

Chapter 6.

Branch-and-Bound

Foundations of Algorithms, 5th Ed.

Richard E. Neapolitan



6.1 The 0-1 Knapsack Problem

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

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

6.2 The Traveling Salesperson Problem



- The Branch-and-Bound algorithm
 - is an improvement on the backtracking algorithm.
 - The design strategy of the branch-and-bound algorithm:
 - uses a state space tree is used to solve a problem.
 - *does not limit* us to any particular *way of traversing* the tree
 - is used for *only for optimization problem*.
 - In backtracking, a depth-first-search is used to traverse a state space tree.
 - In branch-and-bound,
 - a *breadth-first-search* is used to traverse the tree,
 - and a *best-first-search* can be used, more efficiently.



■ The Breadth-First-Search:

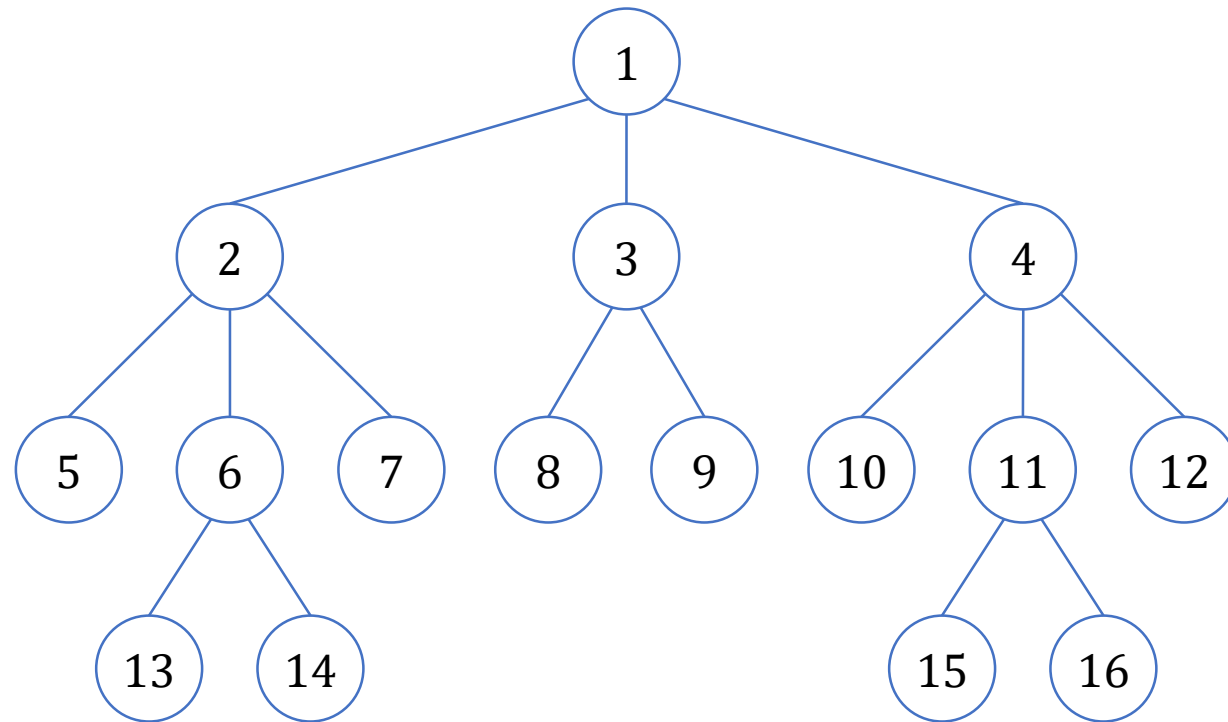
- We can implement the breadth-first-search using a *queue*.

```
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);  
        }  
    }  
}
```



The Branch-and-Bound Strategy

5





6.1 Branch-and-Bound for the 0-1 Knapsack Problem

- Solving the 0-1 Knapsack with the **Branch-and-Bound Pruning**:
 - The backtracking is also actually a branch-and-bound algorithm.
 - However, we can *compare* the *bounds* of *promising nodes*
 - and *visit* the children of the one with the *best bound*.
 - In this way, we often can *arrive* at an *optimal* solution
 - *faster than* we would by visiting in some predefined order (**DFS**).
 - This approach is called
 - ***best-first-search with branch-and-bound pruning***.
 - The implementation of this approach is a simple modification of
 - *breadth-first-search with branch-and-bound pruning*.

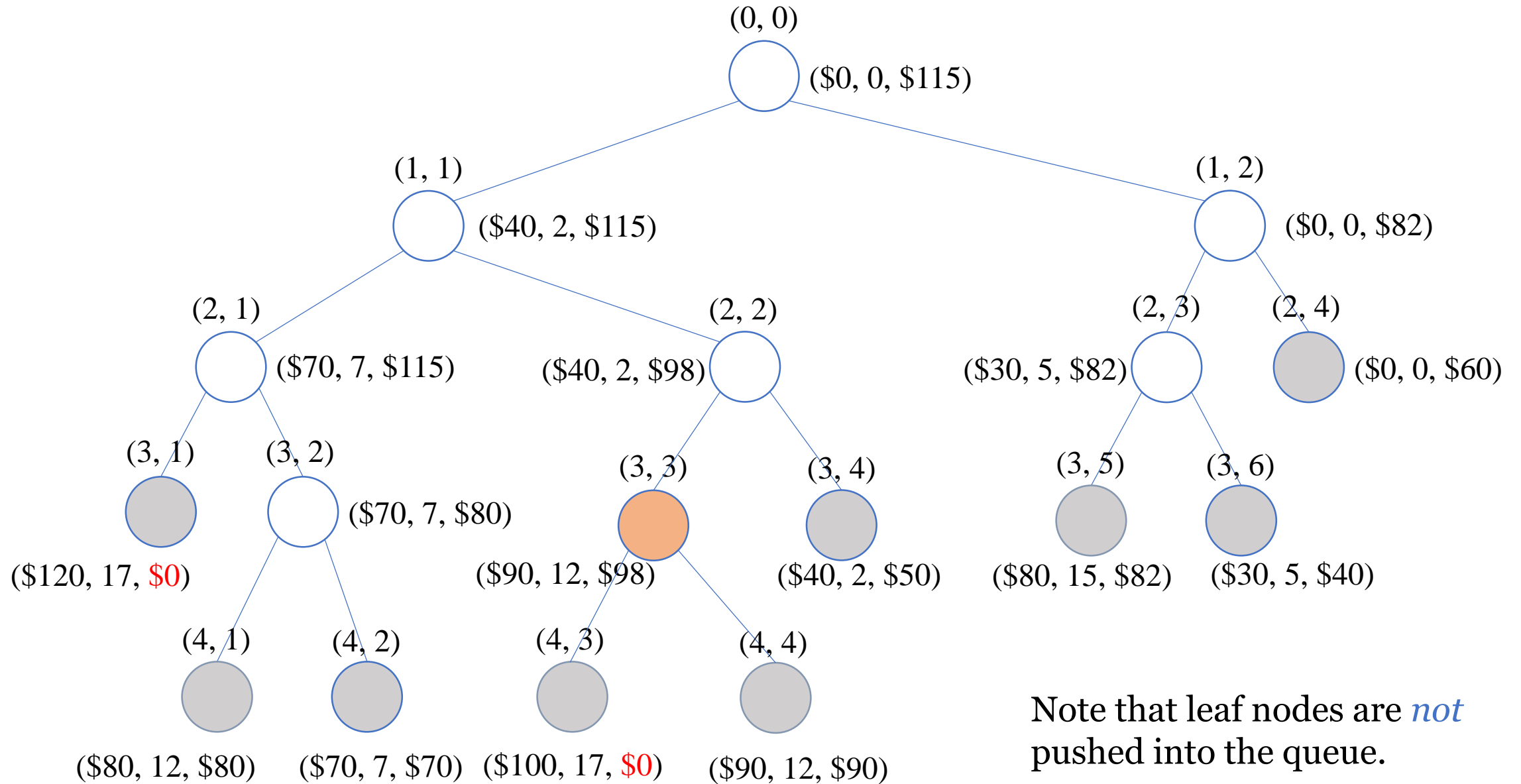


6.1 Branch-and-Bound for the 0-1 Knapsack Problem

- **Breadth-First-Search (BFS)** with Branch-and-Bound Pruning:
 - An illustrative example:
 - $n = 4, W = 16$
 - $p_i = [40, 30, 50, 10]$
 - $w_i = [2, 5, 10, 5]$
 - $\frac{p_i}{w_i} = [20, 6, 5, 2]$: already ordered in nondecreasing order.
 - We proceed exactly as we did using backtracking
 - except that we do a breadth-first-search instead of a depth-first-search.



6.1 Branch-and-Bound for the 0-1 Knapsack Problem





6.1 Branch-and-Bound for the 0-1 Knapsack Problem

9

- A general algorithm for BFS with branch-and-bound pruning:

```
void breadth_first_branch_and_bound(state_space_tree T, int &best) {  
    queue_of_node Q;  
    node u, v;  
  
    initialize(Q);  
    v = root of T;  
    enqueue(Q, v);  
    best = value(v);  
    while (!empty(Q)) {  
        dequeue(Q, v);  
        for (each child u of v) {  
            if (value(u) is better than best)  
                best = value(u);  
            if (bound(u) is better than best)  
                enqueue(Q, u);  
        }  
    }  
}
```



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

- Applying the strategy to the 0-1 Knapsack Problem:
 - Notice that two nodes (3, 1) and (4, 3) have bounds of \$0.
 - Unlike the promising function in backtracking,
 - *bound* function should return an integer.
 - Because we do not have the benefit of recursion,
 - we need to store all the information pertinent to a node at that node.

```
typedef struct node *node_pointer;
typedef struct node {
    int level;    // the node's level in the state space tree
    int weight;
    int profit;
} nodetype;

typedef queue<node_pointer> queue_of_node;
```



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

ALGORITHM 6.1: The BFS with Branch-and-Bound Pruning for the 0-1 Knapsack Problem

```
void knapsack5() {
    queue_of_node Q; node_pointer u, v;

    maxprofit = 0;
    Q.enqueue(create_node(0, 0, 0));
    while (!Q.empty()) {
        v = Q.dequeue();
        u = new_node(v->level + 1,
                    v->weight + w[v->level + 1],
                    v->profit + p[v->level + 1]);
        if (u->weight <= W && u->profit > maxprofit)
            maxprofit = u->profit;
        if (bound(u) > maxprofit)
            Q.enqueue(u);
        u = new_node(v->level + 1, v->weight, v->profit);
        if (bound(u) > maxprofit)
            Q.enqueue(u);
    }
}
```



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

ALGORITHM 6.1: The BFS with Branch-and-Bound Pruning for the 0-1 Knapsack Problem

```
float bound(node_pointer u) {
    int j, k, 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 += w[j];
            result += p[j];
            j++;
        }
        k = j;
        if (k <= n)
            result += (W - totweight) * ((float)p[k] / w[k]);
        return result;
    }
}
```



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

- ***Best-First-Search*** with Branch-and-Bound Pruning:
 - Note that the breadth-first-search strategy
 - has *no advantage over a depth-first-search*.
 - For example,
 - there are only **13 nodes** in the state space tree of backtracking.
 - whereas there are **17 nodes** in the state space tree produced by
 - the breadth-first-search with brand-and-bound pruning.
 - However, we can *improve* our search by using our bound
 - to do more than just determine whether a node is promising.
 - After visiting all the children of a given node,
 - *the node with the best bound* can be expanded *first*.



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

- $n = 4, W = 16$
- $p_i = [40, 30, 50, 10]$
- $w_i = [2, 5, 10, 5]$
- $\frac{p_i}{w_i} = [20, 6, 5, 2]$

$maxprofit = \$0$

$profit = \$0, weight = 0$

$bound = \$115$

Node (0, 0) is *inserted* into the PQ.

PQ

(0, 0)

(\$0, 0, \$115)



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

(0, 0)

maxprofit = \$40

○ (\$0, 0, \$115)

1. Node (0, 0) is *removed* from the PQ.

- *profit* = \$0, *weight* = 0, *bound* = \$115
- *maxprofit* = \$40

2. *Visit* node (1, 1).

- *profit* = \$40, *weight* = 2
- *bound* = \$115
- Insert (1, 1) into the PQ.

3. *Visit* node (1, 2).

- *profit* = \$0, *weight* = 0
- *bound* = \$82
- Insert(1, 2) into the PQ.

4. Determine *promising, unexpanded node* with the *greatest bound*.

- Because node (1, 1) has the greatest bound,
 - we visit node (1, 1) which is removed from the PQ next.

PQ

(1, 1)

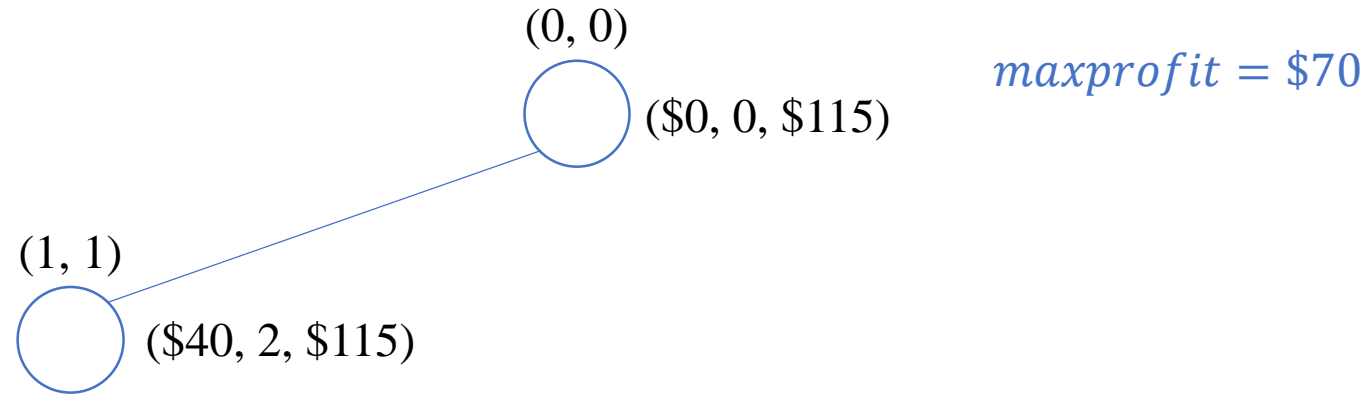
(\$40, 2, \$115)

(1, 2)

(\$0, 0, \$82)



6.1 Branch-and-Bound for the 0-1 Knapsack Problem



5. Node (1, 1) is *removed* from the PQ.

- *profit = \$40, weight = 2, bound = \$115*
- *maxprofit = \$70*

6. *Visit* node (2, 1).

- *profit = \$70, weight = 7*
- *bound = \$115*
- Insert (2, 1) into the PQ.

7. *Visit* node (2, 2).

- *profit = \$40, weight = 2*
- *bound = \$98*
- Insert (2, 2) into the PQ.

PQ

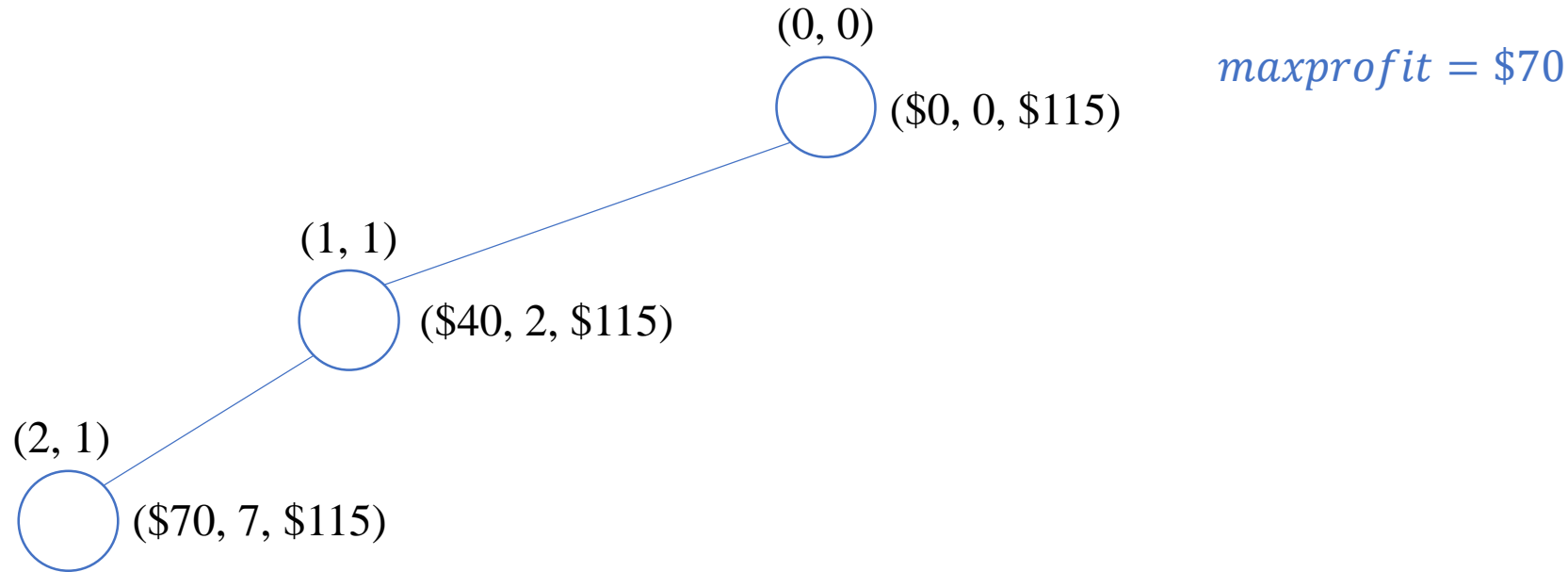
(2, 1)
(\$70, 7, **\$115**)

(2, 2)
(\$40, 2, \$98)

(1, 2)
(\$0, 0, \$82)



6.1 Branch-and-Bound for the 0-1 Knapsack Problem



8. Node (2, 1) is *removed* from the PQ.

- *profit = \$70, weight = 7, bound = \$115*

9. *Visit* node (3, 1).

- *profit = \$120, weight = 17*
- *bound = \$0*
- **Do NOT insert** (3, 1) into the PQ.

10. *Visit* node (3, 2).

- *profit = \$70, weight = 7*
- *bound = \$80*
- **Insert** (3, 2) into the PQ.

PQ

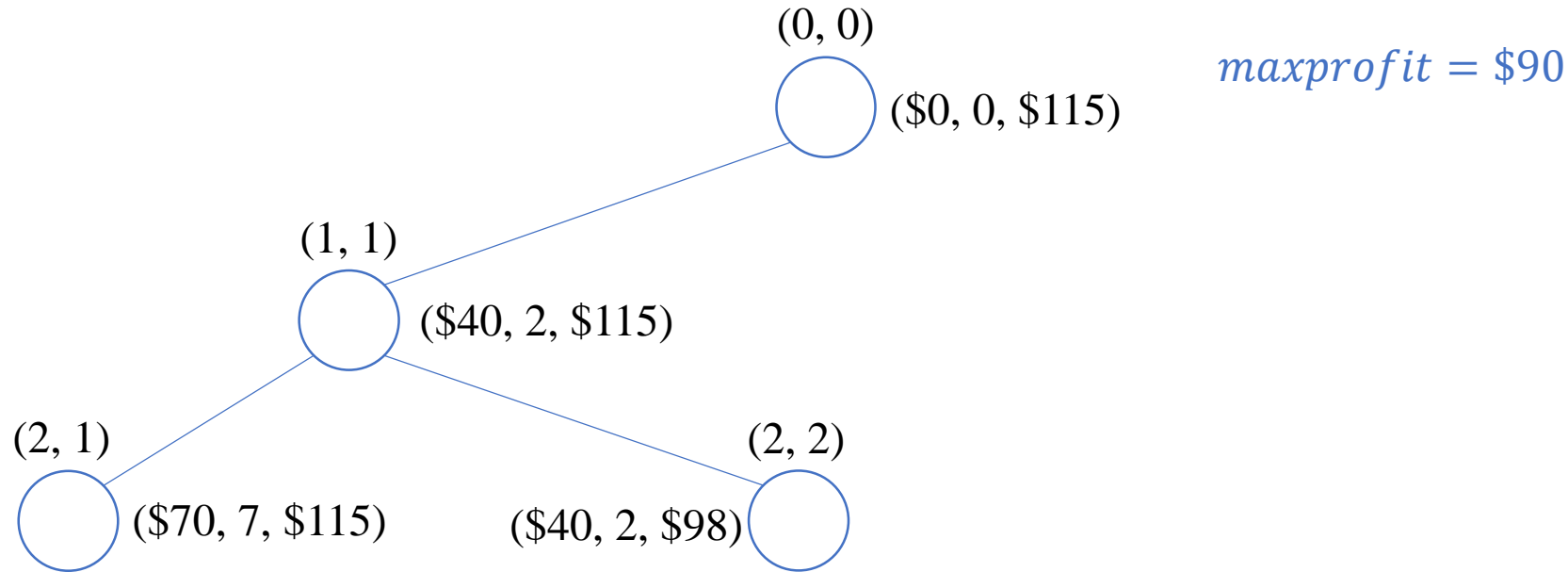
(2, 2)
(\$40, 2, \$98)

(1, 2)
(\$0, 0, \$82)

(3, 2)
(\$70, 7, \$80)



6.1 Branch-and-Bound for the 0-1 Knapsack Problem



11. Node (2, 2) is *removed* from the PQ.

- profit = \$40, weight = 2, bound = \$98
- *maxprofit = \$90*

12. Visit node (3, 3).

- profit = \$90, weight = 12
- bound = \$98
- Insert (3, 3) into the PQ.

13. Visit node (3, 4).

- profit = \$74, weight = 2
- bound = \$50
- Do NOT insert (3, 4) into the PQ.

PQ

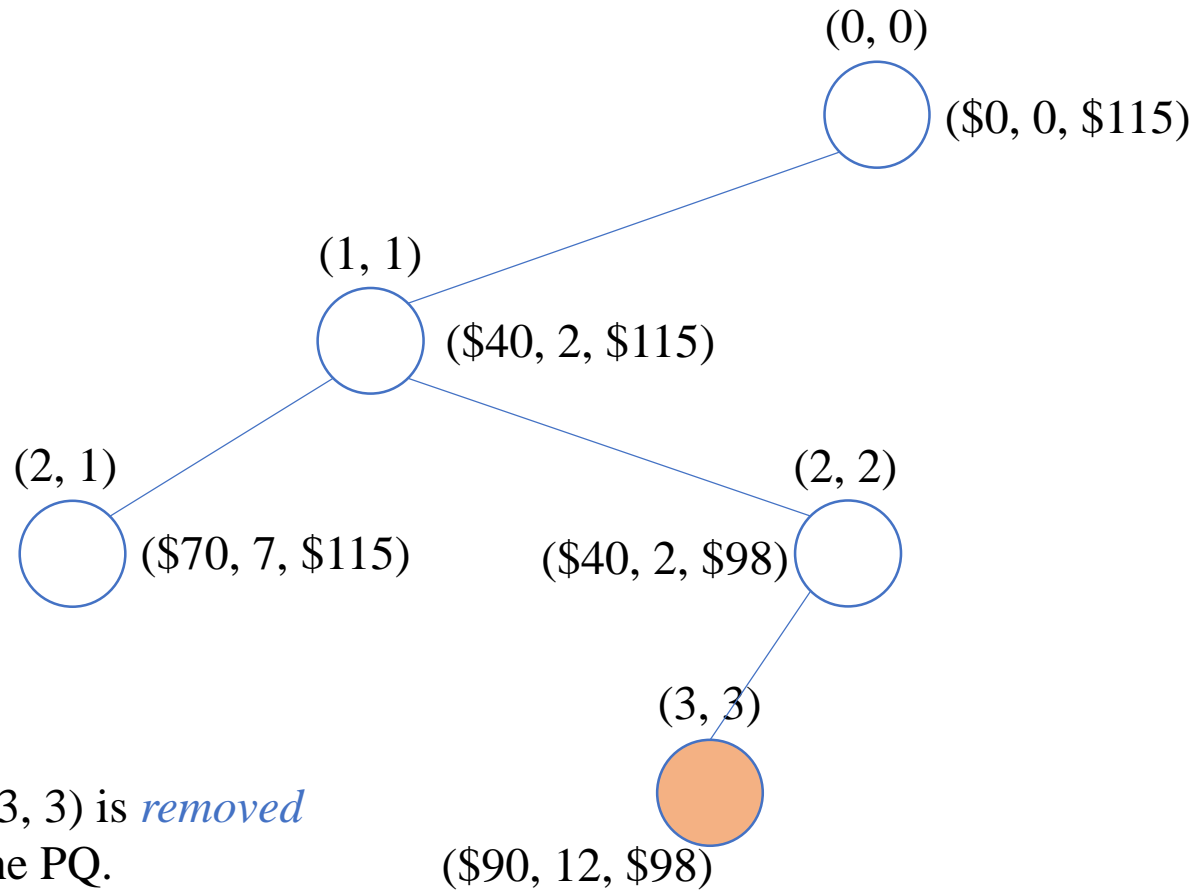
(3, 3)
(\$90, 12, \$98)

(1, 2)
(\$0, 0, \$82)

(3, 2)
(\$70, 7, \$80)



6.1 Branch-and-Bound for the 0-1 Knapsack Problem



14. Node (3, 3) is *removed* from the PQ.

15. Visit node (4, 1).

- \$100, 17, **\$0**
- Do NOT Insert (4, 1) into the PQ.

16. Visit node (4, 2).

- \$90, 12, **\$90**
- Do NOT insert (4, 2) into the PQ.

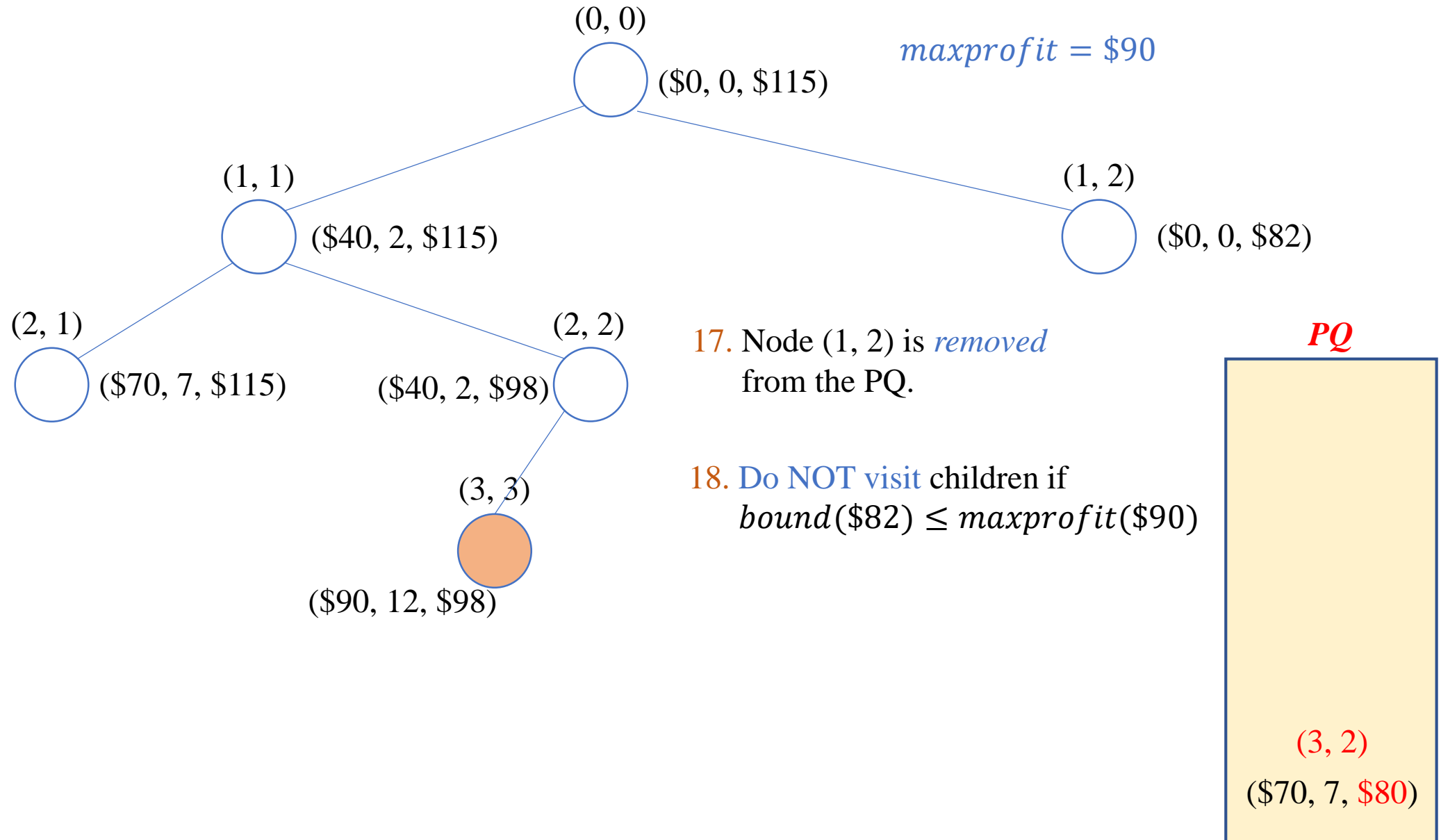
PQ

(1, 2)
(\$0, 0, **\$82**)

(3, 2)
(\$70, 7, \$80)

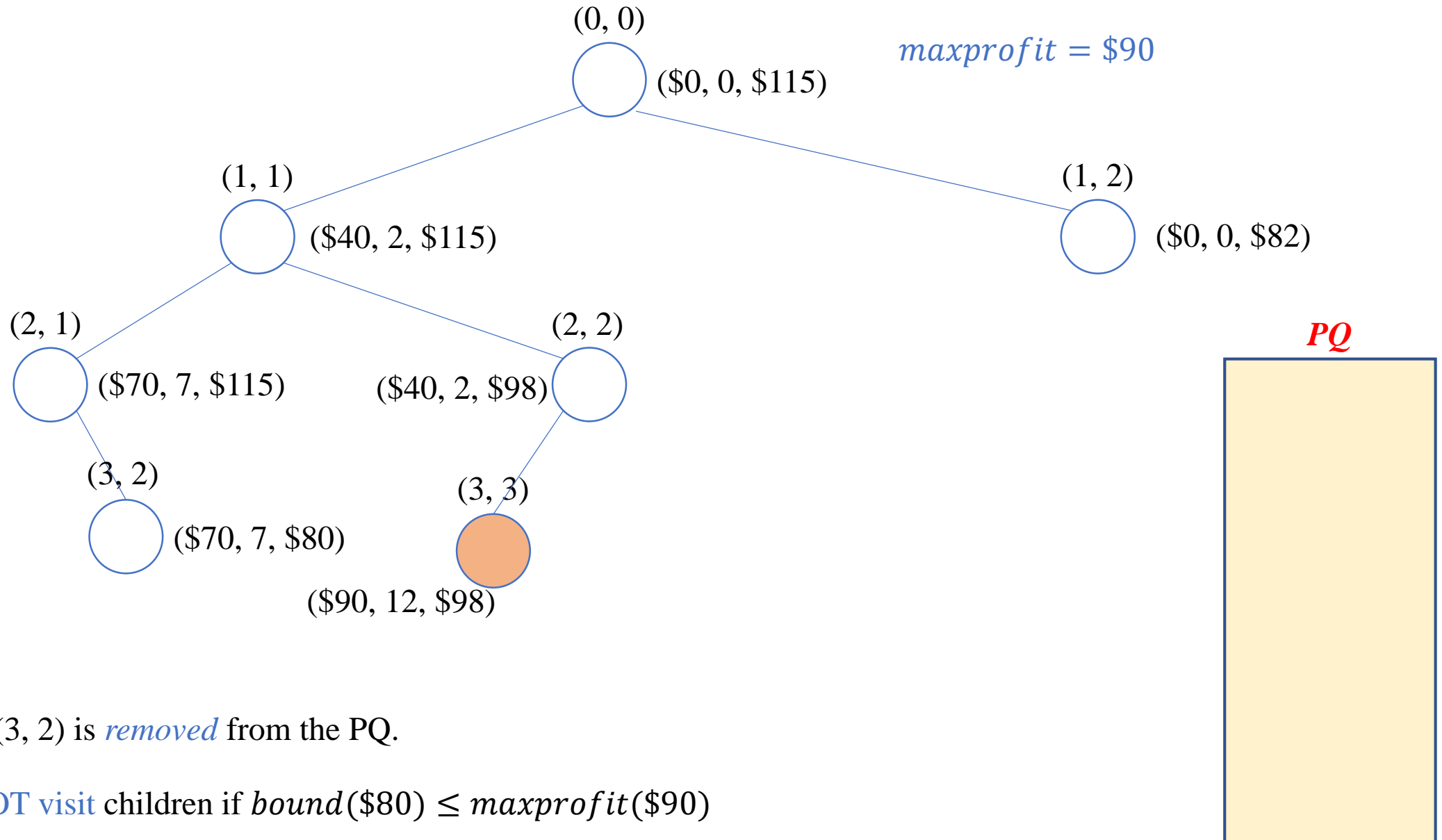


6.1 Branch-and-Bound for the 0-1 Knapsack Problem





6.1 Branch-and-Bound for the 0-1 Knapsack Problem

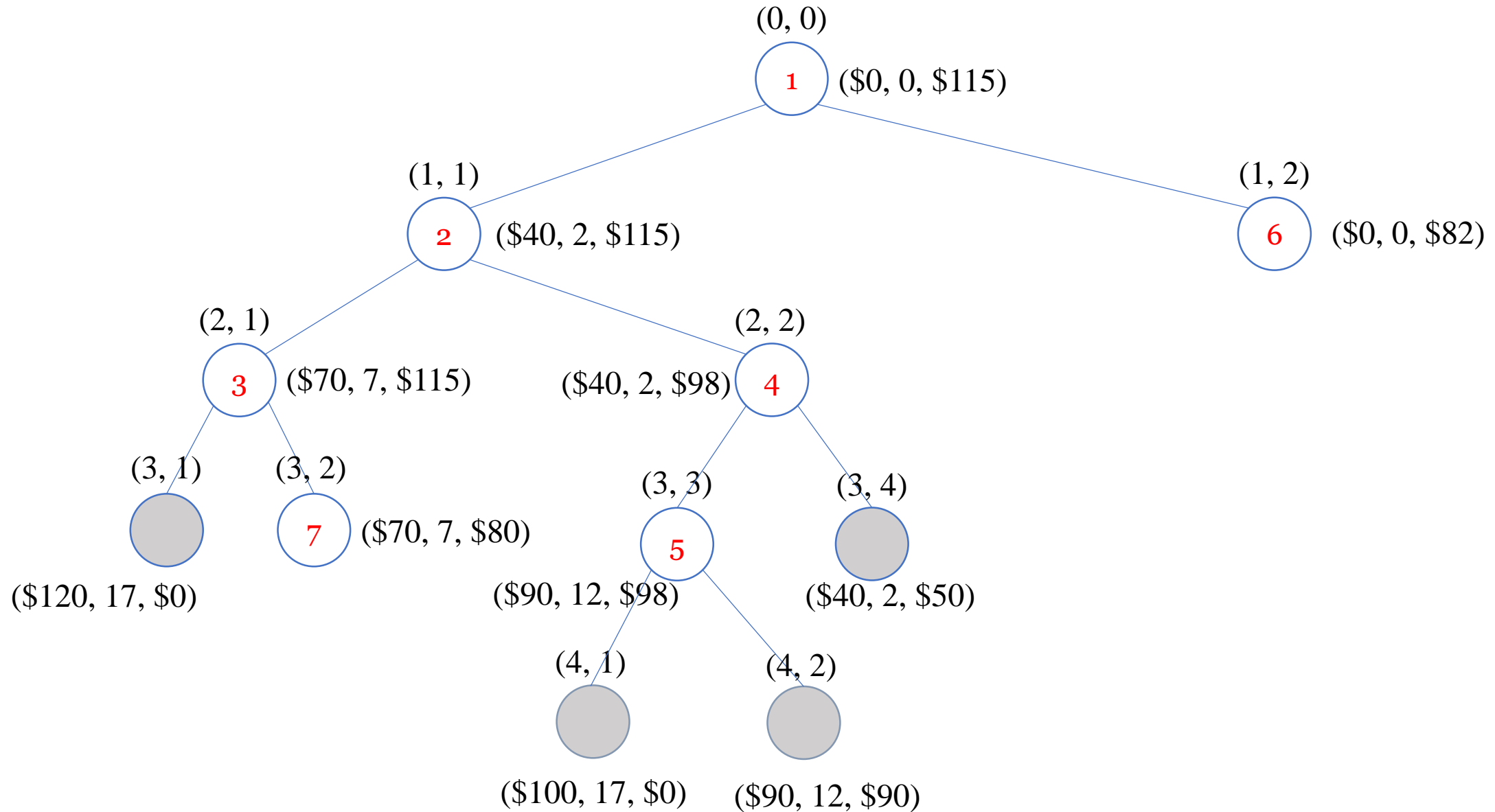


19. Node (3, 2) is *removed* from the PQ.

20. Do NOT visit children if $\text{bound}(\$80) \leq \text{maxprofit}(\$90)$

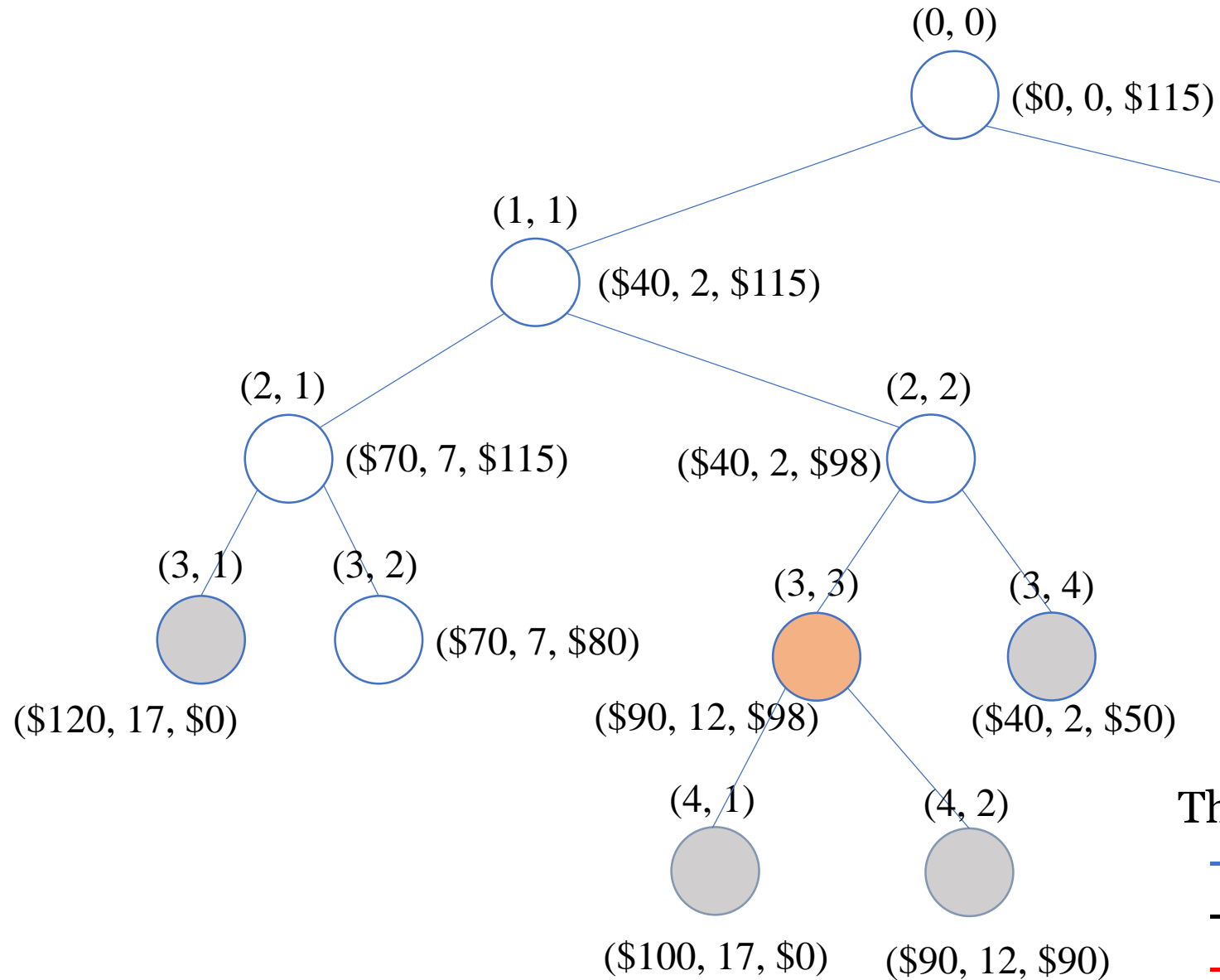


6.1 Branch-and-Bound for the 0-1 Knapsack Problem





6.1 Branch-and-Bound for the 0-1 Knapsack Problem



Note that shaded nodes are *not* inserted into the priority queue, *but* visited to see if it is promising.

The number of nodes in state space tree:

- *depth-first*: 13 nodes
- breadth-first: 17 nodes
- *best-first*: 11 nodes.



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

- The implementation of Best-First-Search:
 - Note that there is no guarantee that
 - the node that appears to be best will actually lead to an optimal solution.
 - ex) (2, 1) is better than (2, 2), but (2, 2) leads to the optimal solution.
 - In general, best-first-search
 - still ends up creating *all of the state space tree* for some instances.
 - Best-first-search is a simple modification to breadth-first-search.
 - Instead of using a *queue*, use a *priority queue*.



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

- A general algorithm for the *best-first-search*:

```
void best_first_branch_and_bound(state_space_tree T, int &best) {
    priority_queue_of_node PQ;
    node u, v;

    initialize(PQ);
    v = root of T;
    best = value(v);
    insert(PQ, v);
    while (!empty(PQ)) {
        remove(PQ, v);
        if (bound(v) is better than best)
            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);
            }
    }
}
```



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

- Applying the strategy to the 0-1 Knapsack Problem:
 - Notice that a node is promising at the time when we insert it into the PQ.
 - However, it can be nonpromising when it is removed from the PQ.
 - So, compare its bound with *maxprofit* after it is removed from the PQ.
 - Because we need the bound for a node
 - at insertion time, at removal time, and to order the nodes in the PQ,
 - we store the bound at the node.

```
typedef struct node *node_pointer;
typedef struct node {
    int level;    // the node's level in the state space tree
    int weight;
    int profit;
    float bound;
} nodetype;
```



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

ALGORITHM 6.2: The Best-First-Search with B&B Pruning for the 0-1 Knapsack Problem

```
void knapsack6() {
    priority_queue_of_node PQ; node_pointer u, v;
    maxprofit = 0;
    PQ.insert(create_node(0, 0, 0));
    while (!PQ.empty()) {
        v = PQ.remove();
        if (v->bound > maxprofit) {
            u = create_node(v->level + 1,
                           v->weight + w[v->level + 1],
                           v->profit + p[v->level + 1]);
            if (u->weight <= W && u->profit > maxprofit)
                maxprofit = u->profit;
            if (u->bound > maxprofit)
                PQ.insert(u);
            u = create_node(v->level + 1, v->weight, v->profit);
            if (u->bound > maxprofit)
                PQ.insert(u);
        }
    }
}
```



6.1 Branch-and-Bound for the 0-1 Knapsack Problem

```
typedef priority_queue<node_pointer, vector<node_pointer>, compare>
    priority_queue_of_node;

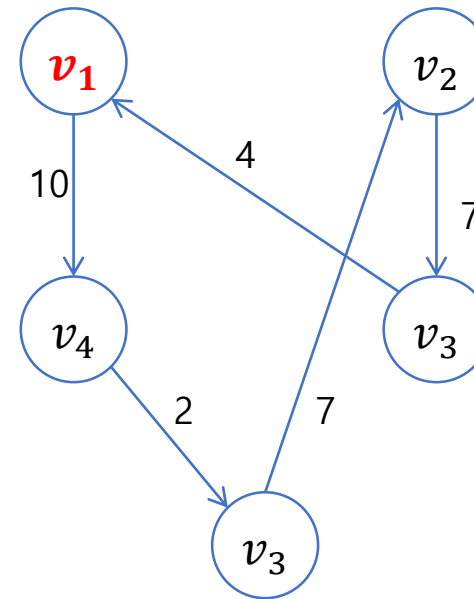
struct compare {
    bool operator()(node_pointer u, node_pointer v) {
        if (u->bound < v->bound)
            return true;
        return false;
    }
};

node_pointer create_node(int level, int weight, int profit) {
    node_pointer v = (node_pointer)malloc(sizeof(nodetype));
    v->level = level;
    v->weight = weight;
    v->profit = profit;
    v->bound = bound(v);
    return v;
}
```

6.2 The Traveling Salesperson Problem

- The Traveling Salesperson Problem Revisited:
 - The goal of this problem is to find an *optimal tour*, that is,
 - the *shortest path* in a directed graph that starts at a given vertex,
 - visits each vertex *exactly once*, and ends up back at the starting vertex.

<i>W</i>	1	2	3	4	5
1	0	14	4	10	20
2	14	0	7	8	7
3	4	5	0	7	16
4	11	7	9	0	2
5	18	7	17	4	0

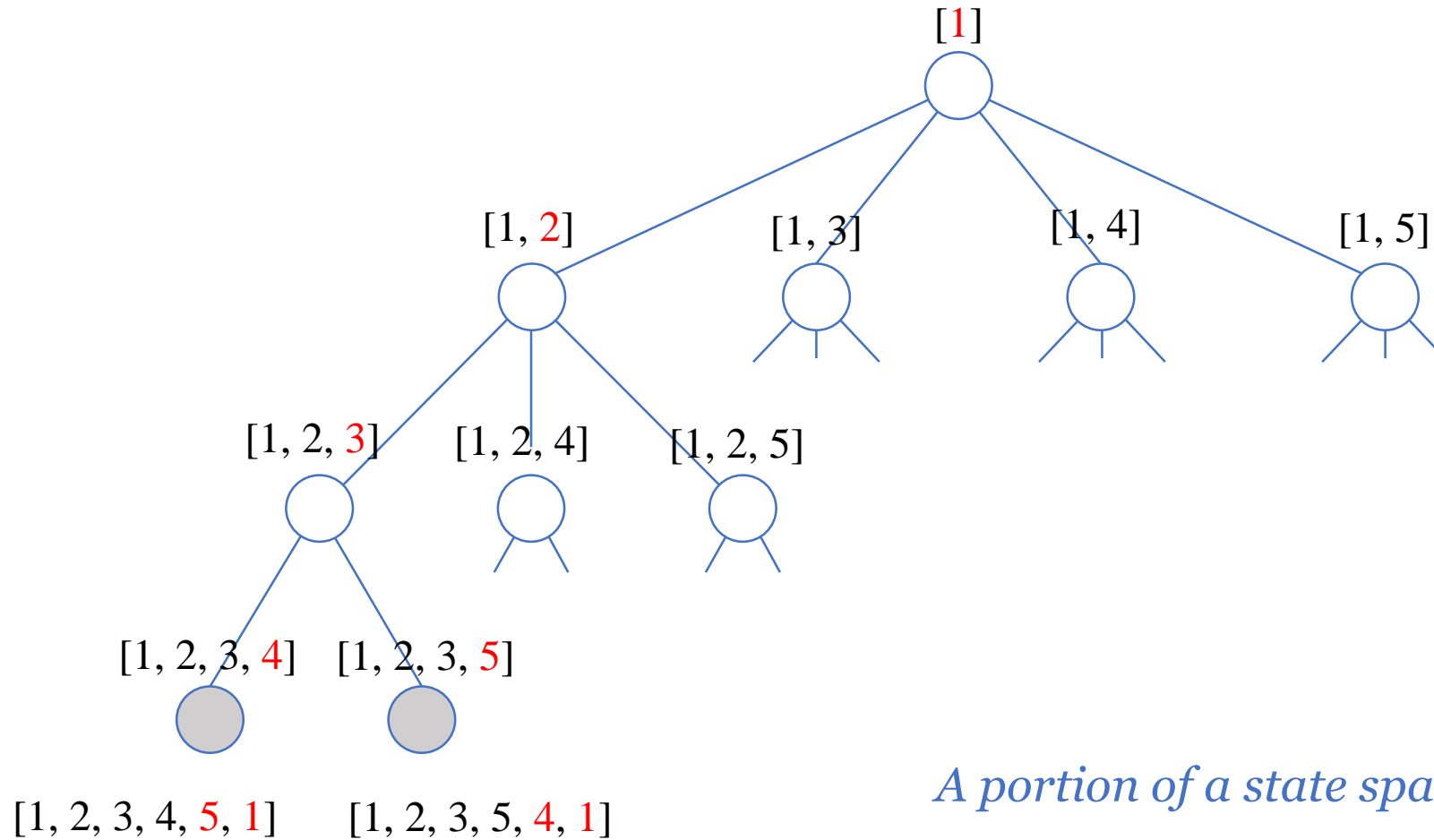


*an optimal tour
starting at v_1 .
(length = 30)*

6.2 The Traveling Salesperson Problem

- The *State Space Tree* for the TSP:
 - An obvious state space tree is one in which
 - each vertex other than the starting vertex
 - is tried as the *first vertex*, after the starting one, *at level 1*.
 - each vertex other than the *starting vertex* and the *first vertex*
 - is tried as the *second vertex at level 2*, and so on.
 - Then, a node represents
 - a *path* chosen up to that node starting from the root node.
 - We stop expanding the tree
 - when there are $n - 1$ vertices in the path stored at a node
 - because, at that time, the *nth vertex* is *uniquely* determined.

6.2 The Traveling Salesperson Problem



6.2 The Traveling Salesperson Problem

- Determining a *bound* for each node:
 - Determine a *lower bound* on the length of any tour
 - that can be obtained by expanding beyond a given node.
 - Then, the node is *promising*
 - only if its bound is *less* than the *current minimum tour length*.
 - We can obtain a *bound for a node*:
 - In any tour, the *length* of the *edge taken* when leaving a vertex
 - must be *at least as great as* the *length* of the *shortest edge*
 - emanating from the vertex.

6.2 The Traveling Salesperson Problem

- Determining a *bound* for each node:
 - Let *cost* be the length of the edge taken.
 - A *lower bound* on the *cost of leaving* vertex v_1 is given by
 - the *minimum* of all the nonzero entries in *row 1* of the adjacency matrix.

<i>W</i>	1	2	3	4	5
1	0	14	4	10	20
2	14	0	7	8	7
3	4	5	0	7	16
4	11	7	9	0	2
5	18	7	17	4	0

<i>W</i>	1	2	3	4	5	
1	0	14	4	10	20	
2	14	0	7	8	7	• $v_2: 7$
3	4	5	0	7	16	• $v_3: 4$
4	11	7	9	0	2	• $v_2: 2$
5	18	7	17	4	0	• $v_2: 4$

- the *lower bound* leaving v_1 : 4

6.2 The Traveling Salesperson Problem


- Determining a *bound* for each node:
 - Because a tour must leave every vertex exactly once,
 - a lower bound on the length of a tour is the sum of these *minimums*.

<i>W</i>	1	2	3	4	5	
1	0	14	4	10	20	• $v_1: 4$
2	14	0	7	8	7	• $v_2: 7$
3	4	5	0	7	16	• $v_3: 4$
4	11	7	9	0	2	• $v_2: 2$
5	18	7	17	4	0	• $v_2: 4$

- A *lower bound* on the length of a *tour* is
 - $4 + 7 + 4 + 2 + 4 = 21$.
- Note that this is not to say that
 - there is a tour with this length.
- Rather, it says that
 - there can be *no tour* with a *shorter length*.

6.2 The Traveling Salesperson Problem

- Determining a *bound* for each node:
 - Suppose we have visited the node $[1, 2]$.
 - The cost of getting to v_2 is the weight on the edge from v_1 to v_2 .
 - Therefore, any tour obtained by expanding beyond this node,
 - has the following lower bounds on the *cost* of leaving v_2 .

$[1, 2]$


W	1	2	3	4	5
1	0	14	4	10	20
2	14	0	7	8	7
3	4	5	0	7	16
4	11	7	9	0	2
5	18	7	17	4	0

- $v_1: 14$
- $v_2: 7$
- $v_3: 4$
- $v_2: 2$
- $v_2: 4$


- Do not include the edge to v_1 :
- v_2 cannot return to v_1 .
- Do not include the edge to v_2 :
- we already have been at v_2 .

- A lower bound obtained by expanding $[1, 2]$:
- $14 + 7 + 4 + 2 + 4 = 31$.

6.2 The Traveling Salesperson Problem

- Determining a *bound* for each node:
 - Suppose we have visited the node $[1, 2, 3]$.

$[1, 2, 3]$



<i>W</i>	1	2	3	4	5
1	0	14	4	10	20
2	14	0	7	8	7
3	4	5	0	7	16
4	11	7	9	0	2
5	18	7	17	4	0

• $v_1: 14$

• $v_2: 7$

• $v_3: 7$

• $v_2: 2$

• $v_2: 4$

- Do not include the edge to v_1 :
 - v_3 cannot return to v_1 .

- Do not include the edge to v_2 and v_3 :
 - we already have been at v_2 and v_3 .

- A lower bound obtained by expanding $[1, 2, 3]$:
 - $14 + 7 + 7 + 2 + 4 = 34$.



6.2 The Traveling Salesperson Problem

- Determining a *bound* for each node:
 - We will use the following data type in the algorithm.
 - The field *path* contains the partial tour stored at the node.
 - We need two functions *length* and *bound*.
 - *length* returns the length of the tour's *path*,
 - *bound* returns the bound for a node using the considerations discussed.

```
typedef vector<int> ordered_set;  
  
typedef struct node *node_pointer;  
typedef struct node {  
    int level;  
    ordered_set path;  
    int bound;  
} nodetype;
```

6.2 The Traveling Salesperson Problem

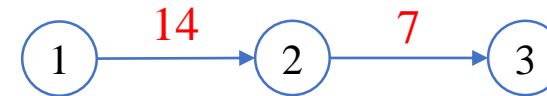
```

int length(ordered_set path) {
    vector<int>::iterator it;
    int len = 0;
    for (it = path.begin(); it != path.end(); it++)
        if (it != path.end() - 1)
            len += W[*it][*(it+1)];
    return len;
}

```

W	1	2	3	4	5
1	0	14	4	10	20
2	14	0	7	8	7
3	4	5	0	7	16
4	11	7	9	0	2
5	18	7	17	4	0

[1, 2, 3]



$$\text{length}([1, 2, 3]) = W[1][2] + W[2][3] = 21$$

6.2 The Traveling Salesperson Problem

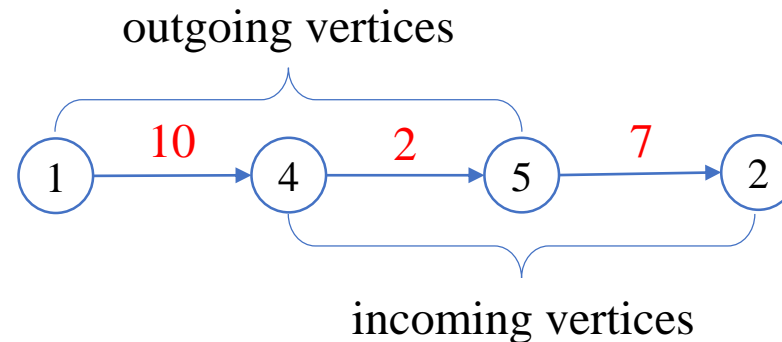
```
int bound(node_pointer v) {  
    // start from the length of path  
    int lower = length(v->path);  
    for (int i = 1; i <= n; i++) {  
        if (hasOutgoing(i, v->path)) continue;  
        int min = INF;  
        for (int j = 1; j <= n; j++) {  
            // Do not include self-loop  
            if (i == j) continue;  
            // Do not include an edge to which i cannot return  
            if (j == 1 && i == v->path[v->path.size() - 1]) continue;  
            // Do not include edges already in the path  
            if (hasIncoming(j, v->path)) continue;  
            // A lower bound (minimum) on the cost of leaving i  
            if (min > W[i][j]) min = W[i][j];  
        }  
        lower += min;  
    }  
    return lower;  
}
```

6.2 The Traveling Salesperson Problem

```
bool hasOutgoing(int v, ordered_set path) {
    vector<int>::iterator it;
    for (it = path.begin(); it != path.end() - 1; it++)
        if (*it == v) return true;
    return false;
}
```

```
bool hasIncoming(int v, ordered_set path) {
    vector<int>::iterator it;
    for (it = path.begin() + 1; it != path.end(); it++)
        if (*it == v) return true;
    return false;
}
```

[1, 4, 5, 2]



6.2 The Traveling Salesperson Problem

<i>W</i>	1	2	3	4	5
1	0	14	4	10	20
2	14	0	7	8	7
3	4	5	0	7	16
4	11	7	9	0	2
5	18	7	17	4	0

$$[1, 3, 2] \quad \text{length}([1, 3, 2]) = 4 + 5 = 9$$

$$\begin{aligned} \text{bound}([1, 3, 2]) &= \text{length}[1,3,2] + \min(v_2) + \min(v_4) + \min(v_5) \\ &= 9 + 7 + 2 + 4 = 22 \end{aligned}$$

$$[1, 3, 4] \quad \text{length}([1, 3, 4]) = 4 + 7 = 11$$

$$\begin{aligned} \text{bound}([1, 3, 4]) &= \text{length}[1,3,4] + \min(v_2) + \min(v_4) + \min(v_5) \\ &= 11 + 7 + 2 + 7 = 27 \end{aligned}$$

$$[1, 4, 5] \quad \text{length}([1, 4, 5]) = 10 + 2 = 12$$

$$\begin{aligned} \text{bound}([1, 4, 5]) &= \text{length}[1,4,5] + \min(v_2) + \min(v_3) + \min(v_5) \\ &= 12 + 7 + 4 + 7 = 30 \end{aligned}$$



6.2 The Traveling Salesperson Problem

- Best-First-Search with Branch-and-Bound Pruning:
 - Initialize the value of the best solution to infinity
 - since there is no candidate solution at the root node.
 - *Candidate solutions* exist *only at leaves* in the state space tree.
 - We need not compute bounds for leaves
 - because the algorithm is written so as not to expand beyond leaves.

6.2 The Traveling Salesperson Problem

- $n = 5$

<i>W</i>	1	2	3	4	5
1	0	14	4	10	20
2	14	0	7	8	7
3	4	5	0	7	16
4	11	7	9	0	2
5	18	7	17	4	0

$minlength = \infty$

$opttour = [\]$

$bound = 21$


Node [1] is *inserted* into the PQ.

PQ

[1]

($bound = 21$)

6.2 The Traveling Salesperson Problem

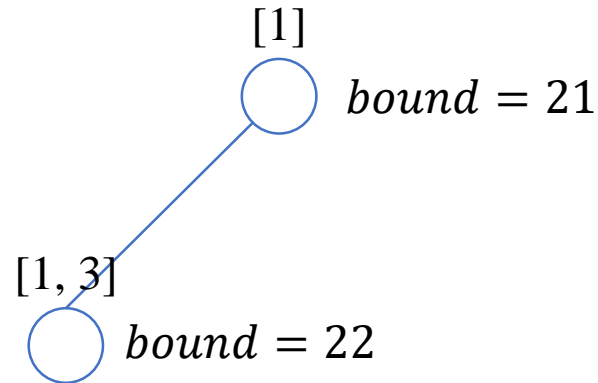
[1]
 *bound* = 21

1. Node [1] is *removed* from the PQ. (the root)
 - a. Bound is 21 and *minlength* = ∞
 - b. Node [1,2] is inserted into the PQ. *bound* = 31.
 - c. Node [1,3] is inserted into the PQ. *bound* = 22.
 - d. Node [1,4] is inserted into the PQ. *bound* = 30.
 - e. Node [1,5] is inserted into the PQ. *bound* = 42.

PQ

[1, 3]
 (*bound* = 22)
 [1,4]
 (*bound* = 30)
 [1,2]
 (*bound* = 31)
 [1,5]
 (*bound* = 42)

6.2 The Traveling Salesperson Problem



2. Node $[1,3]$ is *removed* from the PQ.
 - a. Bound is 22 and $minlength = \infty$
 - b. Node $[1,3,2]$ is inserted into the PQ. *bound = 22*.
 - c. Node $[1,3,4]$ is inserted into the PQ. *bound = 27*.
 - d. Node $[1,3,5]$ is inserted into the PQ. *bound = 39*.

PQ

[1,3,2]
(*bound = 22*)

[1,3,4]
(*bound = 27*)

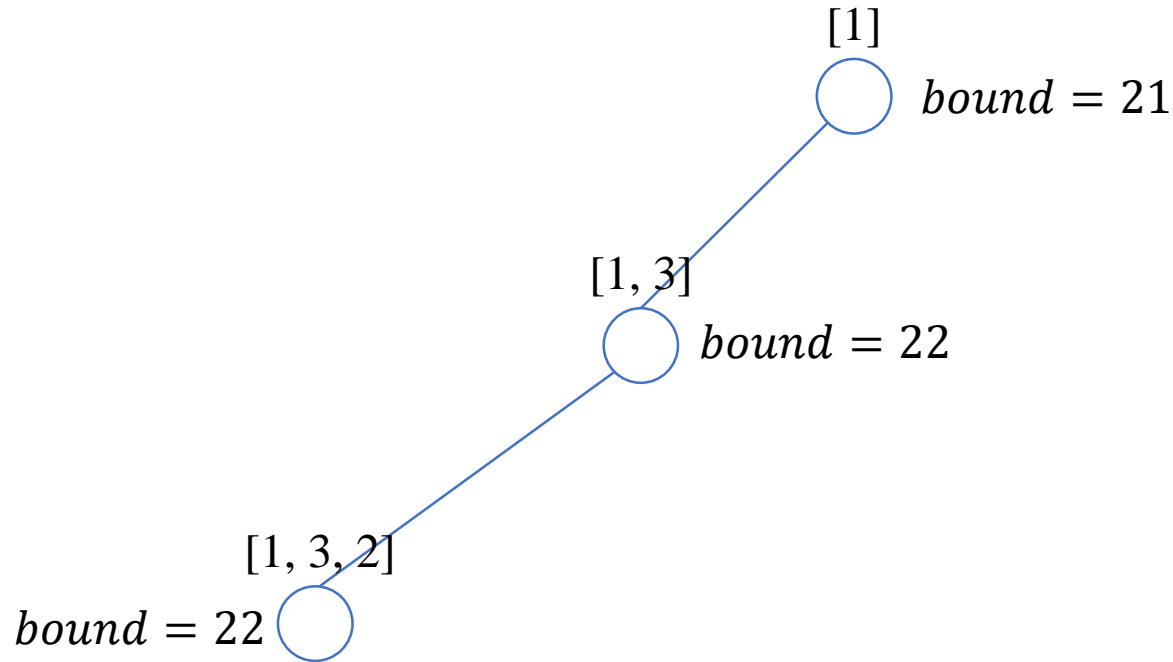
[1,4]
(*bound = 30*)

[1,2]
(*bound = 31*)

[1,3,5]
(*bound = 39*)

[1,5]
(*bound = 42*)

6.2 The Traveling Salesperson Problem



3. Node $[1, 3, 2]$ is *removed* from the PQ.
 - a. Bound is 22 and $minlength = \infty$
 - b. Node $[1, 3, 2, 4]$ is a leaf node. $length = minlength = 37$. $opttour = [1, 3, 2, 4, 5, 1]$.
 - c. Node $[1, 3, 2, 5]$ is a leaf node. $length = minlength = 31$. $opttour = [1, 3, 2, 5, 4, 1]$.

PQ

[1, 3, 4]
(bound = 27)

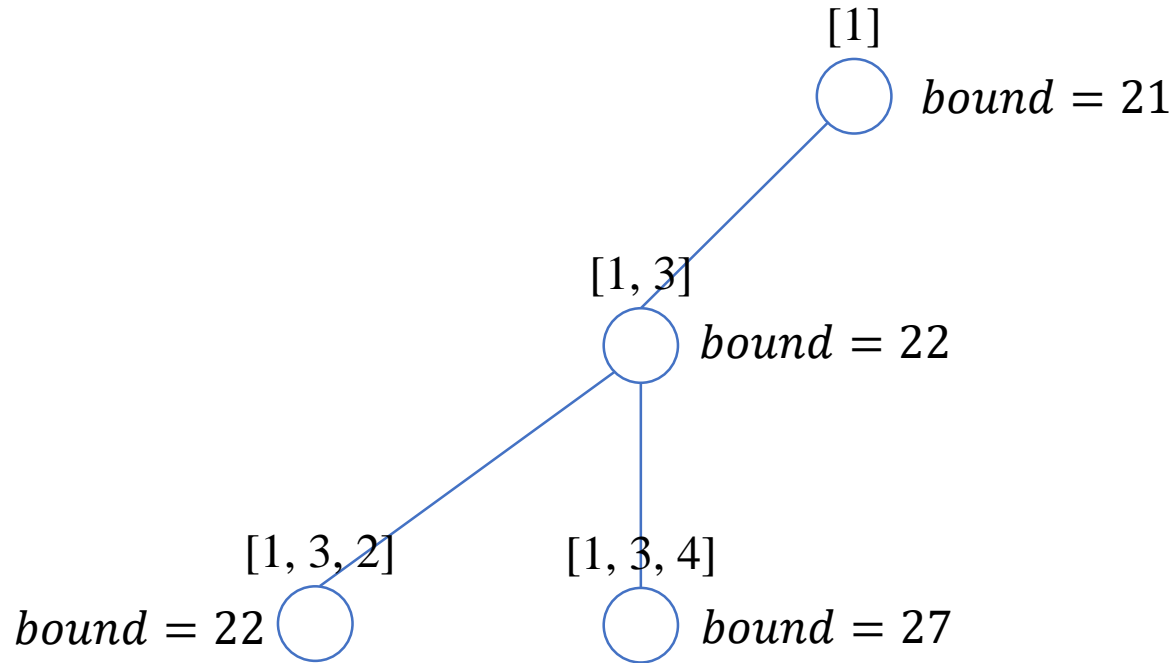
[1, 4]
(bound = 30)

[1, 2]
(bound = 31)

[1, 3, 5]
(bound = 39)

[1, 5]
(bound = 42)

6.2 The Traveling Salesperson Problem



3. Node $[1, 3, 4]$ is *removed* from the PQ.
 - a. Bound is 27 and *minlength* = 31
 - b. Node $[1, 3, 4, 2]$ is a leaf node. *length* = 43. *minlength* = 31.
 - c. Node $[1, 3, 4, 5]$ is a leaf node. *length* = 34. *minlength* = 31.

PQ

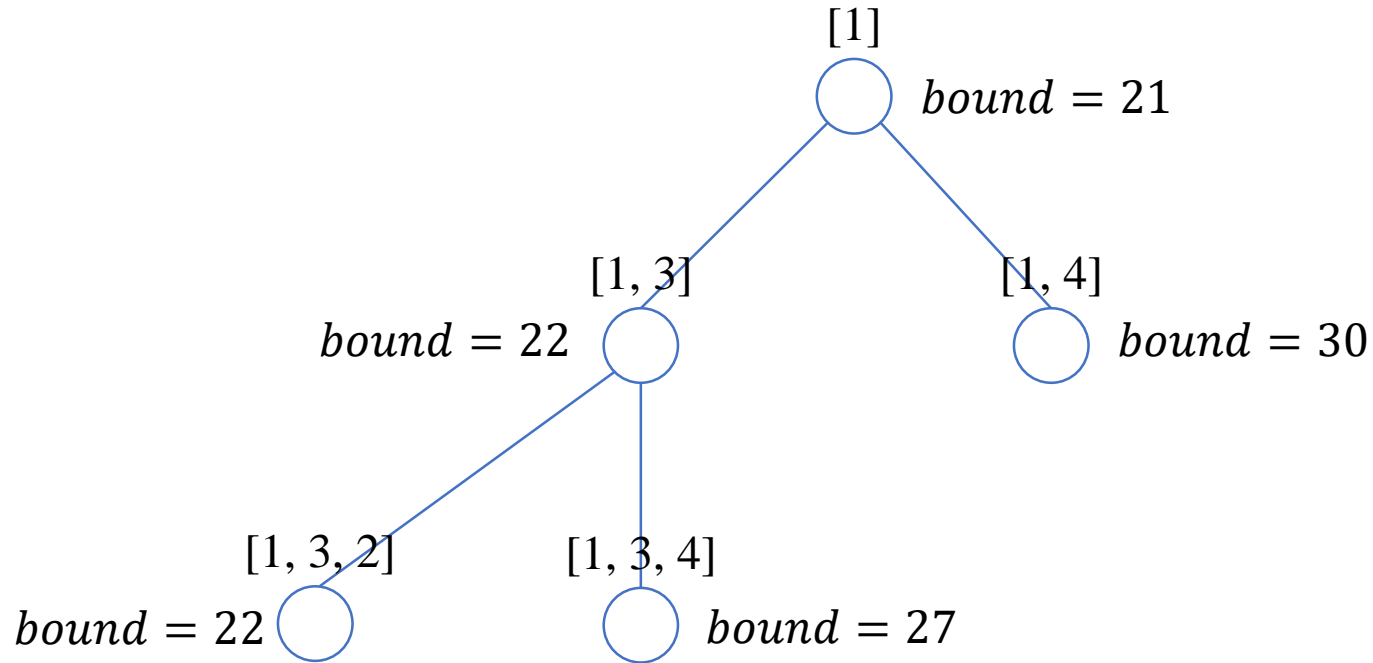
$[1, 4]$
(*bound* = 30)

$[1, 2]$
(*bound* = 31)

$[1, 3, 5]$
(*bound* = 39)

$[1, 5]$
(*bound* = 42)

6.2 The Traveling Salesperson Problem



3. Node [1,4] is *removed* from the PQ.
 - a. Bound is 30 and *minlength* = 31.
 - b. Node [1,4,2] is **NOT** inserted into the PQ. *bound* = 45 > *minlength* = 31.
 - c. Node [1,4,3] is **NOT** inserted into the PQ. *bound* = 38 > *minlength* = 31.
 - d. Node [1,4,5] is inserted into the PQ. *bound* = 30.

PQ

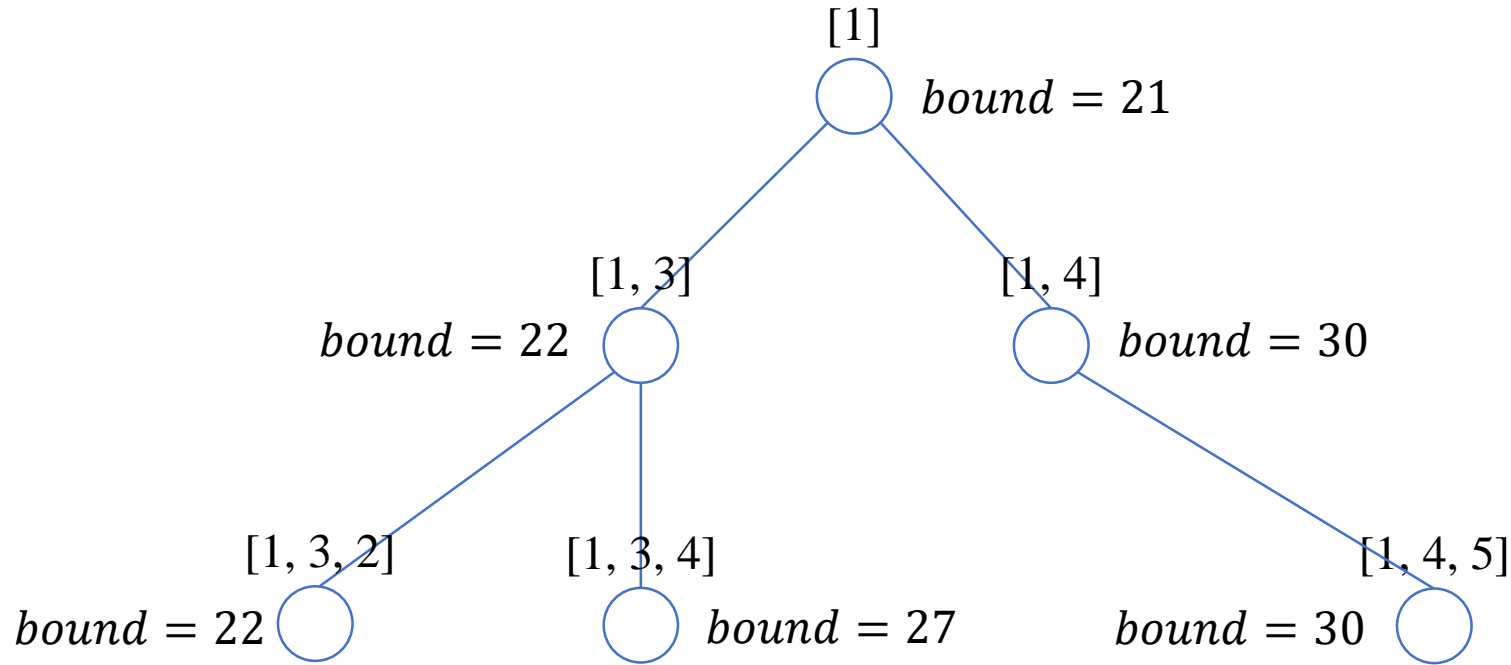
[1,4,5]
(*bound* = 30)

[1,2]
(*bound* = 31)

[1,3,5]
(*bound* = 39)

[1,5]
(*bound* = 42)

6.2 The Traveling Salesperson Problem



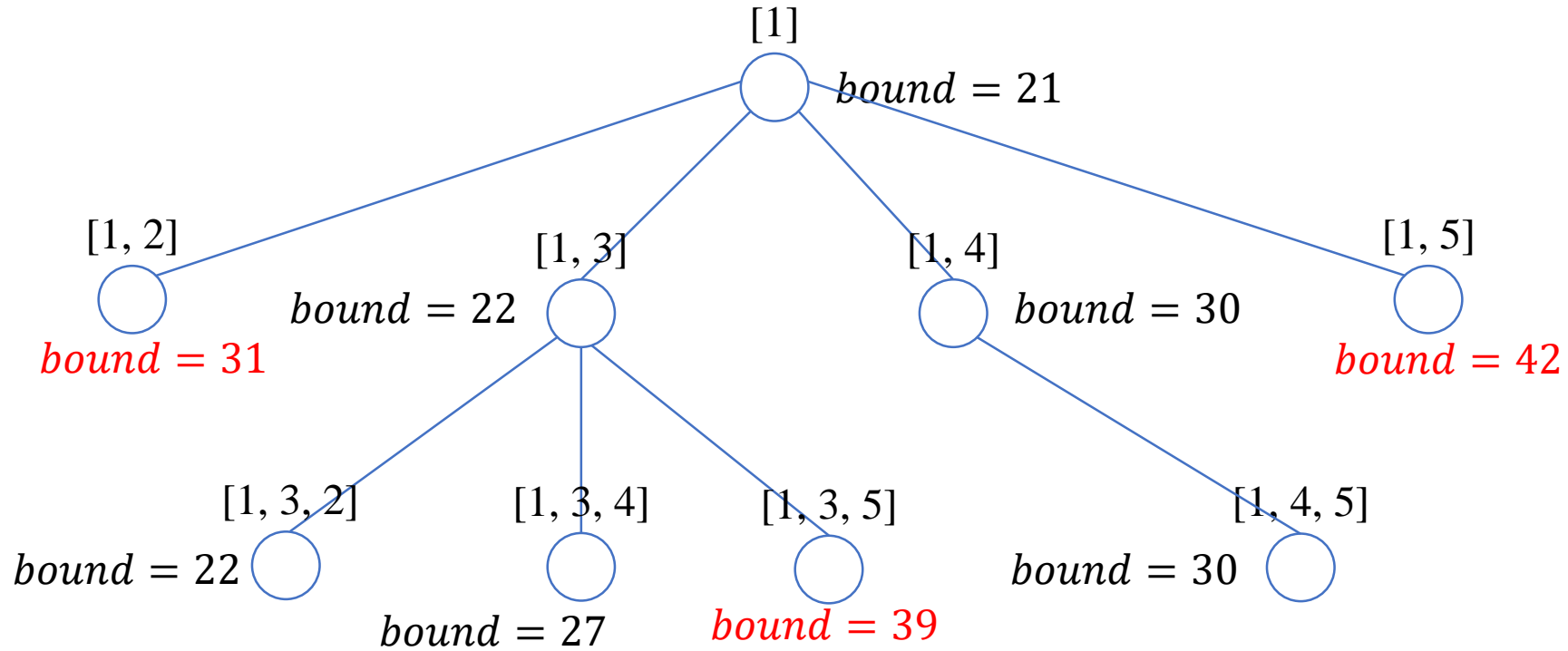
3. Node $[1, 4, 5]$ is *removed* from the PQ.
 - a. Bound is 30 and *minlength* = 31
 - b. Node $[1, 4, 5, 2]$ is a leaf node. *length* = 30. *minlength* = 30. *opttour* = $[1, 4, 5, 2, 3, 1]$.
 - c. Node $[1, 4, 5, 3]$ is a leaf node. *length* = 48. *minlength* = 30.

PQ

$[1, 3, 5]$
(*bound* = 39)

$[1, 5]$
(*bound* = 42)

6.2 The Traveling Salesperson Problem

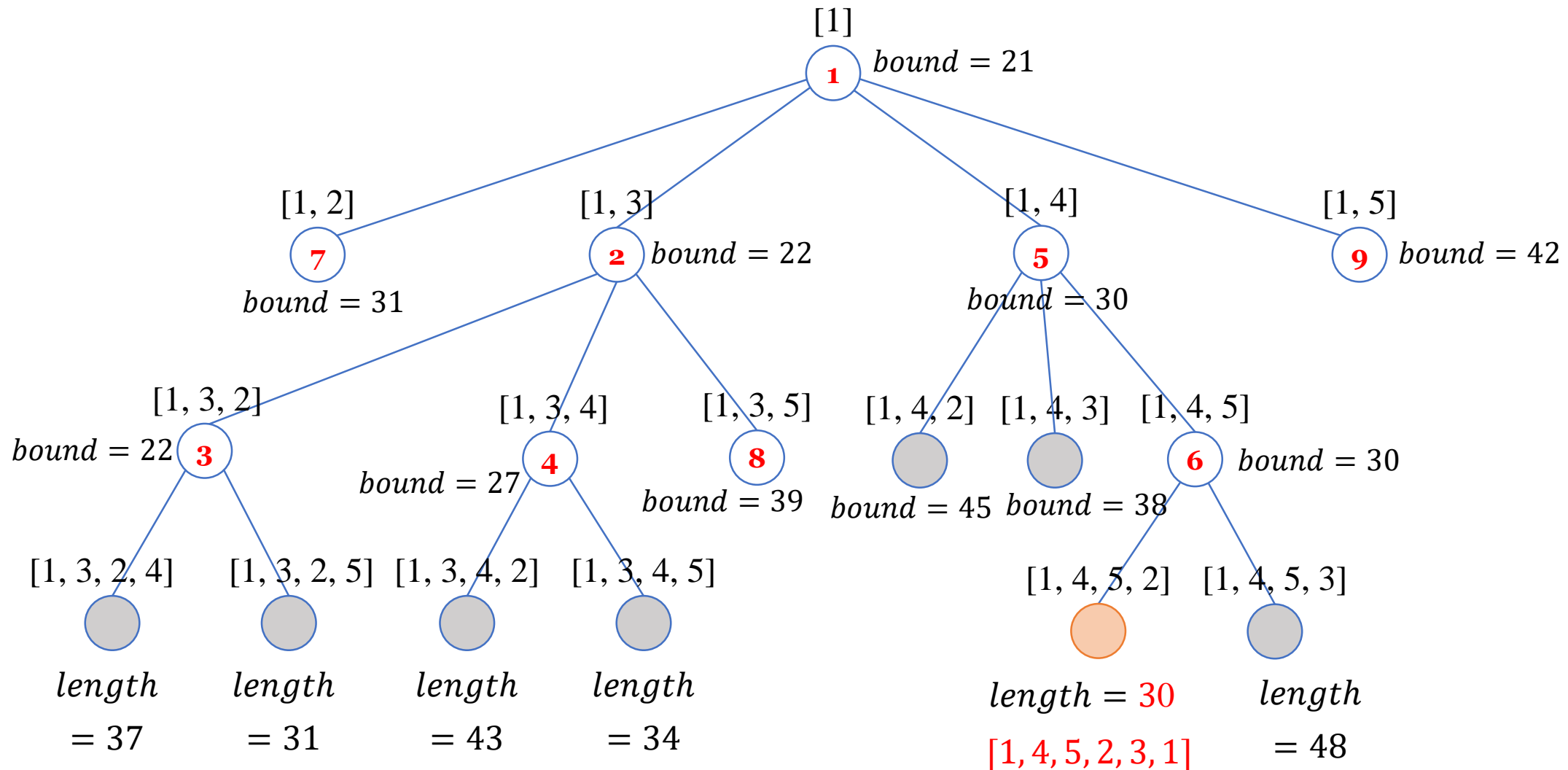


PQ

[1,2]
(bound = 31)
[1,3,5]
(bound = 39)
[1,5]
(bound = 42)

3. Node [1,2] is *removed* from the PQ. $\text{bound} = 31 > \text{minlength} = 30$.
4. Node [1,3,5] is *removed* from the PQ. $\text{bound} = 42 > \text{minlength} = 30$.
5. Node [1,5] is *removed* from the PQ. $\text{bound} = 39 > \text{minlength} = 30$.

6.2 The Traveling Salesperson Problem



6.2 The Traveling Salesperson Problem

ALGORITHM 6.3: The Best-First-Search with Branch-and-Bound Pruning for the TSP.

```
void travel2(ordered_set &opttour, int &minlength) {
    priority_queue_of_node PQ;
    node_pointer u, v;

    minlength = INF;
    v.level = 0; v.path = [1]; v.bound = bound(v);
    PQ.insert(v);
    while (!PQ.empty()) {
        v = PQ.remove();
        if (v->bound < minlength) {

            // .....

        }
    }
}
```

6.2 The Traveling Salesperson Problem

```

for (int i = 2; i <= n; i++) {
    // for all i such that i is not in v.path
    if (isIn(i, v->path)) continue;
    u = create_node(v->level + 1, v->path);
    u->path.push_back(i);
    if (u->level == n - 2) {
        // put the only vertex not in v.path
        u->path.push_back(remaining_vertex(u->path));
        u->path.push_back(1); // make first vertex last one
        if (length(u->path) < minlength) {
            minlength = length(u->path);
            copy(u->path, opttour);
        }
    }
    else {
        u->bound = bound(u);
        if (u->bound < minlength)
            PQ.push(u);
    }
}

```

6.2 The Traveling Salesperson Problem

- The Efficiency for the TSP:
 - Note that there are *17 nodes* visited in the state space tree ,
 - whereas the entire state space tree has *41 nodes*.
 - $1 + 4 + 4 \times 3 + 4 \times 3 \times 2 = 41$.
 - When two or more bounding functions are available,
 - we can compute bounds using all available functions
 - However, remember that our goal is not to visit as few nodes as possible,
 - but rather to maximize the overall efficiency of the algorithm.
 - Final remark on the TSP is *approximation algorithms*:
 - They are not guaranteed to yield optimal solution,
 - but rather yield solutions that reasonably close to optimal.

Any Questions?

