Unit 7.2 Branch and Bound

Algorithms

EE3980

Jun. 2, 2020

0/1 Knapsack Problem

- Given n objects, each with profit p_i and weight w_i, and a sack of maximum weight m, select the objects to be placed into the sack such that the profits of the objects in the sack is maximum. (Note that the object must be placed as a whole, no fraction, into the sack.)
- Recall that the greedy algorithm that allows the fraction of an object to be placed into the sack generate the optimal solution (maximal profits).

Algorithm 4.1.5. Knapsack

```
// n objects with w[i] and p[i] find x[i] that maximizes ∑ p_i x_i with ∑ w_i x_i ≤ m.
// Input: m, n, w[], p[]
// Output: solution vector x[].
Algorithm Knapsack(m, n, w, p, x)
{
    a := Sort(p/w) ; // sort p[a[i]]/w[a[i]] into non-increasing order.
    for i := 1 to n do x[i] := 0 ;
    i := 1 ;
    while (i ≤ n and w[a[i]] ≤ m) do {
        x[i] := 1 ;
        m := m − w[a[i]] ;
        i := i + 1 ;
    }
    if (i ≤ n) then x[i] := m/w[a[i]] ;
}
```

0/1 Knapsack Problem, Bounds

- Note that on line 9 the last object included might be a fraction which violate the requirement of a whole object.
- Thus, excluding this line the profit $p = \sum_{j=1}^{i} p_j$ is the least one can get for the profit.
  - We can use this p as a lower bound (lb)for the profits.
- The profits, P, with the fraction object is the maximum and can be used as the upper bound (ub).
- Thus, assuming the objects are ordered by p/w, the following function generates two bounds for the set of objects

0/1 Knapsack Problem, Bounds Algorithm

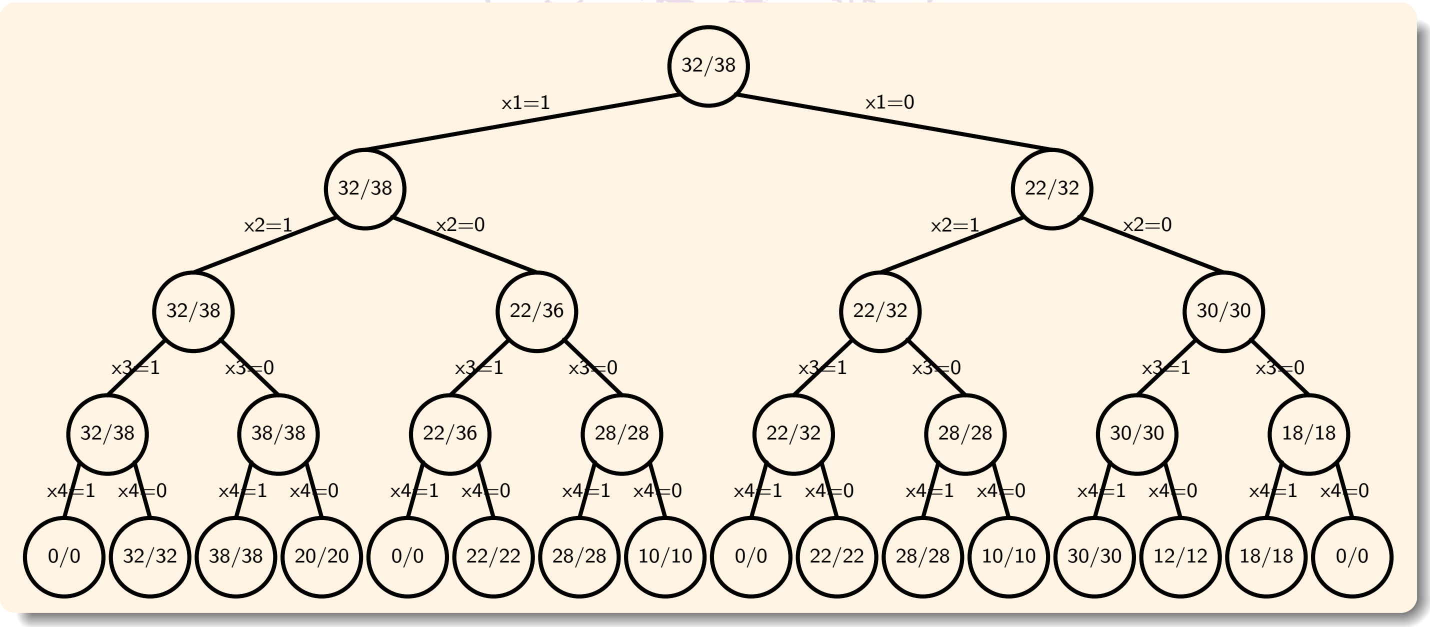
Algorithm 7.2.1. Bounds

```
   // Estimate two bounds lb and ub for n-object 0/1 knapsack problem
   // Input: k, cw c weight, cp c profit
   // Output: lb lower bound, ub upper bound.
 1 Algorithm Bounds(k, cw, cp, lb, ub)
 2 {
 3       i := k + 1 ;
 4       lb := cp ;
 5       while (i ≤ n and cw ≤ m) do {
 6             lb := lb + p[i] ;
 7             cw := cw + w[i] ;
 8             i := i + 1 ;
 9       }
10       if (i > n) then ub := lb ;
11       else ub := lb + (1 − (cw − m)/w[i]) ∗ p[i] ;
12 }
```

- The above algorithm has been generalized such that the decision on the first k objects have been made and cp and cw are the current profits and weights for the first k objects.
- The algorithm estimate the two bounds for the remaining $n - k$ objects.
- Note that arguments lb and ub need to be passed by reference (in C++), or passed by pointer (in C).

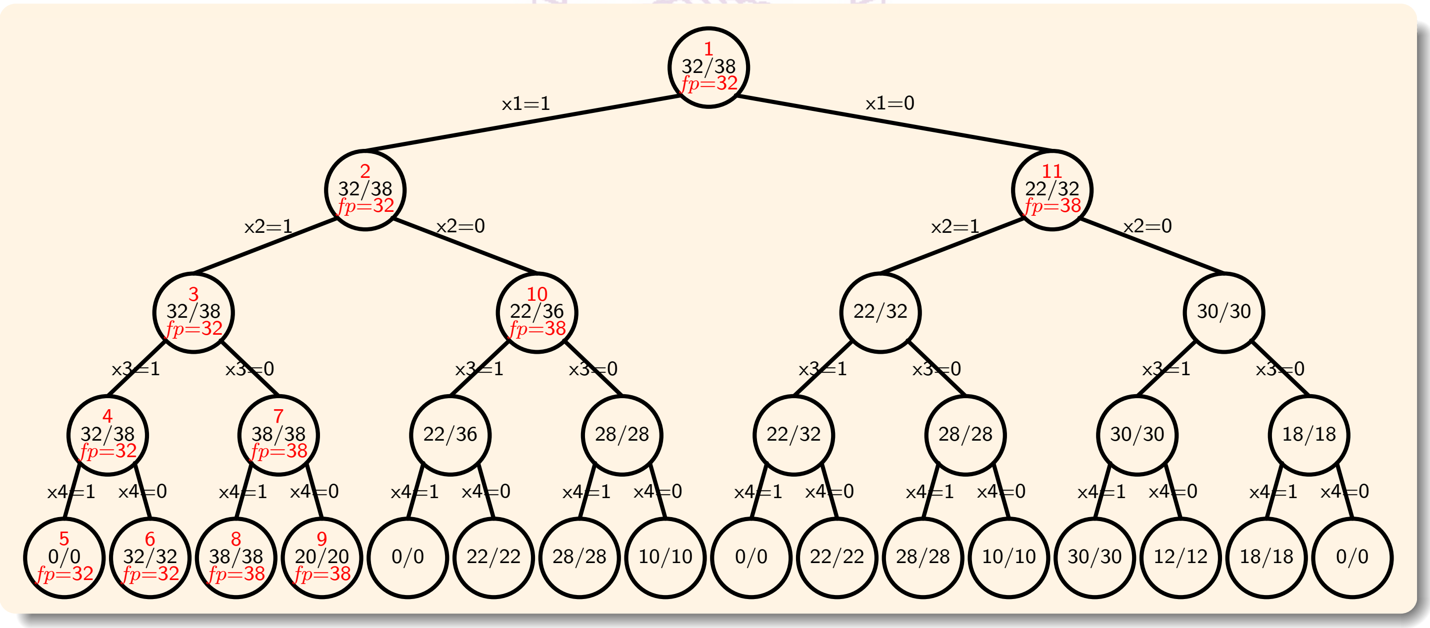
0/1 Knapsack Problem Example

- 0/1 knapsack problem example:
  $n = 4,\ p = (10, 10, 12, 18),\ w = (2, 4, 6, 9),\ m = 15.$
- Complete state space can be shown to be



0/1 Knapsack Problem Example — Depth First

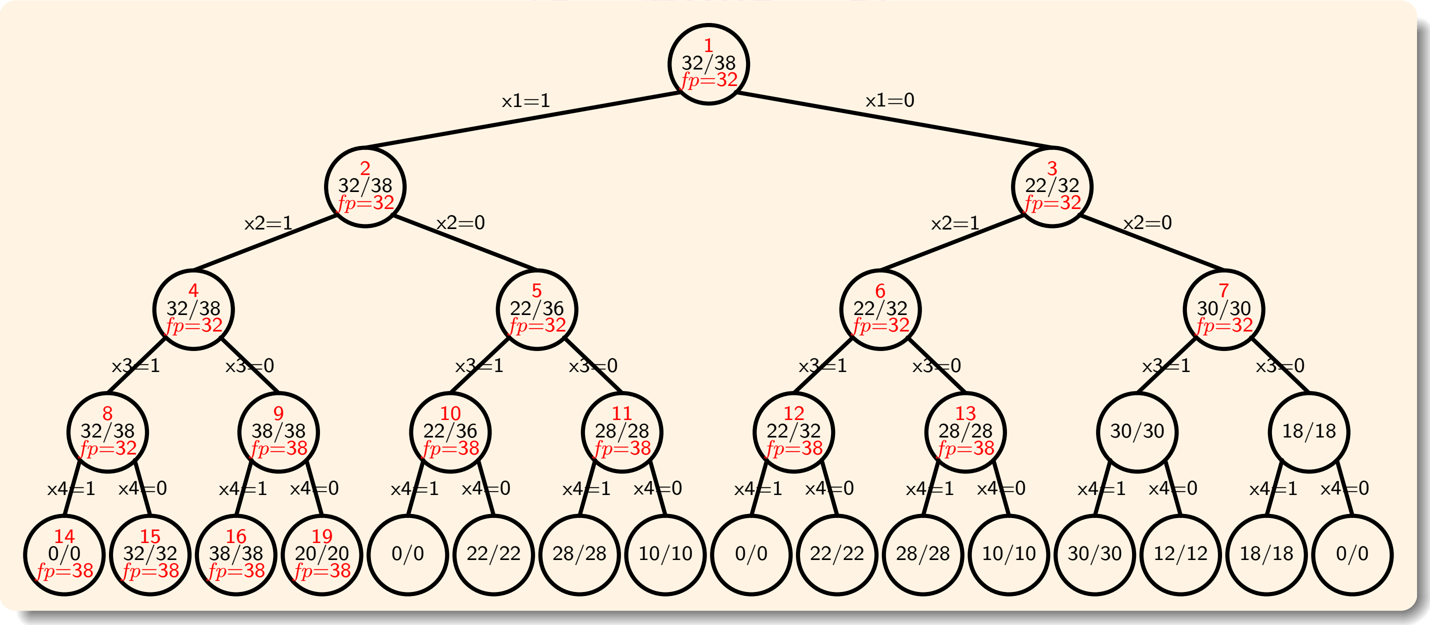
- 0/1 knapsack problem example:
  $n = 4,\ p = (10, 10, 12, 18),\ w = (2, 4, 6, 9),\ m = 15.$
- Using depth-first traversal branch and bound approach, we have



- Branch and bound stops after 11 steps
- The solution is $x = (1, 1, 0, 1),\ fp = 38,\ fw = 15.$

0/1 Knapsack Problem Example — Breadth First

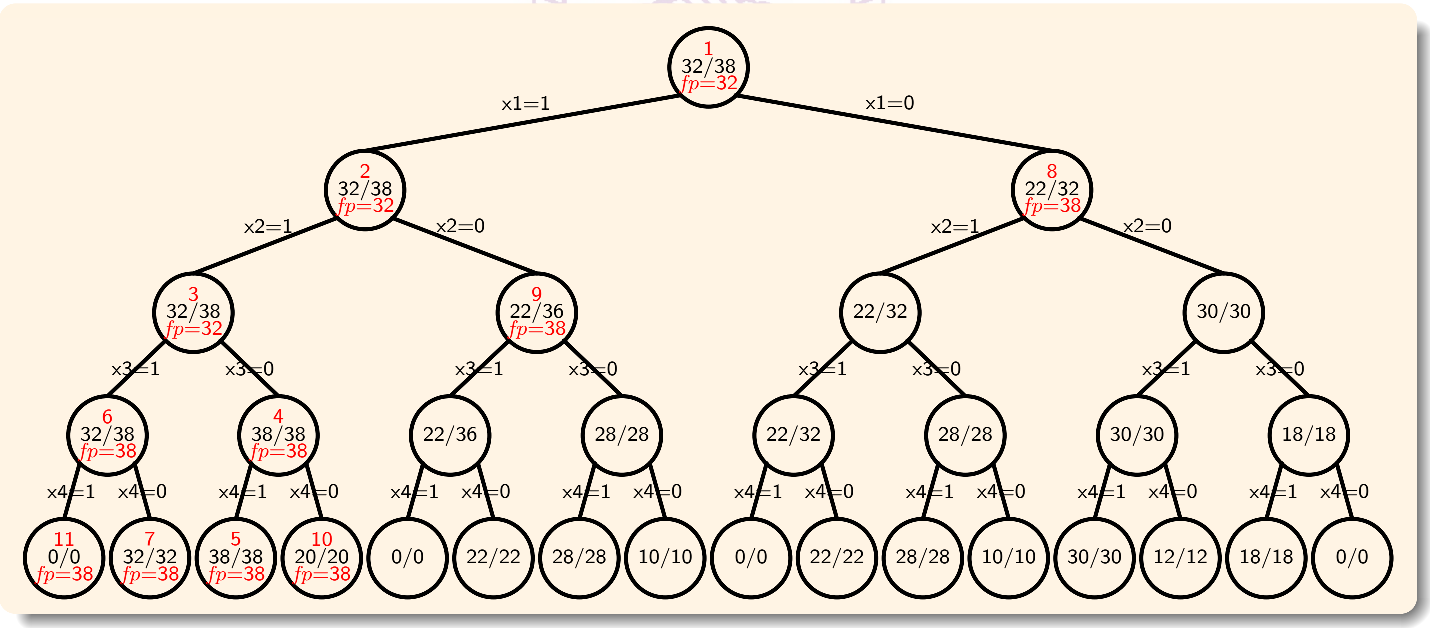
- 0/1 knapsack problem example:
  $n = 4,\ p = (10, 10, 12, 18),\ w = (2, 4, 6, 9),\ m = 15.$
- Using breadth-first traversal branch and bound approach, we have



- Branch and bound stops after 19 steps
- The solution is $x = (1, 1, 0, 1),\ fp = 38,\ fw = 15.$

0/1 Knapsack Problem Example — Least Cost

- 0/1 knapsack problem example:
  $n = 4,\ p = (10, 10, 12, 18),\ w = (2, 4, 6, 9),\ m = 15.$
- Using Least-cost branch and bound approach, we have



- Branch and bound stops after 11 steps
- The solution is $x = (1, 1, 0, 1),\ fp = 38,\ fw = 15.$

Branch and Bound Algorithms

- Branch and bound method is applicable to all state space search methods.
  - All children of a search node are generated before any other live node is explored.
  - Bounding functions are used to help reducing the number of subtrees to be explored.
- Two tree traversal algorithms are applicable to explore the state space.
  - Breadth-first search: also known as first-in-first-out (FIFO) strategy.
    - Need a stack to keep the live nodes.
  - Depth-first search: also known as the last-in-first-out (LIFO) strategy.
- An additional strategy least cost search has been introduced.
  - Each node is associated with a cost that estimates the solution cost.
  - To select the next node to explore, select one with the least cost.
- The following algorithm is a high level description of the LC-search approach.
- The LC-search algorithm uses the following structure.

```
struct listnode {
      double cost , lb , ub ; // cost and estimated lower and upper bounds
      struct listnode *next, *parent;
}
```


Branch and Bound Algorithms — LC Search

Algorithm 7.2.2. LC Search

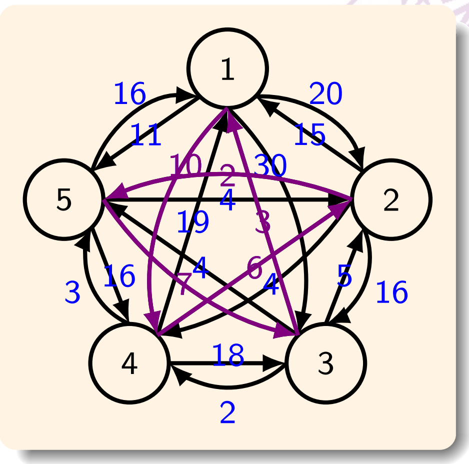
```
// General framework for least cost search.
// Input: tree with root t
// Output: solution path.
Algorithm LCSearch(t)
{
      if t is an answer node then {
            write (t) ;
            return ;
      }
      E := t; // Current search node.
      Initialize the list of live nodes to be empty ;
      while (E ≠ ∅) do {
            for each child x of E do {
                  if x is an answer node then {
                        write ( path from x to t) ;
                        return ;
                  }
                  Add(x) ; // x is a new live node.
                  x → parent := E;
            }
            if there are no live nodes then {
                  write ("No answer.") ;
                  return ;
            }
            E := Least() ;
      }
}
```

Branch and Bound Algorithms — General

- In the above algorithm, two functions are used
  - `Add`: add a new live node to the list.
  - `Least`: find the minimum cost node from the live node list and remove it from the list.
- The `list` data structure is used for LCS for searching of least cost node is needed. In contrast,
  - DFS uses `stack` (LIFO),
  - BFS uses `queue` (FIFO).
  - Selecting the next live node is more consuming in LCS approach.
- All three search approaches can be used in branch-and-bound method.
- For each E-node, in addition to the cost c two more estimates are calculated: a lower bound lb and an upper bound ub.
- In exploring each node, the best cost fc is also tracked.
- Thus, when exploring node E if $lb > fc$ then there is no need to traverse the subtree of E.
  - And, in selecting E node, the one with the minimum lb should be selected.
- By reducing the number of subtrees to be explored, the branch-and-bound algorithm can be fast.

Traveling Salesperson Problem

- Let $G = (V, E)$ be a directed graph, with $|V| = n$ and c_{ij} be the cost of edge $\langle i, j\rangle \in E$, $c_{ij} = \infty$ if $\langle i, j\rangle \notin E$.
- Without loss of generality, we can assume every tour start from vertex 1. So, the solution space is $S = \{(1, \pi, 1) | \pi \text{ is a permutation of } (2, 3, \cdots, n)\}$.
- Of course, for any solution $(1, i_1, i_2, \cdots, i_{n-1}, 1) \in S$, $\langle i_j, i_{j+1}\rangle \in E$, $0 \le j \le n-1$ and $i_0 = i_n = 1$.
- The objective is to find a path with the minimum cost.
- Traveling salesperson problem example



- Cost matrix

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

Traveling Salesperson Problem — Reduced Cost Matrix

- Given a cost matrix, it can reduced as following.
- Note that $c_{i,j}$ is the cost from vertex i to vertex j
  - Thus, if $c_{i,k} = \min_{j=1}^{n} c_{i,j}$, then $c_{i,k}$ is the minimum cost leaving vertex i.
  - And, if $c_{k,j} = \min_{i=1}^{n} c_{i,j}$, then $c_{k,j}$ is the minimum cost entering vertex j.

- Original cost matrix

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{array}{l} \text{row } 1 - 10 \\ \text{row } 2 - 2 \\ \text{row } 3 - 2 \\ \text{row } 4 - 3 \\ \text{row } 5 - 4 \end{array}$$

- Row-reduced cost matrix

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

- Row- and Column-reduced cost matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

- Column 1 is reduced by 1, and column 3 reduced by 3 are performed.
- The total reduction $R = 10+2+2+3+4+1+3 = 25$ is the lower bound for for the salesperson traveling problem.

Traveling Salesperson Problem — Reduced Cost Matrix, II

- The technique of reduced cost matrix to estimate the lower bound of the traveling salesperson problem can be extended to estimating path selection.
- Suppose an edge $\langle i, j \rangle$ is selected, the cost of the path is increased by $c_{i,j}$
  - All other edges $\langle i, k \rangle$, $k \neq j$ cannot be selected. Thus, set $c_{i,k} = \infty$, $1 \leq k \leq n$. (Row i)
  - All edges $\langle k, j \rangle$, $k \neq i$, cannot be selected. Thus, set $c_{k,j} = \infty$, $1 \leq k \leq n$. (Column j)
  - The edge $\langle j, 1 \rangle$ cannot be selected (unless j is the only vertex not selected). Thus, set $c_{j,1} = \infty$.
  - Perform reduced matrix technique to the resulting matrix to get the lower bound, r.
  - Then the lower bound of path cost of selecting edge $\langle i, j \rangle$ is $R + c_{i,j} + r$, where R is the lower bound before selecting edge $\langle i, j \rangle$.

Traveling Salesperson Problem — Reduced Cost Matrix, III

- Example

- Original cost matrix

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

- Selecting edge $\langle 1, 3 \rangle$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

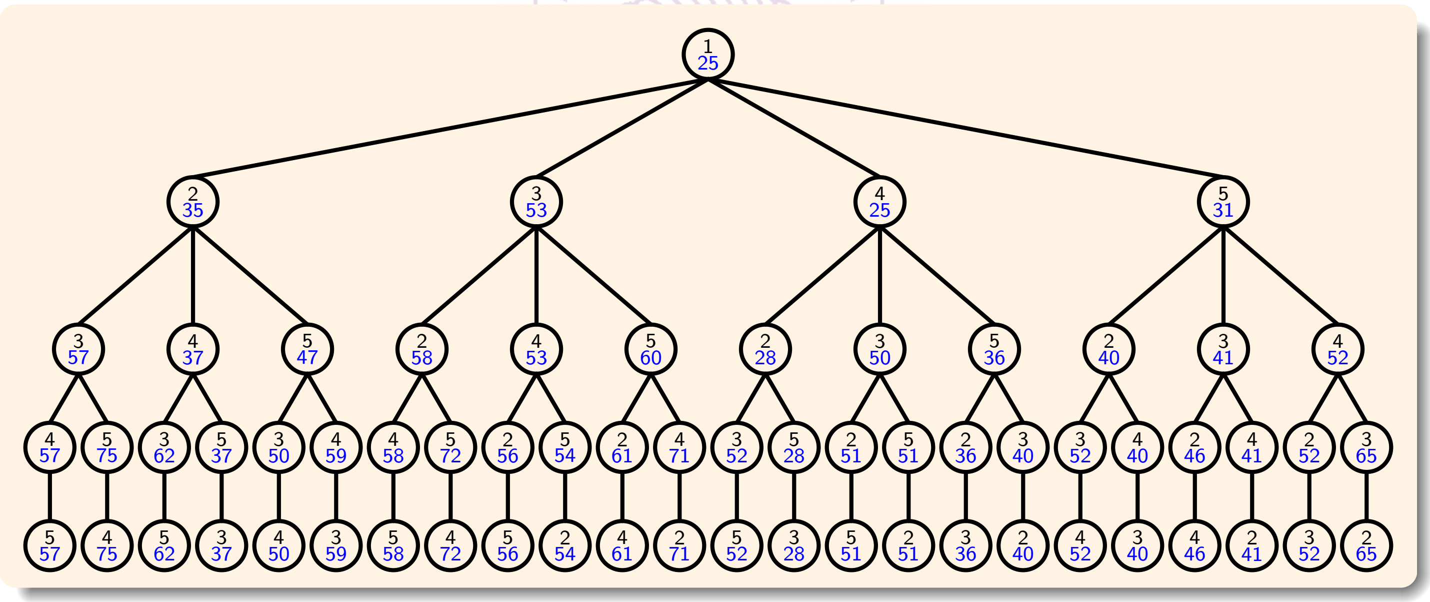
- Cost-reduced cost matrix, $R = 25$.

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

- $c_{1,3} = 17$.
- Row 1 is set to ∞
- Column 3 is set to ∞
- $c_{3,1}$ is set to ∞
- Then column 1 can be reduced by 11. $(r = 11)$
- The lower bound for selecting edge $\langle 1, 3 \rangle$ is $R + c_{1,3} + r = 25 + 17 + 11 = 53$.

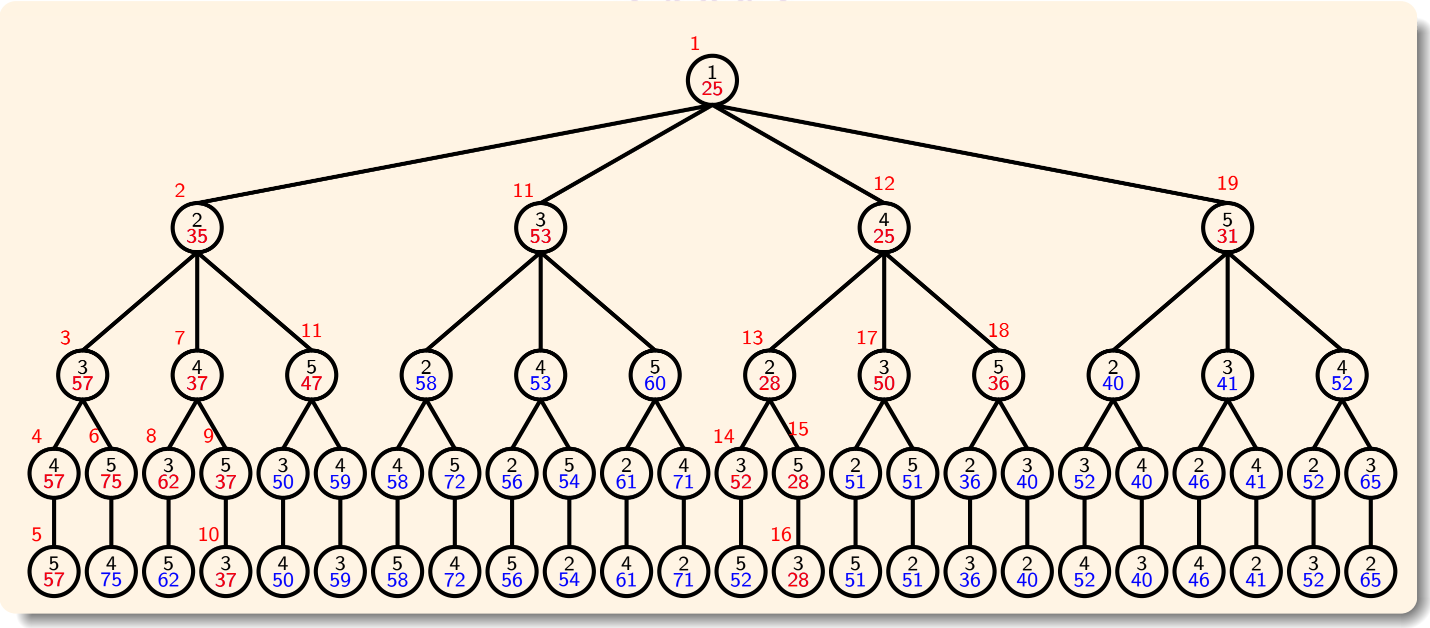
Traveling Salesperson Problem

- The full state space is shown below



Traveling Salesperson Problem — Depth-First Search BB

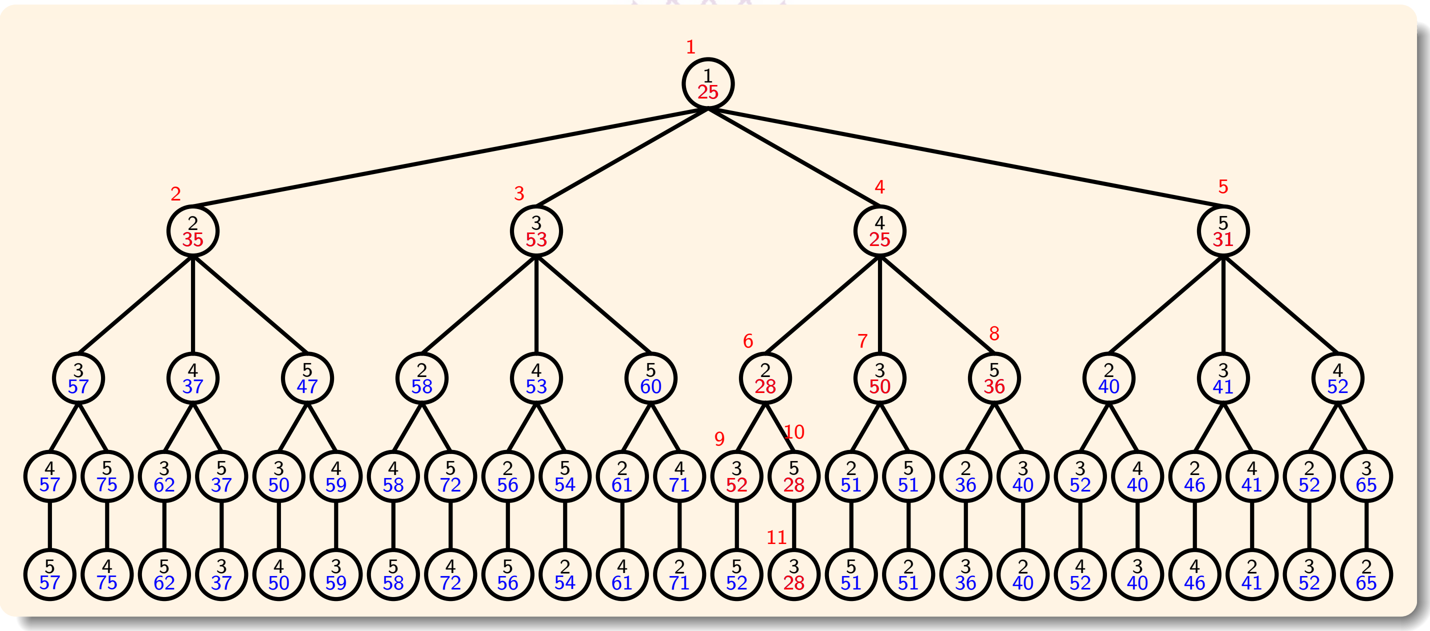
- Using depth-first search with BB, we have



- DFS BB stops in 19 steps
- The solution is $1 - 4 - 2 - 5 - 3 - 1$, total cost is 28.

Traveling Salesperson Problem — Least-Cost Search BB

- Using least-cost search with BB, we have



- LCBB stops in 11 steps
- The solution is $1 - 4 - 2 - 5 - 3 - 1$, total cost is 28.

Theories

- Some theories concerning branch-and-bound approaches.

Theorem 7.2.3.

Let t be a state space tree. The number of nodes of t generated by FIFO, LIFO and LC branch-and-bound algorithms cannot be decreased by the expansion of any node x with $lb(x) >= upper$, where $upper$ is the upper bound on the cost of a minimum-cost solution node in the tree t.

Theorem 7.2.4.

Let U_1 and U_2, $U_1 < U_2$, be two initial upper bounds on the cost of a minimum-cost solution node in the state space tree t. The FIFO, LIFO, and LC branch-and-bound algorithms beginning with U_1 will generate no more nodes than they would if they started with U_2 as the initial upper bound.

Theorem 7.2.5.

The use of a better lower bound function lb in conjunction with FIFO and LIFO branch-and-bound algorithms does not increase the number of nodes generated.

Theories, II

Theorem 7.2.6.

If a better lower bound function is used in a LC branch-and-bound algorithm, the number of nodes generated may increase.

Theorem 7.2.7.

The number of nodes generated during FIFO and LIFO branch-and-bound search for a least-cost solution the number of nodes generated may increase when a stronger dominance relation is used.

Theorem 7.2.8.

Let D_1 and D_2 be two dominance relations. Let D_2 be stronger than D_1 such that $(i, j) \in D_2$, $i \neq j$, implies $lb(i) < lb(j)$. An LC branch-and-bound using D_1 generates at least as many nodes as the one using D_2.

Branch and Bound

- Branch and bound methods belong to the all state space search method.
- To avoid extensive searching of all states, bounding functions for lower bound and upper bound are keys.
- Accurate bounding functions can decrease the state space that needs to be searched.
- Three traversal techniques can be used to explore the state space – depth first search, breadth first search and least cost search.
- Least cost searching is shown to be effective in some problems.
- With good bounding function and effective traversal method, branch and bound can solve real problems with significant time saving.

Summary

- 0/1 knapsack problem
- Branch-and-bound algorithms
- Least-cost branch-and-bound
- The travelling salesperson problem
- Theories on branch-and bound algorithms