Course: Analysis of Algorithms
Code: CS33104
Branch: MCA -3rd Semester

Lecture 7 – Back Tracking and Branch & Bound

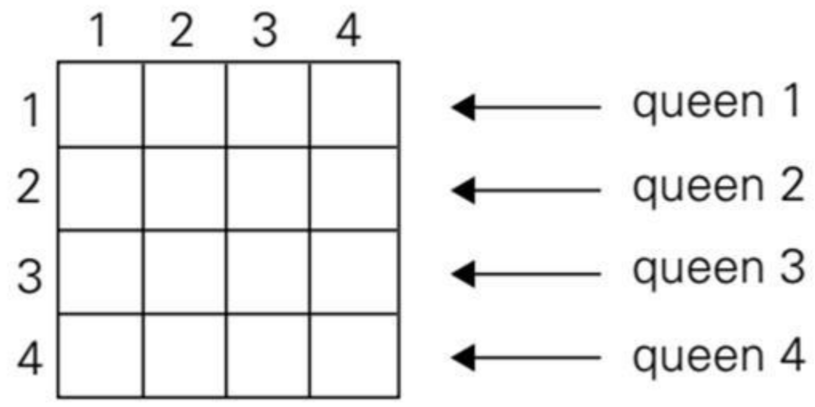
Faculty & Coordinator : Dr. J Sathish Kumar (JSK)

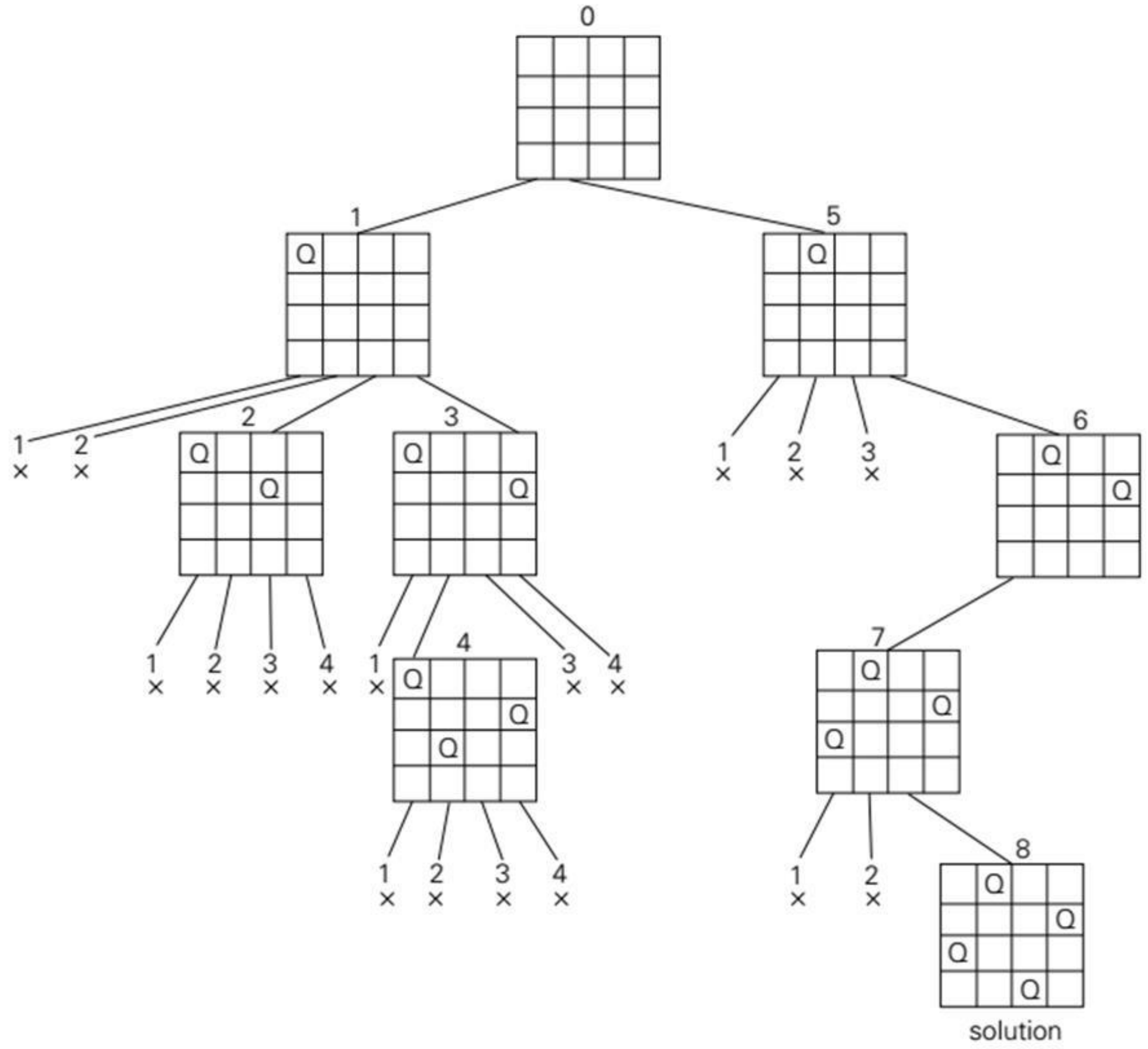
Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Backtracking : *n*-Queens Problem

- The problem is to place *n* queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.
- For $n = 1$, the problem has a trivial solution, and it is easy to see that there is no solution for $n = 2$ and $n = 3$.
- So let us consider the four-queens problem and solve it by the backtracking technique.
- Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board





Finally, it should be pointed out that a single solution to the n-queens problem for any n ≥ 4 can be found in linear time

Backtracking and Branch-and-bound

- Both strategies can be considered an improvement over exhaustive search.
- Unlike exhaustive search, they construct candidate solutions one component at a time and evaluate the partially constructed solutions.
- If no potential values of the remaining components can lead to a solution, the remaining components are not generated at all.
- This approach makes it possible to solve some large instances of difficult combinatorial problems, though, in the worst case, we still face the same curse of exponential explosion encountered in exhaustive search.

Backtracking and Branch-and-bound

- Both backtracking and branch-and-bound are based on the construction of a ***state-space tree*** whose nodes reflect specific choices made for a solution's components.
- Both techniques terminate a node as soon as it can be guaranteed that no solution to the problem can be obtained by considering choices that correspond to the node's descendants.

Backtracking and Branch-and-bound

- The techniques differ in the nature of problems they can be applied to.
- Branch-and-bound is applicable only to optimization problems because it is based on computing a bound on possible values of the problem's objective function.
- Backtracking is not constrained by this demand, but more often than not, it applies to non-optimization problems.
- The other distinction between backtracking and branch-and-bound lies in the order in which nodes of the state-space tree are generated.
- For backtracking, this tree is usually developed depth-first (i.e., similar to DFS).
- Branch-and-bound can generate nodes according to several rules and one of them is best first search or BFS.

Backtracking

- The exhaustive-search technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property.
- Backtracking is a more intelligent variation of this approach.
- The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.
  - If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component.
  - If there is no legitimate option for the next component, no alternatives for *any* remaining component need to be considered.
  - In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

Backtracking

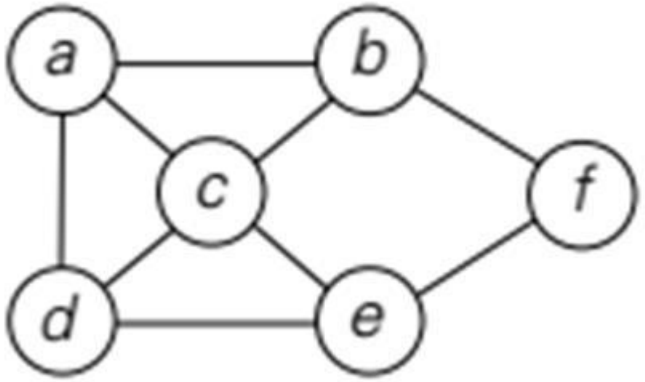
- It is convenient to implement this kind of processing by constructing a tree of choices being made, called the ***state-space tree***.
- Its root represents an initial state before the search for a solution begins.
- The nodes of the first level in the tree represent the choices made for the first component of a solution, the nodes of the second level represent the choices for the second component, and so on.
- A node in a state-space tree is said to be ***promising*** if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called ***nonpromising***.

Backtracking

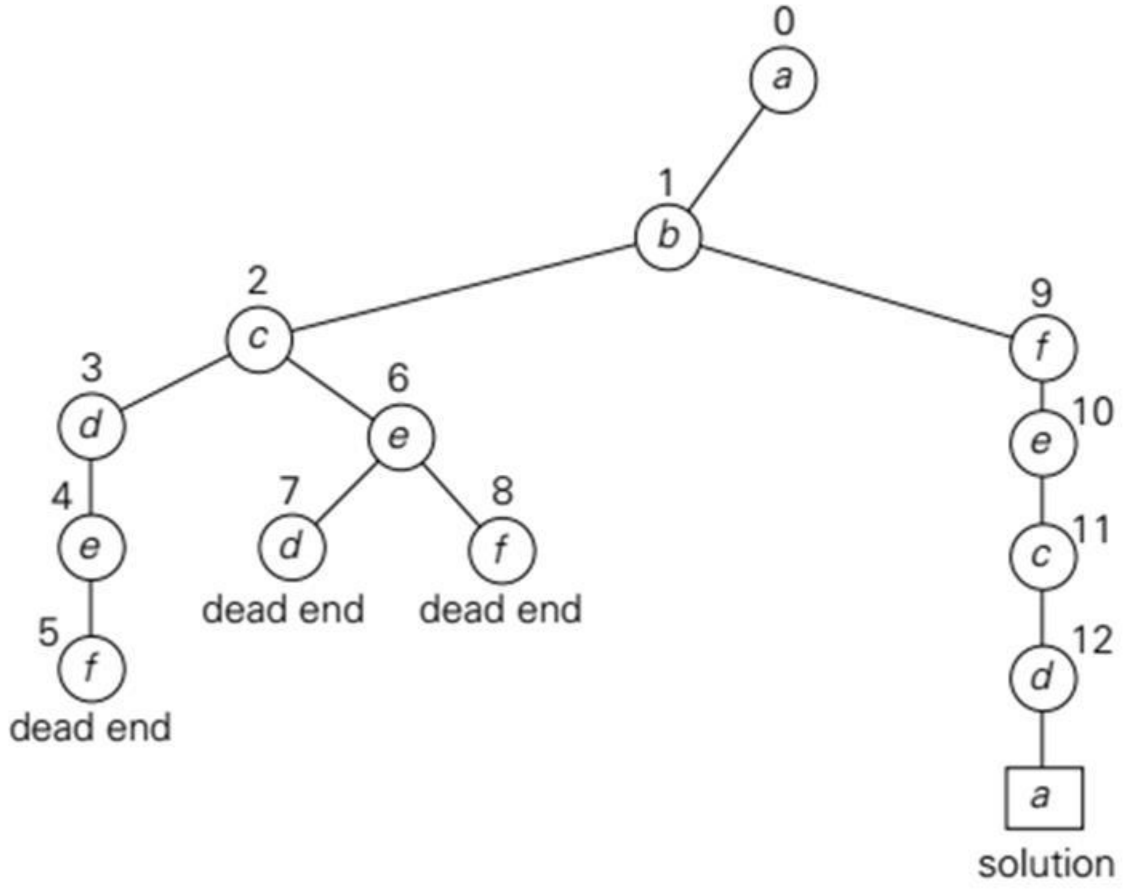
- Leaves represent either non-promising dead ends or complete solutions found by the algorithm.
- In the majority of cases, a state space tree for a backtracking algorithm is constructed in the manner of depth first search.
- If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child.
- If the current node turns out to be non-promising, the algorithm backtracks to the node’s parent to consider the next possible option for its last component;
- If there is no such option, it backtracks one more level up the tree, and so on.
- Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

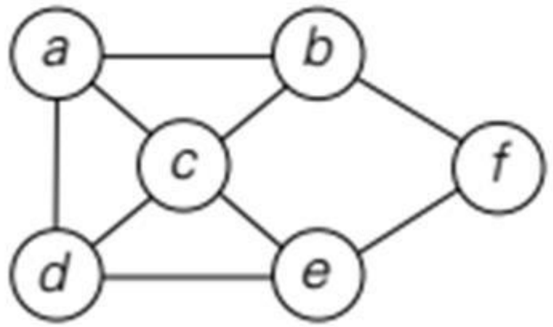
Hamiltonian Circuit Problem

- A directed graph in which the path begins and ends on the same vertex (a closed loop) such that each vertex is visited exactly once is known as a Hamiltonian circuit.



Hamiltonian Circuit Problem





State-space tree for finding a Hamiltonian circuit.

Subset-Sum Problem

- ***Subset-sum problem***: find a subset of a given set $A = \{a1, . . . , an\}$ of n positive integers whose sum is equal to a given positive integer d.
- For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions:
  - {1, 2, 6} and
  - {1, 8}.
- Some instances of this problem may have no solutions.

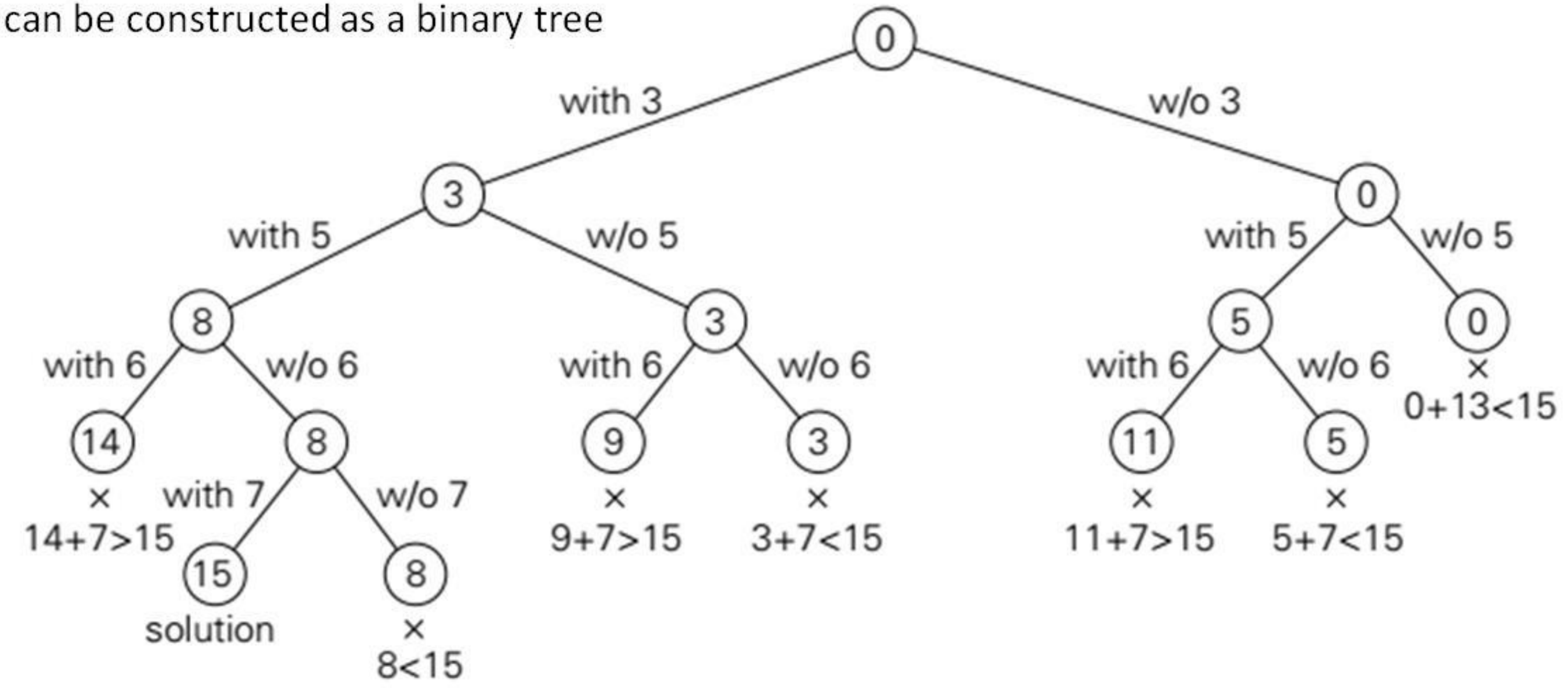
Subset-Sum Problem

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large)},$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small)}.$$

- $A = \{3, 5, 6, 7\}$ and $d = 15$

The state-space tree can be constructed as a binary tree



Branch-and-Bound

- The central idea of backtracking is to cut off a branch of the problem's state-space tree as soon as we can deduce that it cannot lead to a solution.
- This idea can be strengthened further if we deal with an optimization problem.
- An optimization problem seeks to minimize or maximize some objective function (a tour length, the value of items selected, the cost of an assignment, and the like), usually subject to some constraints.

Branch-and-Bound

- Note that in the standard terminology of optimization problems,
  - a feasible solution is a point in the problem’s search space that satisfies all the problem’s constraints
    - (e.g., a Hamiltonian circuit in the traveling salesman problem or a subset of items whose total weight does not exceed the knapsack’s capacity in the knapsack problem),
  - whereas an optimal solution is a feasible solution with the best value of the objective function
    - (e.g., the shortest Hamiltonian circuit or the most valuable subset of items that fit the knapsack).

Branch-and-Bound

- Compared to backtracking, branch-and-bound requires two additional items:
  1. A way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
  2. The value of the best solution seen so far

Branch-and-Bound

- In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:
  1. The value of the node's bound is not better than the value of the best solution seen so far.
  2. The node represents no feasible solutions because the constraints of the problem are already violated.
  3. The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—
     - In this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Assignment Problem

- Assigning *n* people to *n* jobs so that the total cost of the assignment is as small as possible.
- 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.

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

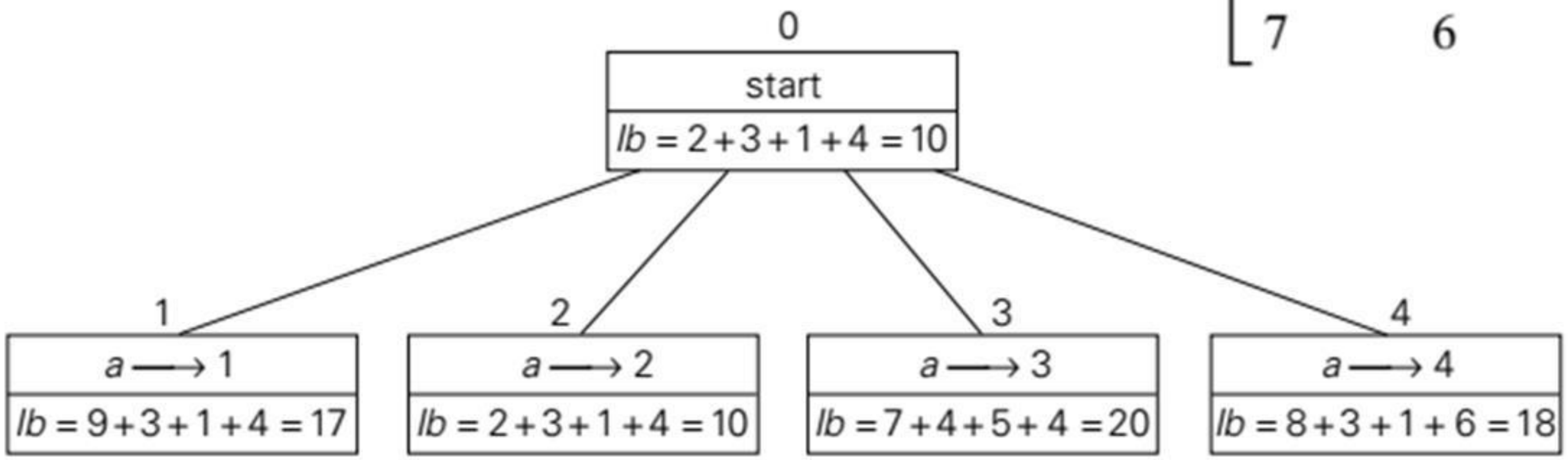
Assignment Problem

- It is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows.
- For the instance here, this sum is 2 + 3 + 1+ 4 = 10.

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

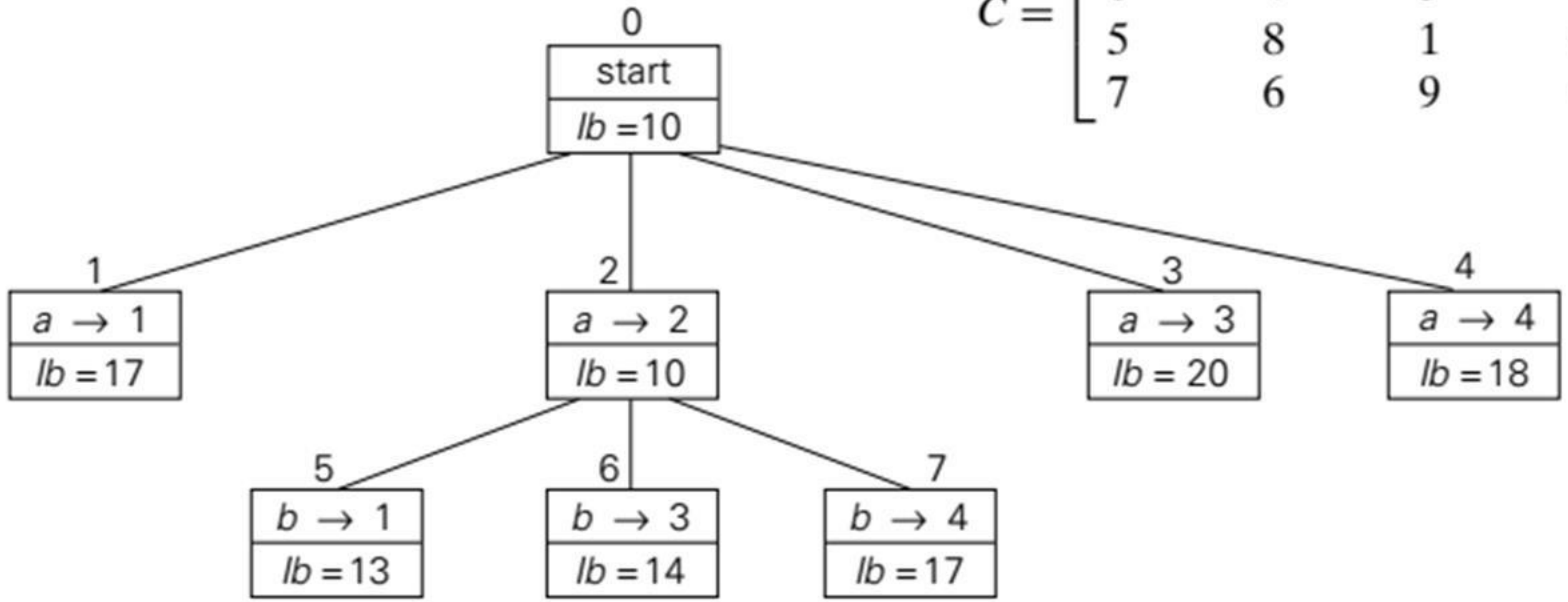
Assignment Problem

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

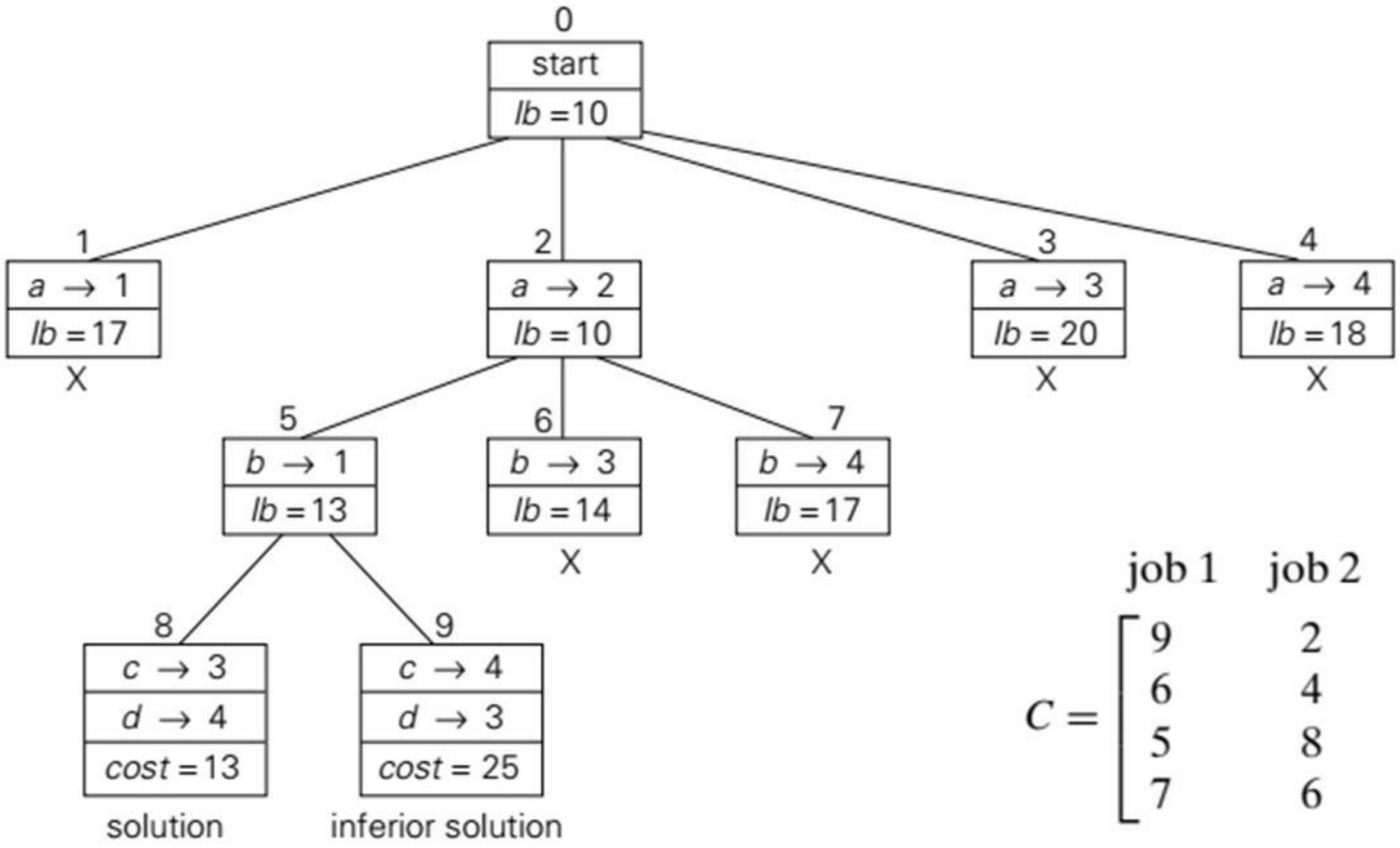


Assignment Problem

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



Assignment Problem



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

Knapsack Problem

- Each node on the *i*th level of this tree, $0 \leq i \leq n$, represents all the subsets of *n* items that include a particular selection made from the first *i* ordered items.
- This particular selection is uniquely determined by the path from the root to the node.
- A branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion.
- We record the total weight *w* and the total value *v* of this selection in the node, along with some upper bound *ub* on the value of any subset that can be obtained by adding zero or more items to this selection.

Knapsack Problem

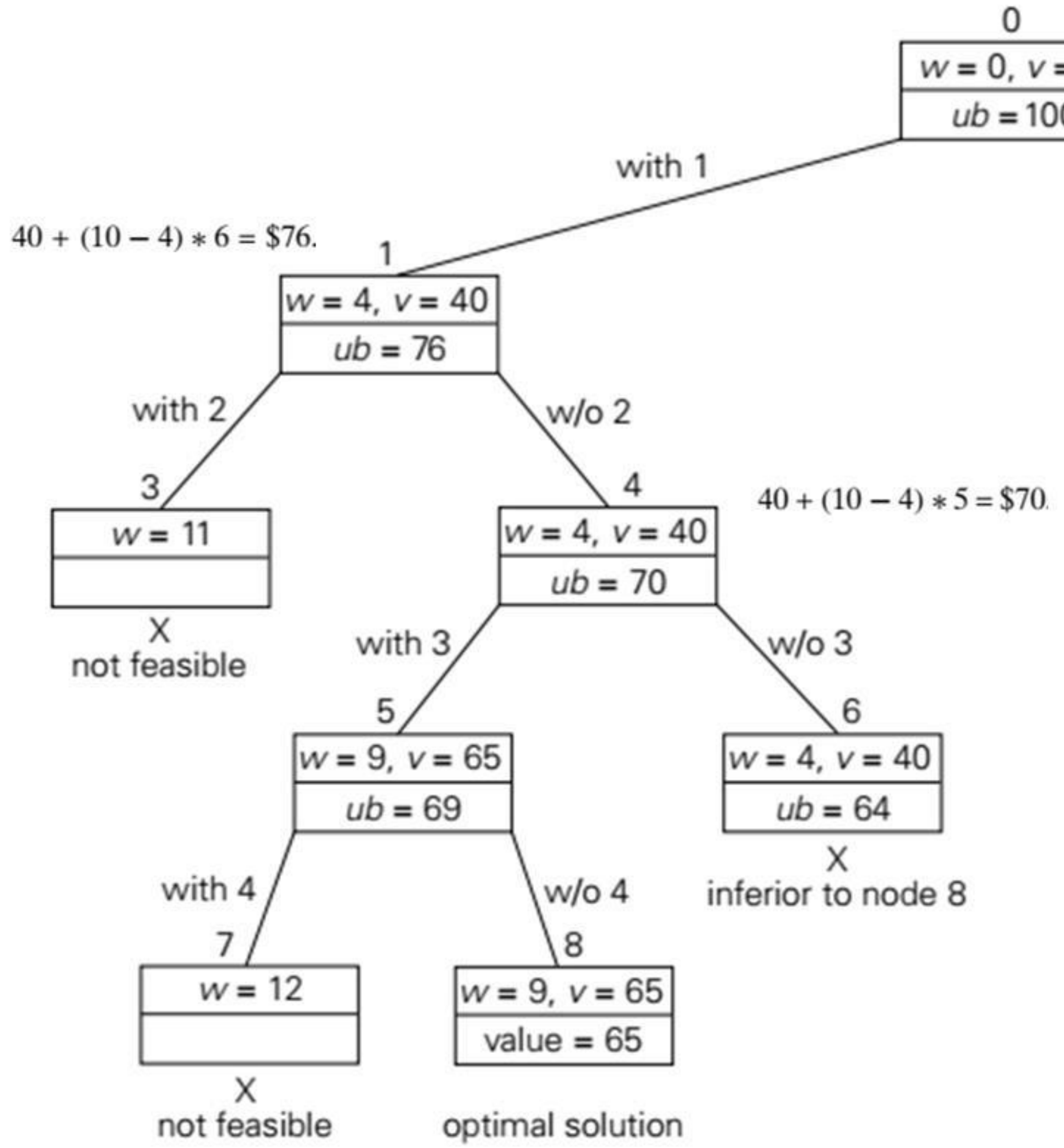
- A simple way to compute the upper bound *ub* is to add to *v*, the total value of the items already selected, the product of the remaining capacity of the knapsack $W - w$ and the best per unit payoff among the remaining items, which is v_{i+1}/w_{i+1}:

$$ub = v + (W - w)(v_{i+1}/w_{i+1})$$

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.

ub= 40 + (10)*6 = 40+60 =100



item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

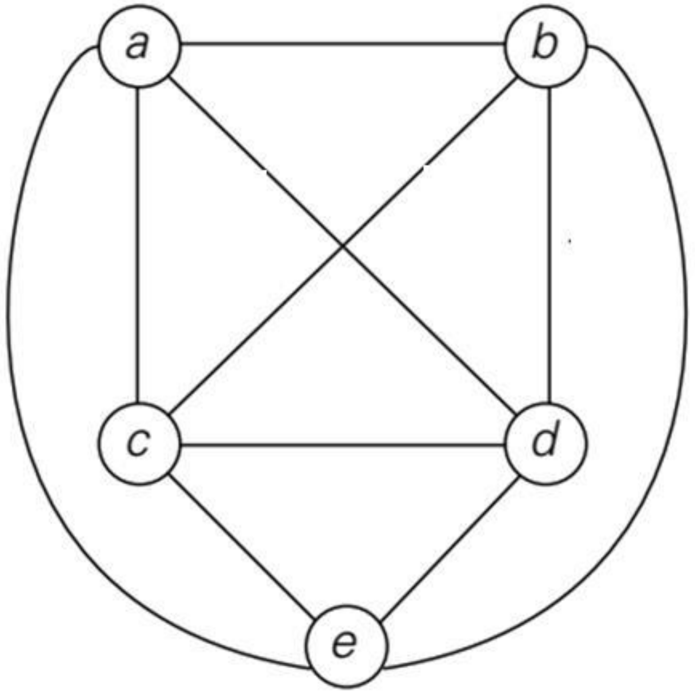
The knapsack's capacity W is 10.

Knapsack Problem

TSP: Branch-and-bound technique

- Solving a problem by branch-and-bound has both the challenge and opportunity of choosing the order of node generation and finding a good bounding function.
- Though the best-first rule we used above is a sensible approach, it may or may not lead to a solution faster than other strategies.
- Artificial intelligence researchers are particularly interested in different strategies for developing state-space trees.
- Finding a good bounding function is usually not a simple task.
- On the one hand, we want this function to be easy to compute.
- On the other hand, it cannot be too simplistic—otherwise, it would fail in its principal task to prune as many branches of a state-space tree as soon as possible.
- Striking a proper balance between these two competing requirements may require intensive experimentation with a wide variety of instances of the problem in question.

TSP: Branch-and-bound technique



	N_0	N_1	N_2	N_3	N_4
N_0	INF	20	30	10	11
N_1	15	INF	16	4	2
N_2	3	5	INF	2	4
N_3	19	6	18	INF	3
N_4	16	4	7	16	INF

TSP: Branch-and-bound technique

	N_0	N_1	N_2	N_3	N_4
N_0	INF	20	30	**<u>10</u>**	11
N_1	15	INF	16	4	**<u>2</u>**
N_2	3	5	INF	**<u>2</u>**	4
N_3	19	6	18	INF	**<u>3</u>**
N_4	16	**<u>4</u>**	7	16	INF

	N_0	N_1	N_2	N_3	N_4
N_0	INF	10	17	0	1
N_1	12	INF	11	2	0
N_2	0	3	INF	0	2
N_3	15	3	12	INF	0
N_4	11	0	0	13	INF

cost = 10 + 2 + 2 + 3 + 4 + 1 + 3 = 25

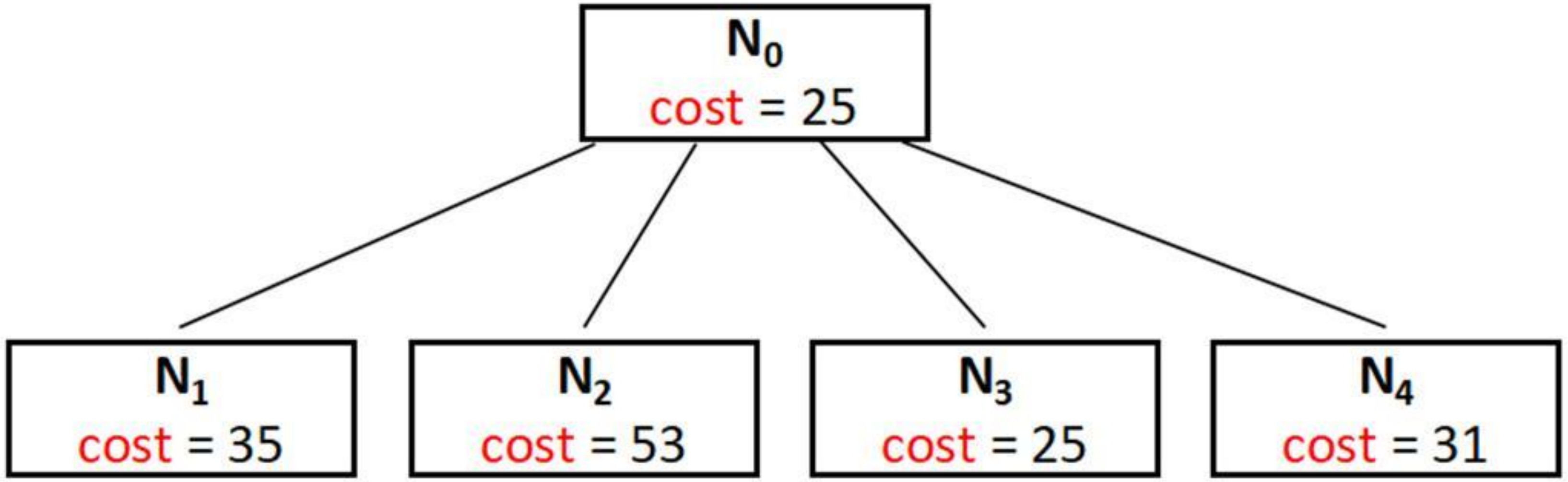
TSP: Branch-and-bound technique

	N_0	N_1	N_2	N_3	N_4
N_0	INF	INF	INF	INF	INF
N_1	INF	INF	11	2	0
N_2	0	INF	INF	0	2
N_3	15	INF	12	INF	0
N_4	11	INF	0	13	INF

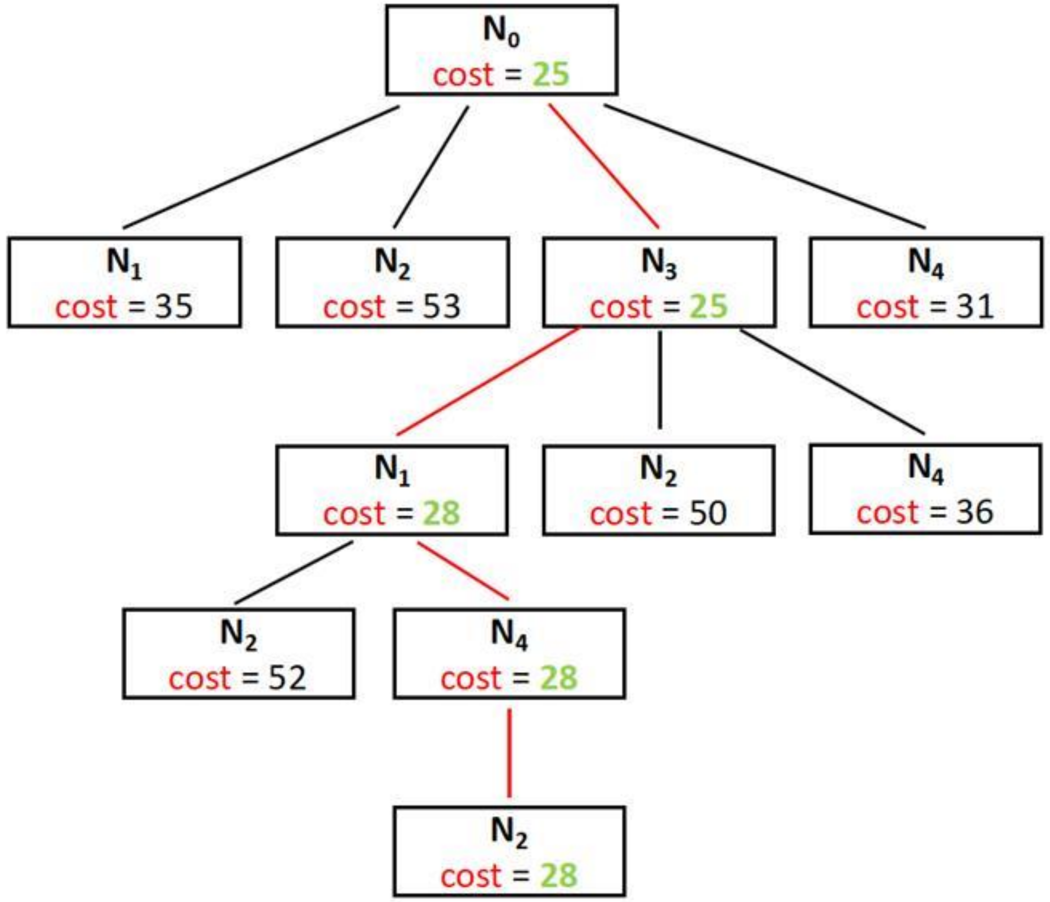
cost = cost of node N0 + cost of the (N0,N1) position(before setting INF) + lower bound of the path starting at N1
= 25 + 10 + 0 = 35

In a similar way we can find that,
For N2 the cost is 53,
For N3 the cost is 25,
And for N4 the cost is 31.

TSP: Branch-and-bound technique



TSP: Branch-and-bound technique



TSP: Branch-and-bound technique

- Best Case it is **O(n^2)** because of matrix reductions.
- Worst Case
    - We are actually creating all the possible extensions of E-nodes in terms of tree nodes. Which is nothing but a **permutation**.
    - Suppose we have **N** cities, then we need to generate all the permutations of the **(N-1)** cities, excluding the root city. Hence the time complexity for generating the permutation which is equal to **O(2^(n-1))**.
    - Hence the final time complexity of the algorithm can be **O(n^2 * 2^n)**.
    - Same as Dynamic Programming Strategy

Assignment #5

- How state-space trees are used for programming such games as chess, checkers, and tic-tac-toe.
- Implement the backtracking and branch and bound algorithms that we discussed in this slides. Run your program for a sample of *n* values to get the numbers of nodes in the algorithm's state-space trees. Compare these numbers with the numbers of candidate solutions generated by the exhaustive search algorithm for this problem.