# Chap 6. Branch-and-Bound

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

**6.2 The Traveling Salesperson Problem**

# Branch-and-Bound

- Branch-and-Bound (분기한정법)
  - Similar to backtracking
    - A state space tree is used to solve a problem
  - Differences

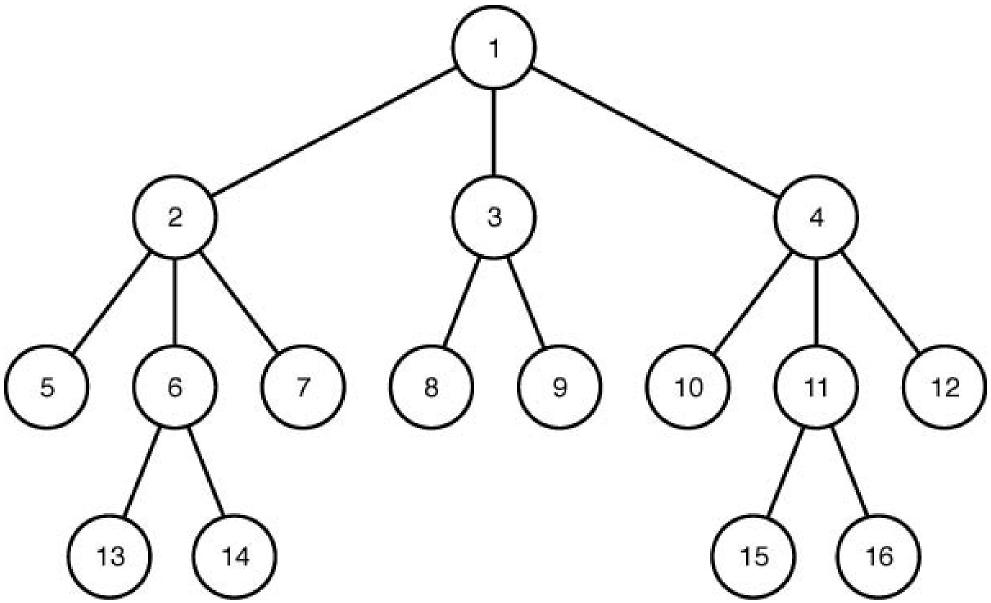
    (1) Doesn’t limit to any particular way of traversing the tree

    (2) Used only for optimization problems

  - Design strategy
    - Compute a number (bound) at a node to determine whether the node is promising
    - The number is a bound on the value of the solution that could be obtained by expanding beyond the node
    - If that bound is no better than the value of the best solution found so far, the node is nonpromising; otherwise promising.

# Breadth-First Search with Branch-and-Bound Pruning

- Breadth-first Search (너비우선검색) 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

- 일반적인 너비 우선 검색 알고리즘
  - 되부름(recursive) 알고리즘을 작성하기는 상당히 복잡하다. **Queue**를 사용

```
void breadth_first_search(tree T) {
    queue_of_node Q;
    node u, v;

    initialize(Q);                          // Initialize Q to be empty
    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);
        }
    }
}
```

# Breadth-First Search with Branch-and-Bound Pruning

```
void breadth_first_branch_and_bound(state_space_tree T, number& best){
    queue_of_node Q;
    node u, v;

    initialize(Q);                          // Initialize Q to be empty
    v = root of T;                          // Visit root
    enqueue(Q,v);
    best = value(v);

    while(!empty(Q)) {
        dequeue(Q,v);
        for(each child u of v) {            // Visit each child
            if(value(u) is better than best)
                best = value(u);
            if(bound(u) is better than best)
                enqueue(Q,u);
        }
    }
}
```

# Breadth-First Search with Branch-and-Bound Pruning

- Ex 6.1 (Same as Ex 5.6)

  $n = 4$, $W = 16$이고,

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

일 때,

- Do a breadth-first search (instead of a depth-first search)
- Let weight & profit be the total weight & total profit up to a node

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

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

# Breadth-First Search with Branch-and-Bound Pruning

- Algorithm 6.1 : The Breadth-First-Search with Branch-and-Bound Pruning Algorithm for the 0-1 Knapsack Problem
  - 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 : maxprofit.

```
Struct node {
        int level;
        int profit;
        int weight;
}
```

# Breadth-First Search with Branch-and-Bound Pruning

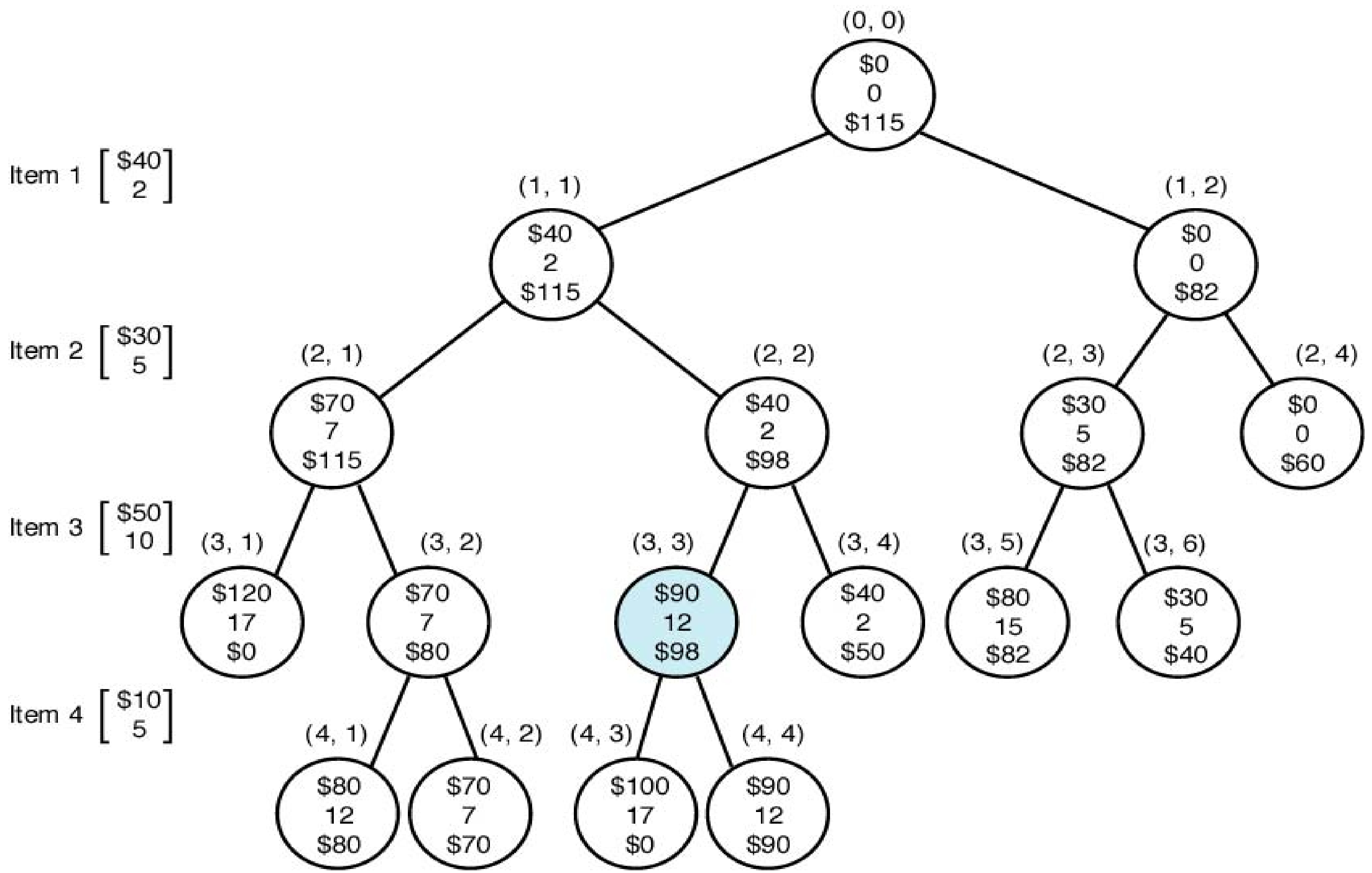
```
void knapsack2(int n, const int p[], cont int w[], int W, int &maxprofit) {
    queue_of_node Q;      node u, v;
    initialize(Q);
    v.level =0; v.profit = 0; v.weight = 0; maxprofit = 0;
    enqueue(Q, v);
    while (!empty(Q)) {
        dequeue(Q, v);   u.level = v.level+1;
        u.profit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

# Breadth-First Search with Branch-and-Bound Pruning

```
float bound(node u) {
      index j, k;  int totweight;  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;
      }
}
```

# Breadth-First Search with Branch-and-Bound Pruning

- 분기한정 가지치기로 너비우선검색을 하여 상태공간트리를 그려보면 검색 마디의 개수는 **17개**이다. – 분기한정 가지치기 되추적 알고리즘(13개)

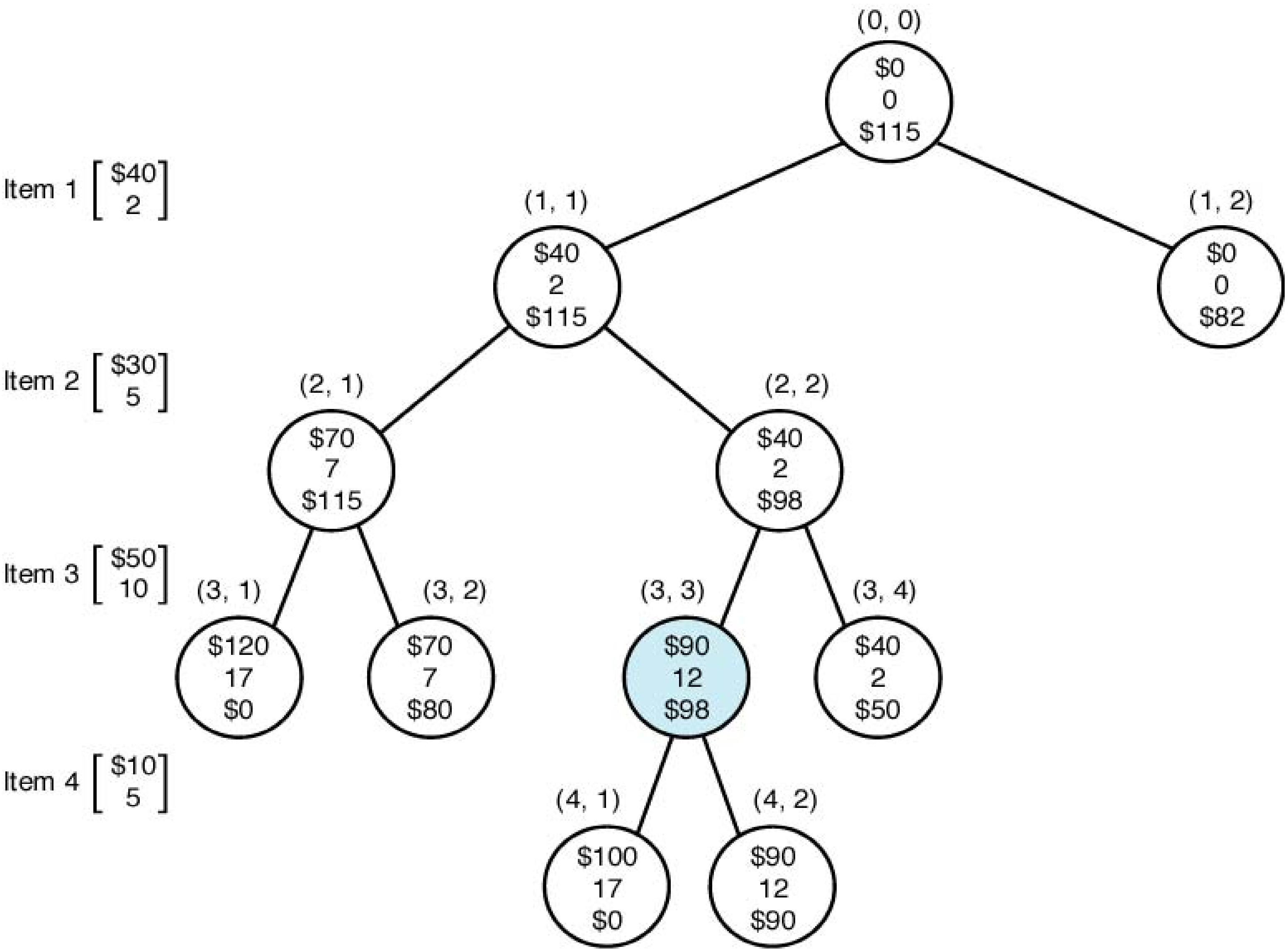


# Best-First Search with Branch-and-Bound Pruning

- 최적의 해답에 더 빨리 도달하기 위한 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 확장

- 최고우선검색(Best-First Search) 전략
  - 최고의 한계를 가진 마디를 우선적으로 선택하기 위해서 우선순위 대기열(Priority Queue)을 사용
  - 우선순위 대기열은 힙(heap)을 사용하여 효과적으로 구현
- Ex 6.2 (Same as Ex 5.6)
  - 앞에서와 같은 예를 사용하여 분기한정 가지치기로 최고우선검색을 하여 상태공간트리를 그려보면, 검색하는 마디의 개수는 **11개**

# Best-First Search with Branch-and-Bound Pruning



# Best-First Search with Branch-and-Bound Pruning

- 분기한정 최고우선검색 알고리즘

```
void best_first_branch_and_bound (state_space_tree T,number best) {
    priority_queue_of_node PQ;
    node u,v;
    initialize(PQ);                          // initialize PQ to empty
    v = root of T;
    best = value(v);
    insert(PQ,v);
    while(!empty(PQ)) {                      // Remove node with best bound
        remove(PQ,v);
        if(bound(v) is better than best)     // Check if node is still promising
            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);
            }
    }
}
```

# Best-First Search with Branch-and-Bound Pruning

- Algorithm 6.2 : The Best-First-Search with Branch-and-Bound Pruning Algorithm for the 0-1 Knapsack Problem

```
struct node {
    int level;
    int profit;
    int weight;
    float bound;
}
```

```
void knapsack3(int n, const int p[], cont int w[], int W, int
  &maxprofit){
    priority queue_of_node  PQ;  node u, v;

    initialize(PQ);
    v.level =0;  v.profit = 0;  v.weight = 0;  v.bound = bound(v);
    maxprofit = 0;
    insert (PQ, v);
```

# Best-First Search with Branch-and-Bound Pruning

```
    while (!empty(Q)) {                         // Remove node with
        remove(PQ, v);                          // best bound
        if(v.bound > maxprofit) {               // Check if node is still
            u.level = v.level+1;                // promising.
            u.profit = v.profit + p[u.level];
            u.weight = v.weight + w[u.level];
            if ((u.weight <= W) && (u.profit > maxprofit))
                maxprofit = u.profit;

            u.bound = bound(u);
            if (u.bound > maxprofit) insert (PQ, u);

            u.profit = v.profit;        // Set u to the child
            u.weight = v.weight;        // that does not include
            u.bound = bound(u);         // the next item.
            if (u.bound > maxprofit) insert (PQ, u);
        }
    }
}
```

# The Traveling Salesperson Problem

- The Traveling Salesperson Problem (Review)
  - 외판원의 집이 위치하고 있는 도시에서 출발하여 다른 도시들을 각각 한번씩만 방문하고, 다시 집으로 돌아오는 가장 짧은 일주여행경로(tour)를 결정하는 문제.
  - 이 문제는 음이 아닌 가중치가 있는, 방향성 그래프로 나타낼 수 있다.
  - 일주여행경로(tour, Hamiltonian circuits)는 한 정점을 출발하여 다른 모든 정점을 한번씩만 거쳐서 다시 그 정점으로 돌아오는 경로이다.
  - 여러 개의 일주여행경로 중에서 길이가 최소가 되는 경로, 즉 최적일주여행경로(optimal tour)를 찾는 것이 외판원문제(TSP)
  - 무작정 알고리즘:
    - 가능한 모든 일주여행경로를 다 고려한 후, 그 중에서 가장 짧은 일주여행경로를 선택한다.
    - 가능한 일주여행경로의 총 개수는 $(n-1)!$

# The Traveling Salesperson Problem

- 동적계획법을 이용한 접근방법 (Review)
  - $V$는 모든 정점의 집합이고, $A$는 V의 부분집합이라고 하자.
    그리고 $D[v_i][A]$는 $A$에 속한 각 정점을 정확히 한번씩 만 거쳐서
    $v_i$에서 $v_1$로 가는 최단경로의 길이라고 하자.
    그러면 위의 예제에서 $D[v_2][\{v_3, v_4\}]$의 값은?
  - 최적 일주여행경로의 길이:

$$D[v_1][V-\{v_1\}] = min_{2\le j\le n}(W[1][j] + D[v_j][V-\{v_1, v_j\}])$$

일반적으로 표시하면 $i \neq 1$이고, $v_i$가 $A$에 속하지 않을 때, 다음과 같이 된다.

$$D[v_i][A] = min_{v_j \in A}(W[i][j] + D[v_j][A-\{v_j\}]) \text{ if } A \neq \phi$$

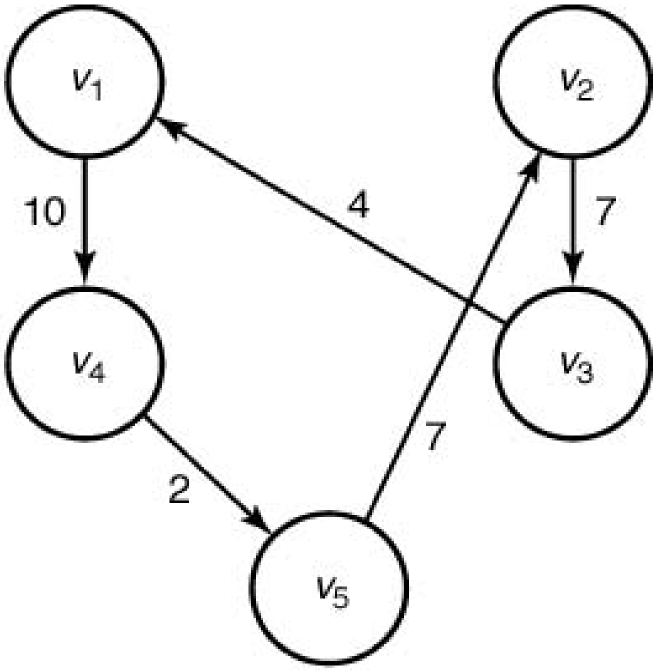
$$D[v_i][\phi] = W[i][1]$$

# The Traveling Salesperson Problem

- 외판원문제: 분기한정법
  - $n$ = 40일 때, 동적계획법 알고리즘은 6년 이상이 걸린다. ($\Theta(n^2 2^n)$) 그러므로 분기한정법을 시도해 본다.

**Ex)** 인풋그래프 및 인접행렬

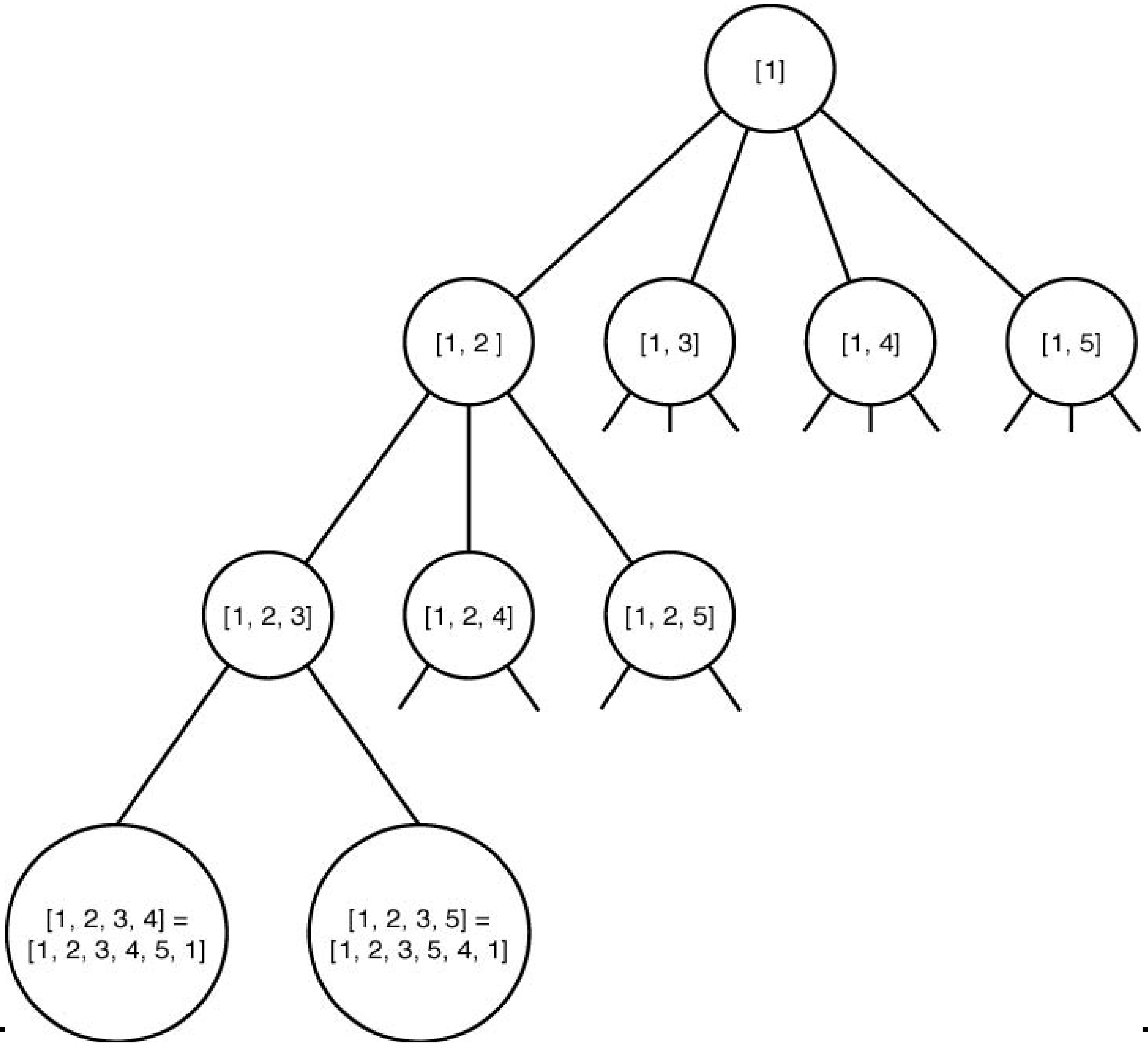
$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$



# The Traveling Salesperson Problem

- 상태공간트리 구축방법
  - 각 마디는 출발정점으로부터의 여행경로를 나타낸다.
  - 뿌리마디의 여행 경로는 [1]이 되고,
    뿌리마디에서 뻗어 나가는 "수준 1"에 있는 여행경로는
    각각 [1,2], [1,3], ..., [1,5]가 되고,
    마디 [1,2]에서 뻗어 나가는 "수준 2"에 있는 여행경로는
    각각 [1,2,3],...,[1,2,5]가 되고, 이런 식으로 뻗어 나가서
    잎마디에 도달하게 되면 완전한 일주여행경로를 가지게 된다.
  - 마디에 저장되어 있는 정점이 4개가 되면 (즉, 수준 n-2가 되면)
    더 이상 자식마디로 뻗어 나갈 필요가 없다.
    왜냐하면, 자식마디는 하나뿐이기 때문이다.

# The Traveling Salesperson Problem



# The Traveling Salesperson Problem

- 분기한정 최고우선검색
  - 분기한정 가지치기로 최고우선 검색을 사용하기 위해서 각 마디의 한계치를 구한다.
  - 이 문제에서는 주어진 마디에서 뻗어 나가서 얻을 수 있는 여행 경로 길이의 하한(최소치)을 구하여 한계치로 한다. 각 마디를 검색할 때 최소여행경로의 길이보다 한계치가 작은 경우 이 마디를 유망하다고 한다.
  - $[1,\ldots,k]$의 여행경로를 가진 마디의 한계치:

    Let $A = V - [1,\ldots,k]$ .

    Bound

    = 경로 $[1,\ldots,k]$의 길이

    \+ ($v_k$ 로부터 $A$에 속한 정점으로 가는 이음선 길이 중 최소치)

    \+ $\Sigma_{i \in A}$($v_i$ 로부터 $\mathbf{A} - \{\mathbf{v_i}, \mathbf{v_k}\} \cup \{\mathbf{v_1}\}$ 에 속한 정점으로 가는 이음선 길이 중 최소치)

# The Traveling Salesperson Problem

- Compute the lower bounds

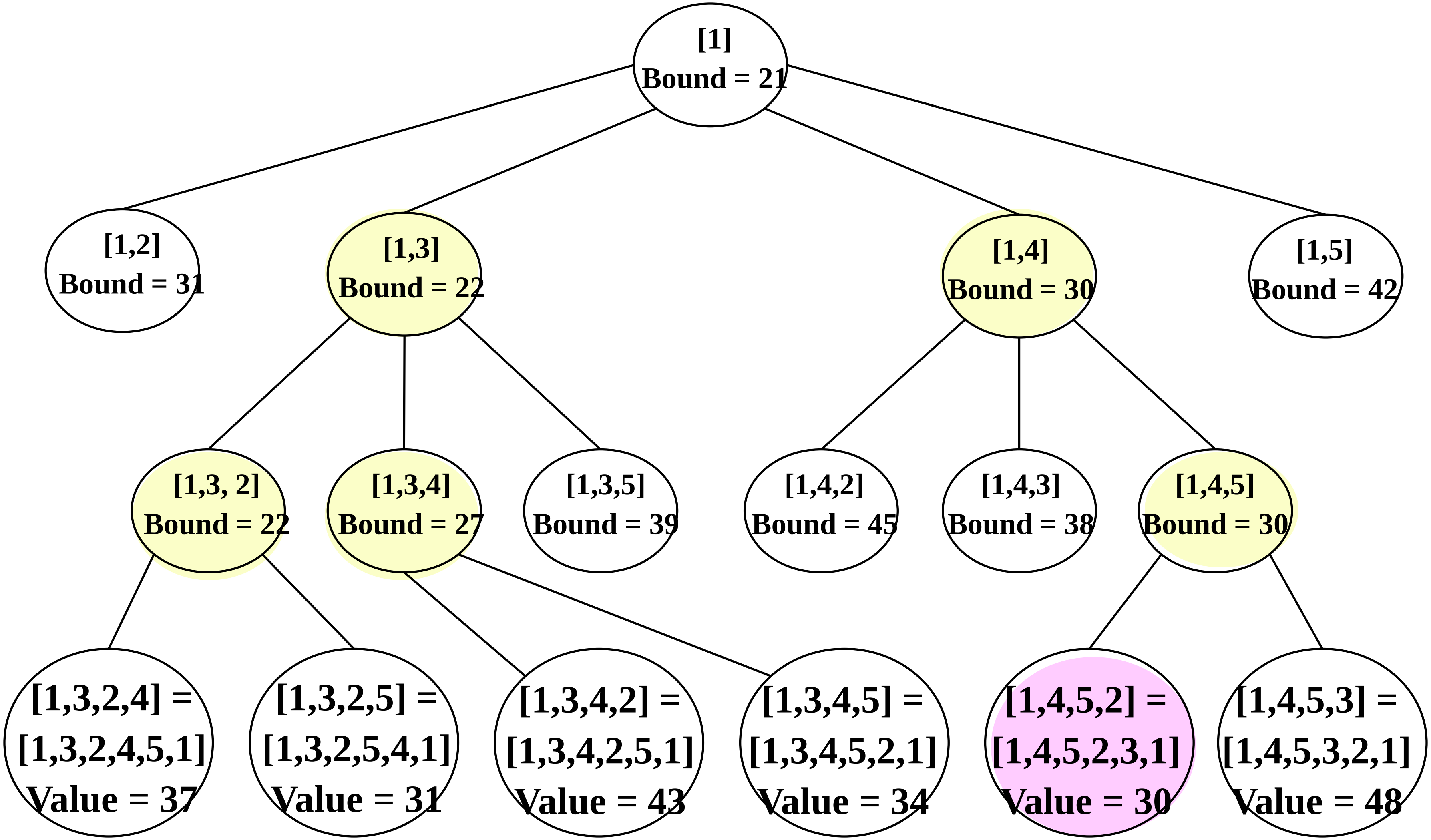
- Lower bounds on the cost of leaving 5 vertices
  - $v_1$ min(14, 4, 10, 20) = 4
  - $v_2$ min(14, 7, 8, 7) = 7
  - $v_3$ min(4, 5, 7, 16) = 4
  - $v_4$ min(11, 7, 9, 2) = 2
  - $v_5$ min(18, 7, 17, 4) = 4
  - 4 + 7 + 4 + 2 + 4 = 21

- Suppose we have visited [1,2]

  Make $v_2$ as a second vertex on the tour
  - $[v_1, v_2]$ = 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
  - 14 + 7 + 4 + 2 + 4 = 31

# The Traveling Salesperson Problem

Ex 6.3 분기한정 가지치기로 최고우선검색을 하여 상태공간트리 구축



# The Traveling Salesperson Problem

- Algorithm 6.3 : The Best-First-Search with Branch-and-Bound Pruning Algorithm for the Traveling Salesperson Problem

  - Problem : Algorithm 3.11 과 동일
  - Inputs : Algorithm 3.11 과 동일
  - Outputs : minlength (the length of optimal tour ), opttour (the path of optimal tour )

# The Traveling Salesperson Problem

```
struct node {
    int level;
    ordered_set path;
    number bound;
}
```

```
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 = INFINITE;
    insert(PQ, v);
```

# The Traveling Salesperson Problem

```
while (!empty(PQ)) {
   remove(PQ, v);
   if (v.bound < minlength) {
       u.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 the only index 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);
           }
       }
   }
}
```

# The Traveling Salesperson Problem

- 분석
  - 이 알고리즘은 방문하는 마디의 개수가 더 적다.
  - 그러나 아직도 알고리즘의 최악 시간복잡도는 변하지 않는다.
    즉, $n = 40$이 되면 문제를 풀 수 없는 것과 다름없다고 할 수 있다.

  - 다른 방법이 있을까?
    - 근사(approximation) 알고리즘
      - 최적의 해답을 준다는 보장은 없지만,
        최적의 해답에 근사한 답을 주는 알고리즘