CST207
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 9: Branch-and-Bound

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

Branch-and-Bound

- The branch-and-bound design strategy is very similar to backtracking in that a state space tree is used to solve a problem.
- The differences are that the branch-and-bound method
  1. does not limit us to any particular way of traversing the tree;
  2. is used only for optimization problems.
- A branch-and-bound algorithm computes a *bound* at a node to determine whether the node is promising.
  - The backtracking algorithm for the 0-1 Knapsack problem is actually a branch-and-bound algorithm.
  - The promising function returns false if the value of bound is not greater than the current value of maxprofit.

Breadth-First Search

- Branch-and-bound is based on breadth-first search (BFS).
- BFS visits the nodes in a tree level by level.
- It is usually implemented by using a queue, rather than recursion (stack).

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

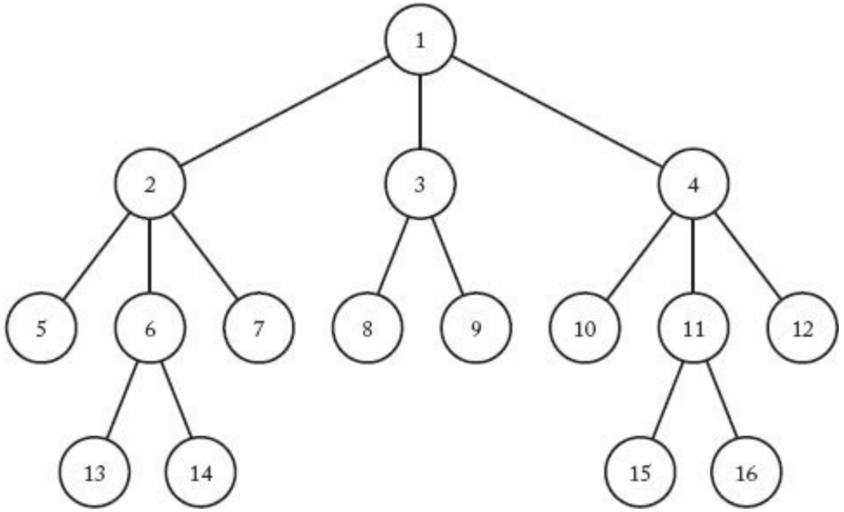


Image source: Figure 6.1, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Outlines

- 0-1 Knapsack Problem
- The Assignment Problem
- The Traveling Salesperson Problem

0-1 KNAPSACK PROBLEM

Bound

- Sort the items in non-increasing order according to the ratio between v_k and w_k.
- Suppose the node is at level i, we first calculate k such that the level k is the one that would bring the sum of the weights *exceeds* W.
- Then we have:

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

$$bound = \underbrace{profit + \sum_{j=i+1}^{k-1} v_j}_{\substack{\text{Profit from first} \\ k-1 \text{ items taken}}} + \underbrace{(W - totweight)}_{\substack{\text{Capacity available} \\ \text{for } k\text{th item}}} \times \underbrace{\frac{v_k}{w_k}}_{\substack{\text{Profit per unit} \\ \text{weight for } k\text{th item}}}.$$

Pruned State Space Tree by BFS

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

- Recall that by using DFS, node (1, 2) was found to be nonpromising and we did not expand beyond the node.
- However, in the case of BFS, node (1, 2) is the third node visited.
  - At the time it is visited, the value of maxprofit is only \$40. Because its bound \$82 exceeds `maxprofit` at this point, we expand beyond the node.
- Unlike DFS, in BFS the value of maxprofit can change by the time we actually visit the children.
  - In this case, `maxprofit` has a value of \$90 by the time we visit the children of node (2, 3).
  - We then waste our time checking these children.

$W = 16$

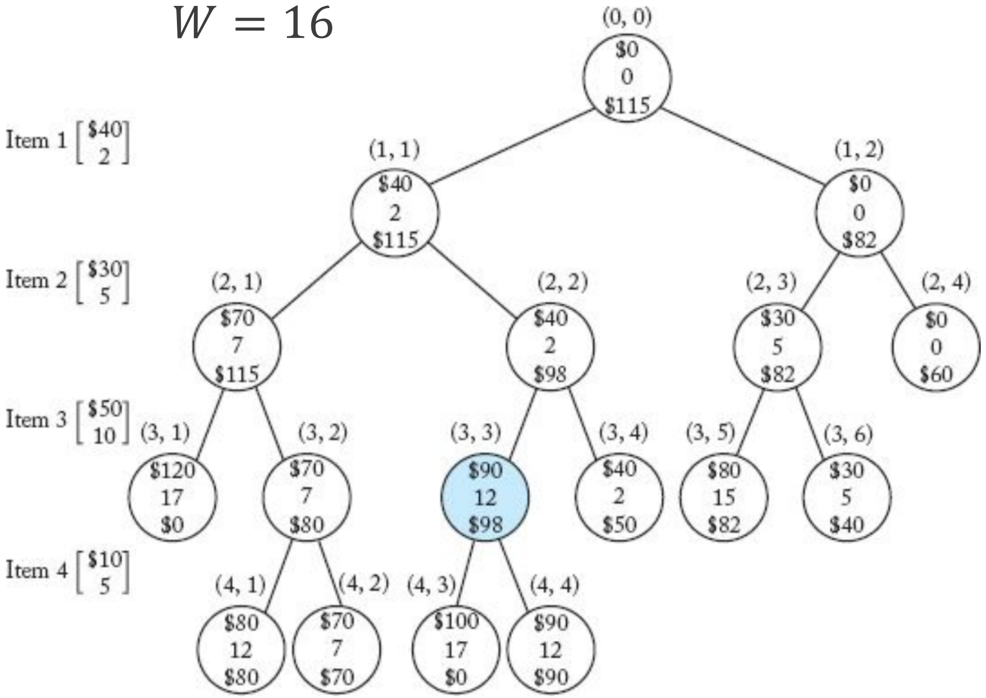


Image source: Figure 6.2, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pseudocode of a General BFS with Branch-and-Bound Algorithm

```
void breadth_first_branch_and_bound (tree T,
                                     number& best)
{
    queue Q;
    node u, u_child;

    initialize(Q);
    u = root of T;
    best = value(u);
    enqueue(Q, u);
    while (!empty(Q)){
        u = dequeue(Q);
        for (each child of u_child of u){
            if (value(u_child) is better than best)
                best = value(u_child);
            if (bound(u_child) is better than best)
                enqueue(Q, u_child);
        }
    }
}
```

Pseudocode of BFS Version of 0-1 Knapsack Problem

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

```
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 + v[j];
            j++;
        }
        k = j;
        if (k <= n)
            result = result + (W – totweight) * v[k] / w[k];

        return result;
    }
}
```

```
void knapsack_breadth (int n,
                       const int v[],
                       const int w[],
                       int W,
                       int& maxprofit)
{
    queue Q;
    node u, u_child;

    initialize(Q);
    u.level = 0; u.profit = 0; u.weight = 0;

    maxprofit = 0;
    enqueue(Q, u);
    while (!empty(Q)){
        u = dequeue(Q);
        u_child.level = u.level + 1;
        // set u to the child that includes the next item.
        u_child.weight = u.weight + w[u_child.level];
        u_child.profit = u.profit + v[u_child.level];
        if (u_child.weight <= W && u_child.profit > maxprofit)
            maxprofit = u_child.profit;
        if (bound(u_child) > maxprofit)
            enqueue(Q, u_child);
        // set u to the child that does not include the next item.
        u_child.weight = u.weight;
        u_child.profit = u.profit;
        if (bound(u_child) > maxprofit)
            enqueue(Q, u_child);
    }
}
```

Best-First Search with Branch-and-Bound Pruning

- Comparison between breadth-first and best-first search:
  - Breadth-first: visit the unexpanded node according to its order in the queue.
  - Best-first: visit the unexpanded node according to its *value* in the queue.
- For 0-1 knapsack problem, best-first search visit the node with maximum bound in the queue first.
  - Select the one who has the greatest hope!

Pruned State Space Tree by Best-First Search

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

1. Visit node (0,0).
2. Visit node (1,1).
   - `maxprofit`=40.
3. Visit node (1,2).
4. Determine promising, unexpanded node with greatest bound.
   - Select node (1,1) to expand.

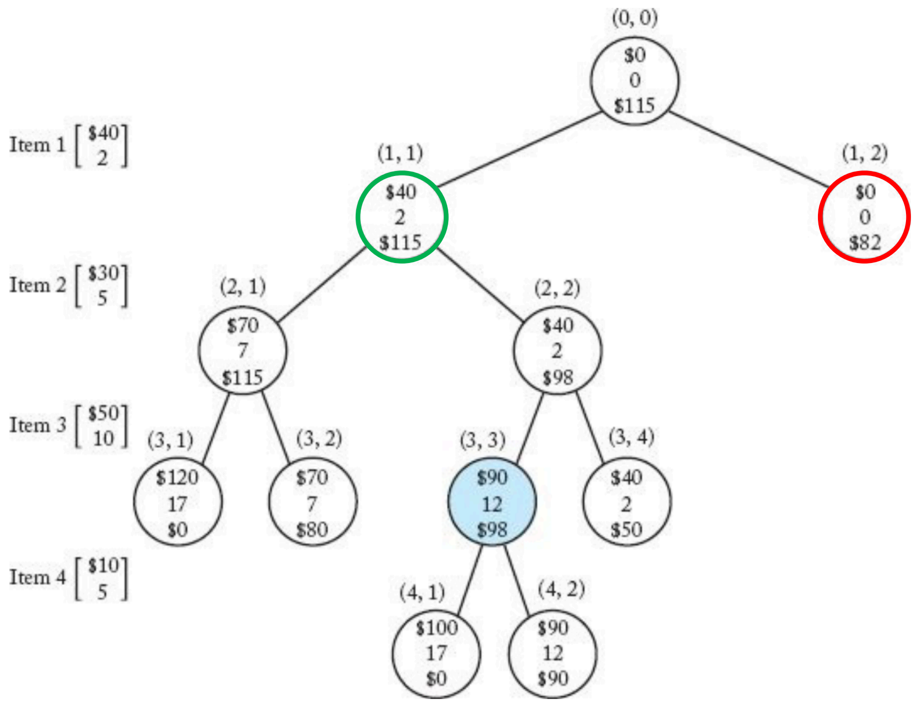


Image source: Figure 6.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pruned State Space Tree by Best-First Search

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

5. Visit node (2, 1).
   - `maxprofit`=70.
6. Visit node (2, 2).
7. Determine promising, unexpanded node with greatest bound.
   - Select node (2,1) to expand.

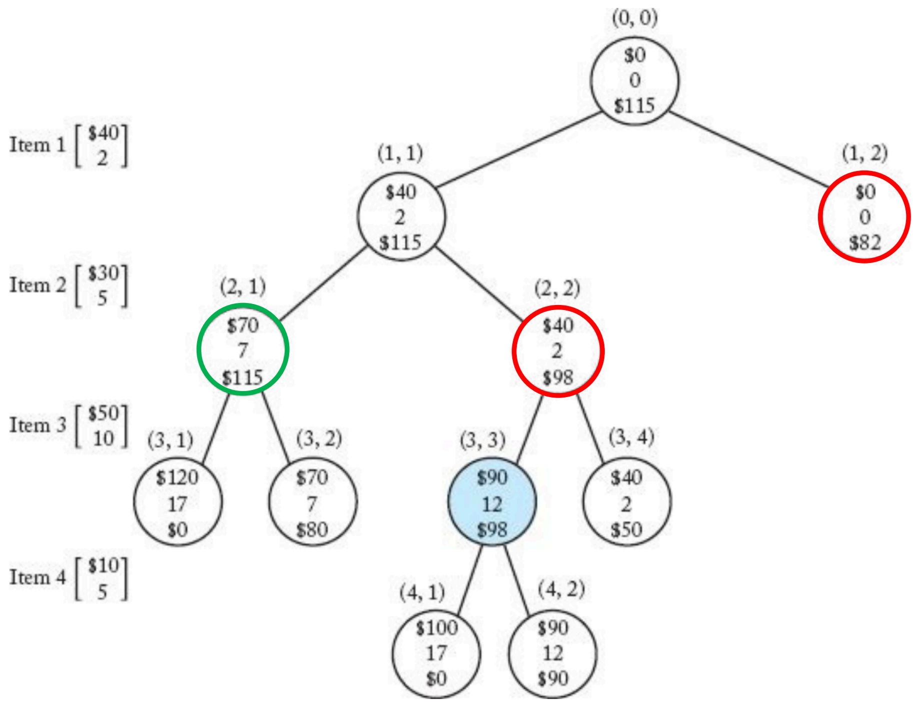


Image source: Figure 6.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pruned State Space Tree by Best-First Search

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

8. Visit node (3, 1).
9. Visit node (3, 2).
10. Determine promising, unexpanded node with greatest bound.
    - Select node (2,2) to expand.

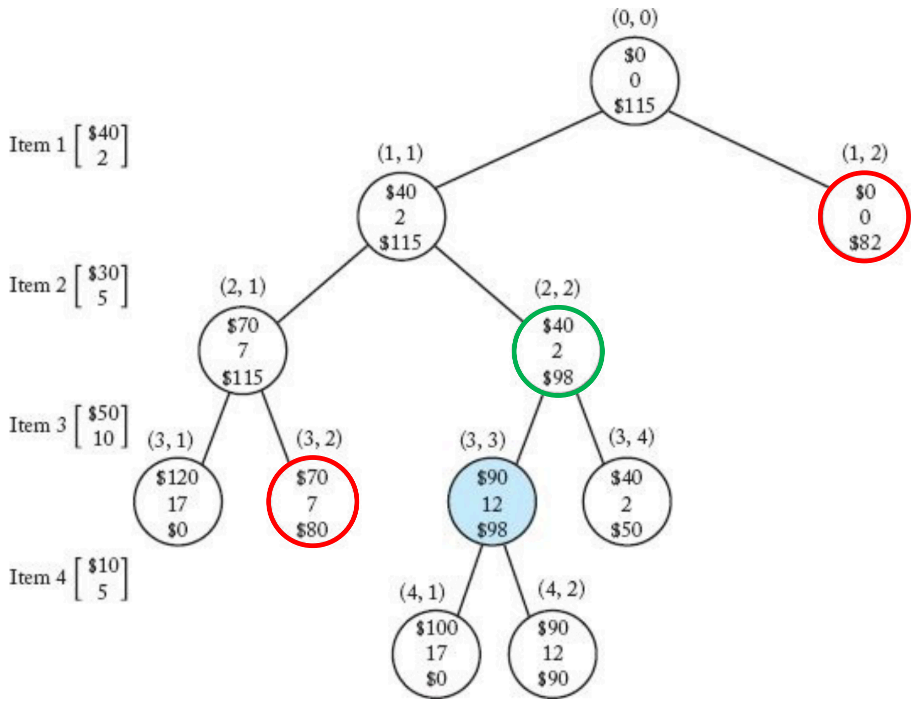


Image source: Figure 6.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pruned State Space Tree by Best-First Search

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

11. Visit node (3, 3).
    - `maxprofit`=90.
12. Visit node (3, 4).
13. Determine promising, unexpanded node with greatest bound.
    - Select node (3, 3) to expand.

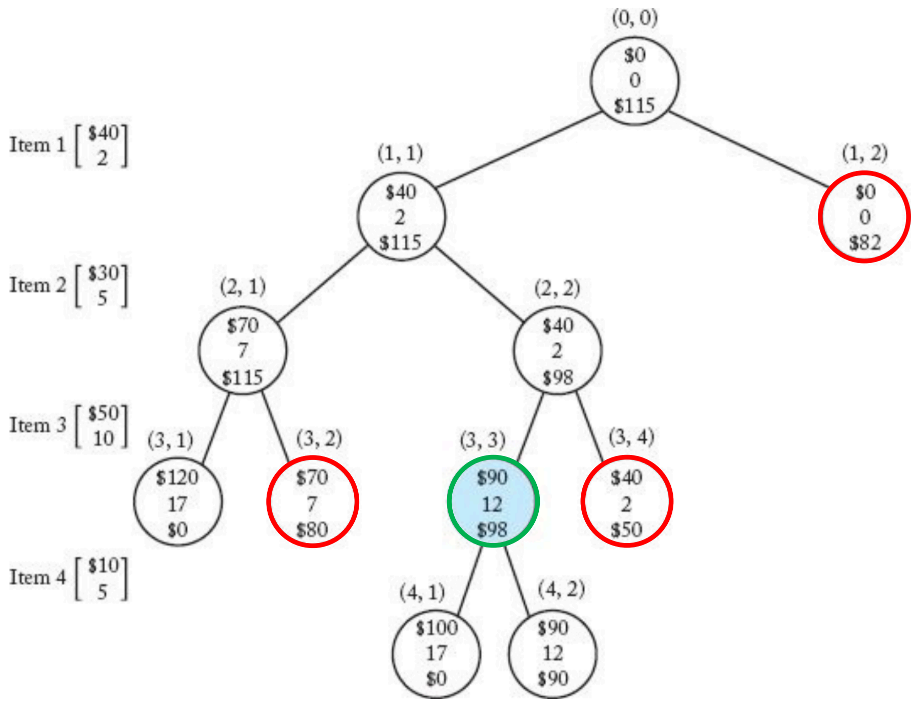


Image source: Figure 6.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pruned State Space Tree by Best-First Search

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

14. Visit node (4,1).
15. Visit node (4,2).
    - No promising and unexpanded node exists because the bound of node (1, 2) is less than `maxprofit`=90.

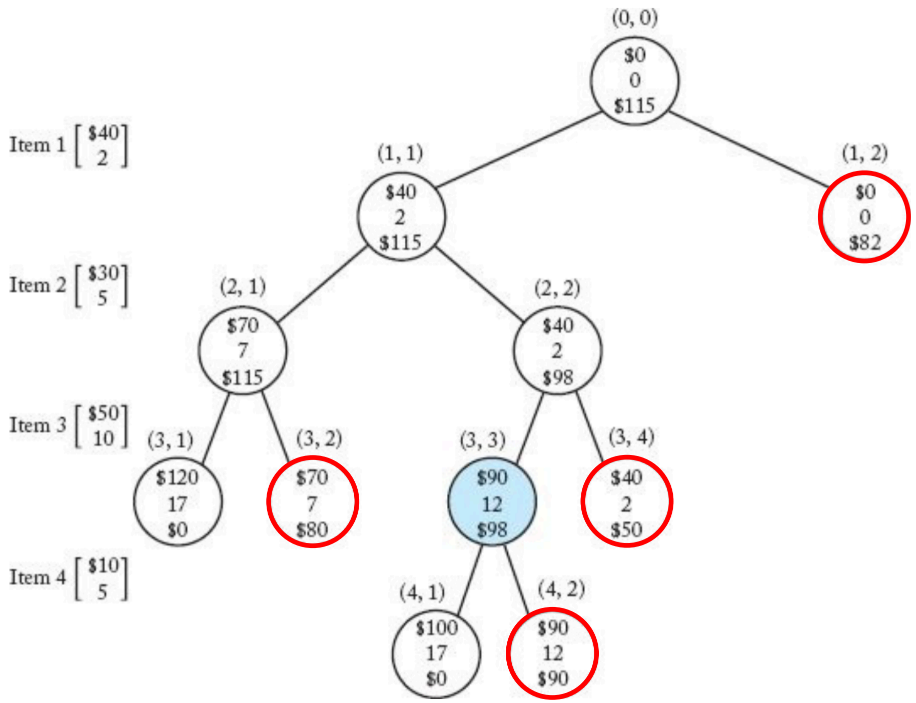


Image source: Figure 6.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pruned State Space Tree by Best-First Search

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

- Using best-first search, we have checked only 11 nodes.
  - 6 less than the number checked using BFS.
  - 2 less than the number checked using DFS.
- However, there is no guarantee that the node that appears to be best will actually lead to an optimal solution.
  - In this example, node (2, 1) appears to be better than node (2, 2), but node (2, 2) leads to the optimal solution.

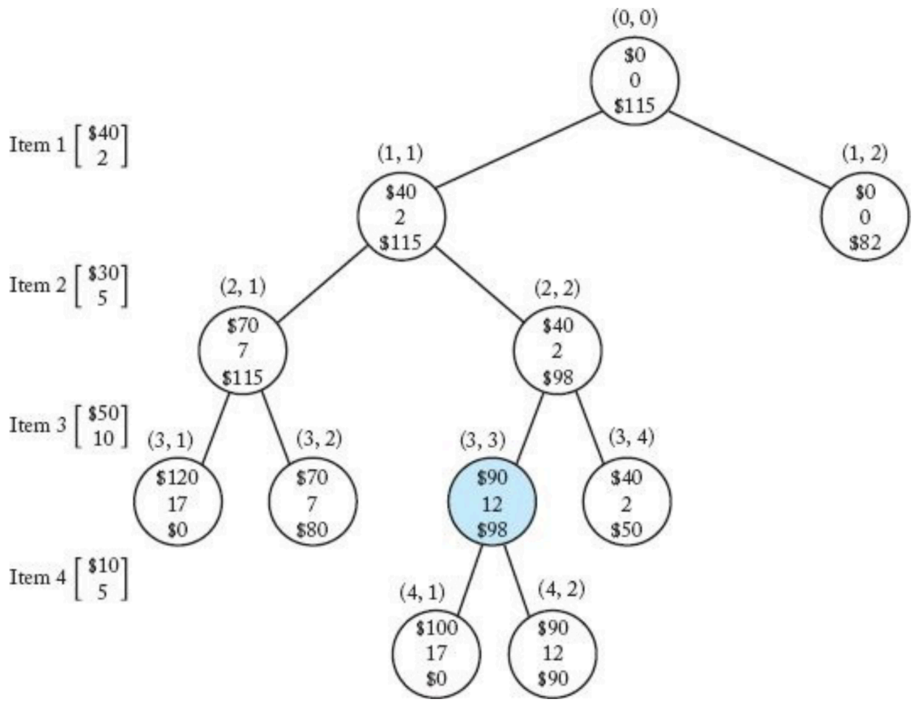


Image source: Figure 6.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pseudocode of a General Best-First Search with Branch-and-Bound Algorithm

```
void best_first_branch_and_bound (tree T,
                                  number& best)
{
    priority_queue PQ;
    node u, u_child;

    initialize(PQ);
    u = root of T;
    best = value(u);
    enqueue(PQ, u);
    while (!empty(PQ)){
        u = dequeue(PQ);
        if (bound(u) is better than best)
            for (each child of u_child of u){
                if (value(u_child) is better than best)
                    best = value(u_child);
                if (bound(u_child) is better than best)
                    enqueue(PQ, u_child);
            }
    }
}
```

Difference with BFS version

Pseudocode of Best-First Search Version

- Function bound is same.

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

```
void kanpsack_best (int n,
                    const int v[],
                    const int w[],
                    int W,
                    int& maxprofit)
{
    priority_queue PQ;
    node u, u_child;

    initialize(PQ);
    u.level = 0; u.profit = 0; u.weight = 0;

    maxprofit = 0;
    enqueue(PQ, u);
    while (!empty(PQ)){
        u = dequeue(PQ);
        if (u.bound > maxprofit){
            u_child.level = u.level + 1;
            // set u to the child that includes the next item.
            u_child.weight = u.weight + w[u_child.level];
            u_child.profit = u.profit + v[u_child.level];
            if (u_child.weight <= W && u_child.profit > maxprofit)
                maxprofit = u_child.profit;
            u_child.bound = bound(u_child)
            if (u_child.bound > maxprofit)
                enqueue(PQ, u_child);
            // set u to the child that does not include the next item.
            u_child.weight = u.weight;
            u_child.profit = u.profit;
            u_child.bound = bound(u_child)
            if (u_child.bound > maxprofit)
                enqueue(PQ, u_child);
        }
    }
}
```

Difference with BFS version

THE ASSIGNMENT PROBLEM

The Assignment Problem

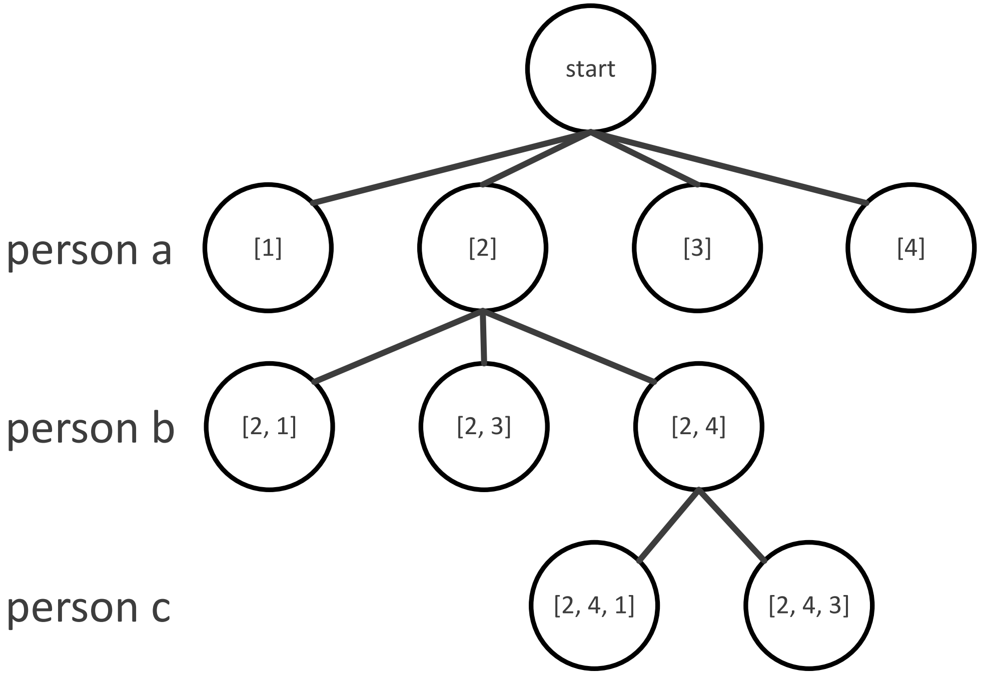
- The assignment problem aims to assign n people to n jobs so that the total cost of the assignment is as small as possible.
  - An instance of the assignment problem is specified by an $n \times n$ cost matrix C.
  - Select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible.

$$\begin{array}{cc} & \begin{array}{cccc}\text{job 1} & \text{job 2} & \text{job 3} & \text{job 4}\end{array} \\ C = & \begin{bmatrix}9 & 2 & 7 & 8\\ 6 & 4 & 3 & 7\\ 5 & 8 & 1 & 8\\ 7 & 6 & 9 & 4\end{bmatrix} \begin{array}{l}\text{person a}\\ \text{person d}\\ \text{person c}\\ \text{person d}\end{array}\end{array}$$

- For this example, the optimal solution is 2+6+1+4=13.

State Space Tree of the Assignment Problem

- The final solution does not depend on the starting person, we will start with person a.
- We stop expanding the tree when we have assigned $n-1$ people because, at that time, the job of the nth person is uniquely determined.
  - For example, if we have assigned [2, 4, 3], person d can only be assigned to job 1.



The Assignment Problem

- It seems that this problem can be solve by greedy approach.
  - Always find the smallest cost in the unselected columns and rows.
- However, a counterexample can be easily obtained:

$$C = \begin{bmatrix} 10 & 10 & 2 & 10 \\ 10 & 10 & 2 & 10 \\ 2 & 2 & 1 & 2 \\ 10 & 10 & 2 & 10 \end{bmatrix}$$

- The greedy solution is 1+10+10+10=31, while the optimal solution is 2+2+10+10=24.

Lower Bound of Total Cost

- In this case, the bound is a lower bound.
- The lower bound is calculated as the sum of minimum cost of each person.

$$\text{person a: } minimum(9,2,7,8) = 2$$
$$\text{person b: } minimum(6,4,3,7) = 3$$
$$\text{person c: } minimum(5,8,1,8) = 1$$
$$\text{person d: } minimum(7,6,9,4) = 4$$

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

- Therefore, a lower bound of the total cost is:

$$2 + 3 + 1 + 4 = 10.$$

- Any solution can’t be smaller than this lower bound.

Lower Bound of Total Cost

- The lower bound in each node will change according to the assignment.
- For example, if person a is assigned to job 1.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

$$\begin{aligned} &\text{person a: } 9 \\ &\text{person b: } minimum(4,3,7) = 3 \\ &\text{person c: } minimum(8,1,8) = 1 \\ &\text{person d: } minimum(6,9,4) = 4 \end{aligned}$$

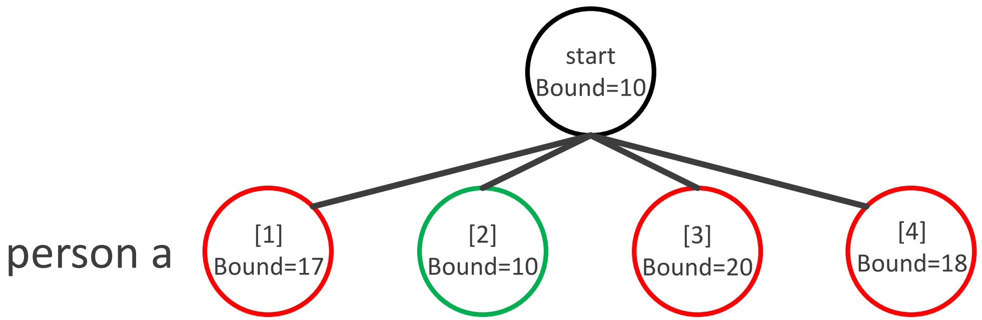
- Therefore, a lower bound of the total cost after person a being assigned to job 1 is:

$$9 + 3 + 1 + 4 = 17.$$

- We can thus use this calculation to build the pruned state space tree with best-first search.

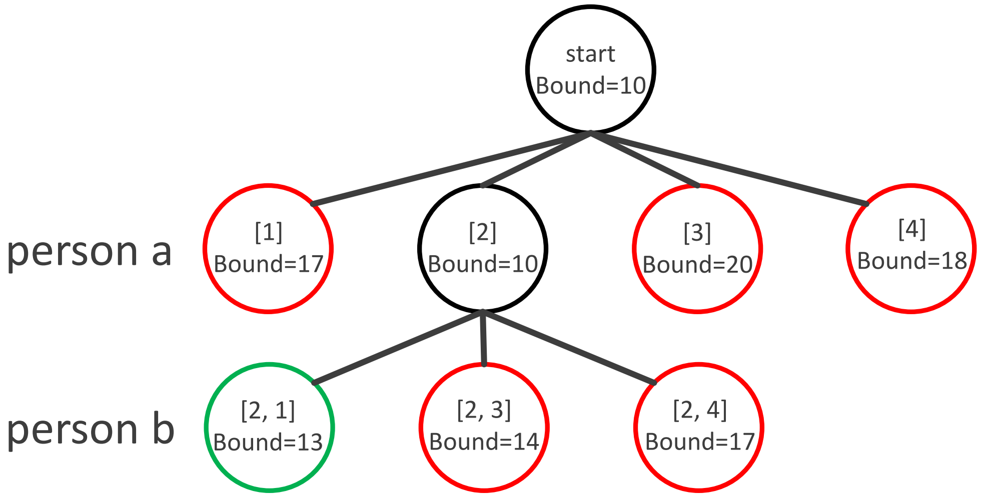
Pruned State Space Tree with Best-First Search

1. Visit root.
2. Visit node containing [1].
3. Visit node containing [2].
4. Visit node containing [3].
5. Visit node containing [4].
6. Determine promising, unexpanded node with the smallest bound.
   - Node containing [2] is selected. We visit its children.



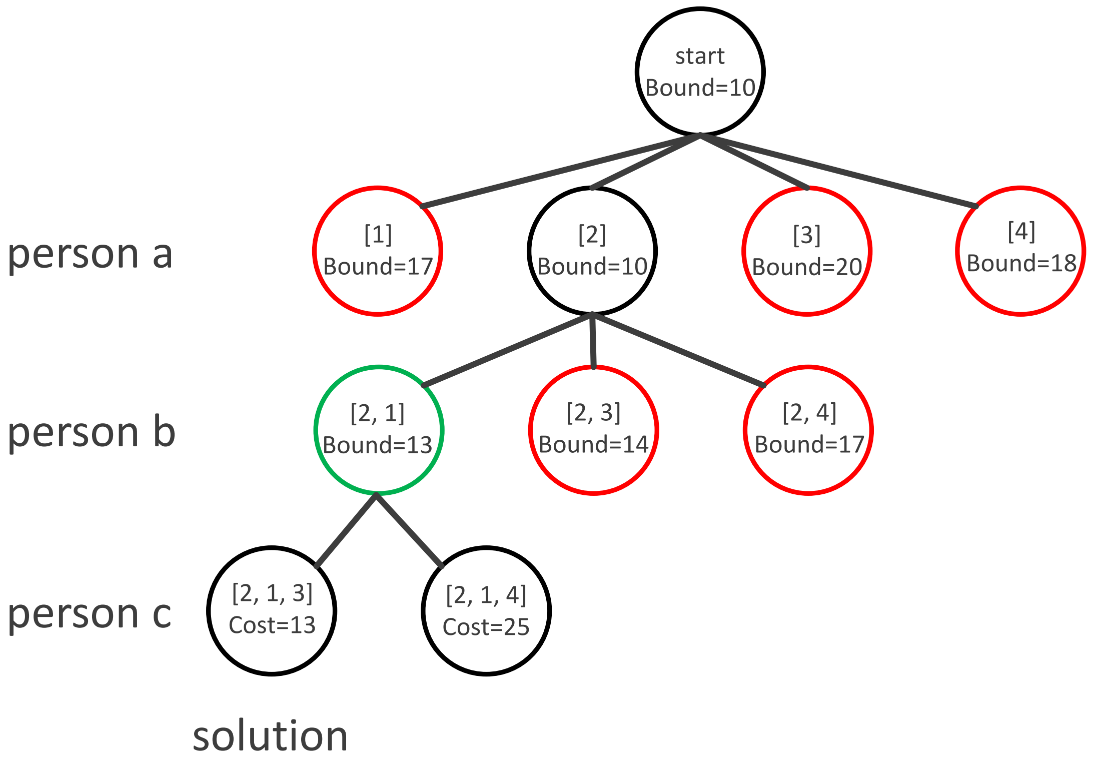
Pruned State Space Tree with Best-First Search

7. Visit node containing [2, 1].
8. Visit node containing [2, 3].
9. Visit node containing [2, 4].
10. Determine promising, unexpanded node with the smallest bound.
    - Node containing [2, 1] is selected. We visit its children.



Pruned State Space Tree with Best-First Search

11. Visit node containing [2, 1, 3].
    - Compute total cost: 13. `mincost=13`.
12. Visit node containing [2, 1, 4].
    - Compute total cost: 25.
13. Determine promising, unexpanded node with the smallest bound.
    - There are no more promising, unexpanded nodes, because all the nodes have higher bound than `mincost`.

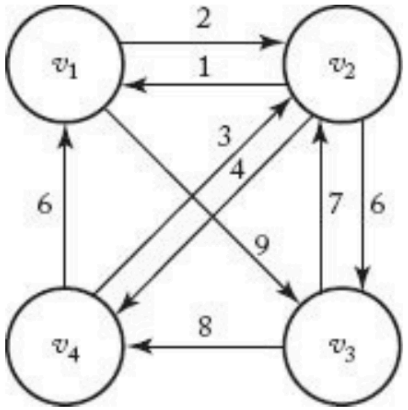


THE TRAVELING SALESPERSON PROBLEM

The Traveling Salesperson Problem

- A *tour* (also called a Hamiltonian circuit) in a directed graph is a path from a vertex to itself that passes through each of the other vertices exactly once.
- An *optimal tour* in a weighted, directed graph is such a path of minimum length.
- The Traveling Salesperson problem (TSP) is to find an optimal tour in a weighted, directed graph when at least one tour exists.
  - Because the weights are considered, it is the optimization version of Hamiltonian circuit problem.
- For example:

$$length[v_1, v_2, v_3, v_4, v_1] = 22$$
$$length[v_1, v_3, v_2, v_4, v_1] = 26$$
$$length[v_1, v_3, v_4, v_2, v_1] = 21$$



	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Image source: Figure 3.16-17, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

State Space Tree

- Because the starting vertex is irrelevant to the length of an optimal tour, we will consider v_1 to be the starting vertex.
- The state space tree can be constructed by:
  - Each vertex other than v_1 is tried as the first vertex at level 1.
  - Each vertex other than v_1 and the one chosen at level 1 is tried as the second vertex at level 2.
  - …
- 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.
  - For example, the far-left leaf represents the tour [1, 2, 3, 4, 5, 1] because once we have specified the path [1, 2, 3, 4], the next vertex must be v_5.

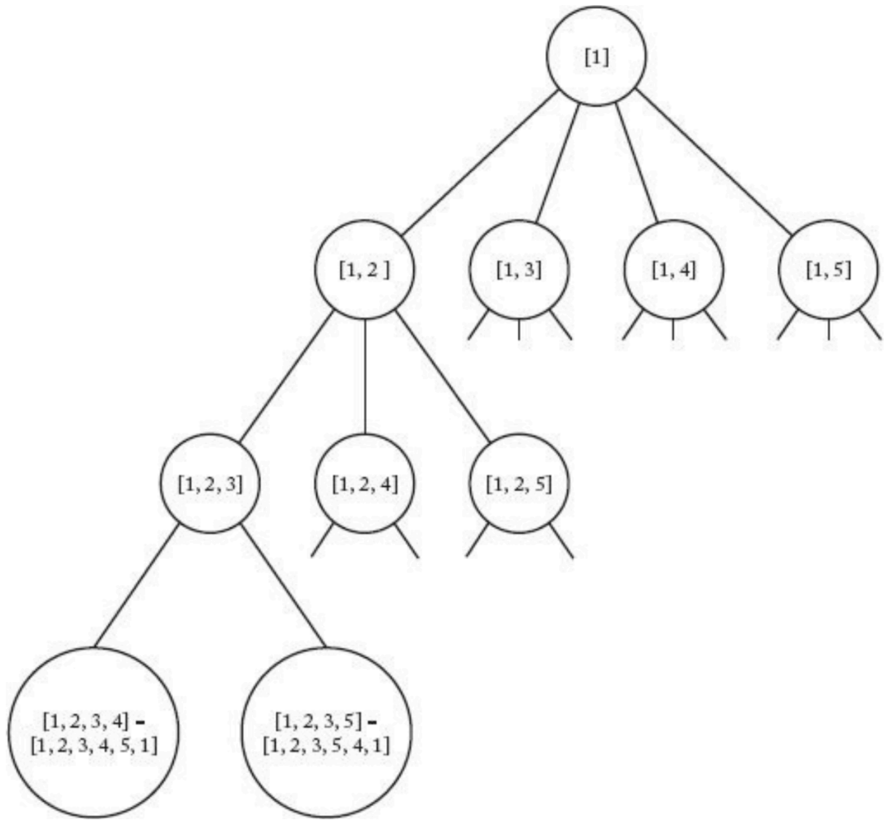


Image source: Figure 6.5, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Lower Bound of Tour Length

- In this case, the bound is a lower bound.
- 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 from that vertex.

$$
\begin{aligned}
v_1&: minimum(14, 4{,}10{,}20) = 4 \\
v_2&: minimum(14{,}7{,}8{,}7) = 7 \\
v_3&: minimum(4{,}5{,}7{,}16) = 4 \\
v_4&: minimum(11{,}7{,}9{,}2) = 2 \\
v_5&: minimum(18{,}7{,}17{,}4) = 4
\end{aligned}
$$

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

This adjacency matrix assumes that every vertex is connected

- Because a tour must leave every vertex exactly once, a lower bound on the length of a tour is the sum of these minimums. Therefore, a lower bound on the length of a tour is

$$4 + 7 + 4 + 2 + 4 = 21.$$

- 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.

Lower Bound of Tour Length

- Suppose we have visited the node containing [1, 2].
- Any tour obtained by expanding beyond this node has the following lower bounds on the costs of leaving the vertices:

$$v_1:\ 14$$
$$v_2:\ minimum(7,8,7) = 7$$
no path to v_1
$$v_3:\ minimum(4,7,16) = 4$$
$$v_4:\ minimum(11,9,2) = 2$$
$$v_5:\ minimum(18,17,4) = 4$$
no path to v_2

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

This adjacency matrix assumes that every vertex is connected

- A lower bound on the length of any tour, obtained by expanding beyond the node containing [1, 2], is the sum of these minimums, which is

$$14 + 7 + 4 + 2 + 4 = 31.$$

Pruned State Space Tree with Best-First Search

1. Visit node containing [1].
2. Visit node containing [1, 2].
3. Visit node containing [1, 3].
4. Visit node containing [1, 4].
5. Visit node containing [1, 5].
6. Determine promising, unexpanded node with the smallest bound.
   - Node containing [1, 3] is selected. We visit its children.

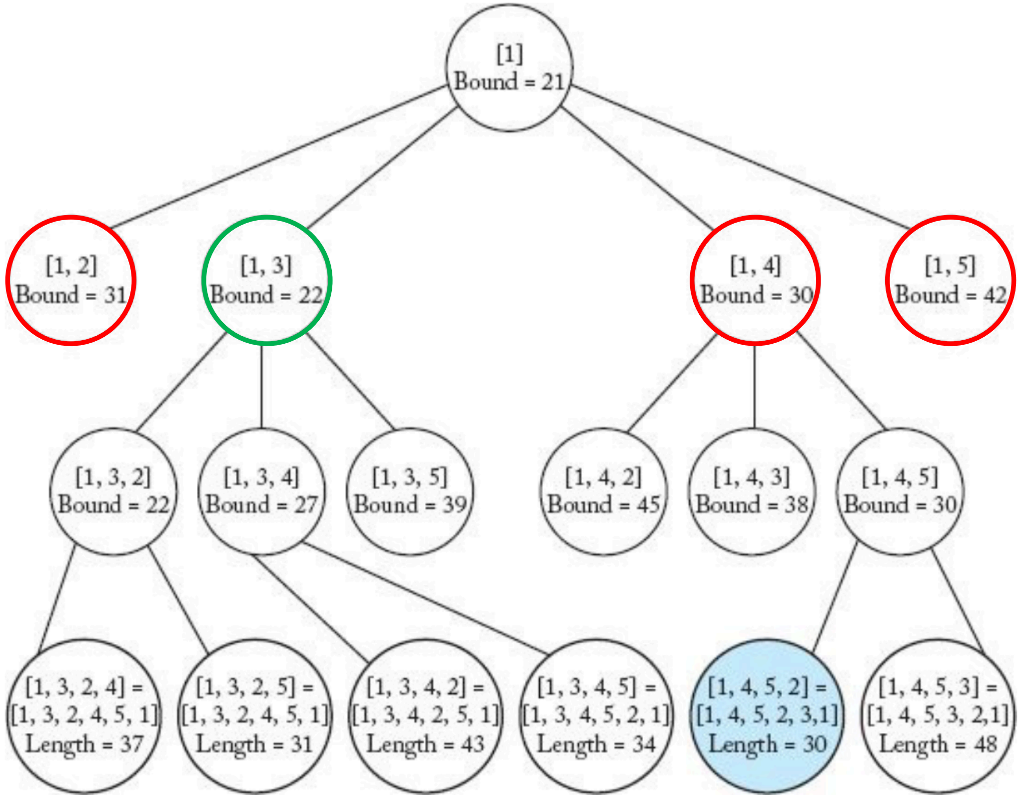


Image source: Figure 6.6, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pruned State Space Tree with Best-First Search

7. Visit node containing [1, 3, 2].
8. Visit node containing [1, 3, 4].
9. Visit node containing [1, 3, 5].
10. Determine promising, unexpanded node with the smallest bound.
    - Node containing [1, 3, 2] is selected. We visit its children.

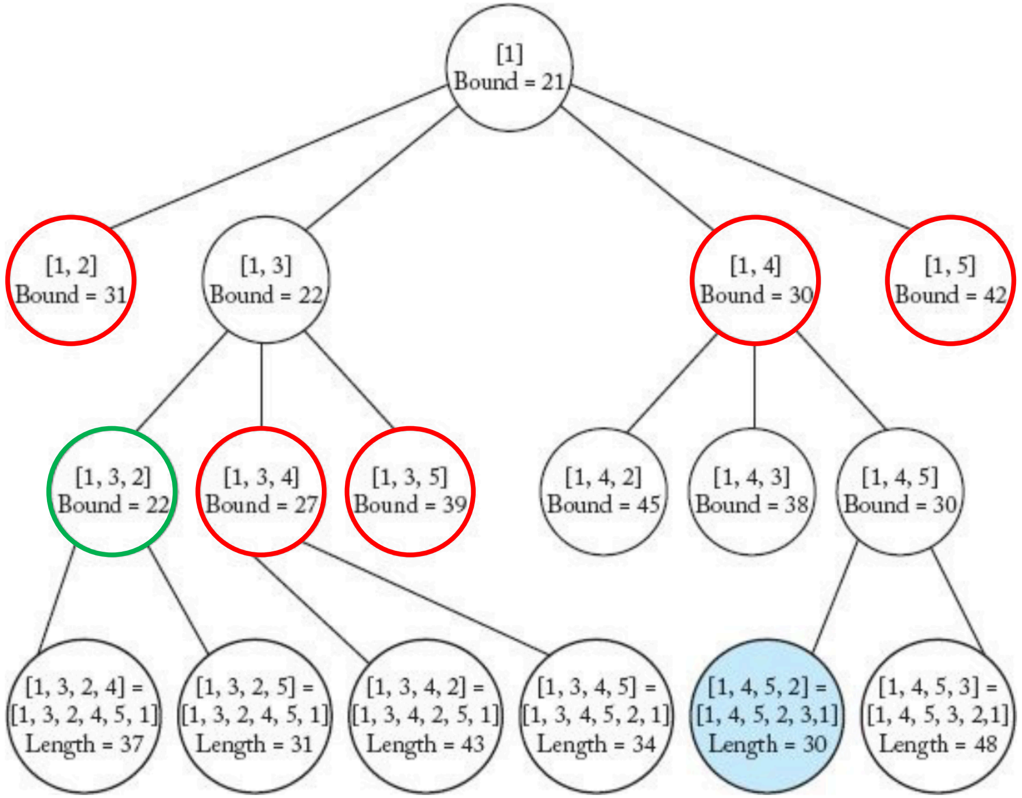


Image source: Figure 6.6, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pruned State Space Tree with Best-First Search

11. Visit node containing [1, 3, 2, 4].
    - Compute tour length, `minlength`=37.
12. Visit node containing [1, 3, 2, 5].
    - Compute tour length, `minlength`=31.
13. Determine promising, unexpanded node with the smallest bound.
    - Node containing [1, 3, 4] is selected. We visit its children.

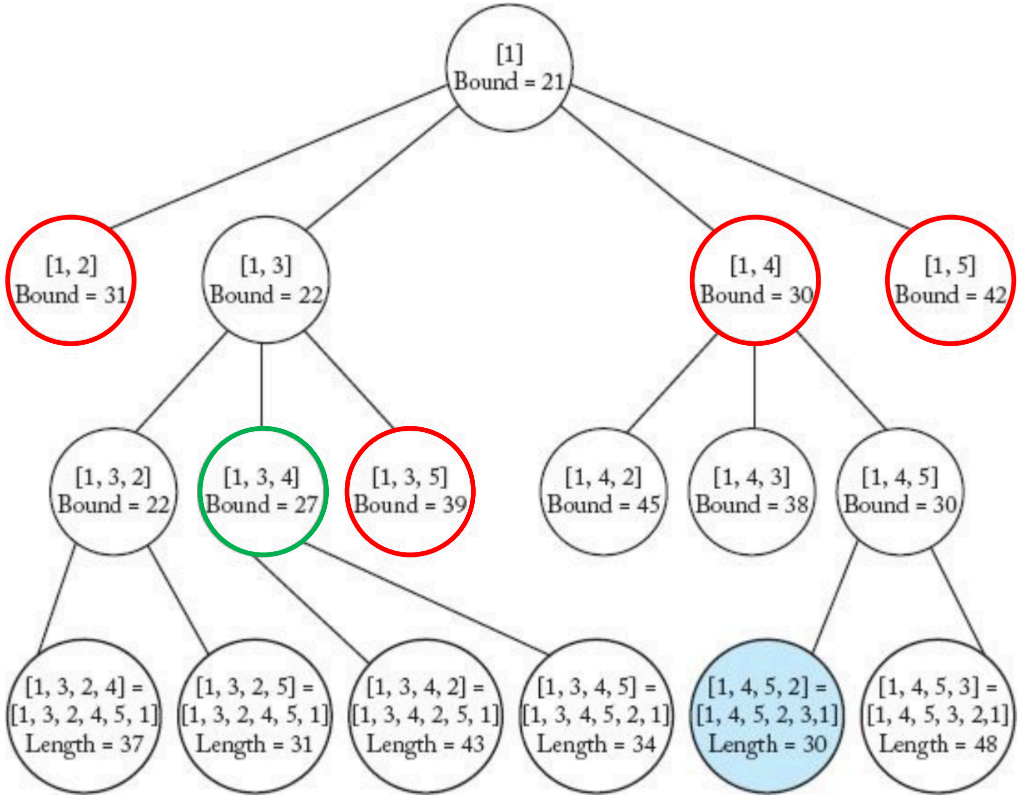


Image source: Figure 6.6, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pruned State Space Tree with Best-First Search

14. Visit node containing [1, 3, 4, 2].
    - Compute tour length, `minlength`=31.
15. Visit node containing [1, 3, 4, 5].
    - Compute tour length, `minlength`=31.
16. Determine promising, unexpanded node with the smallest bound.
    - Node containing [1, 4] is selected. We visit its children.

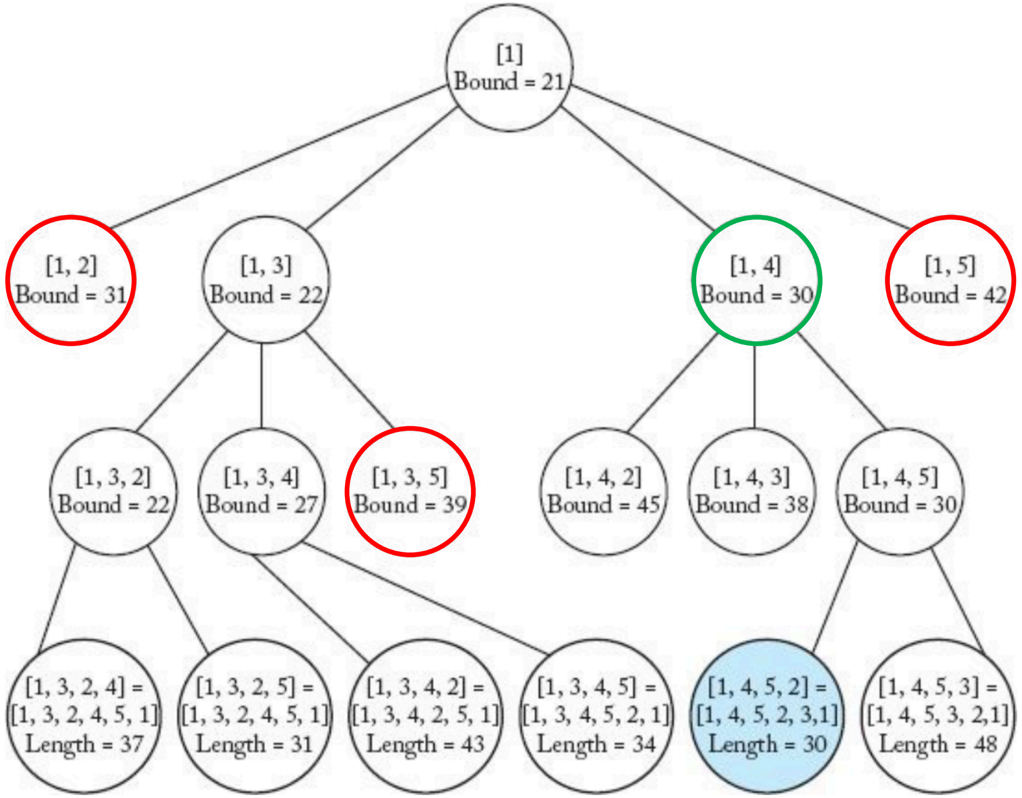


Image source: Figure 6.6, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pruned State Space Tree with Best-First Search

17. Visit node containing [1, 4, 2].
18. Visit node containing [1, 4, 3].
19. Visit node containing [1, 4, 5].
20. Determine promising, unexpanded node with the smallest bound.
    - Node containing [1, 4, 5] is selected. We visit its children.

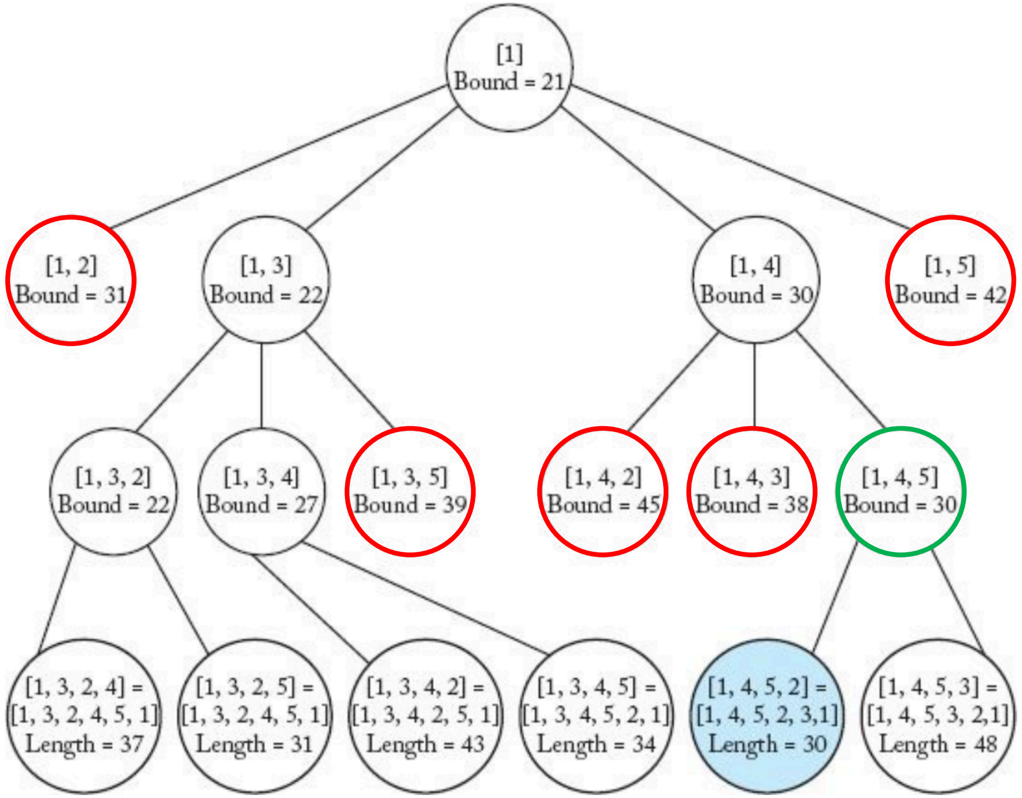


Image source: Figure 6.6, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pruned State Space Tree with Best-First Search

21. Visit node containing [1, 4, 5, 2].
    - Compute tour length, `minlength`=30.
22. Visit node containing [1, 4, 5, 3].
    - Compute tour length, `minlength`=30.
23. Determine promising, unexpanded node with the smallest bound.
    - There are no more promising, unexpanded nodes, because all the nodes have higher bound than `minlength`.

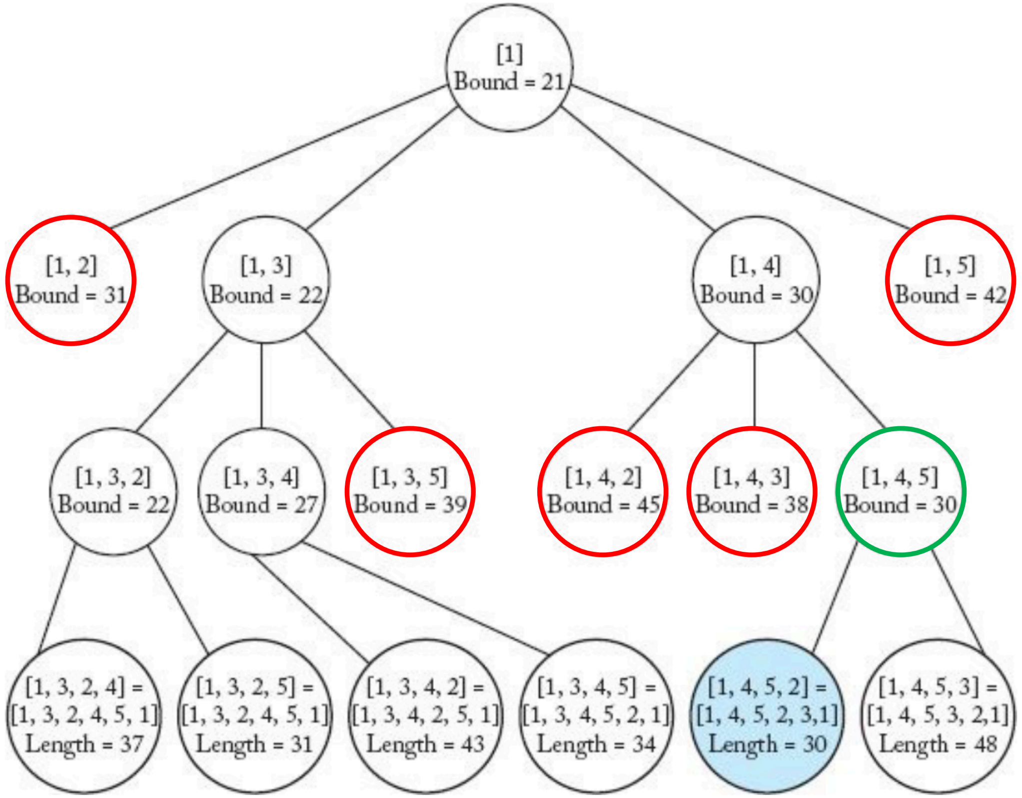


Image source: Figure 6.6, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pseudocode of Best-First Search Version of TSP

- Again, select the one who has the greatest hope!
  - The key idea of branch-and-bound.
- `bound` and `length` are easy to implement.

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

```
void travel (int n,
             cont number W[][],
             ordered_set& opt_tour,
             number& minlength)
{
    priority_queue PQ;
    node u, u_child;

    initialize(PQ);
    u.level = 0;
    u.path = [1];
    u.bound = bound(u);
    minlength = inf;
    enqueue(PQ, u);
    while (!empty(PQ)){
        u = dequeue(PQ);
        if (u.bound < minlength){
            u_child = u.level + 1;
            for (all i such that 2 <= i <= n && i is not in u.path){
                u_child.path = u.path;
                put i at the end of u_child.path;
                if (u_child.level == n - 2){
                    put index of only vertex not in u_child.path at the end of u.path;
                    put 1 at the end of u_child.path;
                    if (length(u_child) < minlength){
                        minlength = length(u_child);
                        opt_tour = u_child.path;
                    }
                }
                else{
                    u_child.bound = bound(u_child);
                    if (u_child.bound < minlength)
                        enqueue(PQ, u_child);
                }
            }
        }
    }
}
```

Conclusion

After this lecture, you should know:

- What is the difference between breadth-first search and best-first search.
- What is the difference between backtracking and branch-and-bound.
- What kind of problem that we can use branch-and-bound.
- How can we use the bound to eliminate unnecessary node checking.

Assignment

- No tutorial this week. Just implementing 0-1 knapsack problem by branch-and-bound in Python and submit to Attendance Quiz.
- Assignment 4 is released. The deadline is **18:00, 15th June**.

Thank you!

- Any question?
- Don’t hesitate to send email to me for asking questions and discussion. ☺

Acknowledgement: Thankfully acknowledge slide contents shared by Prof. Xuemin Hong