Branch & Bound

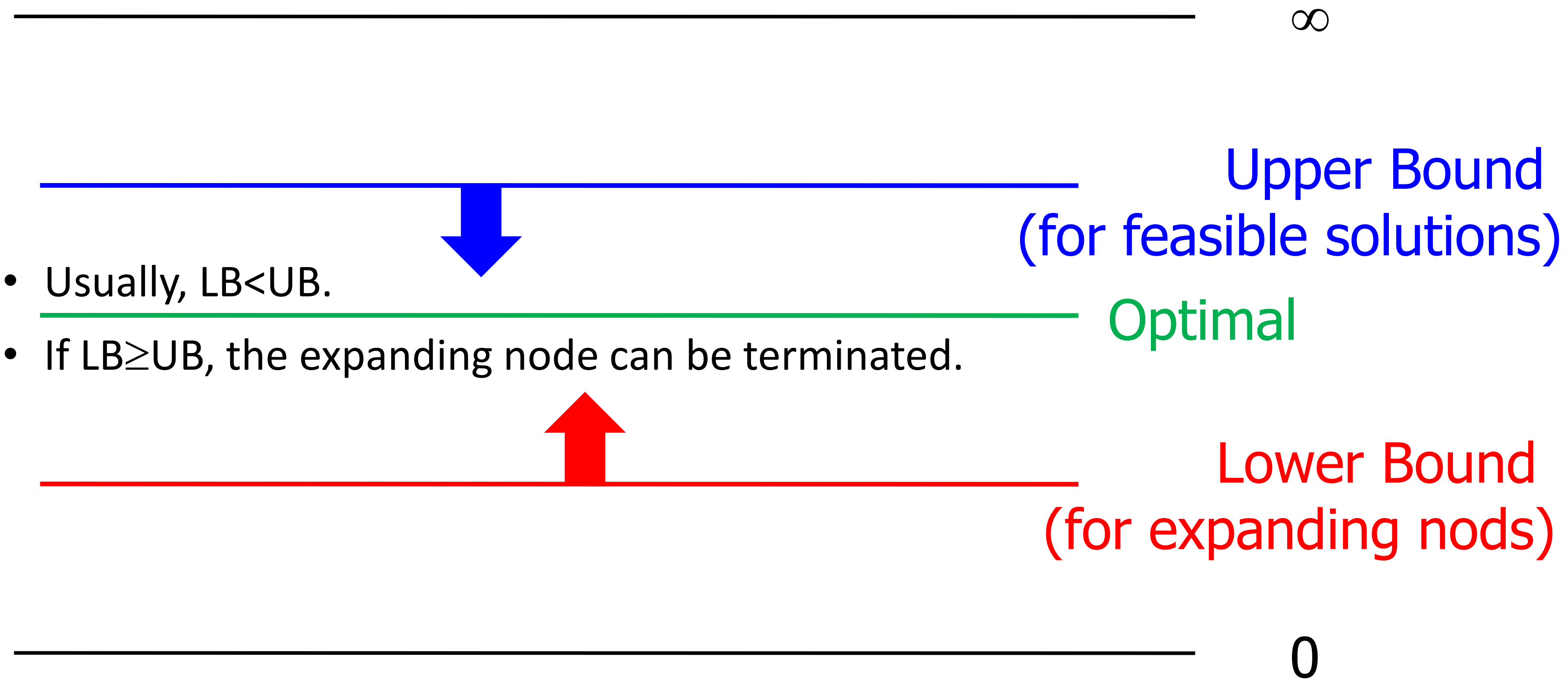
Introduction

- Branch and Bound is a state space search method in which all the children of a node are generated before expanding any of its children (Possibly BFS).
- It is similar to backtracking technique but uses BFS-like search.
- Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node .
- Systematic enumeration of all candidate solutions, discarding large subsets of fruitless candidates by using upper and lower estimated bounds of quantity being optimized.
- Least-cost branch and bound directs the search to parts of the space most likely to contain the answer. So it could perform better than backtracking.
- Used to find optimal solution to many optimization problems.

Introduction

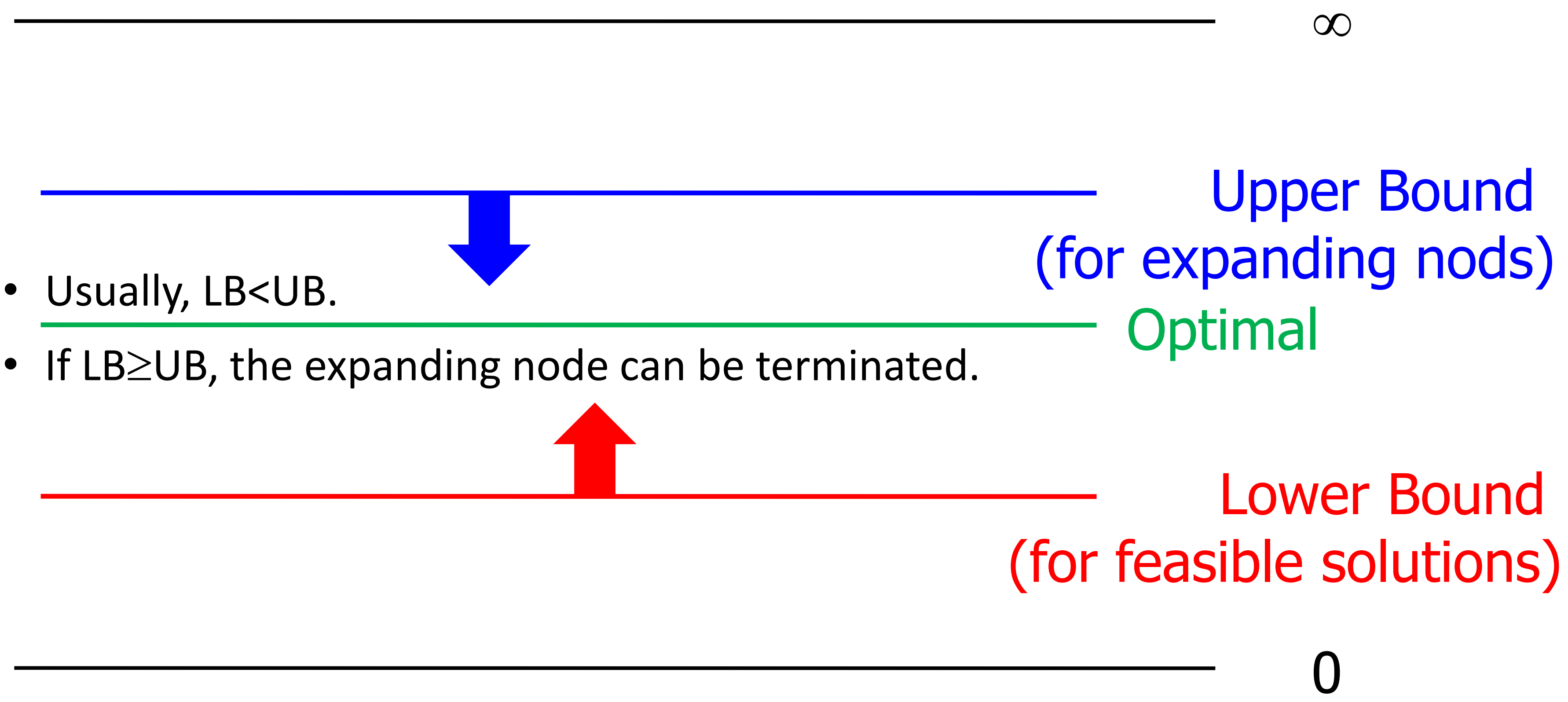
- Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems.
- These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.
- On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average.
- **Combinatorial optimization problem** is an optimization problem, where an optimal solution has to be identified from a finite set of solutions.
- B&B is a rather general optimization technique that applies where the greedy method and dynamic programming fail.
- The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded (i.e., their children generated). Rather, a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found.

For Minimization Problems



- Usually, LB<UB.
- If LB≥UB, the expanding node can be terminated.

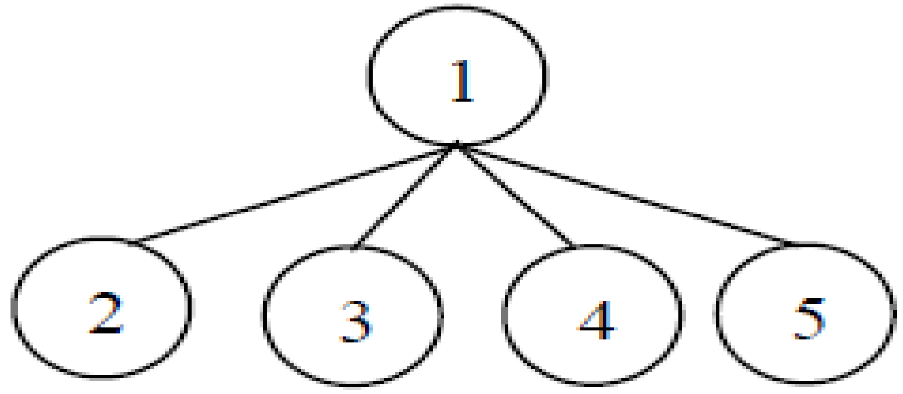
For Maximization Problems



- Usually, LB<UB.
- If LB≥UB, the expanding node can be terminated.

Terminology

- **Live node:** is a node that has been generated but whose children have not yet been generated.
- **E-node (expanded node)**: is a live node whose children are currently being explored (or, expanded).
- **Dead node**: is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.



Live Node: 2, 3, 4, and 5

Bounding Functions are used to kill live nodes without generating all their children.

General Method

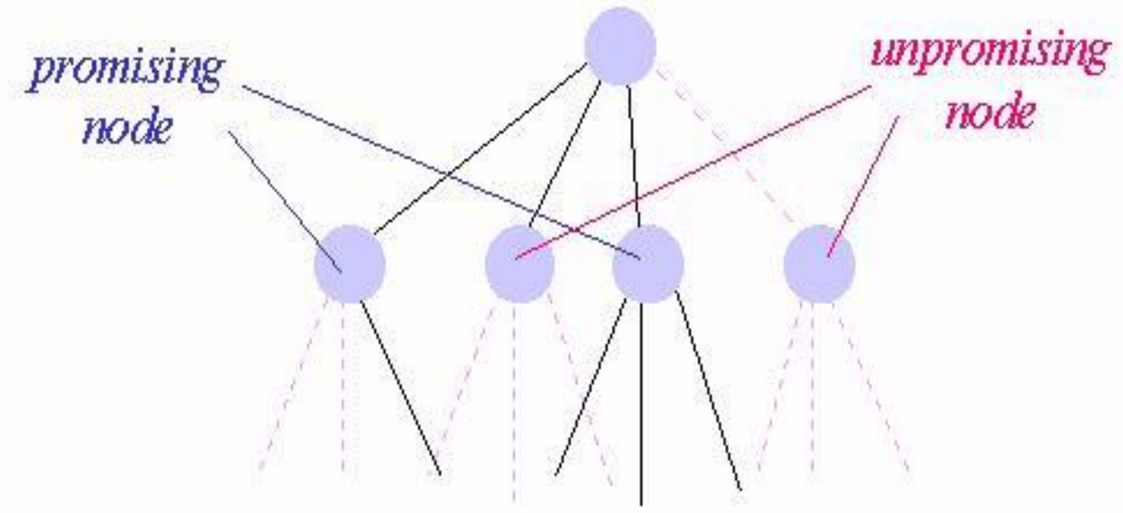
- ❑Start by considering the root node and applying a lower-bounding and upper-bounding procedure to it.
- ❑If the bounds match, then an optimal solution has been found and the algorithm is finished.
- ❑If they do not match, then algorithm runs on the child nodes.

Elements of B&B

- Branching:
  - A set of solutions, which is represented by a node, can be partitioned into mutually exclusive sets. Each subset in the partition is represented by a child of the original node.
- Bound:
  - An essential element of B&B is a greedy way to estimate the value of choosing one node over another. This is called the **bound or bounding heuristic.**

Elements of B&B

- A B&B algorithm computes 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 the state space tree beyond the current node.
- When performing a breadth-first traversal, the bound is used only to prune the unpromising nodes from the state-space tree.
- The approach is to check all the nodes reachable from the currently active node (breadth-first) and then to choose the most promising node (best-first) to expand next.



B&B strategies

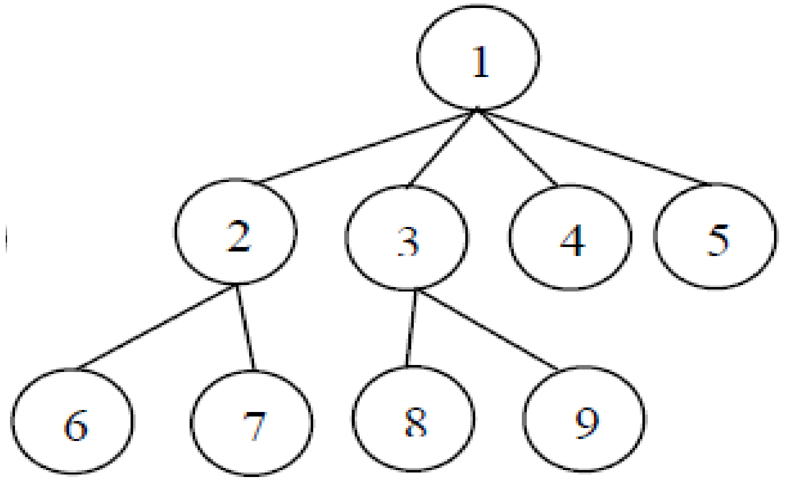
FIFO Branch & Bound (or BFS) : BFS is an FIFO search in terms of live nodes.

∗ List of live nodes is a queue.

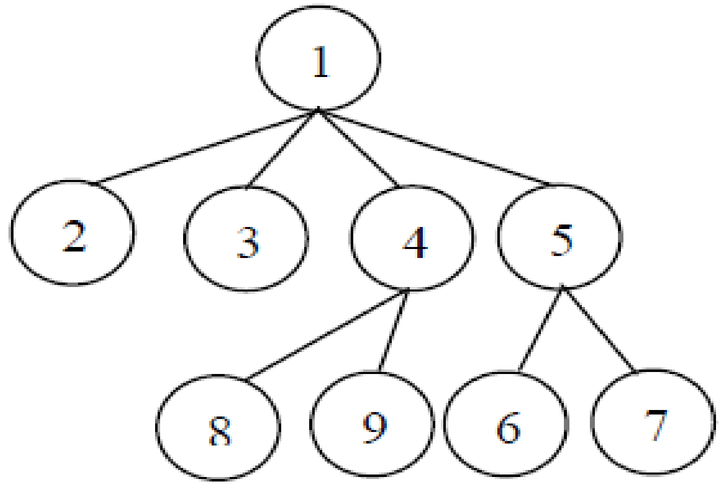
LIFO Branch & Bound (or D-Search): – DFS is an LIFO search in terms of live nodes

∗ List of live nodes is a stack.

- **Least Cost Search (LC-Search)** : Uses Approximate Cost Function



FIFO Branch & Bound (BFS)
Children of E-node are
inserted in a queue.



LIFO Branch & Bound (D-Search)
Children of E-node are inserted in a
stack.

LC-Search (Least Cost Search):

- The selection rule for the next E-node in FIFO or LIFO branch-and-bound is sometimes “blind”.

  i.e. the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

- The search for an answer node can often be speeded by using an “intelligent” ranking function, also called **an Approximate Cost Function.**
- Each node, X, in the search tree is associated with a cost: C(X)
- C(X) = cost of reaching the current node, X (E node), from the root + the cost of reaching an answer node from X.

$$\mathbf{C(X) = g(X) + h(X)}$$

- Get an approximation of C(x), C ^ (x) such that C ^ (x) ≤C(x), and C ^ (x) = C(x) if x is a solution-node.
- The approximation part of C ^ (x) is h(x)=the cost of reaching a solution-node from X, not known.
- Least-cost search: The next E-node is the one with least C ^

LC-Search (Example):

Initial State

1	2	3
5	6	
7	8	4

$\longrightarrow$

Final State

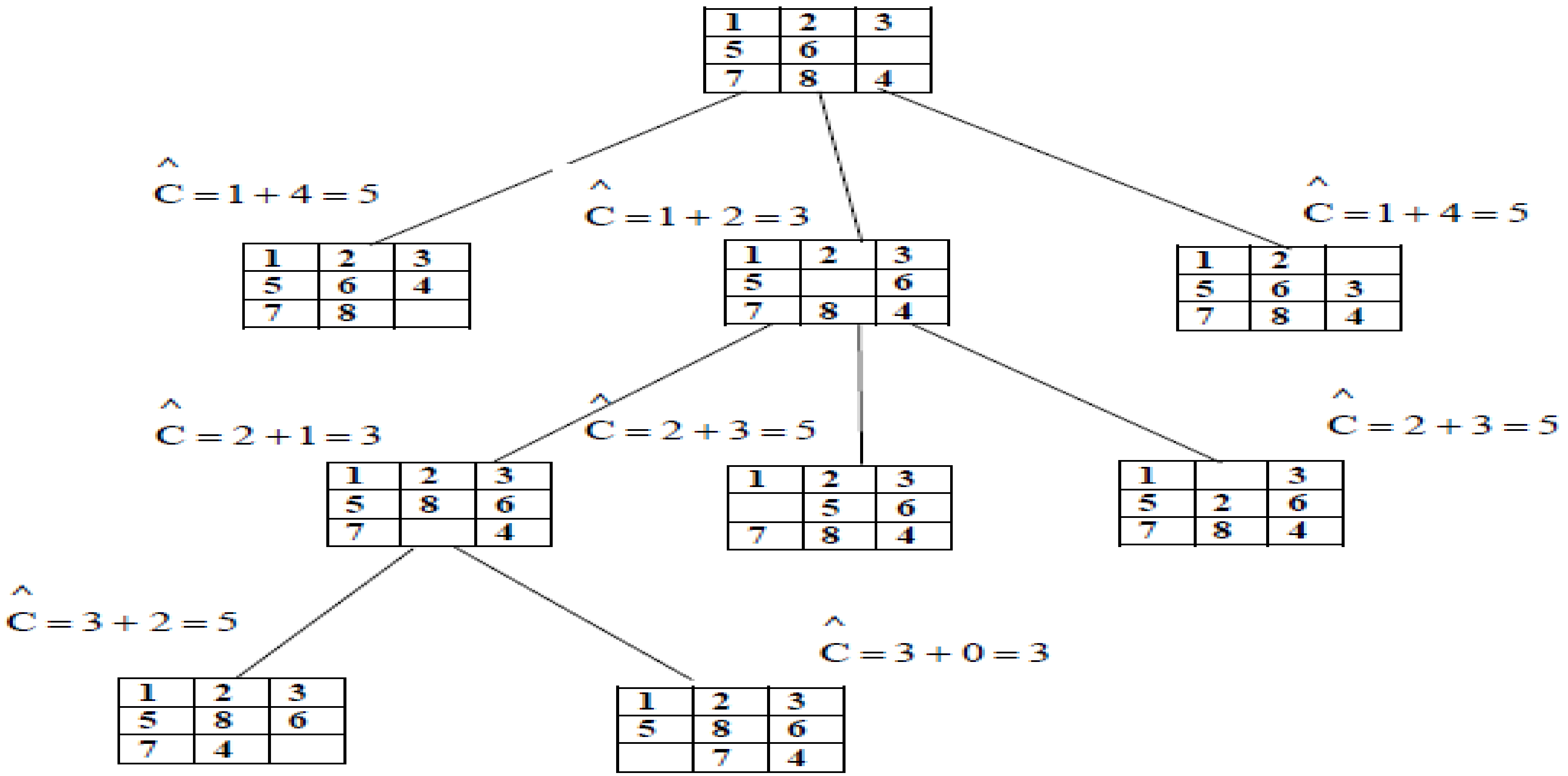
1	2	3
5	8	6
	7	4

- 8-puzzle :

Cost function: C ^ = g(x) +h(x)

where h(x) = the number of misplaced tiles and g(x) = the number of moves so far

- Assumption: move one tile in any direction cost 1.



Binary Knapsack problem

Input: Weight of N items $\{w_1, w_2, ..., w_n\}$
Knapsack limit to maximum Capacity S

Output: Selection for knapsack: $\{x_1,x_2,\ldots x_n\}$ where $x_i \in \{0,1\}$.

Bound: Upper bound for the revenue
we greedily grab items, adding their profits to bound and their weights to total weight until we get to an item that if grabed would bring total weight above W.

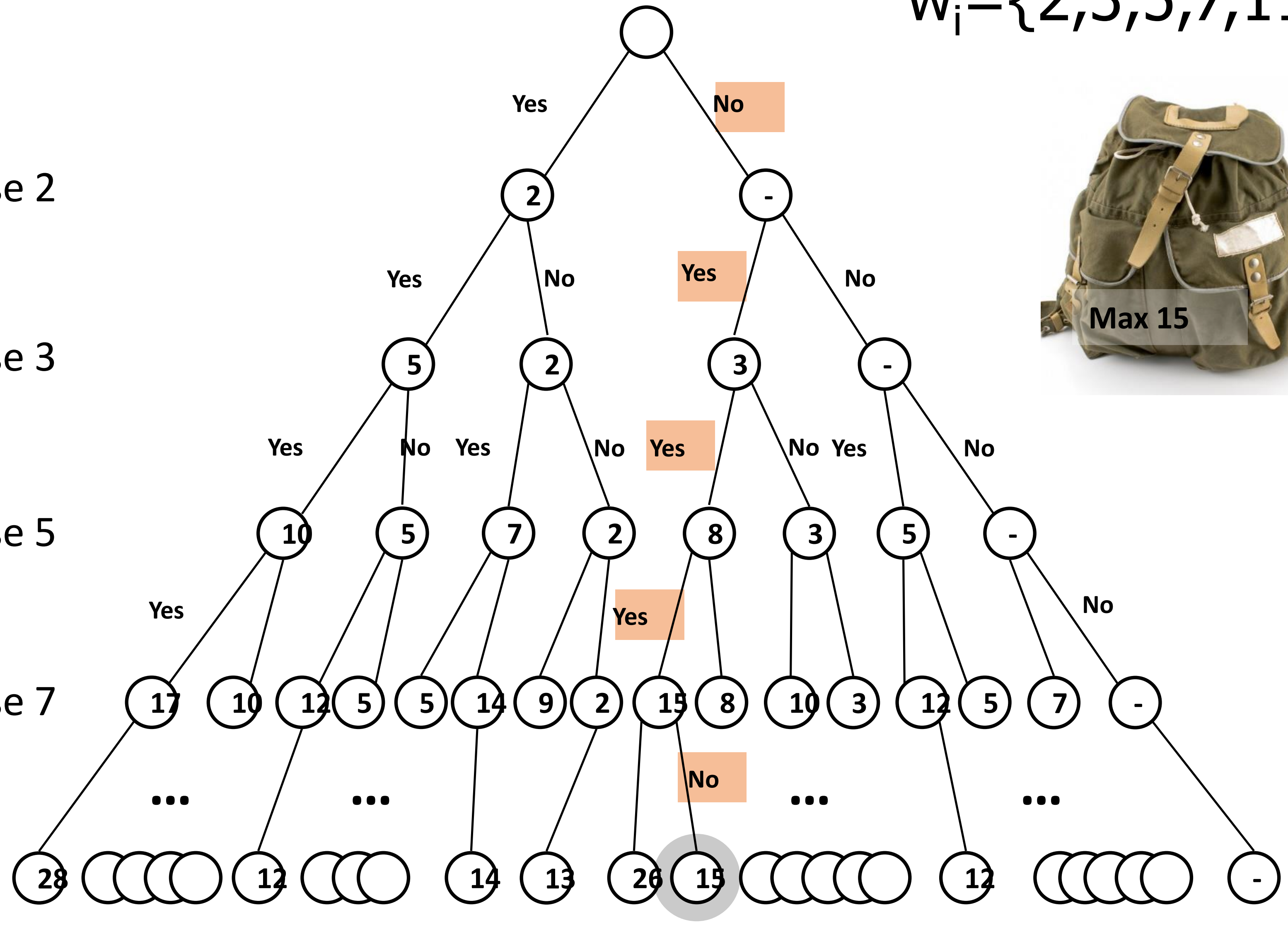
$$bound = (profit + \sum_{j=i+1}^{k-1} p_j) + (W - totalweight) \times \frac{p_k}{w_k}$$

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

Where, **weight**: the sum of the weights of the items that have been included
profit: the sum of the profits of the items included up to the node

Knapsack as decision tree

$w_i=\{2,3,5,7,11\}$



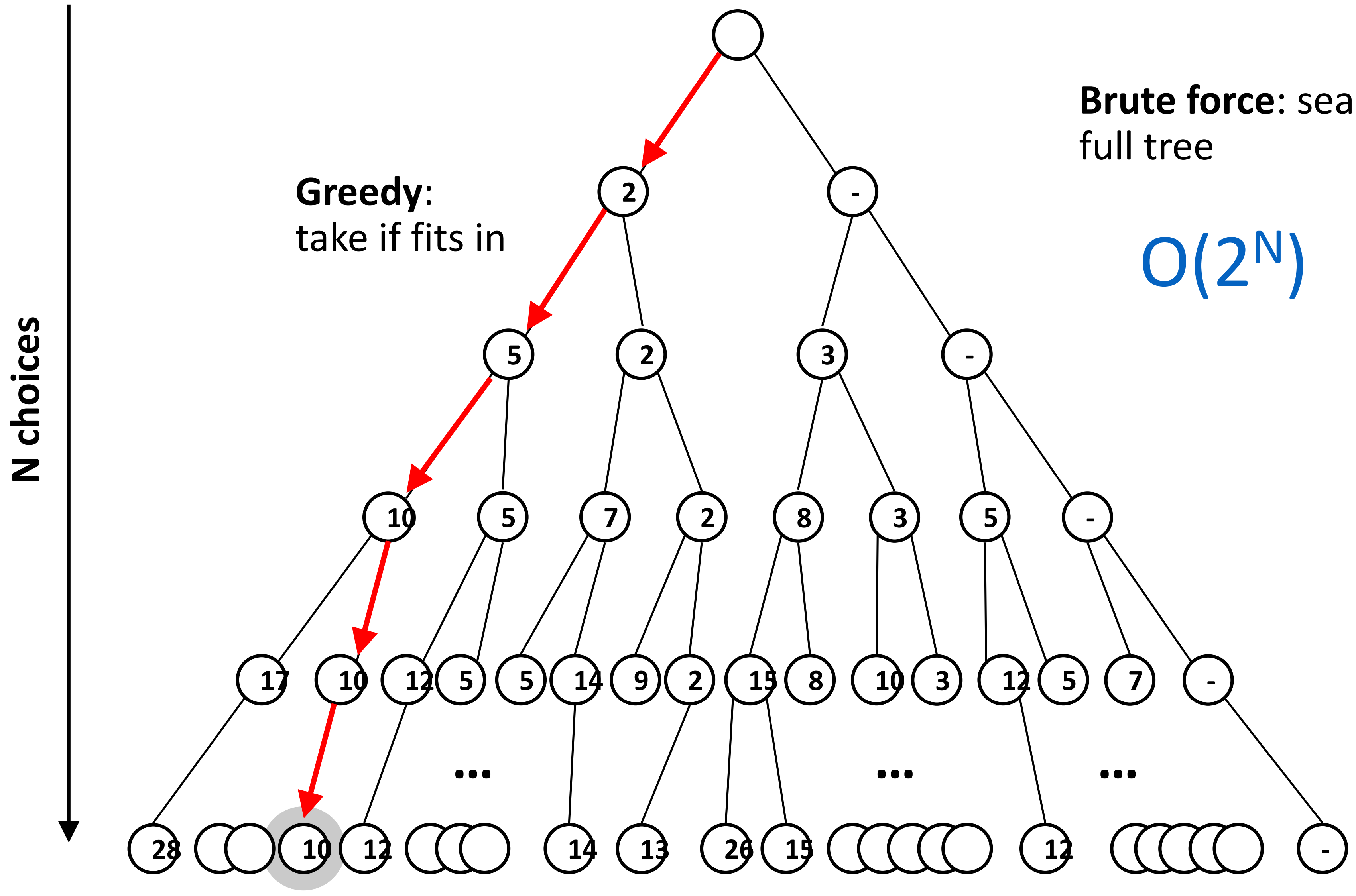
Choose 2

Choose 3

Choose 5

Choose 7

Bounding criterion



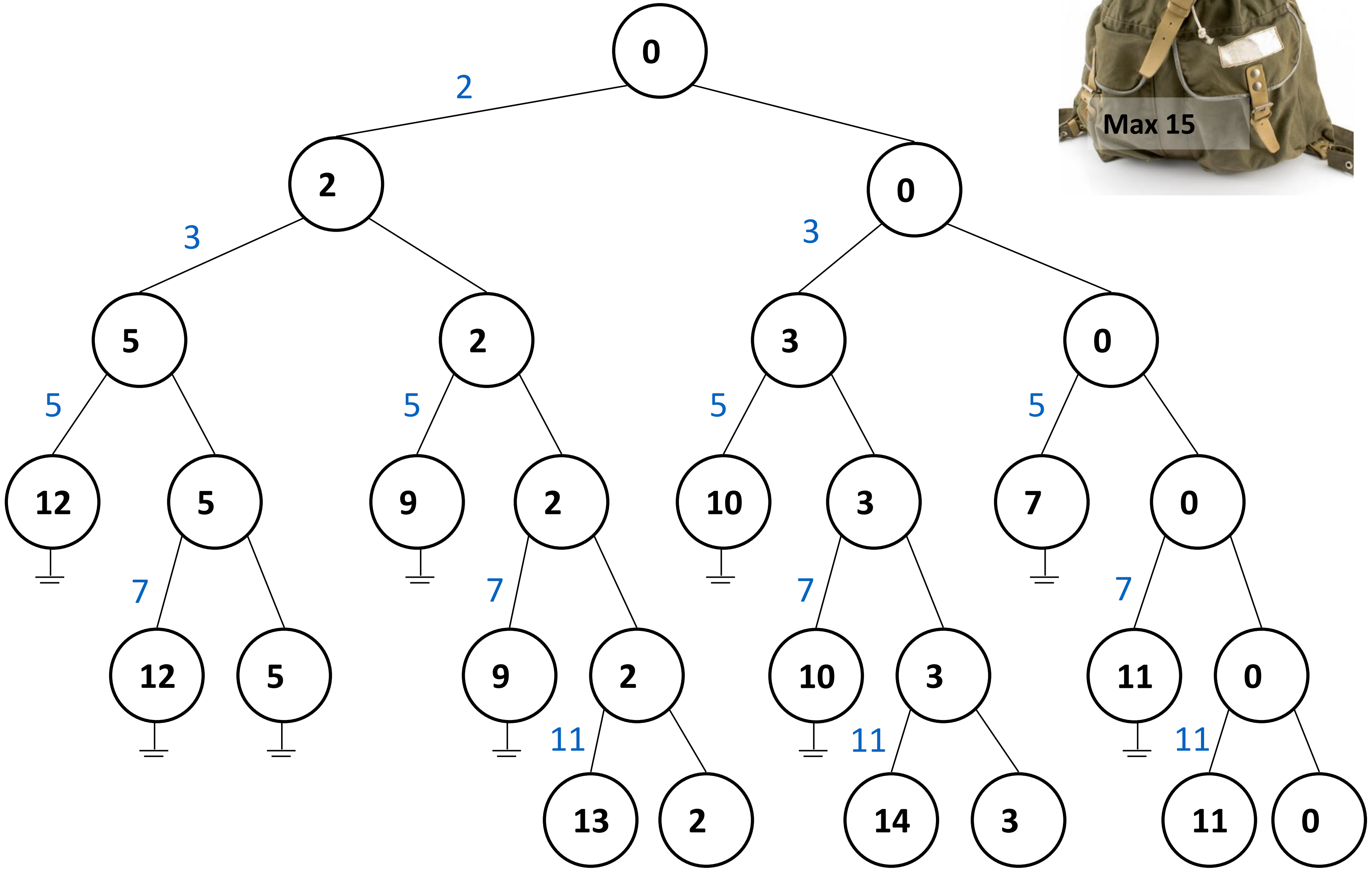
Greedy:
take if fits in

Brute force: search full tree

$O(2^N)$

Knapsack problem: Branch-and-bound

$w_i=\{2,3,5,7,11\}$



Algorithm

- Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
- Initialize maximum profit, maxProfit = 0
- Create an empty queue, Q.
- Create a dummy node of decision tree and enqueue it to Q.
    - Profit and weight of dummy node are 0.
- Do following while Q is not empty.
    - Extract an item from Q. Let the extracted item be u.
    - Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
    - Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
    - Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

Example

Item	Wi	Vi	$Pi = \frac{vi}{wi}$
I1	5	30	6
I3	20	100	5
I5	40	160	4
I4	30	90	3
I2	10	20	2

Step 1- calculate the upper bound using fractional knapsack

I1+I3+I5

5+20+35=60

30+100+140=270

Ub for node B

I3+I5

20+40=60

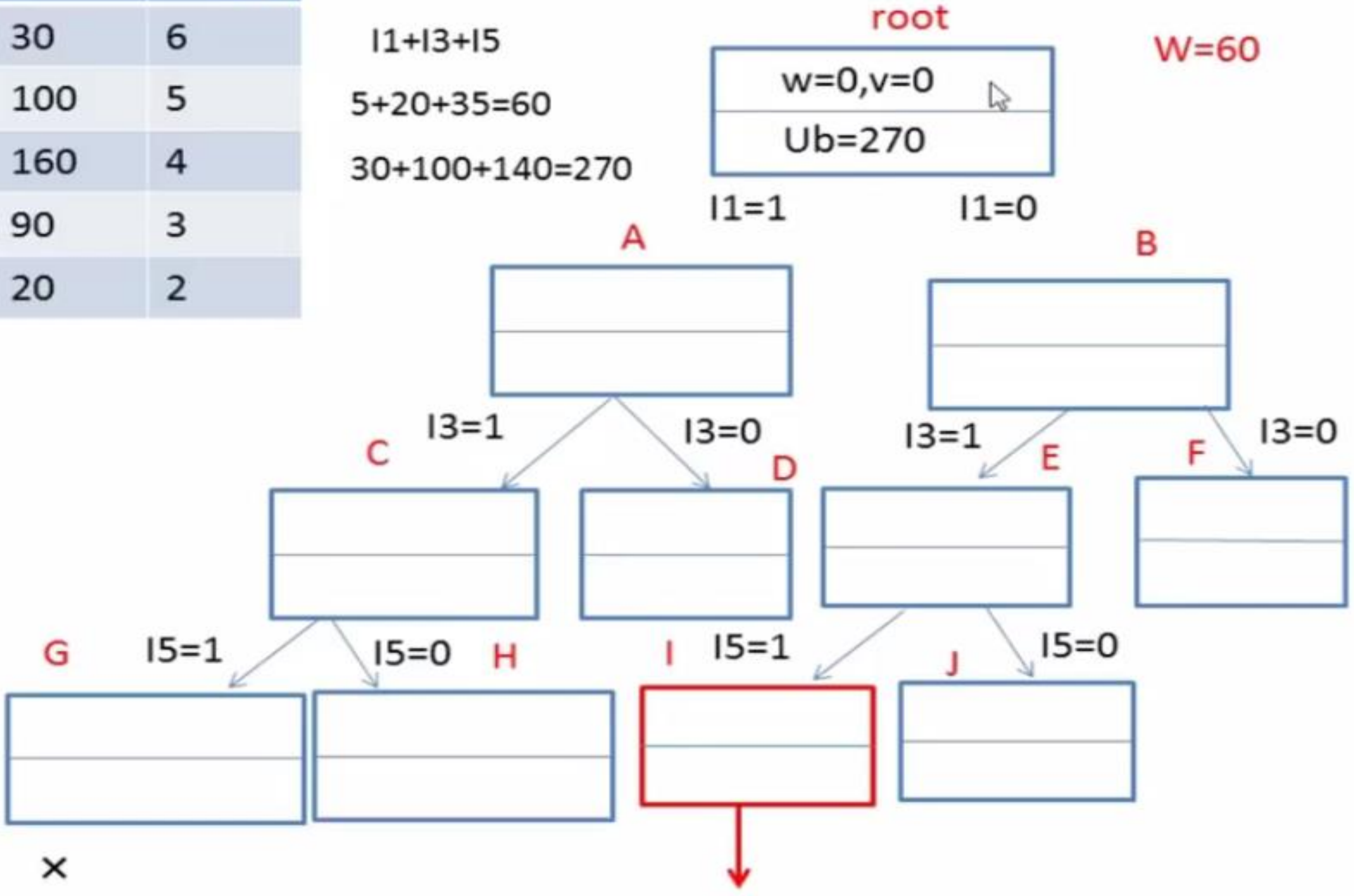
100+160=260

Ub for node D

I1+I5+I4

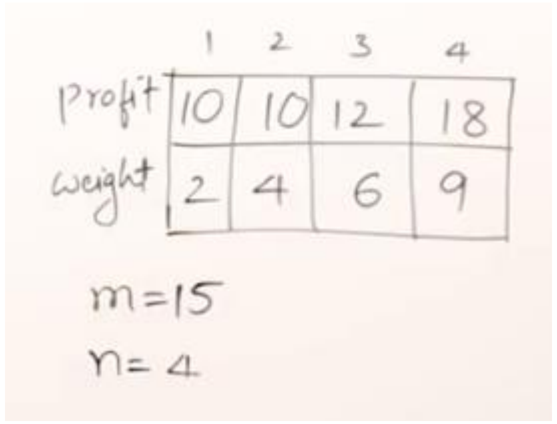
5+40+15=60

30+160+45=235



Final solution
I1=0,I3=1,I5=1
Maximum Value= 260

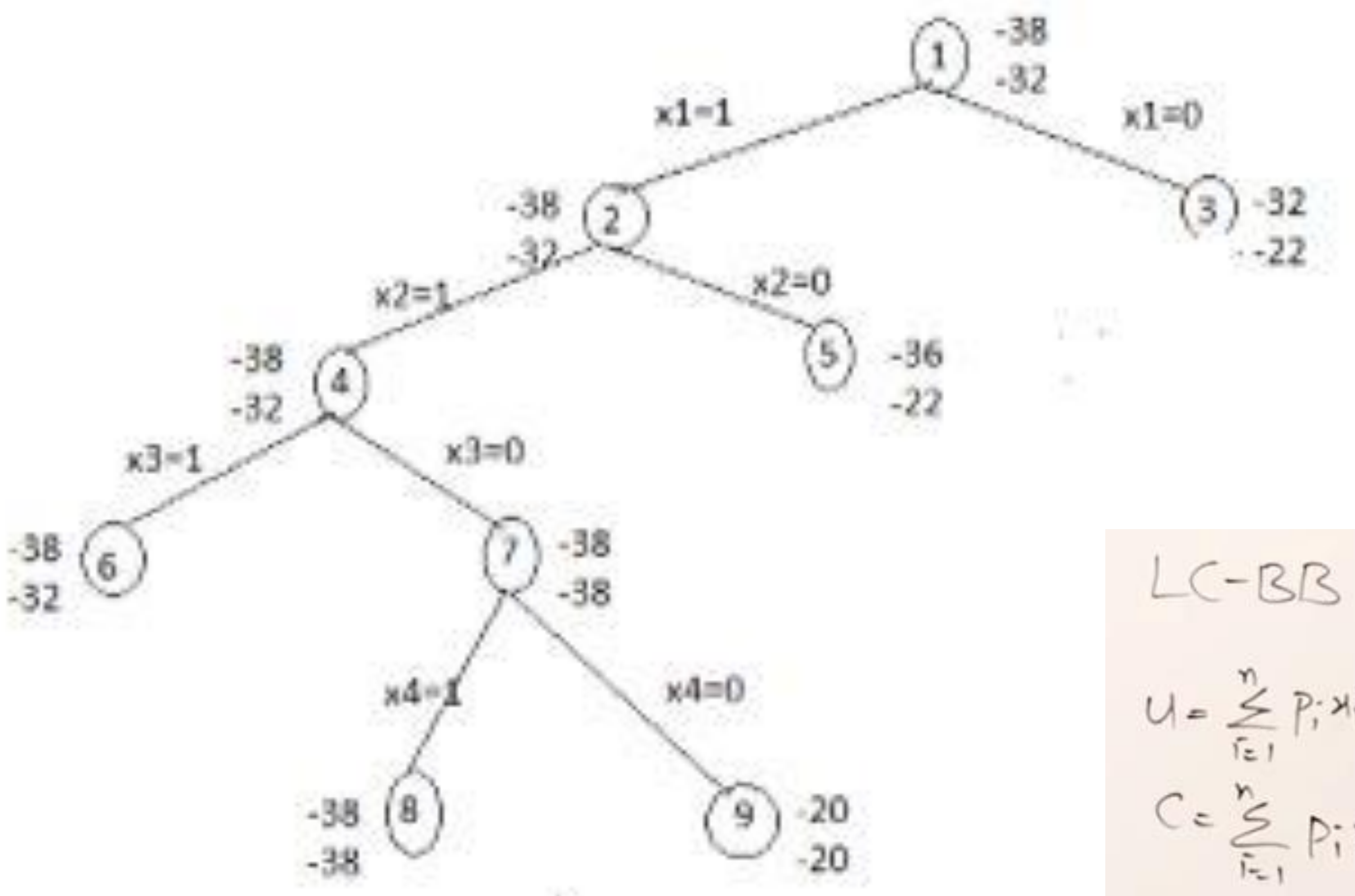
Another Example using Least Cost Search

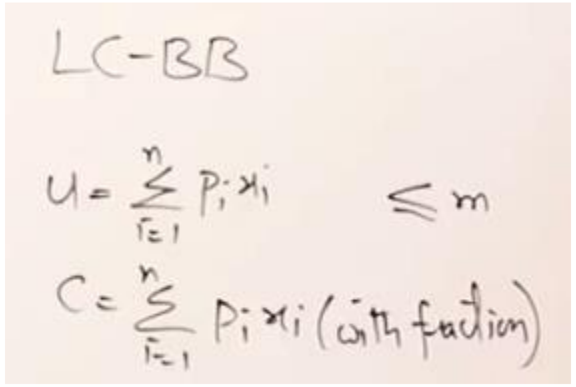


	1	2	3	4
profit	10	10	12	18
weight	2	4	6	9

$m = 15$

$n = 4$





LC-BB

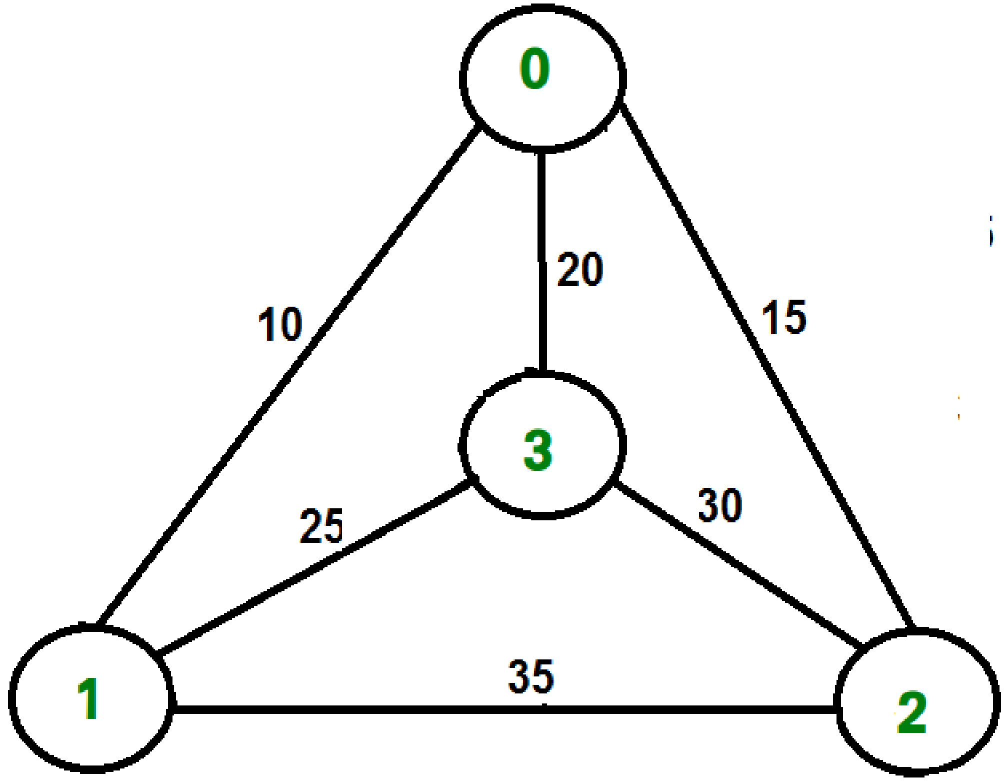
$U = \sum_{i=1}^{n} p_i x_i \quad \leq m$

$C = \sum_{i=1}^{n} p_i x_i$ (with faction)

Figure2: SPACE STATE TREE FOR KNAPSACK PROBLEM USING LC-Branch and Bound

The Traveling Salesman Problem

- Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.
- For example, A TSP tour in the graph is 0-1-3-2-0.
- The cost of the tour is 10+25+30+15 which is 80.



Traveling Salesman Problem

- Given a graph, the TSP Optimization problem is to find a tour, starting from any vertex, visiting every other vertex and returning to the starting vertex, with minimal cost.
- It is NP-hard.
- We try to avoid n! exhaustive search by the branch-and-bound technique on the average case. (Recall that $O(n!)>O(2^n)$.)
- Given a complete, weighted graph on n nodes, find the least weight Hamiltonian cycle, a cycle that visits every node once.
- Though this problem is easy enough to explain, it is very difficult to solve.
- Finding all the Hamiltonian cycles of a graph takes exponential time. Therefore, TSP is in the class NP.

Branch-and-Bound Algorithm

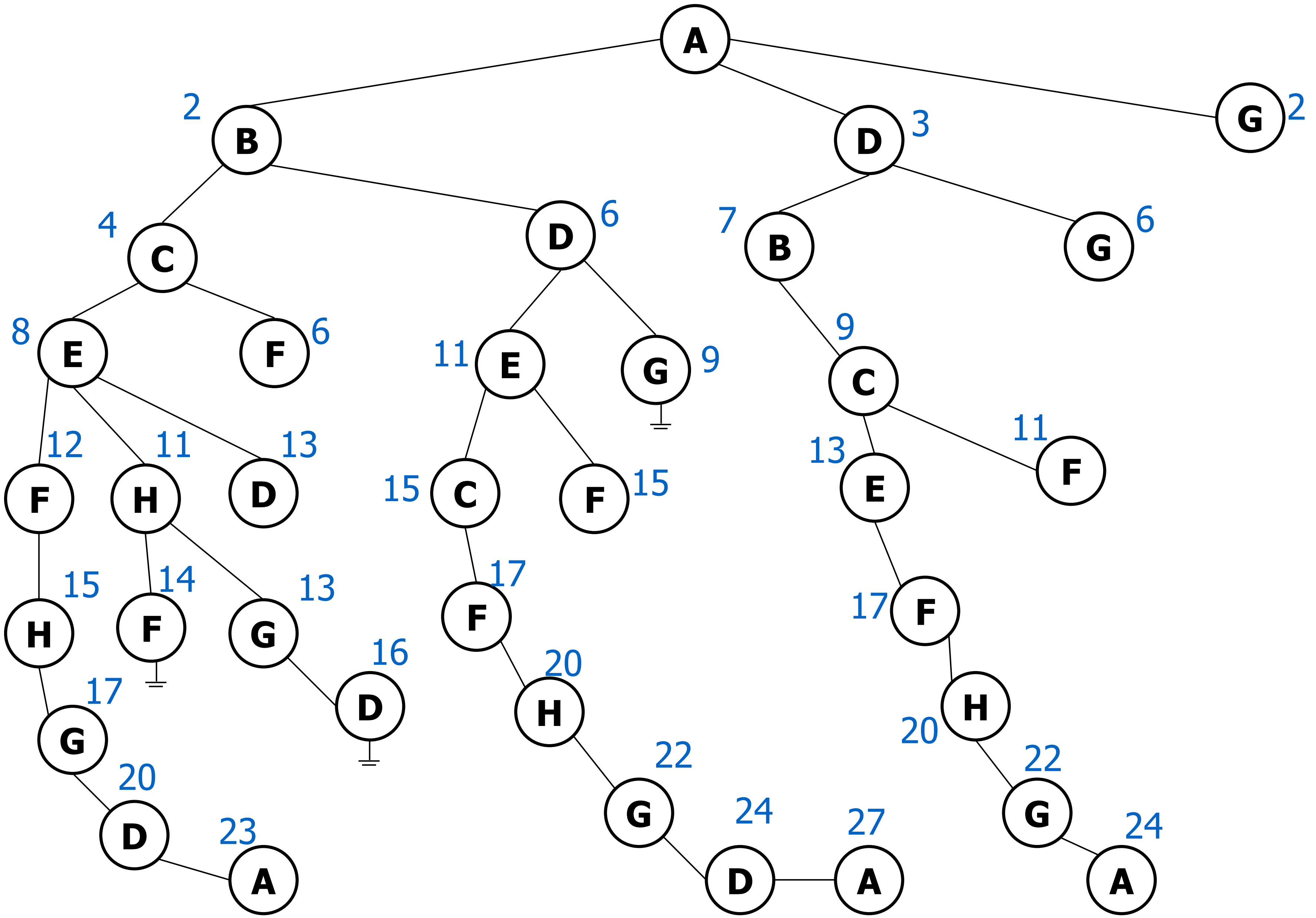
```
T(k) = a tour on k cities ,
Search(k, T(k-1))
        if k=n
                record the tour details
                the bound B=length of the tour
        else
                Find the k-1 possibilities of adding k to all of the possible
places in the tour

        For every tour where the tour length is less then B,
Search(k+1,T(k))
```

The basic idea

- There is a way to split the solution space (branch)
- There is a way to predict a lower bound for a class of solutions.
- There is also a way to find a upper bound of an optimal solution.
- If the lower bound of a solution exceeds the upper bound, this solution cannot be optimal and thus we should terminate the branching associated with this solution.
- We split a solution into two groups:
    - One group including a particular arc
    - The other excluding the arc
- Each splitting incurs a lower bound and we shall traverse the searching tree with the "lower" lower bound.

State space Tree



Example

- Given A Cost Matrix for a Traveling Salesperson Problem.

j / i	1	2	3	4	5	6	7
1	∞	3	93	13	33	9	57
2	4	∞	77	42	21	16	34
3	45	17	∞	36	16	28	25
4	39	90	80	∞	56	7	91
5	28	46	88	33	∞	25	57
6	3	88	18	46	92	∞	7
7	44	26	33	27	84	39	∞

Step 1 to reduce: Search each row for the smallest value

- The Cost Matrix for a Traveling Salesperson Problem.

to j

from i

$$\begin{array}{c|ccccccc}
{}^{i}\,{}^{j} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & \infty & 3 & 93 & 13 & 33 & 9 & 57 \\
2 & 4 & \infty & 77 & 42 & 21 & 16 & 34 \\
3 & 45 & 17 & \infty & 36 & 16 & 28 & 25 \\
4 & 39 & 90 & 80 & \infty & 56 & 7 & 91 \\
5 & 28 & 46 & 88 & 33 & \infty & 25 & 57 \\
6 & 3 & 88 & 18 & 46 & 92 & \infty & 7 \\
7 & 44 & 26 & 33 & 27 & 84 & 39 & \infty
\end{array}$$

Step 2 to reduce: Search each column for the smallest value

- Reduced cost matrix:

i \ j	1	2	3	4	5	6	7	
1	∞	0	90	10	30	6	54	(-3)
2	0	∞	73	38	17	12	30	(-4)
3	29	1	∞	20	0	12	9	(-16)
4	32	83	73	∞	49	0	84	(-7)
5	3	21	63	8	∞	0	32	(-25)
6	0	85	15	43	89	∞	4	(-3)
7	18	0	7	1	58	13	∞	(-26)

reduced:84

The traveling salesperson optimization problem

i \ j	1	2	3	4	5	6	7	
1	∞	0	83	9	30	6	50	
2	0	∞	66	37	17	12	26	
3	29	1	∞	19	0	12	5	
4	32	83	66	∞	49	0	80	
5	3	21	56	7	∞	0	28	
6	0	85	8	42	89	∞	0	
7	18	0	0	0	58	13	∞	
			(-7)	(-1)			(-4)	reduced: 12

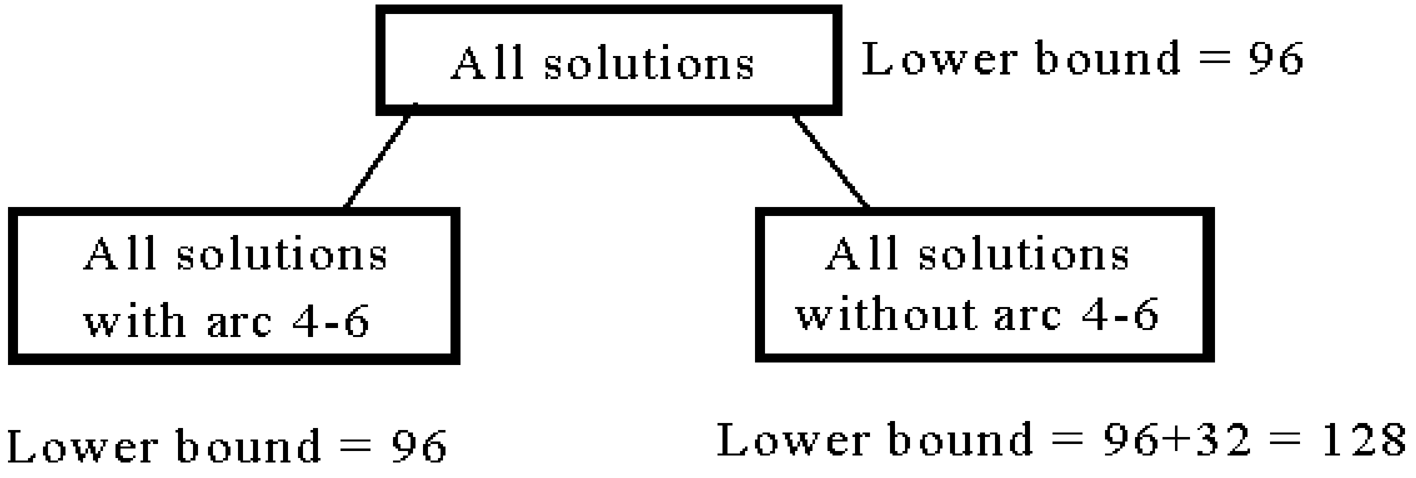
BB Solution Tree

- The total cost of 84+12=96 is subtracted. Thus, we know the lower bound of feasible solutions to this TSP problem is 96.
- Total cost reduced: 84+7+1+4 = 96 (lower bound)

decision tree:

If we use arc 4-6 to split, the difference on the lower bounds is 96+32 =128.

If we use arc 3-5 to split, the difference on the lower bounds is 17+1 = 18.


All solutions
Lower bound = 96
All solutions with arc 4-6
All solutions without arc 4-6
Lower bound = 96
Lower bound = 96+32 = 128

The Highest Level of a Decision Tree.

Heuristic to select an arc to split the solution space

- If an arc of cost 0 (x) is selected, then the lower bound is added by 0 (x) when the arc is included.
- If an arc <i,j> is not included, then the cost of the second smallest value (y) in row i and the second smallest value (z) in column j is added to the lower bound.
- Select the arc with the largest (y+z)-x

We only have to set c4-6 to be ∞.

For the right subtree (Arc 4-6 is excluded)

i \ j	1	2	3	4	5	6	7
1	∞	0	83	9	30	6	50
2	0	∞	66	37	17	12	26
3	29	1	∞	19	0	12	5
4	32	83	66	∞	49	∞	80
5	3	21	56	7	∞	0	28
6	0	85	8	42	89	∞	0
7	18	0	0	0	58	13	∞

For the left subtree
(Arc 4-6 is included)

i \ j	1	2	3	4	5	7
1	∞	0	83	9	30	50
2	0	∞	66	37	17	26
3	29	1	∞	19	0	5
5	3	21	56	7	∞	28
6	0	85	8	∞	89	0
7	18	0	0	0	58	∞

A Reduced Cost Matrix if Arc 4-6 is included.

1. 4th row is deleted.
2. 6th column is deleted.
3. We must set c6-4 to be ∞. (The reason will be clear later.)

For the left subtree

- The cost matrix for all solution with arc 4-6:

j i	1	2	3	4	5	7	
1	∞	0	83	9	30	50	
2	0	∞	66	37	17	26	
3	29	1	∞	19	0	5	
5	0	18	53	4	∞	25	(-3)
6	0	85	8	∞	89	0	
7	18	0	0	0	58	∞	

A Reduced Cost Matrix for that in Table 6-6.

- Total cost reduced: 96+3 = 99 (new lower bound)

When implementing the branch-and-bound approach there is no restriction on the type of state-space tree traversal used. Backtracking, for example, is a simple type of B&B that uses depth-first search.

•The branch-and-bound problem solving method is very similar to backtracking in that a state space tree is used to solve a problem. The differences are that B&B

(1) does not limit us to any particular way of traversing the tree and

- (2) is used only for optimization problems.