UNIT 8. LIMITATIONS OF ALGORITHM POWER

In the previous chapters, various algorithms and design techniques (like brute force, divide and conquer etc) have been considered. But, every methodology and algorithm has limitations. Some problems cannot be solved by any algorithm. Some problems can be solved algorithmically, but not in a polynomial time. And few problems have lower bound for their efficiency. There are certain ways of establishing lower bounds on efficiency of algorithms.

8.1 Decision Trees

A decision tree is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences. Decision trees are helpful in establishing the lower bounds on efficiency of comparison-based algorithms like sorting and searching. Consider a decision tree given in Figure 8.1, which is used for finding minimum of three numbers. Here, each leaf represents the possible outcome of the algorithm. The work of the algorithm on a particular input of size can be traced by a path from the root to a leaf in its decision tree. Number of comparisons made by the algorithm is equal to the number of edges in that path. Hence, maximum number of comparisons required for the algorithm is equal to the height of the decision tree.

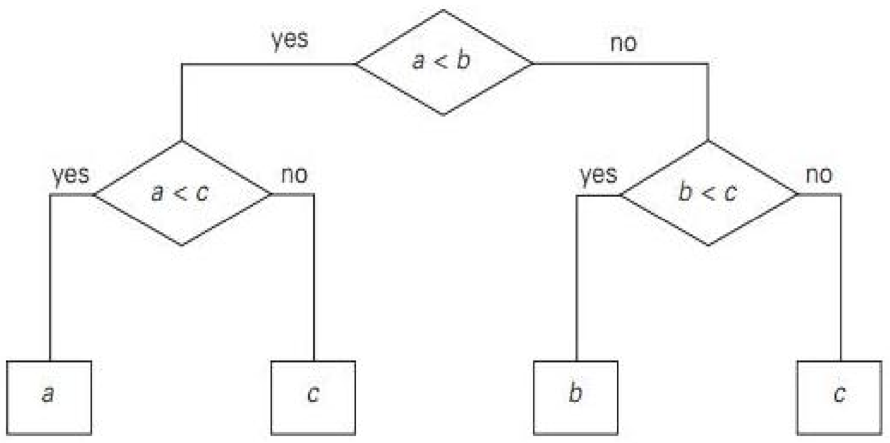


Figure 8.1 Decision tree for finding minimum of three numbers

Decision Trees for Sorting Algorithms:

Most of the sorting algorithms are based on comparison of elements in a given array. Hence, decision tree can be drawn for solving sorting problems. Selection sort is one of the sorting techniques based on comparison of elements. Figure 8.2 is a decision tree for applying selection sort on 3 elements viz. *a, b, c.*

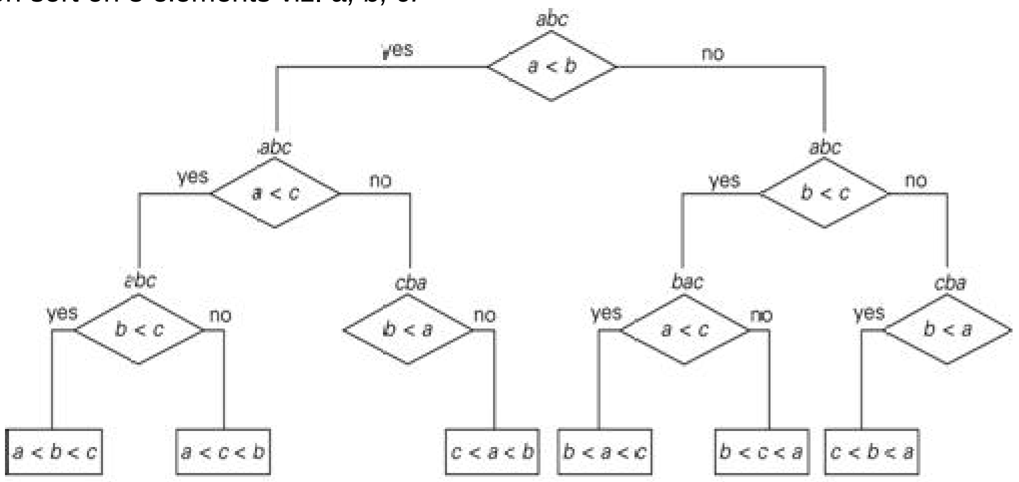


Figure 8.2 Decision tree for three-element selection sort

It can be observed that in the worst case, number of comparisons can be given as –

$$C(n) \geq \lceil \log_2 n \rceil$$

Decision Tree for three-element Insertion sort:
The figure 8.3 shows the decision tree for implementing insertion sort for an array of three elements.

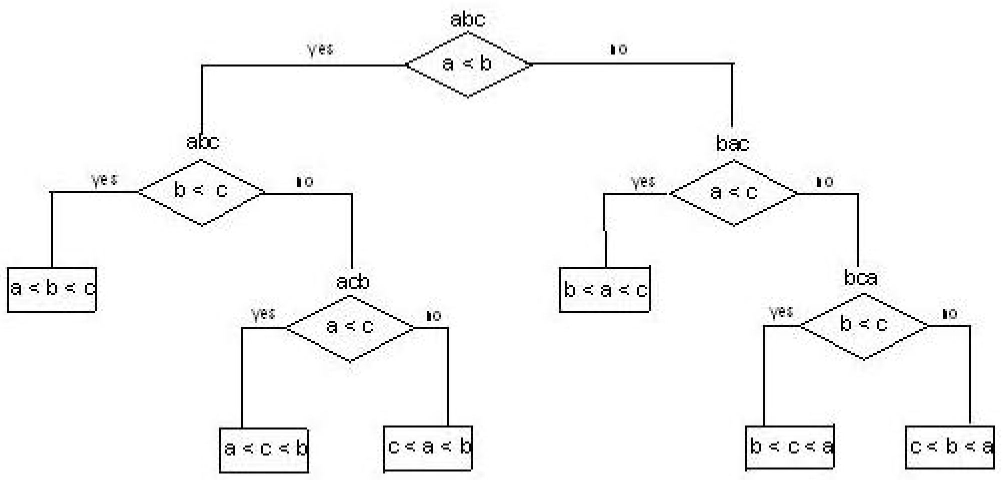


Figure 8.3 Decision tree for three-element insertion sort

8.2 *P*, *NP* and *NP – Complete* Problems

In the study of the computational complexity of the problems, the major concern is whether a given problem can be solved by some algorithm in a polynomial time or not.

We can say that an algorithms solves the problem in polynomial time if its worst-case time complexity is O(*p(n)*), where *p(n)* is a polynomial of input size *n.* The problems which are solvable in polynomial time are called as ***tractable*** and problems that cannot be solved in polynomial time are called as ***intractable.***

Class *P* is a class of decision problems that can be solved in polynomial time by deterministic algorithms. This class of problems is called ***polynomial.***

Class *NP* is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called as ***Nondeterministic polynomial.***

Most of the decision problems are in *NP*. That is,

$$P \subseteq NP$$

But, *NP* also contains few decision problems like Hamiltonian circuit problem, travelling salesman problem, knapsack problem etc. This leads to the most important open question of theoretical computer science: ***Is P a proper subset of NP, or are they same?*** That is,

Whether *P = NP ?*

The meaning of P = NP implies that many combinatorial decision problems can be solved by a polynomial-time algorithm, though no such algorithm is invented till today. Moreover, many well-known decision problems are known to be ***NP-Complete.*** A decision problem *D* is said to be ***NP-Complete*** if,

- It belongs to class *NP*
- Every problem in *NP* is polynomially reducible to *D*.

A Decision problem *D1* is said to be ***polynomially reducible*** to a decition problem *D2,* if there exists a function *t* that transforms instances of *D1* to instances of *D2* such that

- *t* maps all yes instances of *D1* to yes instances of D2 and all no instances of D1 to no instances of D2
- *t* is computable by a polynomial-time algorithm

Informally, an NP-complete problem is a problem in NP that is as difficult as any other problem in this class, because, any other problem in NP can be reduced to it in a polynomial time as shown in Figure 8.4. In this diagram, arrows indicate the polynomial-time reductions of NP problems to an NP-complete problem.

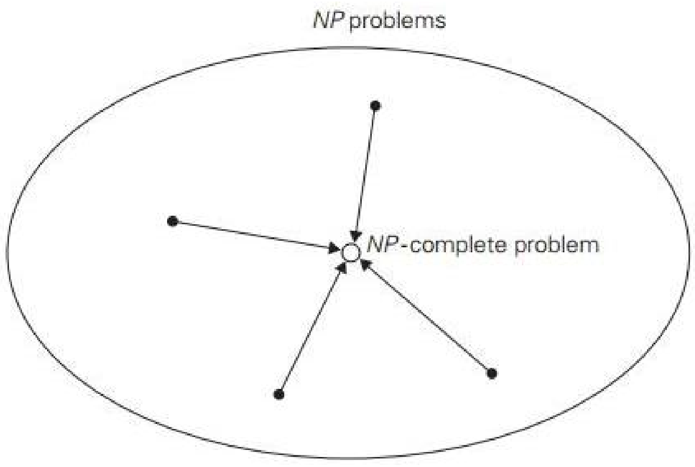


Figure 8.4 Notion of an NP-complete problem.

Copingup with Limitations of Algorithmic Power

In previous chapter, we have seen several algorithms and various design techniques, as very powerful tool for solving real time problems. But the power of algorithms is limited. Some problems cannot be solved by any algorithms. Some problems can be solved algorithmically, but not in polynomial time. Even though, some problems can be solved in polynomial time, there are lower bounds on the efficiency of algorithms.

To deal with such situations, two more algorithms design techniques are used —

* Back tracking
* Branch and Bound.

These techniques are thought as an improvement over exhaustive search.

Both of these techniques 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 descendents.

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 usually applied on non-optimizing problems.

Another difference between these two techniques lies in the order in which nodes of the state-space tree are generated. For backtracking, this tree is usually developed depth first (similar to DFS). Branch and bound can generate nodes according to several rules, the most natural one is the best-first rule.

Backtracking

The primary 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. Here, we will discuss ~~three~~ problems -

(1) n-Queen's problem
(2) Subset-sum problem
(3) Hamiltonian circuit problem

1. n-Queen's Problem

The problem is to place 'n' Queens on a $n \times n$ chess-board so that no two queens attack each other by being in the same row, same column or on the same diagonal.

For $n = 1$, the solution would be $\boxed{Q}$

For $n=2$ and $n=3$, there are no solutions.

Because -

$n=2$

Q	

No place for 2nd queen

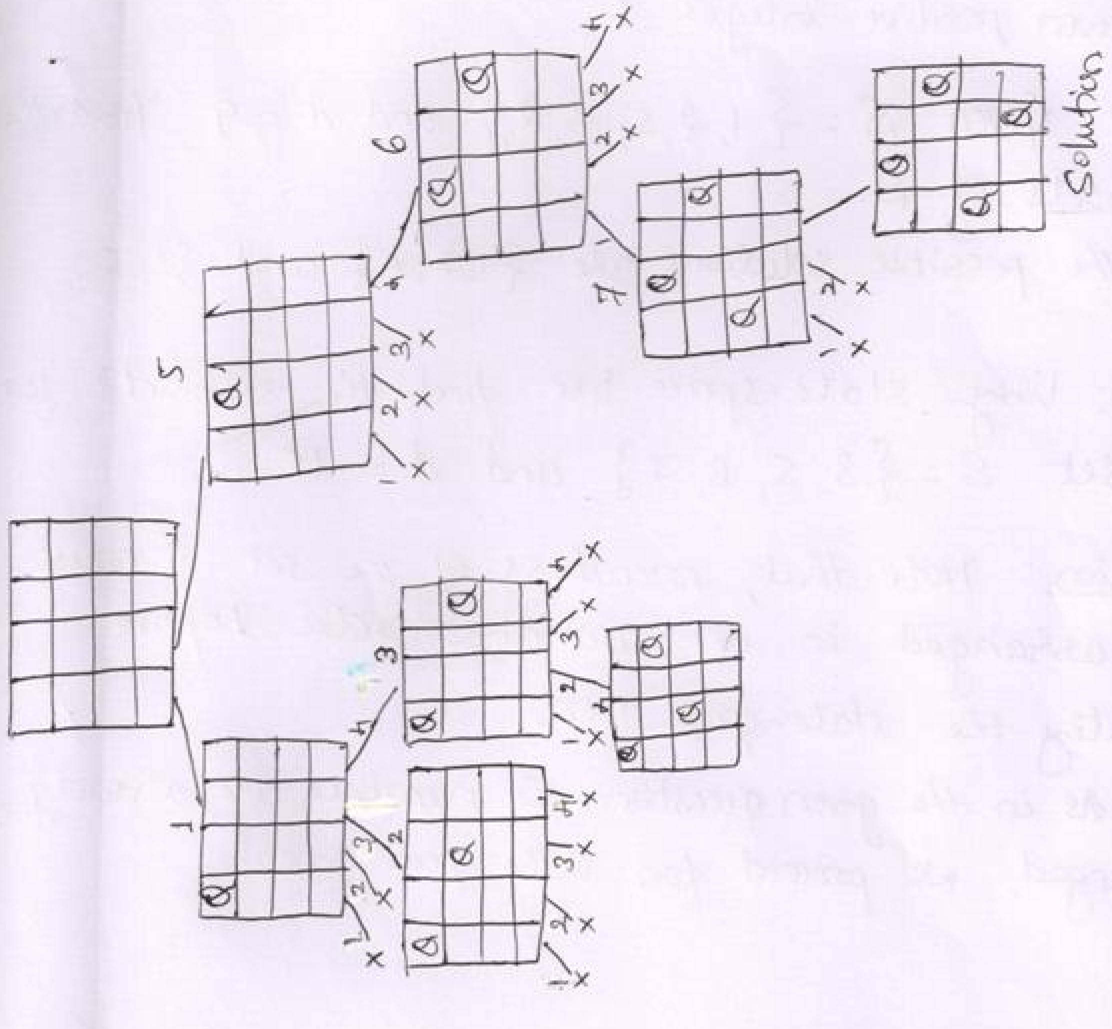
$n=3$

Q		
		Q

No place for 3rd queen

For n=4, there are two solutions. One of the solutions is shown in following state-space tree.

(Note that, the second solution for this problem will be mirror-image of following solution.)



(2) Subset - Sum Problem:

Here, the task is to find a subset of a given set whose elements add-up to a given integer. That is -

Ex Find a subset of a given set $S = \{s_1, s_2, \cdots s_n\}$ of n positive integers whose sum is equal to a given positive integer d.

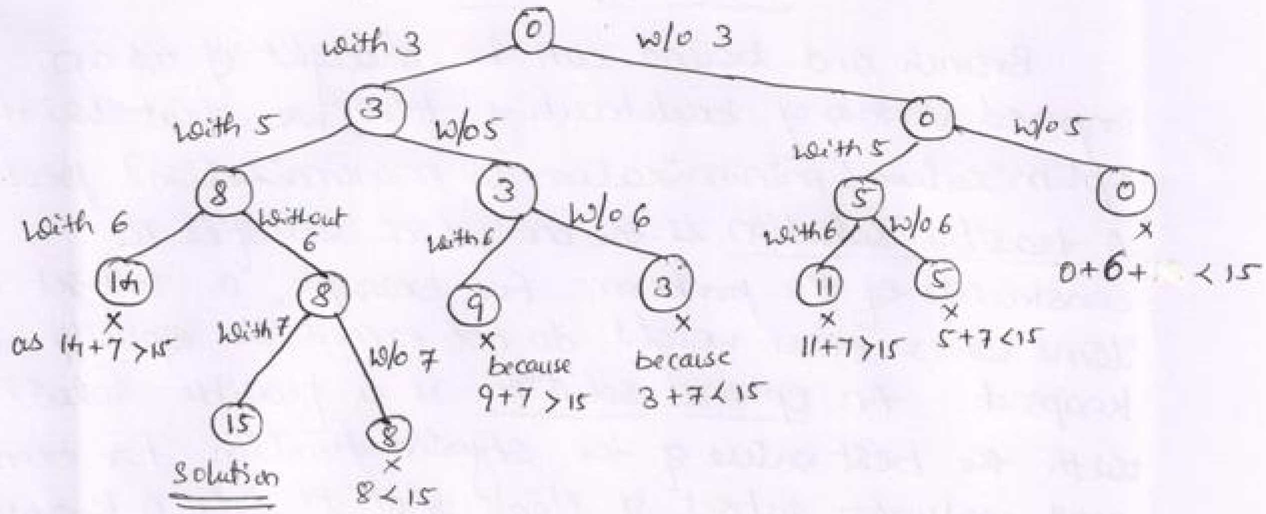
Ex1: Given $S = \{1, 2, 5, 6, 8\}$ and $d = 9$ find the subsets of S.

The possible solutions are $\{1, 2, 6\}$ and $\{1, 8\}$.

Ex2: Using state-space tree, find the subsets for a set $S = \{3, 5, 6, 7\}$ and $d = 15$.

solution: Note that, members of the set S must be arranged in an ascending order before creating the state-space tree.

As in the given question the numbers are already arranged, we proceed for state-space tree.

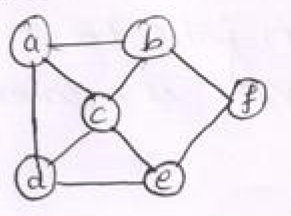


Thus the solution is $\{3, 5, 7\}$.

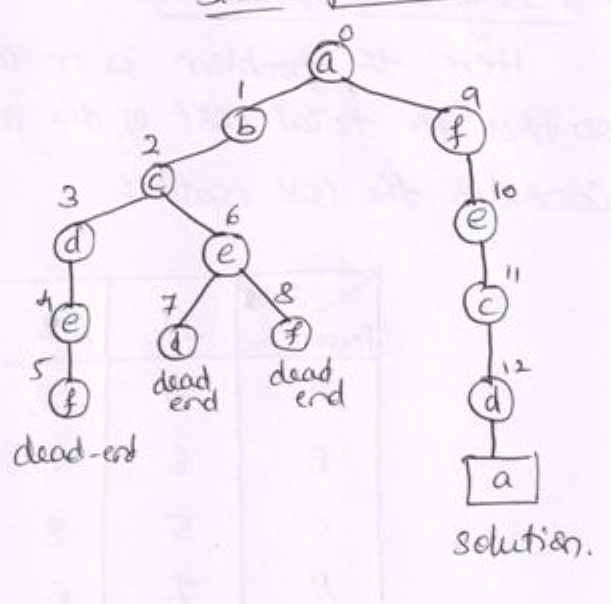
(3) Hamiltonian Circuit Problem

The problem is to check whether the given graph (directed/undirected) contains a Hamiltonian circuit. (Note: If a graph has a path which starts at one node, visits all the node exactly once and comes back to the starting node, then it is hamiltonian circuit).

Ex:



State-space tree



Branch & Bound

Branch and bound can be thought of as an improved version of backtracking. Here, we deal with optimization (minimization or maximization) problem. A feasible solution is the one which satisfies the constraints of the problem. For example, a subset of items whose total weight do not exceed the capacity of knapsack. An optimal solution is a feasible solution with the best value of the objective function. for example, most valuable subset of items that fit into a knapsack.

Branch-and-bound requires two additional items compared with backtracking:

* For every node of a state space tree, a way to provide a bound (either lower or upper) on the best value of the objective function.
* The value of the best solution found so far.

Here, we will discuss 3 problems –

(1) Assignment Problem
(2) Knapsack Problem
(3) Travelling Salesman Problem.

1. Assignment Problem:

Here, the problem is to assign n jobs to n people so that the total cost of the assignment is minimum.

Consider the cost matrix:

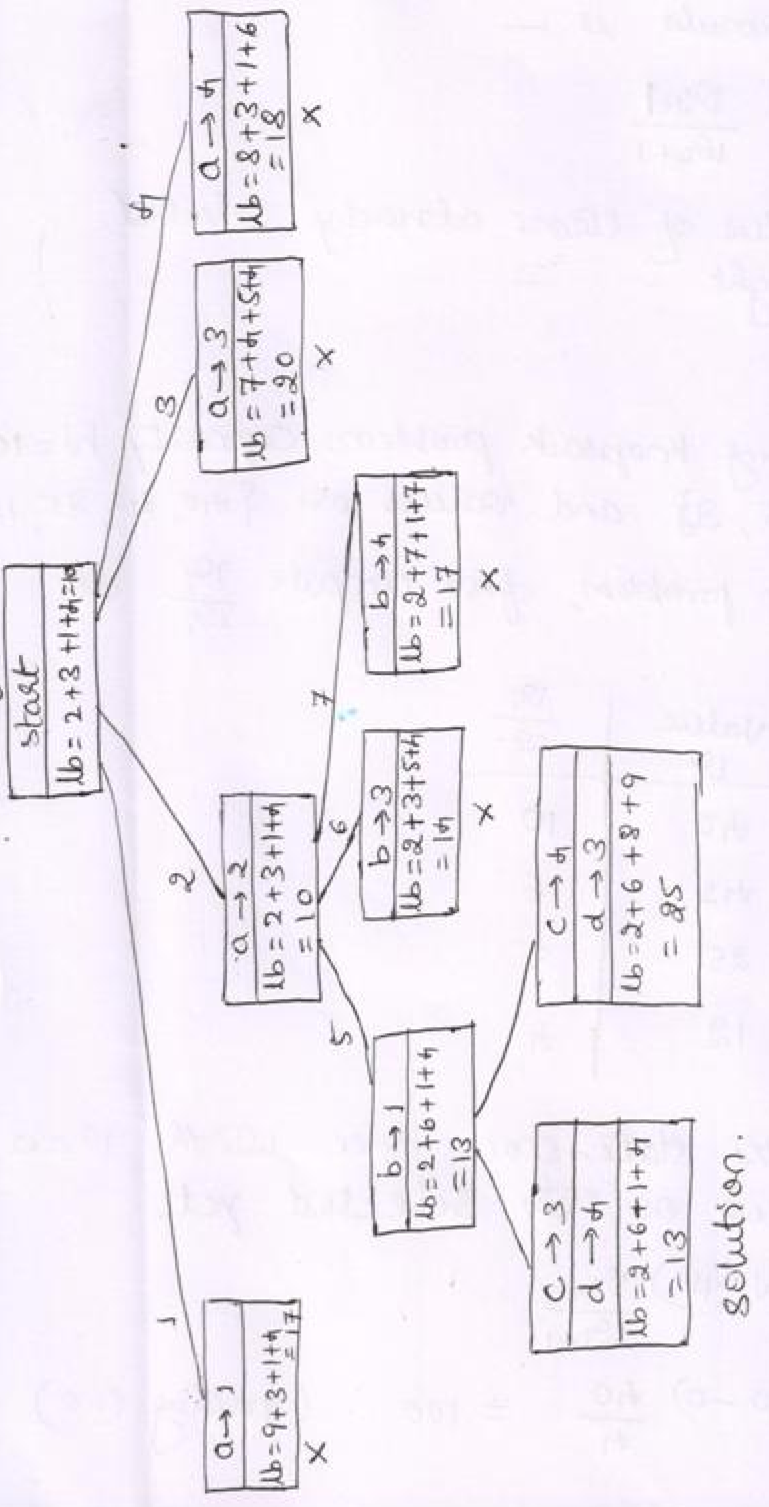
Person \ Jobs	J1	J2	J3	J4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

<u>Solution</u>: compute the lower bound by adding smallest elements in every row. So, here, lower bound would be –

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

Now, at every step, we have to keep finding minimal solution without violating the constraints of the problem.

The state-space tree is given below –



Hence, the cost of given assignment problem is = 13

And, the job allocation would be –

Person a → Job 2

Person b → Job 1

Person c → Job 3

Person d → Job 4

(2) Knapsack Problem:

There are n items with weights $w_1, w_2, \ldots w_n$, and values $v_1, v_2, v_3, \ldots v_n$. The total capacity of knapsack is W. The problem is to maximize the total profit such that total weight of the items is less than or equal to W.

As it is a maximization problem, we need to find upper bound. The formula is —

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

Here, v — total value of items already selected.
w — " weight — ——

Example:

Solve the following knapsack problem: capacity W = 10. Weights are: {4, 7, 5, 3} and values are {40, 42, 25, 12}

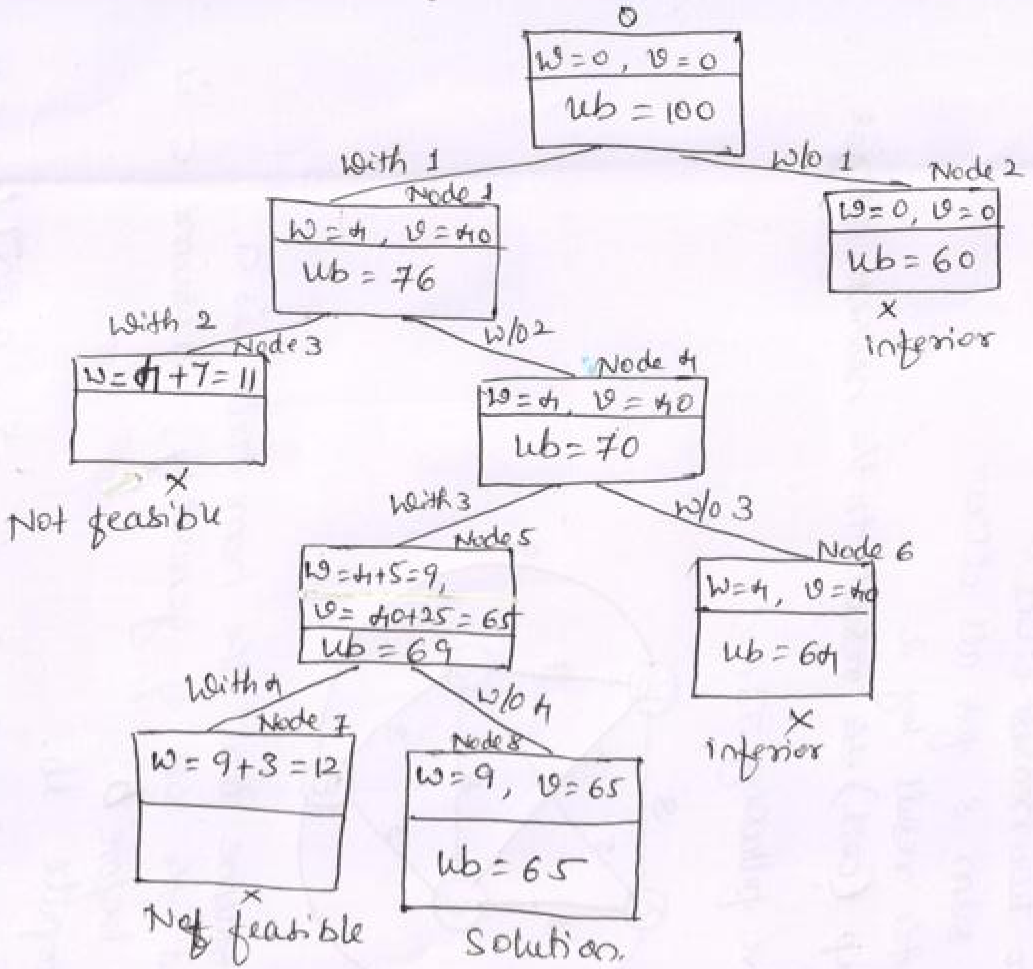
Solution: For a given problem, first compute $\frac{v_i}{w_i}$ as below —

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

We have to start the state-space tree with $v=0$ and $w=0$, because no item selected yet.

And, $ub = v + (W-w) \frac{v_{i+1}}{w_{i+1}}$

$$= 0 + (10-0) \cdot \frac{40}{4} = 100 \quad (\text{treating } i=0)$$



Upper bound calculation

Node 1: $i=1,\ w=4,\ v=40$

$ub = v + (W-w)\dfrac{v_2}{w_2}$

$= 40 + (10-4).\dfrac{42}{7}$

$= 76$

Node 2: $i=1,\ w=0,\ v=0$

$ub = 0 + (10-0).\dfrac{42}{7} = 60$

Node 4: $i=2,\ w=4,\ v=40$

$ub = 40 + (10-4).\dfrac{v_3}{w_3}$

$= 40 + 6 \times \dfrac{25}{5} = 70$

Node 5: $i=3,\ w=9,\ v=65$

$ub = 65 + (10-9).\dfrac{v_4}{w_4}$

$= 65 + 1.\dfrac{12}{3} = 69$

Node 6: $i=3,\ w=4,\ v=40$

$ub = 40 + (10-4).\dfrac{12}{3} = 64$

Node 8: As there are no additional items, ub is treated as sum of all items selected so far.
Hence $ub = 65$.

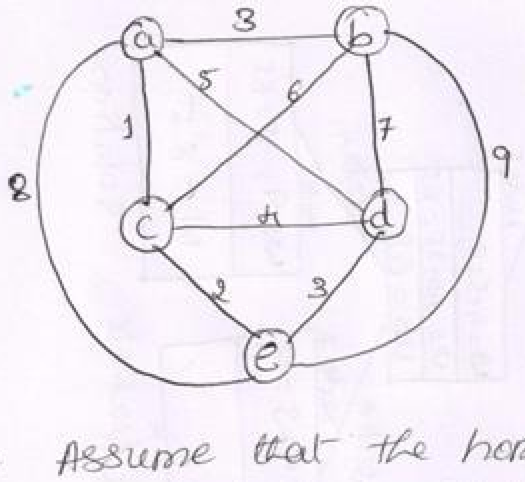
(3) Travelling Salesman Problem:

TSP is represented by a complete graph of n nodes. Here, n cities are connected to each other. A salesman should start from one city (home town) and visit every city exactly once to come back to his home town. A distance matrix D gives the distances between every pair of cities. The travel path of the salesman should be of minimum distance.

Computation of lower bound for TSP using branch-and-bound technique involves following steps –

* For each city i, find sum S_i of distances from city i to two nearest cities.
* Compute sum S for all cities.
* Divide the result by 2.
* Round up (ceil) the result to the nearest integer.

Example: Solve following TSP:

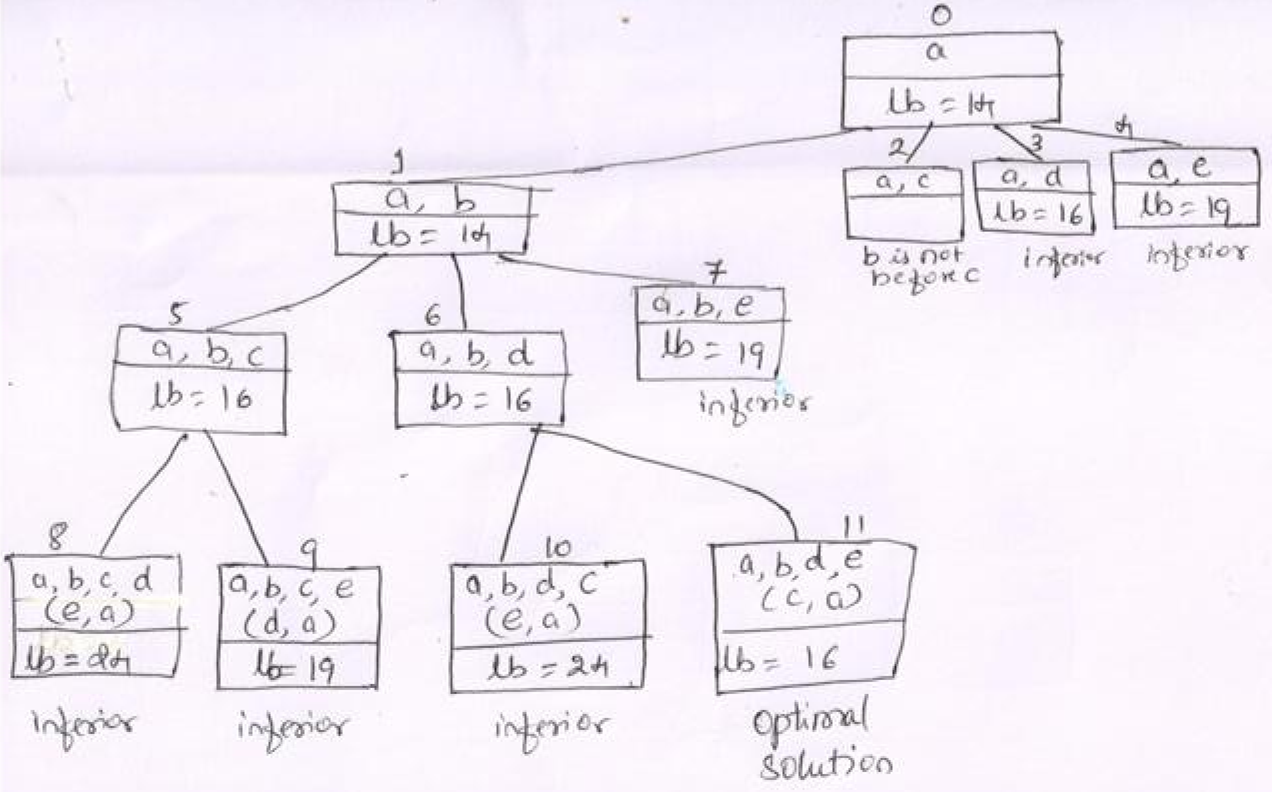


Solution: Assume that the home town is a. Also, without the loss of generosity, assume c is not visited before b.

Now compute lb.

$$lb = [\underbrace{(1+3)}_{a} + \underbrace{(3+6)}_{b} + \underbrace{(1+2)}_{c} + \underbrace{(4+3)}_{d} + \underbrace{(2+3)}_{e}]/2 = 14$$

Nearest distances from



Hence the travel path of a salesman would be –

$a \to b \to d \to e \to c \to a$

And the distance travelled = 16

lower-bound calculation

Node 3 : $a \to d$

$lb = \dfrac{[(5+1)+(3+6)+(1+2)+(5+3)+(2+3)]}{2}$

$= 31/2 = 16$ (ceil operation)

Node 4 : $a \to e$

$lb = [(8+1)+(3+6)+(1+2)+(4+3)+(8+2)]/2$

$= 38/2 = 19$

Node 5 : $a \to b \to c$

$lb = [(3+1)+(6+3)+(6+1)+(4+3)+(2+3)]/2$

$= 32/2 = 16$

Node 6 : $a \to b \to d$

$lb = [(3+1)+(7+3)+(1+2)+(7+3)+(2+3)]/2$

$= 32/2 = 16$

Node 8 : $a \to b \to c \to d \to e \to a$

$lb = [(3+8)+(6+3)+(4+6)+(4+3)+(3+8)]/2$

$= 48/2 = 24$

Node 9 : $a \to b \to c \to e \to d \to a$

$lb = [(3+5)+(3+6)+(6+2)+(2+3)+3+5)]/2$

$= 38/2 = 19$

Node 10 : $a \to b \to d \to c \to e \to a$

$lb = [(3+8)+(3+7)+(7+4)+(4+2)+(2+8)]/2$

$= 48/2 = 24$

Node 11 : $a \to b \to d \to e \to c \to a$

$lb = [(3+1)+(3+7)+(7+3).+(3+2)+(2+1)]/2$

$= 32/2 = 16$