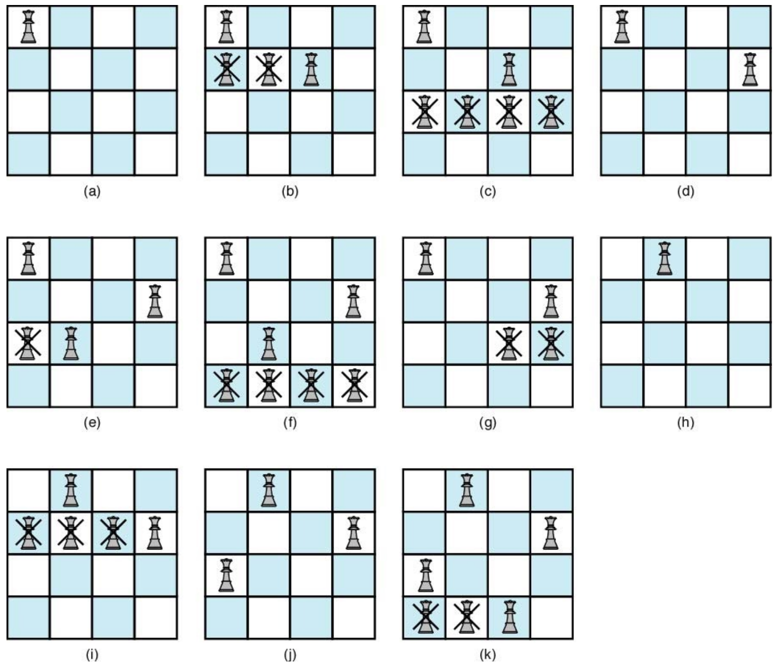


(a) (b)



4-Queens 문제 (2/3)

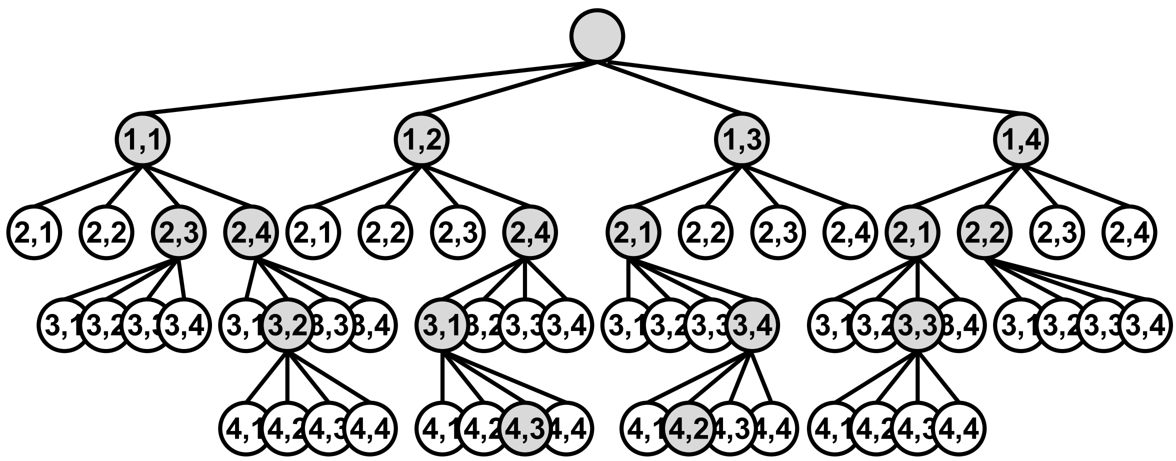




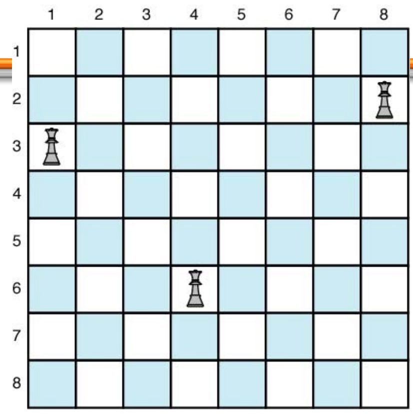
4-Queens 문제 (3/3)

4-Queen 문제의 가지를 친 상태 공간 트리

$4 + 4^2 + 4^3 + 4^4 = 340$ vs. 60



n-Queens (1/2)



- promising 함수
  - 기본적으로 다른 행에 배치함
  - 같은 열에 있으면 안 됨
  - 대각선 검사
    - col(i): i번째 행에 있는 Queen의 열 위치
      - col(6) – col(3) = 4 – 1 = 3 = 6 – 3
      - col(6) – col(2) = 4 – 8 = –4 = 2 – 6

```
promising(cols[], row)
r := 1
ret := true // 유망여부
while r < row and ret do
    if cols[r] = cols[row] or abs(cols[r] – cols[row]) = row – r then
        ret := false
    r += 1
return ret
```


n-Queens (2/2)

- 시간 복잡도
  - n-Queens 문제의 모든 노드 수:

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

    - 예) $n = 8$: 19,173,961
  - 유망한 노드의 수
    - 두 개의 queen은 같은 열에 위치할 수 없음: $n!$
      - 예) $n = 8$: 40,320
    - 실제는?
      - 대각선 때문에 제외되는 것이 있음
      - 유망하지 않은 노드 중 검사되는 것도 있음 (슬라이드 8에서 노란색이 아닌 노드)

알고리즘 성능 비교

n	A	B	C	D
4	341	24	61	17
8	19,173,961	40,320	15,721	2,057
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}

- A: 전수조사 알고리즘이 검사하는 노드의 수(재귀 호출 수)
- B: 각 행의 말이 모두 다른 열에 위치하도록 하는 $n!$ 경우의 수를 조사하는 알고리즘이 검사하는 노드의 수
- C: 되추적이 검사하는 노드의 수
- D: 되추적이 검사하는 노드의 수 중 유망하다고 판단된 노드의 수

Monte Carlo 알고리즘 (1/4)

- 되추적은 전수조사보다 검사하는 양을 줄여주지만,
- 그 효과는 문제마다 문제의 인스턴스마다 상이함
  - 주어진 문제의 인스턴스에 대해 검사하는 양을 미리 알 수 있으면 되추적의 사용 여부를 결정할 수 있음
  - 이것을 해주는 것이 몬테칼로(Monte Carlo) 알고리즘임
- 몬테칼로 알고리즘은 어떤 입력이 주어졌을 때 점검하게 되는 상태 공간 트리의 전형적인 경로를 무작위로 생성하여 이 경로 상에 점검하게 되는 노드의 수를 계산함
  - 이 과정을 여러 번 반복하여 계산된 결과의 평균값을 이용하여 되추적 알고리즘의 성능을 추정함
  - 무작위로 생성하기 때문에 확률 알고리즘임

Monte Carlo 알고리즘 (2/4)

- 몬테칼로 알고리즘을 이용하여 되추적 알고리즘을 분석하기 위한 요구 조건
  - 조건 1. 상태 공간 트리에서 같은 레벨에 있는 모든 노드의 유망성 여부를 점검하는 절차가 같아야 함
  - 조건 2. 상태 공간 트리에서 같은 레벨에 있는 모든 노드의 자식 수는 같아야 함
- n-Queens 문제는 위 조건을 만족함
- 몬테칼로 알고리즘으로 되추적 알고리즘을 분석하기 위해서는 랜덤하게 경로를 생성하고, 이 경로에서 검사하는 노드의 수를 이용하여 점검하는 전체 노드의 수를 추정함

Monte Carlo 알고리즘 (3/4)

- 경로를 랜덤하게 생성하는 방법
  - 루트 노드부터 단말까지 랜덤하게 선택한 유망한 노드를 따라 내려감
    - m_i: 레벨 i에서 선택된 노드의 유망한 자식 노드의 수
- t_i : 레벨 i에 있는 노드의 자식 노드의 총 개수
- 되추적 알고리즘에 의해 점검하는 노드의 총 개수의 추정치는 다음과 같음

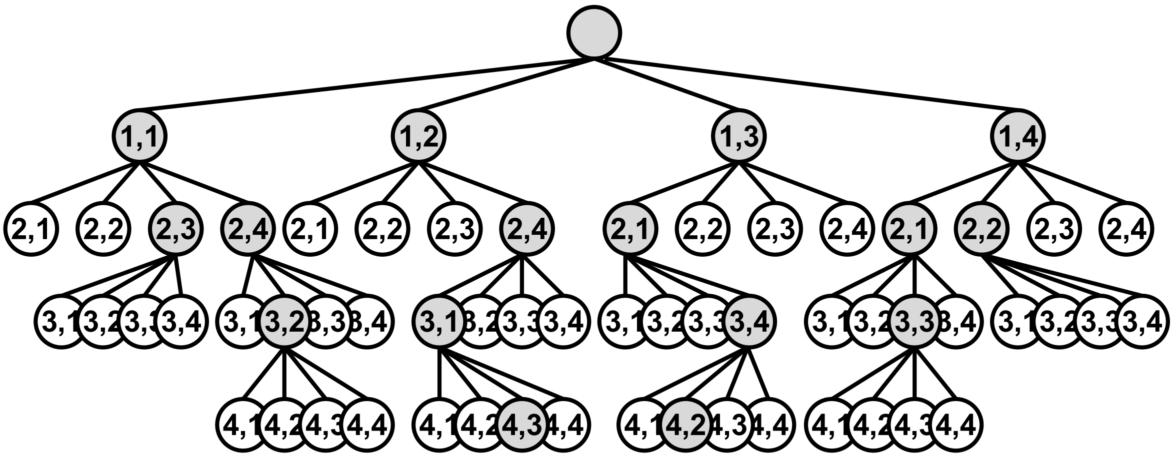
$$1 + t_0 + m_0 t_1 + m_0 m_1 t_2 + \cdots + m_0 m_1 \ldots m_{i-1} t_i + \cdots$$

- 비용. 트리의 높이

Monte Carlo 알고리즘 (4/4)

t_i: 레벨 i에 있는 노드의 자식 수
m_i: 레벨 i에 있는 유망한 자식 노드의 수

$1+t_0+m_0t_1+m_0m_1t_2+\ldots+m_0m_1\ldots m_{i-1}t_i+\ldots$



(1.2를 랜덤하게 선택한 경우)
→ 1+4+4×4+4×1×4+4×1×1×4 = 53

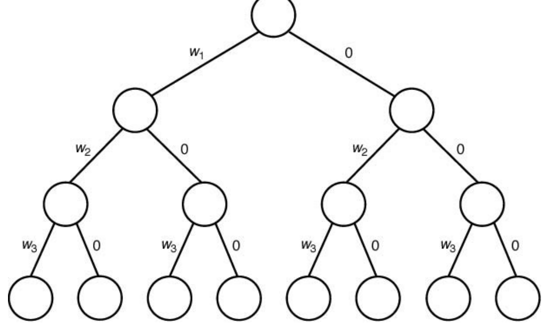
(1.1, 2.4를 랜덤하게 선택한 경우)
→ 1+4+4×4+4×2×4+4×2×1×4 = 85

실제: 61

부분집합의 합 구하기 (1/6)

- 문제(subset sum). n개의 양의 정수와 양의 정수 W가 주어졌을 때, 부분집합의 원소들의 합이 W가 되는 모든 부분 집합을 찾기
- 입력. n개의 양의 정수, 목표 합 W
- 출력. 합이 W가 되는 모든 부분집합
- 예) {5, 6, 10, 11, 16}, 21
  - {5, 6, 10}, {5, 16}, {10, 11}
- 상태 공간 트리를 어떻게?
  - 각 원소는 포함될 수 있고, 포함되지 않을 수 있음
  - 정수의 값을 오름차순으로 정렬하여 접근하면 쉽게 유망하지 않는 노드를 식별할 수 있음
    - {16, 11, 10, 6, 5}: 16을 포함한 후에 11을 포함할 수 없다고 나머지를 모두 가지치기 할 수 없음. 거꾸로 이면 6, 10을 포함하여 16이 된 상태에서 11을 포함할 수 없으면 16도 포함할 수 없음

$2^{n+1}-1$ 노드



부분집합의 합 구하기 (2/6)

- 2장, 10장에서 살펴본 cansum 등의 문제와 어떤 연관?
  - 2장은 가지치기 없이 전수조사 (물론 전혀 가지치기가 없는 것은 아님, 요소를 빼다 0보다 작아지면 전수조사를 중단함)
  - 이 문제는 한 요소를 여러 번 사용할 수 없음
  - 대신 중복된 요소가 포함될 수 있음
- 0-1 배낭 채우기 문제와 어떤 연관?
  - 모든 item의 무게당 이득이 같으면 최적해는 배낭을 최대한 채울 수 있는 item의 집합이 됨
    - 배낭이 수용할 수 있는 최대 무게만큼 채울 수 있다면 이득도 최대가 됨 ⟹ 부분집합의 합 문제와 동일

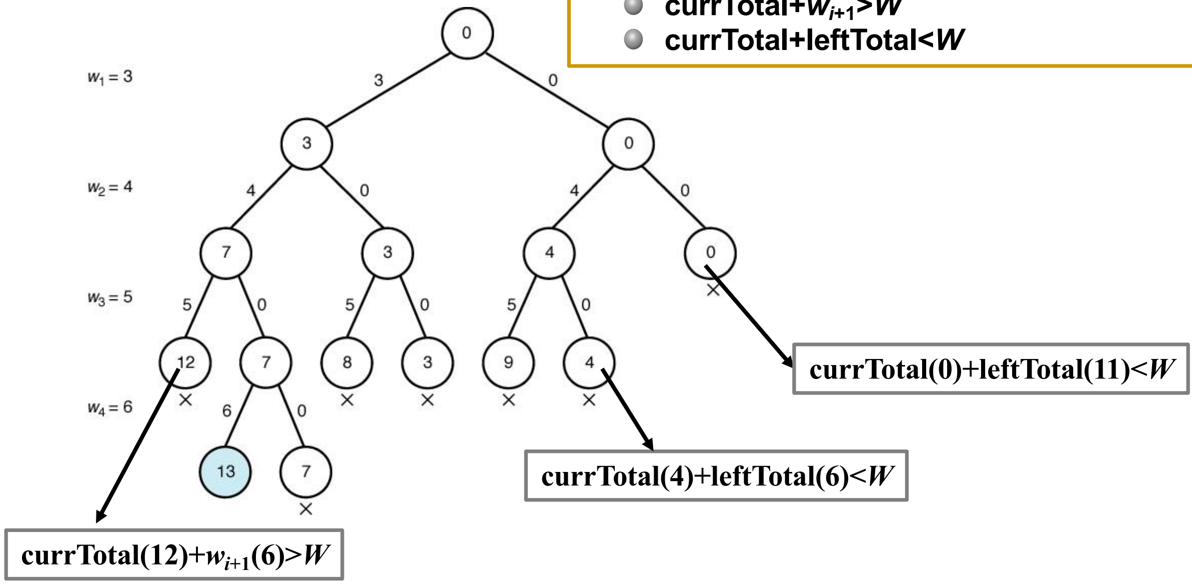
부분집합의 합 구하기 (3/6)

- 답은 어떻게 축적?
  - 수를 포함할 수 있고, 포함하지 않을 수 있음
    - 이 측면에서 0-1 배낭과 유사함
  - boolean 배열
- 지금까지의 합은 어떻게 유지
  - cansum처럼 재귀 호출 인자로 지금까지의 합을 전달함
- promising 함수는? 가지치기를 많이 할수록 효과적임

부분집합의 합 구하기 (4/6)

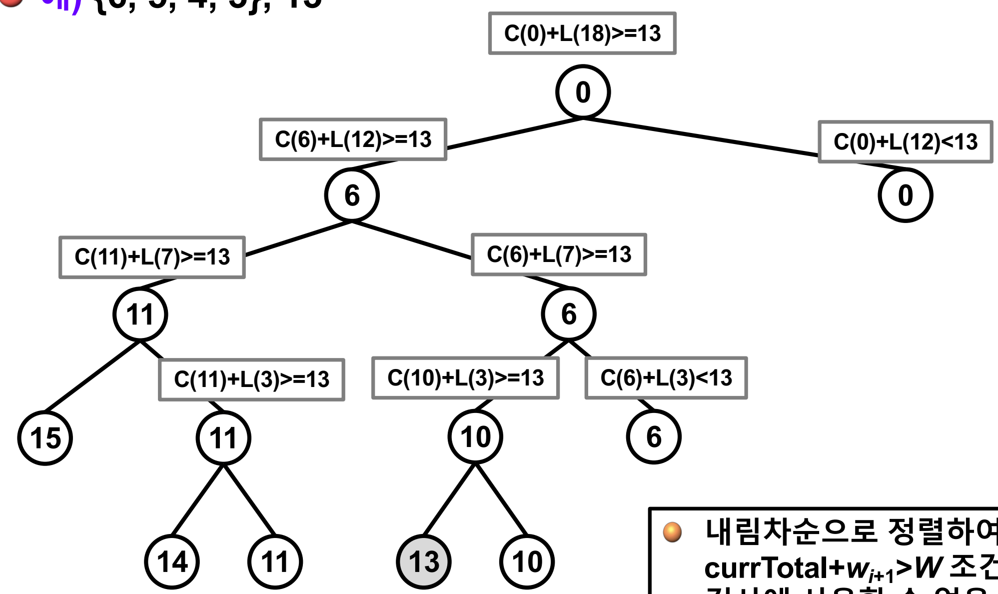
- 예) {3,4,5,6}, 13

- currTotal: 지금까지 포함한 w_i들의 합
- leftTotal: 아직까지 고려하지 않은 w_i들의 합
  - leftTotal의 초기값: $\sum w_i$
- 유망하지 않은 노드의 조건
  - currTotal+w_{i+1}>W
  - currTotal+leftTotal<W



부분집합의 합 구하기 (5/6)

- 예) {6, 5, 4, 3}, 13



- 내림차순으로 정렬하여 접근하면 currTotal+w_{i+1}>W 조건은 유망여부 검사에 사용할 수 없음
- 정렬의 의미가 없음 (전수조사+leftTotal 활용)

부분집합의 합 구하기 (6/6)

promising(A[], **currTotal**, $leftTotal$, **W**, $index$)

return $currTotal + leftTotal \geq W$ **and** $(currTotal = W$ **or** $currTotal + A[index] \leq W)$

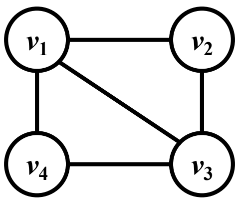
subset_sum(A[], $included$[], $currTotal$, $leftTotal$, W, $index$)

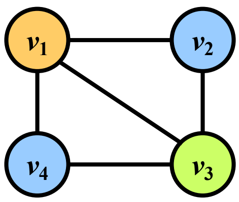
if **promising(nums**, $currTotal$, $leftTotal$, W, $index$) **then**
    if $currTotal = W$ **then**
        write solution
    else
        $leftTotal -= A[index]$
        $included[index] \coloneqq$ **true**
        subset_sum(A, $included$, $currTotal + A[index]$, $leftTotal$, W, $index + 1$);
        $included[index] \coloneqq$ **false**
        subset_sum(A, $included$, $currTotal$, $leftTotal$, W, $index + 1$);

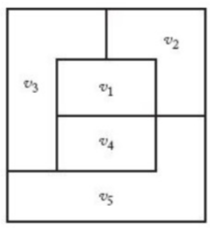
- 끝 조건으로 $index = n$을 사용하지 않음
- $index = n$이면 leftTotal은 0이 됨

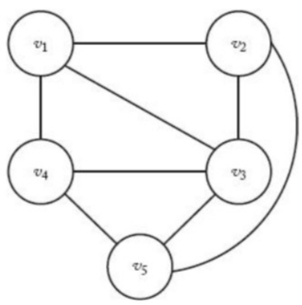
그래프 색칠하기 (1/3)

- m 색칠하기 그래프 문제
  - 인접한 노드는 같은 색으로 색칠할 수 없다는 제약 조건 하에 무방향 그래프의 노드를 최대한 m개의 색으로 색칠하는 문제
- **입력.** 무방향 그래프, m
- **출력.** 색칠할 수 있으면 색 조합
- **예)**





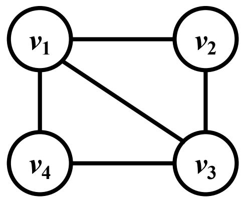


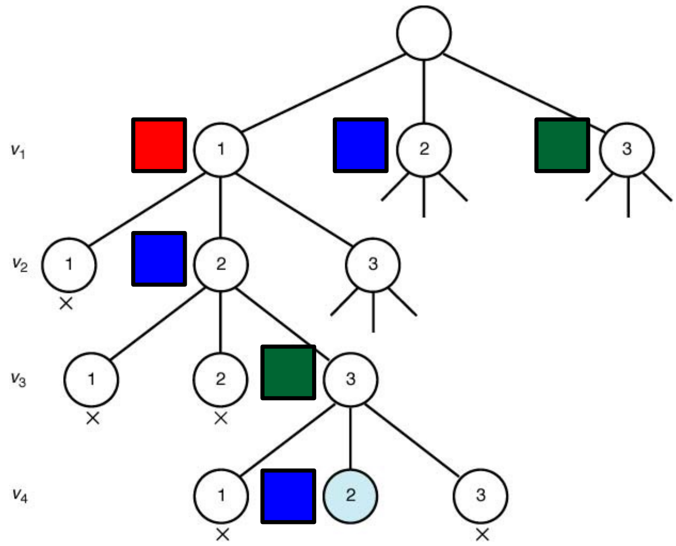


- 응용: 지도 색칠하기 (모든 지도는 그것에 상응하는 **평면 그래프**(planar graph)가 있음
  - 평면 그래프: 그래프의 간선들이 서로 교차하지 않도록 그릴 수 있는 그래프

그래프 색칠하기 (2/3)

- 그래프 색칠하기의 상태 공간 트리 ($m = 3$)





그래프 색칠하기 (3/3)

promising$(G, colors[], m, index)$
for all w **in** $G[index]$ **do**   **if** $w < index$ **and** $colors[w] = colors[index]$ **then return false** **return true**

mColoring$(G, colors[], m, index, sols)$

for $c := 1$ **to** m **do**
    $colors[index] := c$
    if *promising*$(G, colors, m, index)$ **then**
        if $index = n$ **then**
            $sols$*.pushback(*$colors$*.clone())*
            return
        else *mColoring*$(G, colors, m, index + 1)$

0-1 배낭 채우기 문제 (1/3)

- 동적 프로그래밍으로 해결하였음
- 빈틈없이 채우기 문제는 탐욕적 기법으로 해결 가능함
  - 0-1 배낭 채우기 문제의 최대 이익은 빈틈없이 채우기 문제의 최대 이익보다 클 수 없음
- 가지치기를 어떻게?
  - 가장 직관적인 기준.
    - $\text{currentWeight}+w_i > W$: 물건을 포함하였을 때 배낭 용량을 초과한 경우
  - 다른 기준은?
    - bound $\leq$ maxProfit이면 이 노드는 유망하지 않음
      - maxProfit: 지금까지 구한 최적해의 이익
      - bound: 그 노드를 따라 해를 구성하였을 때 빈틈없이 채우기 문제로 얻을 수 있는 최대 이익

0-1 배낭 채우기 문제 (2/3)

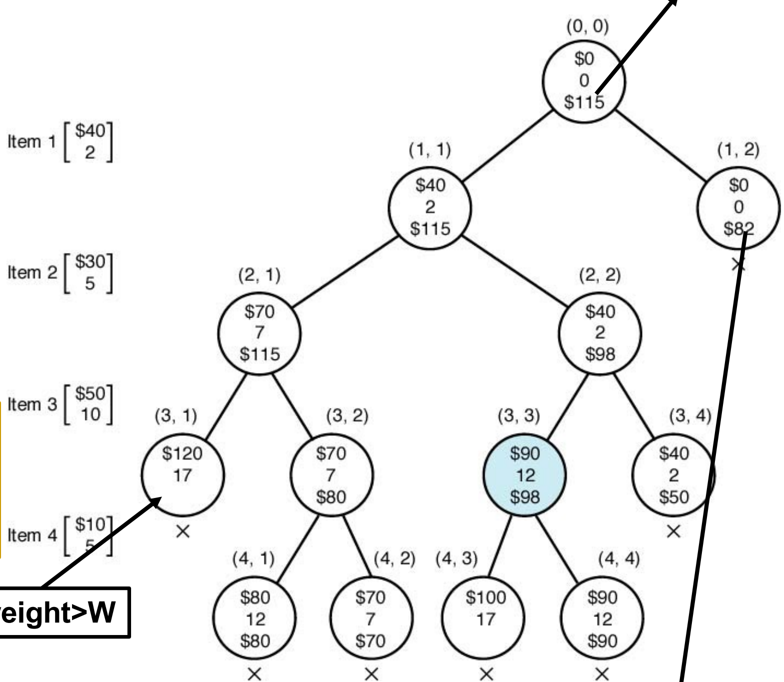
40(2)+30(5)+45(9)=115(16)

빈틈없이 배낭 채우기를 통해 계산

- 예) {(40, 2), (30, 5), (50, 10), (10, 5)}, 16
  - score = 20, 6, 5, 2

i	v_i	w_i	v_i/w_i
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2

무게당 이득을 기준으로 정렬한 이유: 빈틈없이 배낭 채우기 문제로 가지치기 이 문제에서는 무게당 이득이 높은 것부터 차례로 포함



weight>W

30(5)+50(10)+2(1)=82(16)

bounds $\leq$ maxprofit

```
computeBound(items[], W, index, currProfit, currWeight)
bound := currProfit
totalWeight := currWeight
while index ≤ n and totalWeight + items[index].weight ≤ W do
    bound += items[index].profit
    totalWeight += items[index].weight
    index += 1
if index ≤ n then bound += (W − totalWeight)×items[index].score
return bound
```

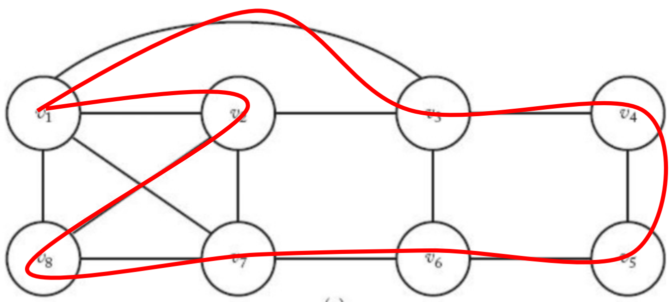
// score = profit/weight

```
promising(item[], W, index, currProfit, currWeight, ret)
return currWeight < W and
    computeBound(items, W, index + 1, currProfit, currWeight) > ret.max
```

```
knapsack(items[], included[], W, index, currProfit, currWeight, ret)
if currWeight ≤ W and currProfit > ret.max then
    ret.max := currProfit
    ret.sol := included.clone()
if promising(items, W, index, currProfit, currWeight, ret) then
    included[index + 1] := true
    knapsack(items, included, W, index + 1, currProfit + items[index + 1].profit,
        currWeight + items[index + 1].weight, ret);
    included[index + 1] := false
    knapsack(items, included, W, index + 1, currProfit, currWeight, ret);
```

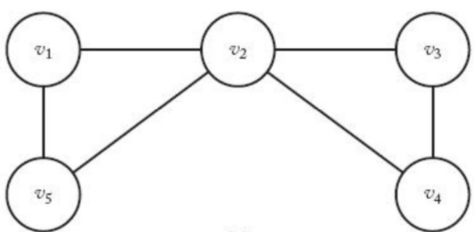

해밀톤 회로 문제 (1/3)

- 해밀톤 회로(Hamiltonian Circuit) 문제
  - 해밀톤 회로: 무방향 그래프에서 한 노드에서 시작하여 모든 노드를 정확하게 한 번 방문하고 시작 노드로 되돌아오는 경로



(a)

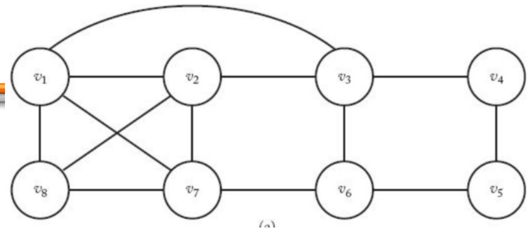
외판원 문제와의 관계.
모든 해밀톤 회로를 찾은 후에 가중치 합이 가장 짧은 회로가 외판원 문제의 답



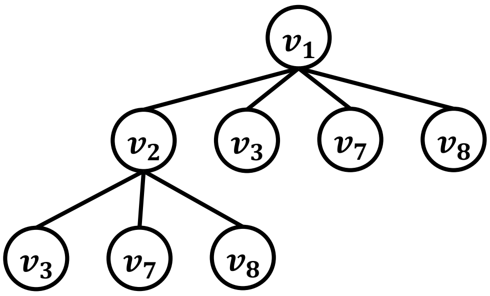
(b)

해밀톤 회로가 존재하지 않음

해밀톤 회로 문제 (2/3)



- 상태 공간 트리



- 루트 노드: 시작노드

- 조건.
  - 경로에 있는 i번째 노드는 $i-1$번째 노드의 인접 노드이어야 함
  - $n-1$번째 노드는 출발 노드의 인접 노드이어야 함
  - i번째 노드는 그 앞에 오는 $i-1$개의 노드 중 하나가 될 수 없음

해밀톤 회로 문제 (3/3)

promising(G[][], $output$[], $index$)

```
if index = 1 then return true
prev := output[index − 1]
curr := output[index]
if index = n − 1 and not G[prev][1] then return false
if index > 1 and not G[prev][curr] then return false
return true
```

hamiltonian(G[][], S, $output$[], $index$)

```
if promising(G, output, index) then
    if index = n − 1 then
        return output
    else
        for j := 2 to n do
            if j ∉ S then
                S.add(j)
                output[index + 1] := j
                hamiltonian(G, S, output, index + 1)
                S.remove(j)
```

- 시간 복잡도: $O(n^n)$