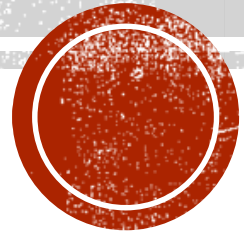
Lect06. Branch and Bound



Branch and Bound

- Illustrating Branch-and-Bound with the 0-1 Knapsack Problem
- The Traveling Salesperson Problem

What is Branch-and-Bound?

Features

1. An improvement on the backtracking algorithm.
2. Similar to backtracking, a state space tree is used.
3. Does not limit us to any particular way of traversing the tree.
4. It is used only for the optimization problem.

Principle of Branch-and-Bound

- Branch-and-Bound algorithm computes a number (bound) at a node to determine whether the node is promising.
- The bound is a limitation of solutions to be get branching from the node
- If the bound is not good comparing optimal solution, there is no necessary to branch anymore and the node is nonpromising.

P.S: the 0-1 Knapsack problem in Section 5.7 is actually a branch-and-bound algorithm.

0-1 knapsack problem

- 분기한정 가지치기로 깊이우선검색 (= 되추적)
  - 상태공간트리를 구축하여 되추적 기법으로 문제를 푼다.
  - 뿌리마디에서 왼쪽으로 가면 첫번째 아이템을 배낭에 넣는 경우이고, 오른쪽으로 가면 첫번째 아이템을 배낭에 넣지 않는 경우이다.
  - 동일한 방법으로 트리의 수준 1에서 왼쪽으로 가면 두 번째 아이템을 배낭에 넣는 경우이고, 오른쪽으로 가면 그렇지 않는 경우이다.
  - 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 검색을 하는 과정 동안 항상 그 때까지 찾은 최적의 해를 기억해 두어야 한다.

General backtracking for solving optimization

```
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.

0-1 Knapsack Problem: Algorithm

- 알고리즘 스켓치:
  - Let:
    - *profit* : 그 마디에 오기까지 넣었던 아이템의 값어치의 합.
    - *weight* : 그 마디에 오기까지 넣었던 아이템의 무게의 합.
    - *bound* : 마디가 수준 i에 있다고 하고, 수준 k에 있는 마디에서 총무게가 W를 넘는다고 하자. 그러면

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

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

    - *maxprofit* : 지금까지 찾은 최선의 해답이 주는 값어치
  - w_i와 p_i를 각각 i번째 아이템의 무게와 값어치라고 하면, p_i / w_i 의 값이 큰 것부터 내림차순으로 아이템을 정렬한다. (일종의 탐욕적인 방법이 되는 셈이지만, 알고리즘 자체는 탐욕적인 알고리즘은 아니다.)
  - *maxprofit* := $0; *profit* := $0; *weight* := 0

0-1 Knapsack Problem

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

- 보기: $n = 4, W = 16$이고 일 때, 되추적을 사용하여 구축되는 가지친 상태공간트리를 그려 보시오.

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

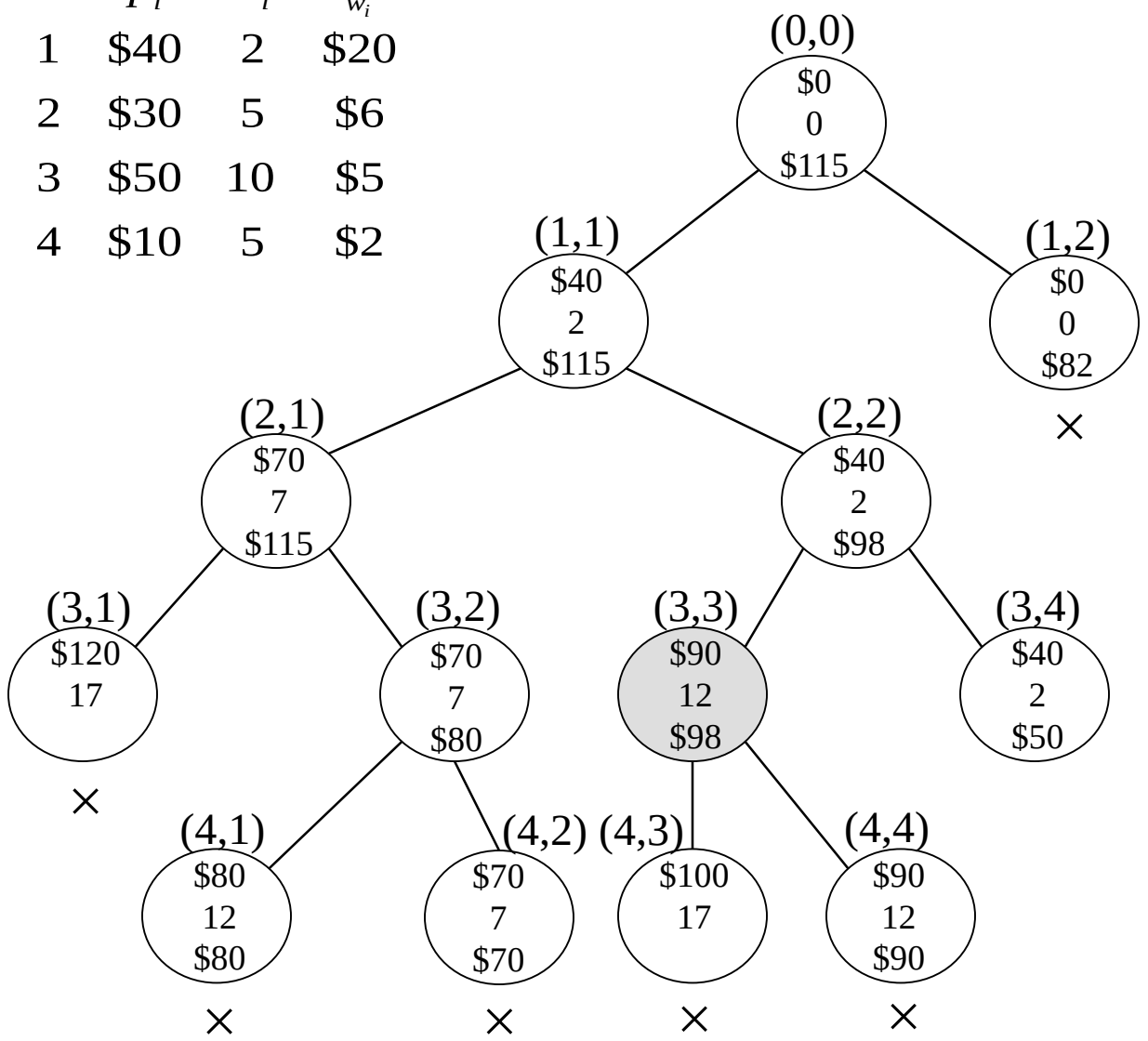
i	p_i	w_i	$\frac{p_i}{w_i}$
1	$40	2	$20
2	$30	5	$6
3	$50	10	$5
4	$10	5	$2

item 1 $\begin{bmatrix}\$40\\2\end{bmatrix}$

item 2 $\begin{bmatrix}\$30\\5\end{bmatrix}$

item 3 $\begin{bmatrix}\$50\\10\end{bmatrix}$

item 4 $\begin{bmatrix}\$10\\5\end{bmatrix}$



0-1 Knapsack problem: Analysis

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

Breadth-First Branch-and-Bound

Breadth-First Branch-and-Bound

- 너비우선검색(Breadth-first Search)순서:

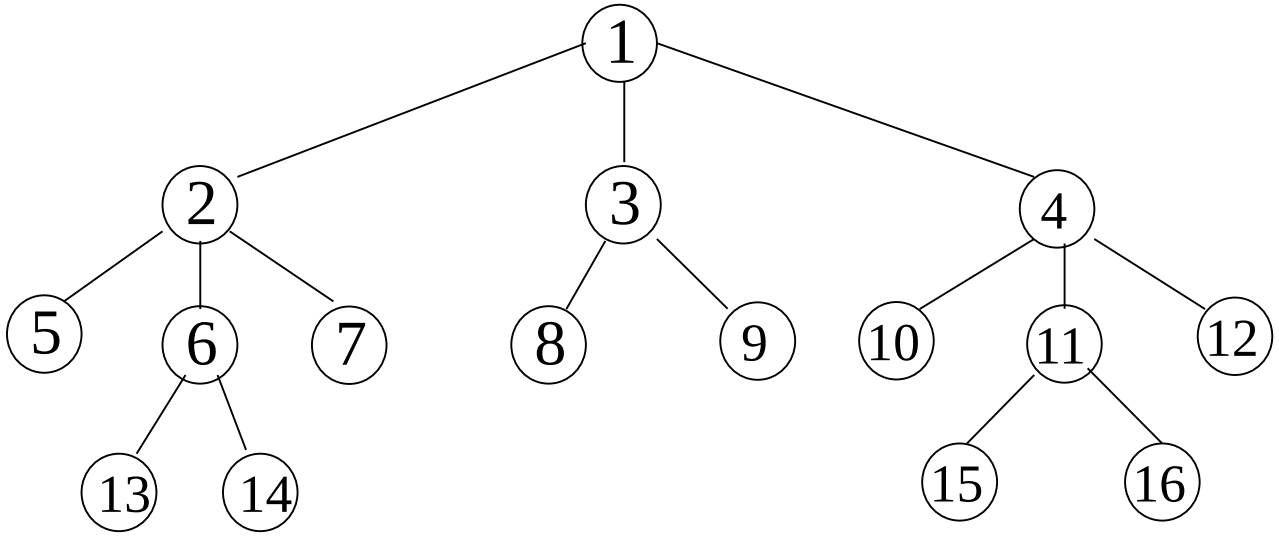
(1) 뿌리마디를 먼저 검색한다.

(2) 다음에 수준 1에 있는 모든 마디를 검색한다.
(왼쪽에서 오른쪽으로)

(3) 다음에 수준 2에 있는 모든 마디를 검색한다
(왼쪽에서 오른쪽으로)

(4) ...

Breadth-First Search of a Tree



BFS algorithm

- 되부름(**recursive**) 알고리즘을 작성하기는 상당히 복잡하다. 따라서 **대기열(queue)**을 사용한다.

```
void breadth_first_search(tree T) {
  queue_of_node Q;
  node u, v;
  initialize(Q);
  v = root of T;
  visit v;
  enqueue(Q,v);
  while(!empty(Q)) {
    dequeue(Q,v);
    for(each child u of v) {
      visit u;
      enqueue(Q,u);
    }
  }
}
```

BFS algorithm with Bound_and_Branch

```
void breadth_first_branch_and_bound(state_space_tree T,
                                    number& best) {
  queue_of_node Q;
  node u, v;
  initialize(Q);              // Q는 빈 대기열로 초기화
  v = root of T;              //뿌리마디를 방문
  enqueue(Q,v);
  best = value(v);
  while(!empty(Q)) {
    dequeue(Q,v);
    for(each child u of v) { // 각 자식마디를 방문
      if(value(u) is better than best)
        best = value(u);
      if(bound(u) is better than best)
        enqueue(Q,u);
    }
  }
}
```

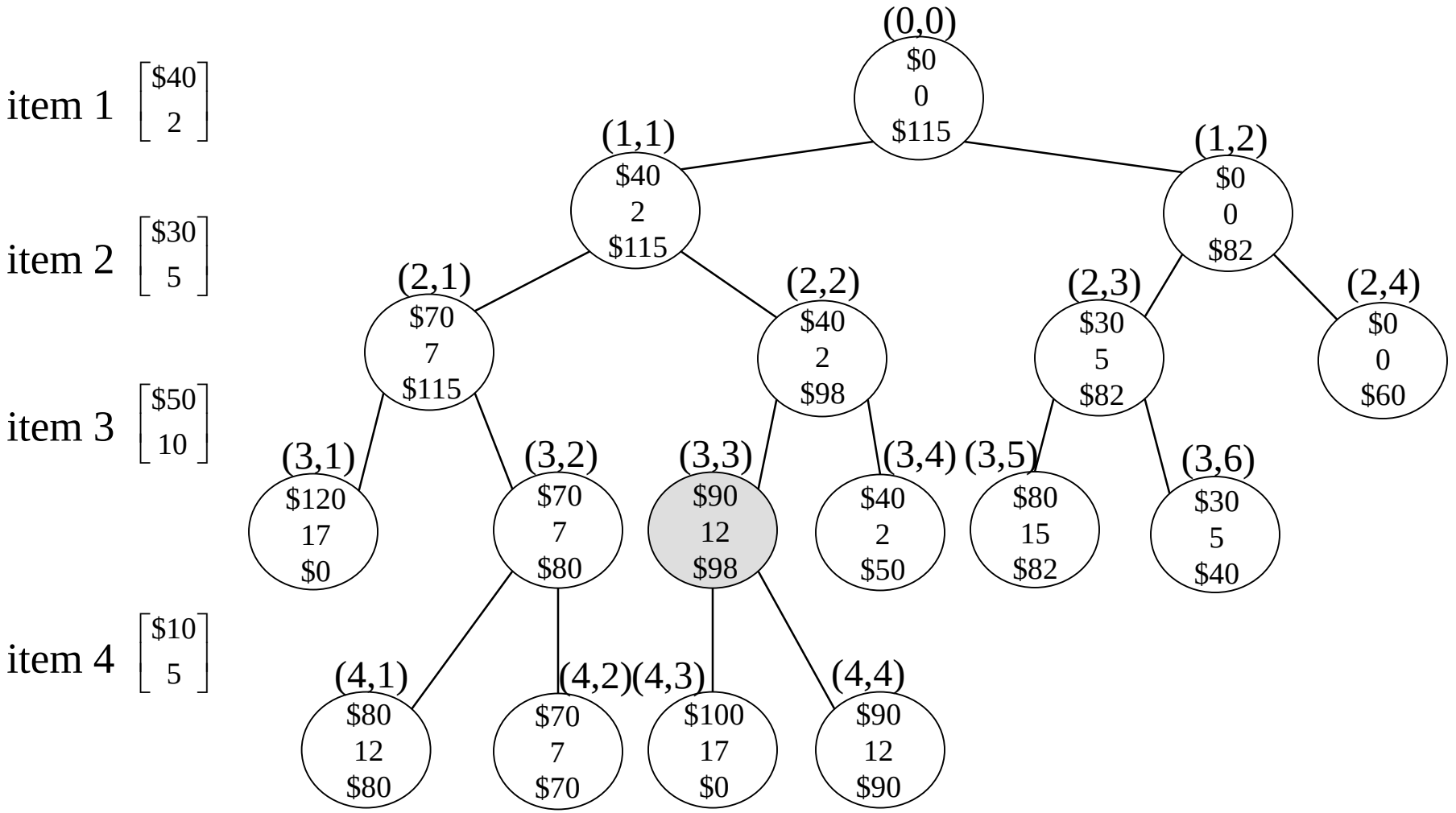
Algorithm 6.1

```
void knapsack2(int n, const int p[], const int w[], int W, int maxprofit)
{    queue_of_node Q;     node  u, v ;
     initialize(Q);          v.level = 0 ;   v.profit = 0 ;   v.weight = 0 ;
     maxprofit = 0;        enqueue(Q, v);
     while (!empty(Q)){
          dequeue(Q, v);
          u.level = v.level+1 ;
          u.weight = v.weight +w[u.level] ;
          u.profit = v.profit+p[u.level] ;
          if (u.weight <=W && u.profit>maxprofit )
               maxprofit = u.profit;
          if (bound(u) > maxprofit)
              enqueue(Q, u);
          u.weight = v.weight ;          u.profit = v.profit ;
          if (bound(u) > maxprofi)
               enqueue(Q, u);
     }
}
```

Algorithm 6.1 (Cont'd)

```
float bound(node u)
{    index j, k ;     int  totalweight ;    float result ;
     if (u.weight >=W)
         return 0;
     else {  result = u.profit ;
             j = u.level+1 ;    totweight = u.weight ;
             while (j <=n && totweight+w[j]<=W ){
                     totweight= totweight+w[j];
                     result= result+p[j];
                     j++;
              }
             k = j ;
             if (k <= n)
               result= result+(W–totweight)×p[k]/w[k];
             return result ; //cf. backtracking, in promising(i),
          }                        // return (return>maxprofit);
}
```

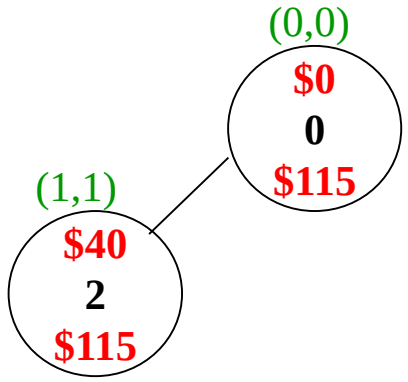
- 보기: 앞에서와 같은 예를 사용하여 분기한정 가지치기로 너비우선검색을 하여 가지친 상태공간트리를 그려보면 다음과 같이 된다. 이때 검색하는 마디의 개수는 **17**이다. 되추적 알고리즘보다 좋지 않다!



Example 6.1(BFS with Branch-and-Bound)

Suppose that $n = 4$, $W=16$, and we have the following

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



Find the solutions.

Sol :

1. Set *maxprofit* = 0
2. Visit node(0,0)
   profit = 0
   weight = 0
   totweight=0+2+5=7
   bound = 0+40+30+(16-7)*50/10=115
3. Visit node (1,1) : *profit* = 40 *weight* = 2 *maxprofit* = 40
   bound = 40+30+(16-7)*50/10, *totweight*=2+5=7

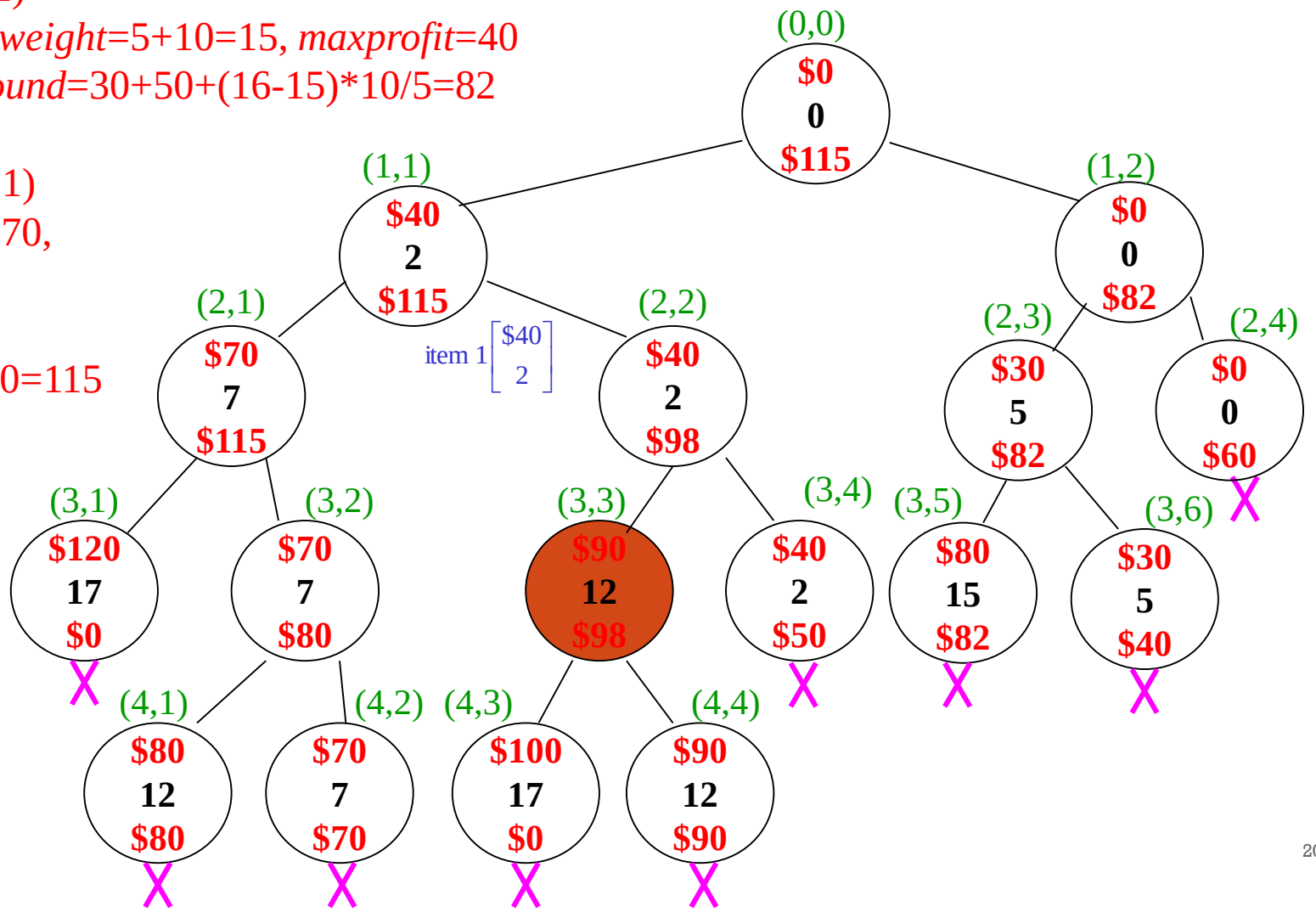
Example 6.1 (Cont'd)

n = 4, *W*=16,

item 2 [$30, 5] item 3 [$50, 10] item 4 [$10, 5]

4. Visit node(1,2)
   profit = 0, *totweight*=5+10=15, *maxprofit*=40
   weight = 0 *bound*=30+50+(16-15)*10/5=82

5. Visit node (2,1)
   profit=40+30=70,
   weight =2+5=7
   bound=
   70+(16-7)*50/10=115
   maxprofit=70



Example 6.1 (Cont'd)

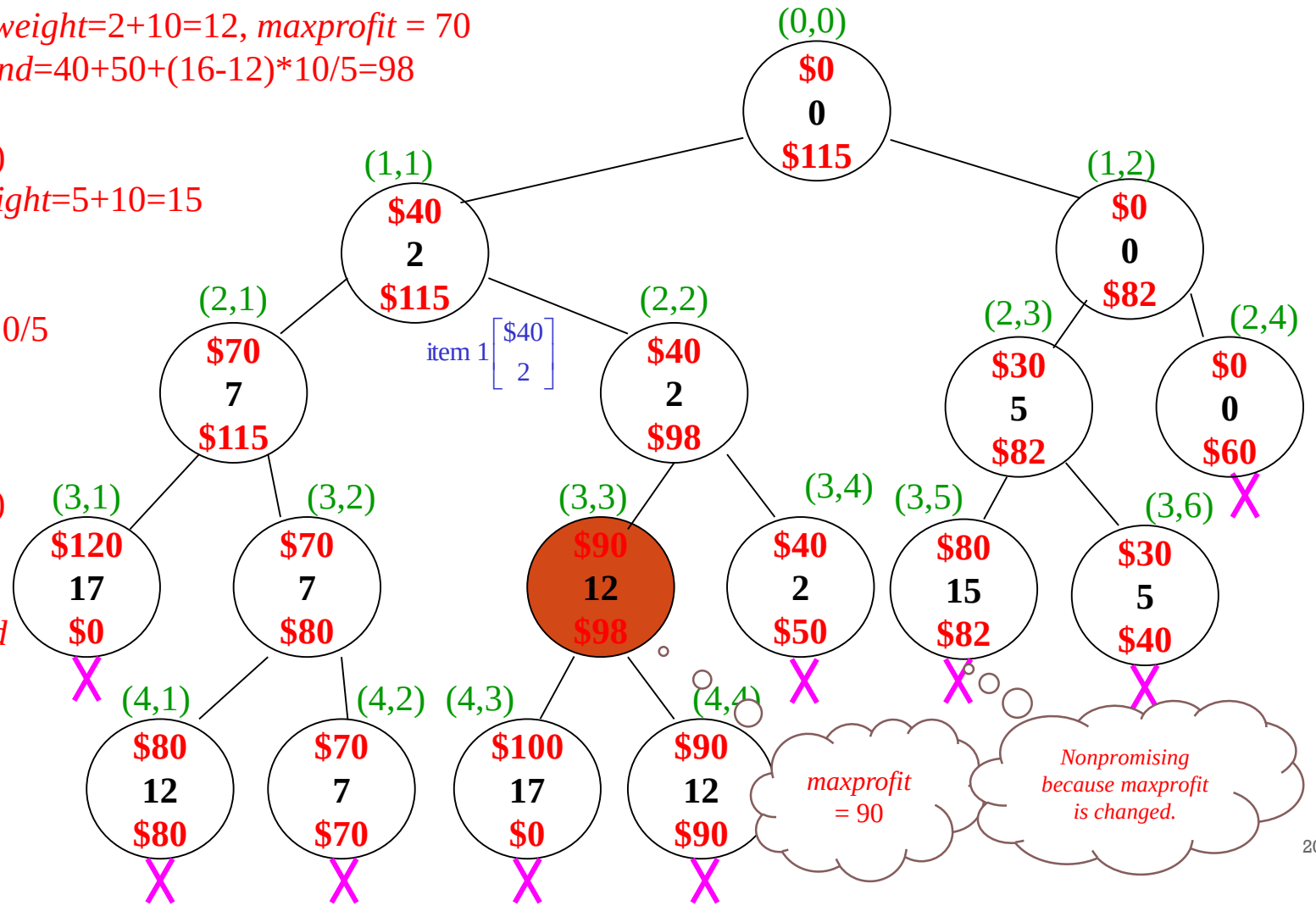
n = 4, *W*=16,

item 2 [$30, 5] item 3 [$50, 10] item 4 [$10, 5]

6. Visit node(2,2)
   profit = 40, *totweight*=2+10=12, *maxprofit* = 70
   weight = 2 *bound*=40+50+(16-12)*10/5=98

7. Visit node (2,3)
 profit=30, *totweight*=5+10=15
 weight =5
 bound=
30+50+(16-15)*10/5
=82
 maxprofit=70

8. Visit node (2,4)
 profit=30
 bound=60
 maxprofit>*bound*
 : no expand



Best-First Search with Brach_and_Bound

Illustrating Branch-and-Bound with the 0-1 Knapsack Problem

1. Breadth-First Search with Branch-and-Bound Pruning for the 0-1 Knapsack problem.
2. Best-First Search with Branch-and-Bound Pruning for the 0-1 Knapsack problem : an improvement of 1.
   - *profit* : 그 마디에 오기까지 넣었던 아이템의 값어치의 합.
   - *weight* : 그 마디에 오기까지 넣었던 아이템의 무게의 합.
   - *bound* : 마디가 수준 *i*에 있다고 하고, 수준 *k*에 있는 마디에서 총무게가 *W*를 넘는다고 하자. 그러면

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

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

   - *maxprofit* : 지금까지 찾은 최선의 해답이 주는 값어치
   - w_i와 p_i를 각각 *i*번째 아이템의 무게와 값어치라고 하면, p_i / w_i 의 값이 큰 것부터 내림차순으로 아이템을 정렬한다. (일종의 탐욕적인 방법이 되는 셈이지만, 알고리즘 자체는 탐욕적인 알고리즘은 아니다.)
   - *maxprofit* := $0; *profit* := $0; *weight* := 0

A node is nonpromising if this bound is less than or equal to *maxprofit* or *weight* $\geq W$.

Best-First Search algorithm with Branch_and_Bound

- 최적의 해답에 더 빨리 도달하기위한 전략:
  1. 주어진 마디의 모든 자식마디를 검색한 후,
  2. 유망하면서 확장되지 않은(unexpanded) 마디를 살펴보고,
  3. 그 중에서 가장 좋은(최고의) 한계치(bound)를 가진 마디를 확장한다.
- 최고우선검색(Best-First Search)은 너비우선검색에 비해서 좋아짐

Best-First Search Strategy

- 최고의 한계를 가진 마디를 우선적으로 선택하기 위해서 우선순위 대기열 (Priority Queue)을 사용한다.
- 우선순위 대기열은 힙(heap)을 사용하여 효과적으로 구현할 수 있다.

Branch-and-Bound Pruning

- Best-first search with branch-and-bound pruning:

  A simple modification of the approach called breadth-first search (review algorithm) with branch-and-bound pruning.

```
void breadth_first_tree_search(tree T)
{    queue_of_node Q;        node u, v ;
     initialize(Q);    v = root of T ;
     visit v;    enqueue(Q, v);
     while (!empty(Q)){
          dequeue(Q, v);
          for (each child u of v){
               visit u;
               enqueue(Q, u);
          }
     }
}
```

Best-First Algorithm with Branch_and_Bound

```
void best_first_branch_and_bound(state_space_tree T,
                                 number best) {
  priority_queue_of_node PQ;
  node u,v;
  initialize(PQ);                    // PQ를 빈 대기열로 초기화
  v = root of T;
  best = value(v);
  insert(PQ,v);
  while(!empty(PQ)) {                // 최고 한계값을 가진 마디를 제거
    remove(PQ,v);
    if(bound(v) is better than best)      // 마디가 아직 유망한 지 점검
      for(each child u of v) {
        if(value(u) is better than best)
          best = value(u);
        if(bound(u) is better than best)
          insert(PQ,u);
      }
  }
}
```

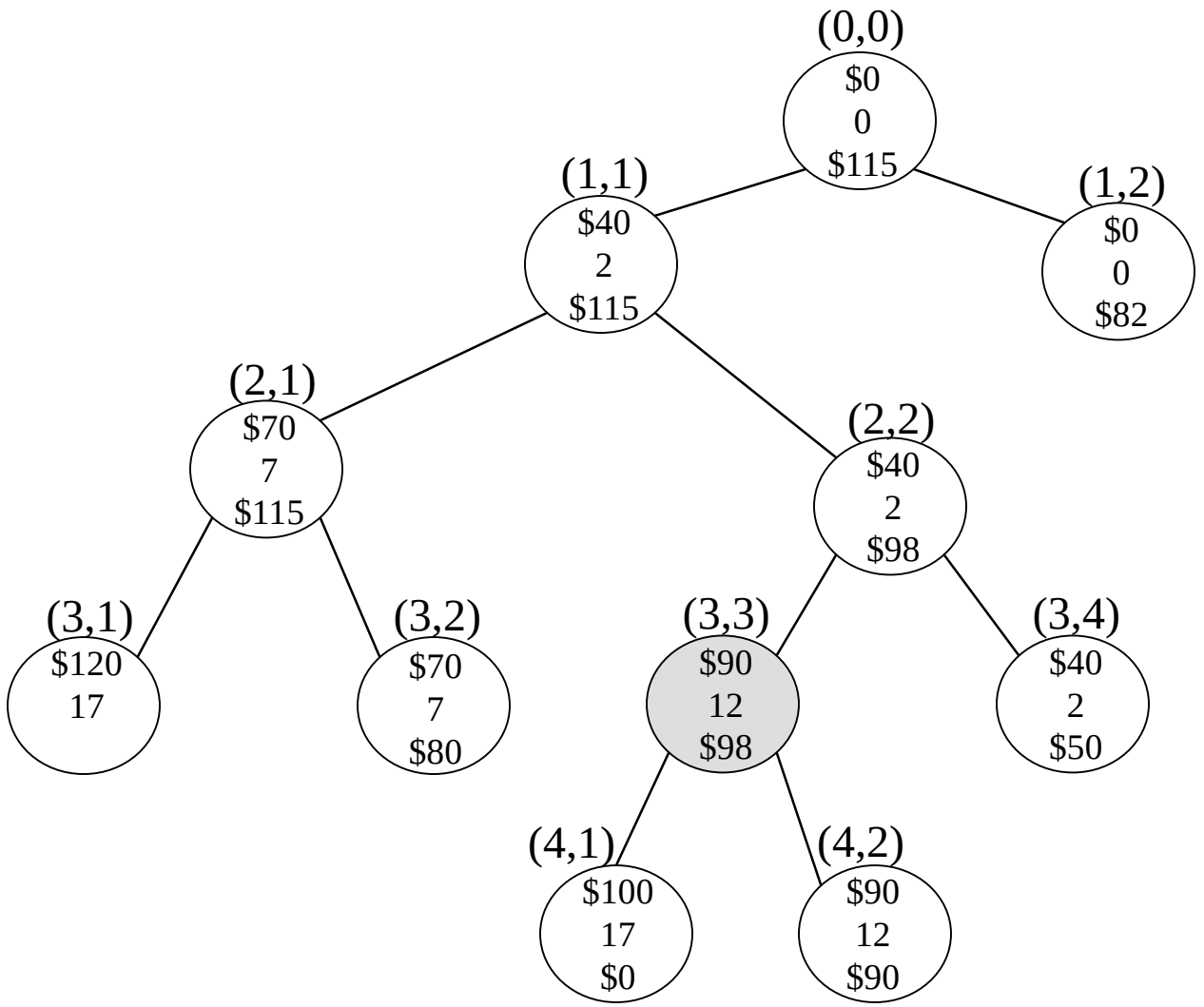
- 보기: 앞에서와 같은 예를 사용하여 분기한정 가지치기로 최고우선검색을 하여 가지친 상태공간 트리를 그려보면, 다음과 같이 된다. 이때 검색하는 마디의 개수는 **11**이다.

item 1 $\begin{bmatrix}\$40\\2\end{bmatrix}$

item 2 $\begin{bmatrix}\$30\\5\end{bmatrix}$

item 3 $\begin{bmatrix}\$50\\10\end{bmatrix}$

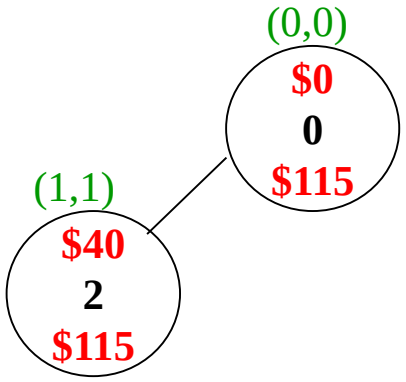
item 4 $\begin{bmatrix}\$10\\5\end{bmatrix}$



Example 6.2(Best-First Search with Branch-and-Bound)

Suppose that $n = 4$, $W=16$, and we have the following

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



Find the solutions.

Sol：

1. Set *maxprofit* = 0
2. Visit node(0,0)

   profit = 0 … *maxprofit*=0

   weight = 0, *totweight=0+2+5=7*

   bound = 0+40+30+(16-7)*50/10=115

3. Visit node (1,1) ：

profit = 40 *weight* = 2 *profit* = 40 *weight* = 2 *maxprofit* = 40

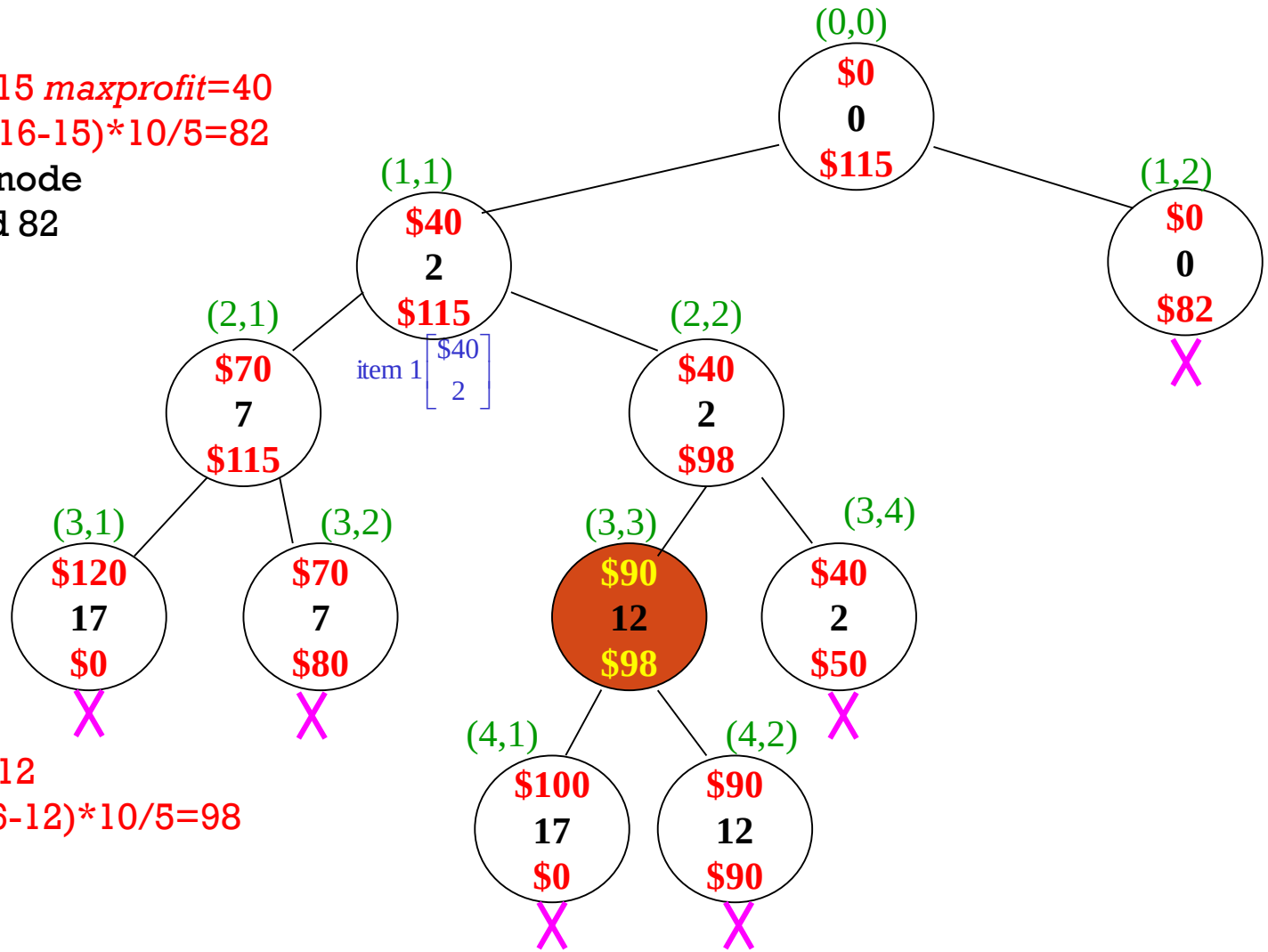
bound = 40+30+(16-7)*50/10=115, *totweight*=2+5=7

Example 6.2 (Cont'd)

$n = 4, W=16,$

item 2 [$30, 5] item 3 [$50, 10] item 4 [$10, 5]

4. Visit node(1,2)
   profit = 0, *totweight*=5+10=15 *maxprofit*=40
   weight = 0 *bound*=30+50+(16-15)*10/5=82
5. Not extended & promising node
   (1,1)-bound 115, (1,2)-bound 82
   child of (1,1) : (2,1) & (2,2)
6. Visit node (2,1)
   profit=40+30=70,
   weight =2+5=7
   bound=
   70+(16-7)*50/10=115
   maxprofit=70



7. Visit node(2,2)
   profit = 40, *totweight*=2+10=12
   weight = 2 *bound*=40+50+(16-12)*10/5=98

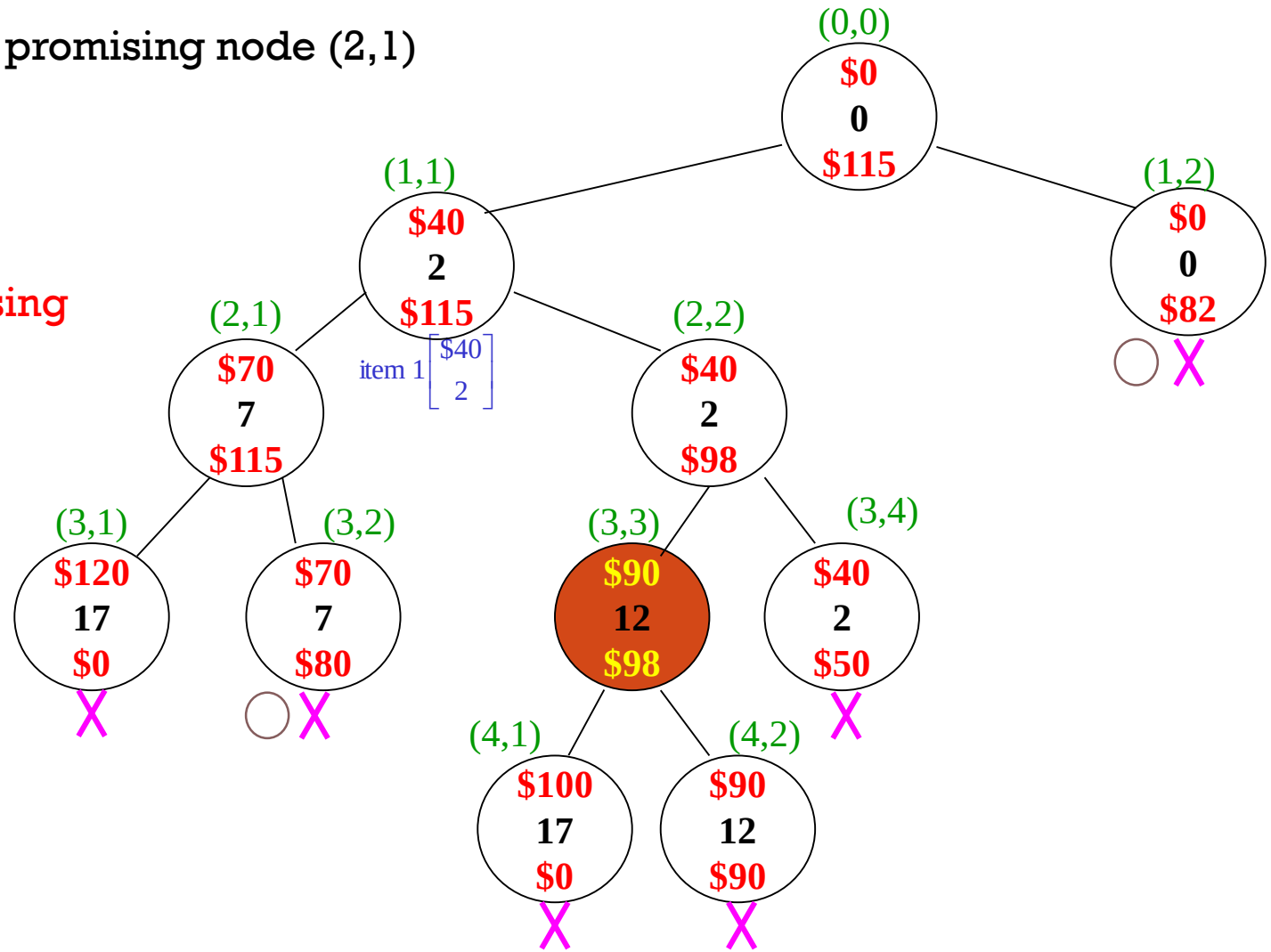
Example 6.2 (Cont'd)

n = 4, *W*=16,

item 2 [$30, 5] item 3 [$50, 10] item 4 [$10, 5]

8. Not extended, max bound & promising node (2,1)
   its children (3,1) (3,2)
9. Visit node (3,1)
   profit=40+30+50=120,
   totweight=2+5=7
   weight =2+5+10=17:nopromising
   bound=0 : no expand
   maxprofit=70

10. Visit node (3,2)
    profit=70
    w=2+5=7
    bound=40+30+10=80

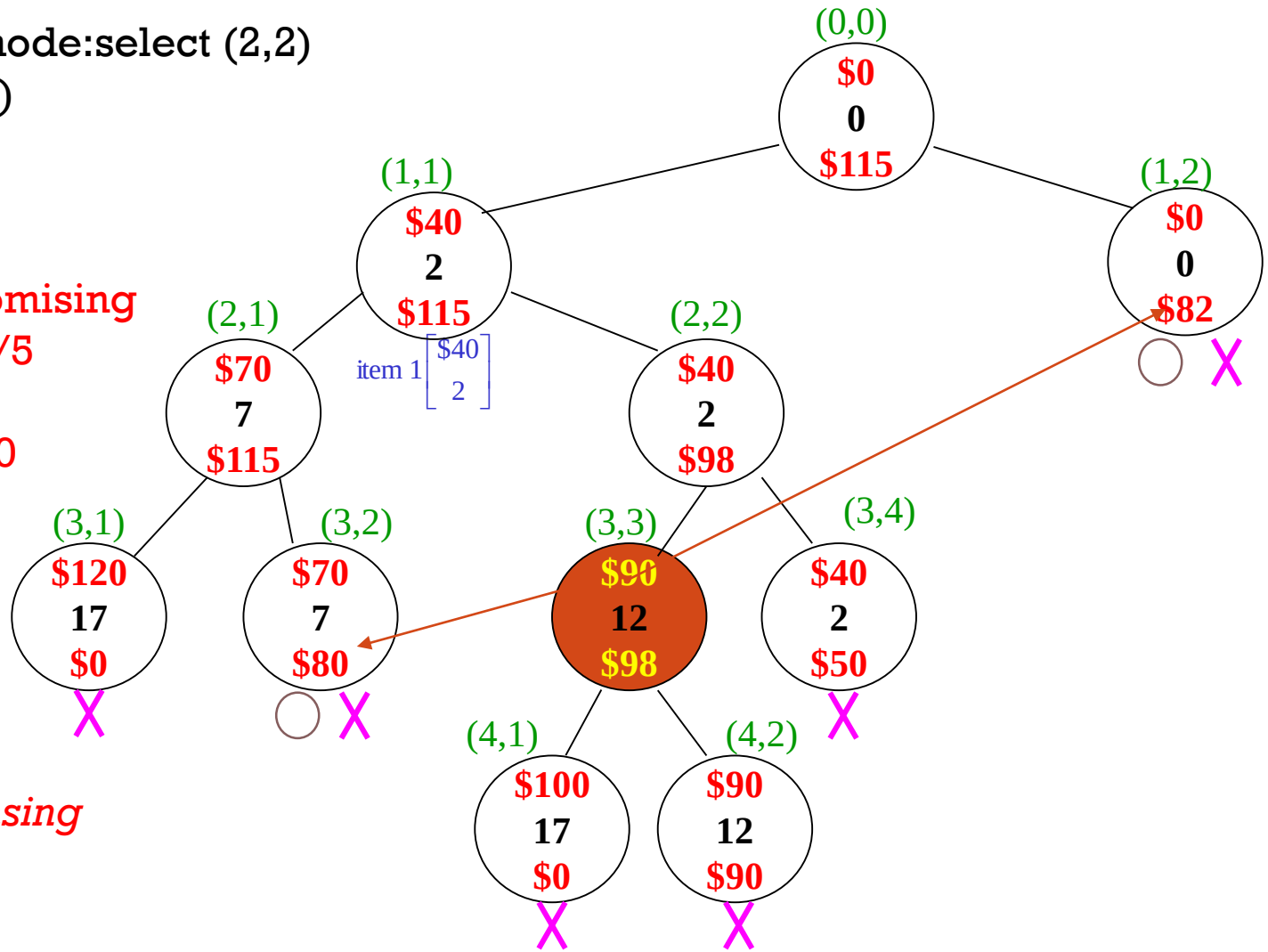


Example 6.2 (Cont'd)

n = 4, *W*=16,

item 2 [$30, 5] item 3 [$50, 10] item 4 [$10, 5]

11. Max bound promising node:select (2,2)
    its children (3,3) ans (3,4)
12. Visit node (3,3)
 profit=40+50=90,
 totweight=2+5=7
 weight =2+5+10=17:nopromising
 bound=40+50+(16-12)*10/5
 = 98
 profit> *maxprofit*(=70)=>90
13. ((3,2)&(1,2) node
 bound<maxprofit
 : nonpromising
14. Visit node (3,4)
 profit=40
 w=2 *bound*=40+10=50
 bound<*maxprofit:nonpromsing*

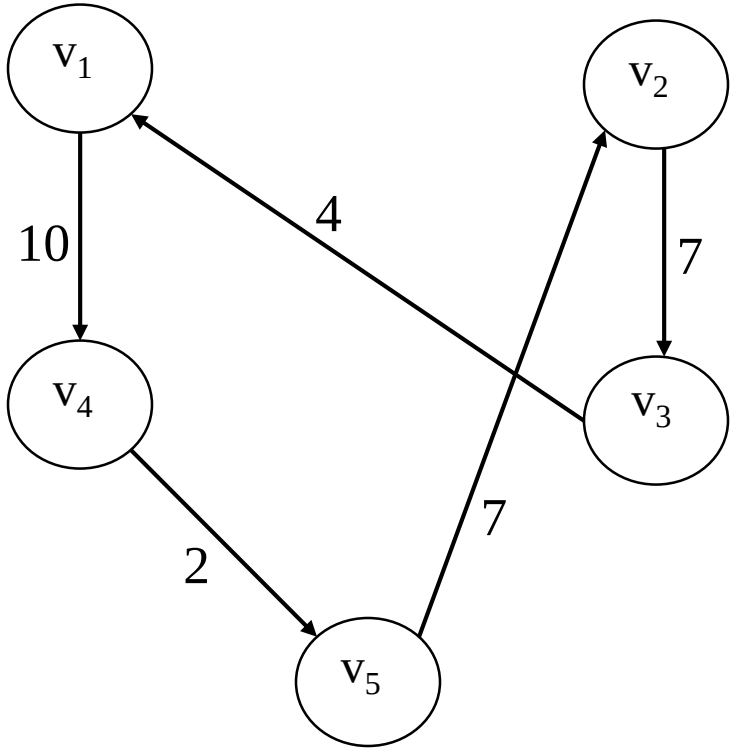


The Traveling Salesperson Problem

TSP with Branch_and_Bound

- $n = 40$일 때, 동적계획법 알고리즘은 6년 이상이 걸린다. 그러므로 분기한정법을 시도해 본다.
- **보기:** 다음 인접행렬로 표현된 그래프를 살펴보시오.

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$



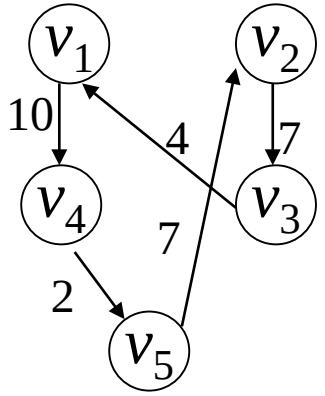
The Traveling Salesperson Problem

How to find the optimal tour more efficient than using backtracking?

Branch-and-Bound can be considered for searching the state space tree.

Best-First Search：the best bound at present is searched first

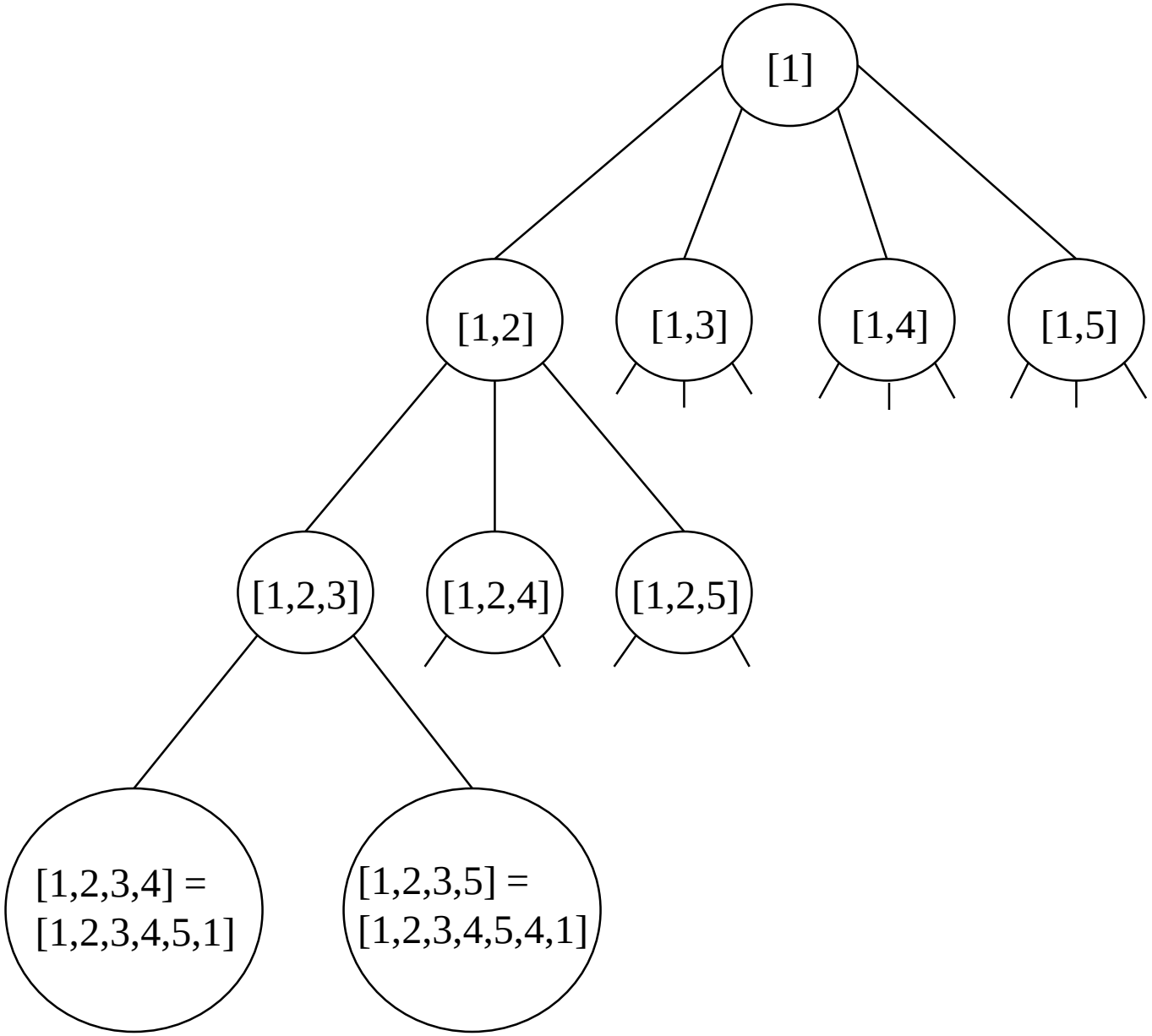
$$W = \begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$



Optimal tour

State Space Tree for TSP

- 각 마디는 출발마디로부터의 일주여행경로를 나타내게 되는데, 몇 개 만 예를 들어 보면, 뿌리마디의 여행경로는 [1]이 되고, 뿌리마디에서 뻗어 나가는 수준 1에 있는 여행경로는 각각 [1,2], [1,3], …, [1,5]가 되고, 마디 [1,2]에서 뻗어 나가는 수준 2에 있는 마디들의 여행경로는 각각 [1,2,3],…,[1,2,5]가 되고, 이런 식으로 뻗어 나가서 잎마디에 도달하게 되면 완전한 일주여행경로를 가지게 된다.
- 따라서 최적일주여행경로를 구하기 위해서는 잎마디에 있는 일주여행경로를 모두 검사하여 그 중에서 가장 길이가 짧은 일주여행경로를 찾으면 된다.
- 참고: 위 예에서 각 마디에 저장되어 있는 마디가 4개가 되면 더 이상 뻗어 나갈 필요가 없다. 왜냐하면, 남은 경로는 더 이상 뻗어 나가지 않고도 알 수 있기 때문이다.


[1]
[1,2]
[1,3]
[1,4]
[1,5]
[1,2,3]
[1,2,4]
[1,2,5]
[1,2,3,4] =
[1,2,3,4,5,1]
[1,2,3,5] =
[1,2,3,4,5,4,1]

Best-Search with Branch_and_Bound

- 분기한정 가지치기로 **최고우선 검색을** 사용하기 위해서 **각 마디의 한계치**를 구할 수 있어야 한다. 이 문제에서는 주어진 마디에서 뻗어 나가서 얻을 수 있는 **여행경로의 길이의 하한(최소치)을 구하여 한계치**로 한다. 그리고 각 마디를 검색할 때 최소여행경로의 길이 보다 한계치가 작은 경우 그 마디는 유망하다고 한다. 최소여행경로의 초기값은 ∞로 놓는다. 따라서 완전한 여행경로를 처음 얻을 때 까지는 한계치가 무조건 최소여행경로의 길이 보다 작게 되므로 모든 마디는 유망하다.
- 각 마디의 한계치는 어떻게 구하나?

[1,…,k]의 여행경로를 가진 마디의 한계치는 다음과 같이 구한다. **Let:**

$A = V$ - ([1,…,k] 경로에 속한 모든 마디의 집합)

bound = [1,…,k] 경로 상의 총거리

+ v_k에서 A에 속한 정점으로 가는 이음선의 길이들 중에서 최소치

+ $\Sigma_{i \in A}$(v_i에서 $A \cup \{v_1\} - \{v_i\}$에 속한 정점으로 가는 이음선의 길이들 중에서 최소치)

Computation of Bound for a Node

1. Starts at v_1, the 2nd vertex can be v_2, v_3, v_4, v_5.

   v_1 : min{14,4,10,20}=4 v_2 : min{14,7,8,7}=7

   v_3 : min{4,5,7,16}=4 v_4 : min{11,7,9,2}=2

   v_5 : min{18,7,17,4}=4

   The minimum bound is 21.

   There can be no tour with a shorter length.

2. Suppose we have visited $[v_1, v_2]$

   v_1 : 14

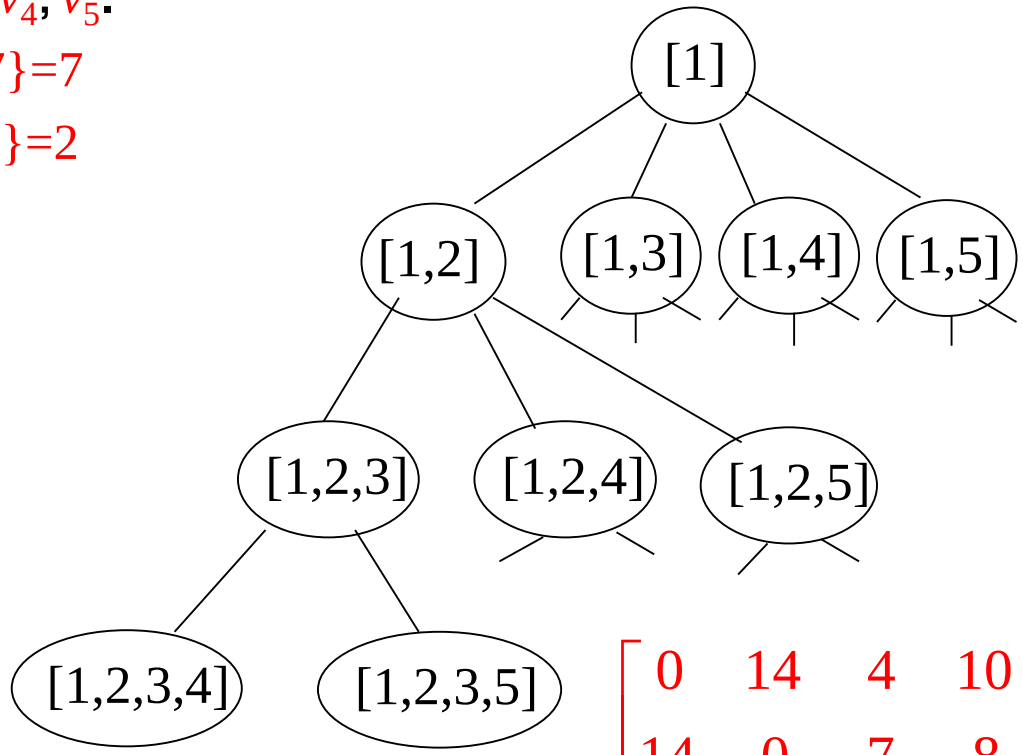
   v_2 : min{7,8,7}=7

   v_3 : min{4,7,16}=4

   v_4 : min{11,9,2}=2

   v_5 : min{18,17,4}=4

   The minimum bound is 31.



$$W = \begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$

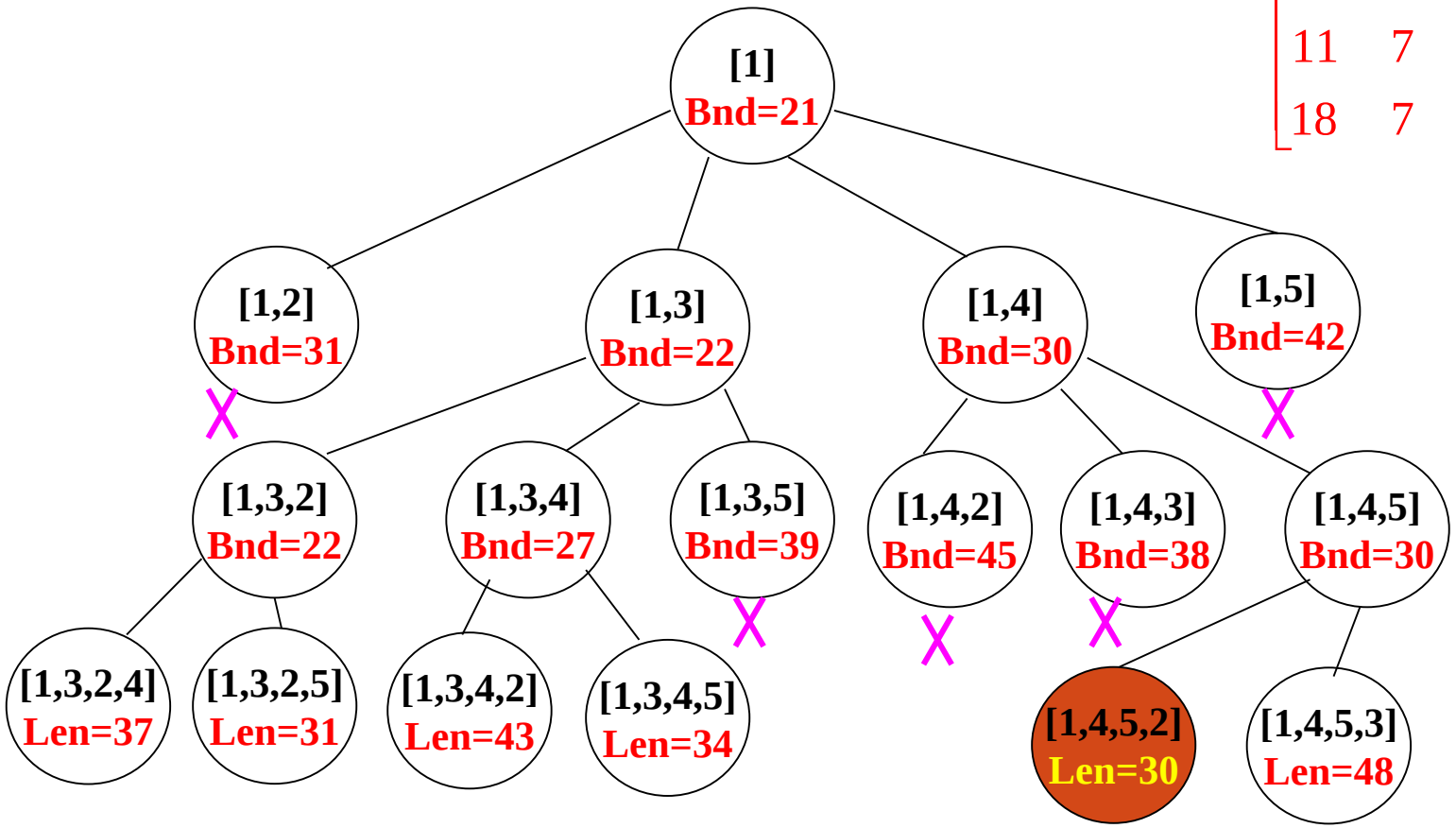
Exampl11e 6.3

1. Visit node [1] : bound=21, minlength=∞
2. Visit node [1,2] : bound=31,
3. Visit node [1,3] : bound=22, ← Best First Search Node
4. Visit node [1,4] : bound=30,
5. Visit node [1,5] : bound=42,
6. Visit node [1,3,2] : bound=22, ← Best First Search Node
7. Visit node [1,3,4] : bound=27,
8. Visit node [1,3,5] : bound=39,
9. Visit node [1,3,2,4] : bound=37, minlength=37
10. Visit node [1,3,2,5] : bound=31, minlength=31

$$W = \begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$

Example 6.3 (Cont'd)

$$W = \begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$



Algorithm 6.3

```
void travel2 (int  n,
              const number  W[][],
              ordered_set&  opttour
              number&  minlength)
{  priority_queue_of_node  PQ;
   node  u, v;
   initialize(PQ);
   v.level = 0;
   v.path = [1];
   v.bound = bound(v);
   minlength = ∞;
   insert(PQ, v);
   while (!empty(PQ ))
       {  remove(PQ, v);
          if (v.bound < minlength)
             {  v.level = v.level +1;
for (all i such that 2≤ i ≤ n
                 && i is not in v.path)
    {  u.path = v.path;
       put i at the end of u.path;
       if (u.level== n–2)
          {  put index of only vertex not
             in u.path at the end of u.path ;
             put 1 at the end of u.path ;
             if (length(u)<minlength)
                { minlength=length(u);
                  opttour = u.path ; }
          }
       else
          {  u.bound = bound(u);
             if (u.bound< minlength)
                insert(PQ, u);
          }
    } }}}
```

Analysis

- 이 알고리즘은 방문하는 마디의 개수가 더 적다.
- 그러나 아직도 알고리즘의 시간복잡도는 지수적이거나 그보다 못하다!
- 다시 말해서 $n = 40$이 되면 문제를 풀 수 없는 것과 다름없다고 할 수 있다.
- 다른 방법이 있을까?
  - 근사(approximation) 알고리즘: 최적의 해답을 준다는 보장은 없지만, 무리 없이 최적에 가까운 해답을 주는 알고리즘.