

Unit V

Branch and Bound

Branch and Bound:

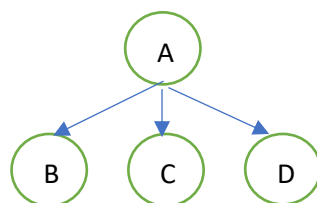
The Branch and Bound Technique is a problem solving strategy, used for optimization problems, where the goal is to minimize a certain value. The optimized solution is obtained by means of a state space tree (A state space tree is a tree where the solution is constructed by adding elements one by one, starting from the root. This method is best when used for combinatorial problems).

In this technique, the first step is to create a function U (which represents an upper bound to the value that node and its children shall achieve), that we intend to minimize. We call this function the objective function. Note that the branch and bound technique can also be used for maximization problems, since multiplying the objective function by -1 converts the problem to a minimization problem.

Live Node: Live node is a node that has been generated but whose children have not yet been generated.

E-Node: E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Branch-and-bound: Branch-and-bound refers to all state space search methods in which all children of an E-node are generated before any other live node becomes E-node.



In the above tree when A derives the nodes B C and D. Which node has to be selected next to explore ?

Based on the order in which the tree is to be searched following are different Branch and Bound techniques

FIFO Branch and Bound

LIFO Branch and Bound

LC Branch and Bound

Least Cost (LC) search:

In both LIFO and FIFO Branch and Bound the selection rule for the next E-node is fixed. These selection rule for the next E-node 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 be speeded by using an “intelligent” ranking function $c(x)$ for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:

$$c(x) = f(h(x)) + g(x) \text{ where, } c(x) \text{ is the cost of } x.$$

$h(x)$ is the cost of reaching x from the root and

$f(.)$ is any non-decreasing function.

$g(x)$ is an estimate of the additional effort needed to reach an answer node from x .

A search strategy that uses a cost function $c(x) = f(h(x)) + g(x)$ to select the next E-node would always choose for its next E-node a live node with least $c(x)$ is called a LC-search (Least Cost search)

Control abstraction for LC Search:

Let t be a state space tree and $c(x)$ a cost function for the nodes in t . If x is a node in t , then $c(x)$ is the minimum cost of any answer node in the subtree with root x .

LC-search uses $c(x)$ to find an answer node. The algorithm uses two functions Least() and Add() to delete and add a live node from or to the list of live nodes, respectively. Least() finds a live node with least $c()$. This node is deleted from the list of live nodes and returned.

Algorithm LCSearch outputs the path from the answer node it finds to the root node t . This is easy to do if with each node x that becomes live, we associate a field parent which gives the parent of node x . When the answer node g is found, the path from g to t can be determined by following a sequence of parent values starting from the current E-node (which is the parent of g) and ending at node t .

```
Algorithm LCSearch(t)
{
    //Search t for an answer node
    if *t is an answer node then output *t and return;
    E := t;          //E-node.
    initialize the list of live nodes to be empty;
    repeat
    {
        for each child x of E do
        {
            if x is an answer node then output the path from x to t and return;
            Add (x);                      //x is a new live node.
            (x → parent) := E;            // pointer for path to root
        }
        if there are no more live nodes then
        {
            write ("No answer node");
            return;
        }
        E := Least();
    } until (false);
}
```

LC Search for 15 Puzzle Problem:

The 15 puzzle consists of 15 squares numbered from 1 to 15 that are placed in a box leaving one position out of the 16 empty. The goal is to reposition the squares by sliding empty once at a time into the configuration shown above.

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Problem State

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal State

A depth first state space tree can be used to find the solution. when the next moves are attempted in the order: move the empty space up, right, down and left. The search of the state space tree continues.

We associate a cost $c(x)$ with each node in the state space tree.

$c(x)$ is as follows:

$$C(x) = f(h(x)) + g(x)$$

where, $f(h(x))$ is the length of the path from the root to node x and

$g(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x .

Here, $g(x)$ is the number of nonblank tiles not in their goal position.

Solve the following 15 Puzzle problem

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

The LC-search, begin with the root as the E-node and generate all childnodes 2, 3, 4 and 5 as shown in below figure.

$$C(x) = f(h(x)) + g(x)$$

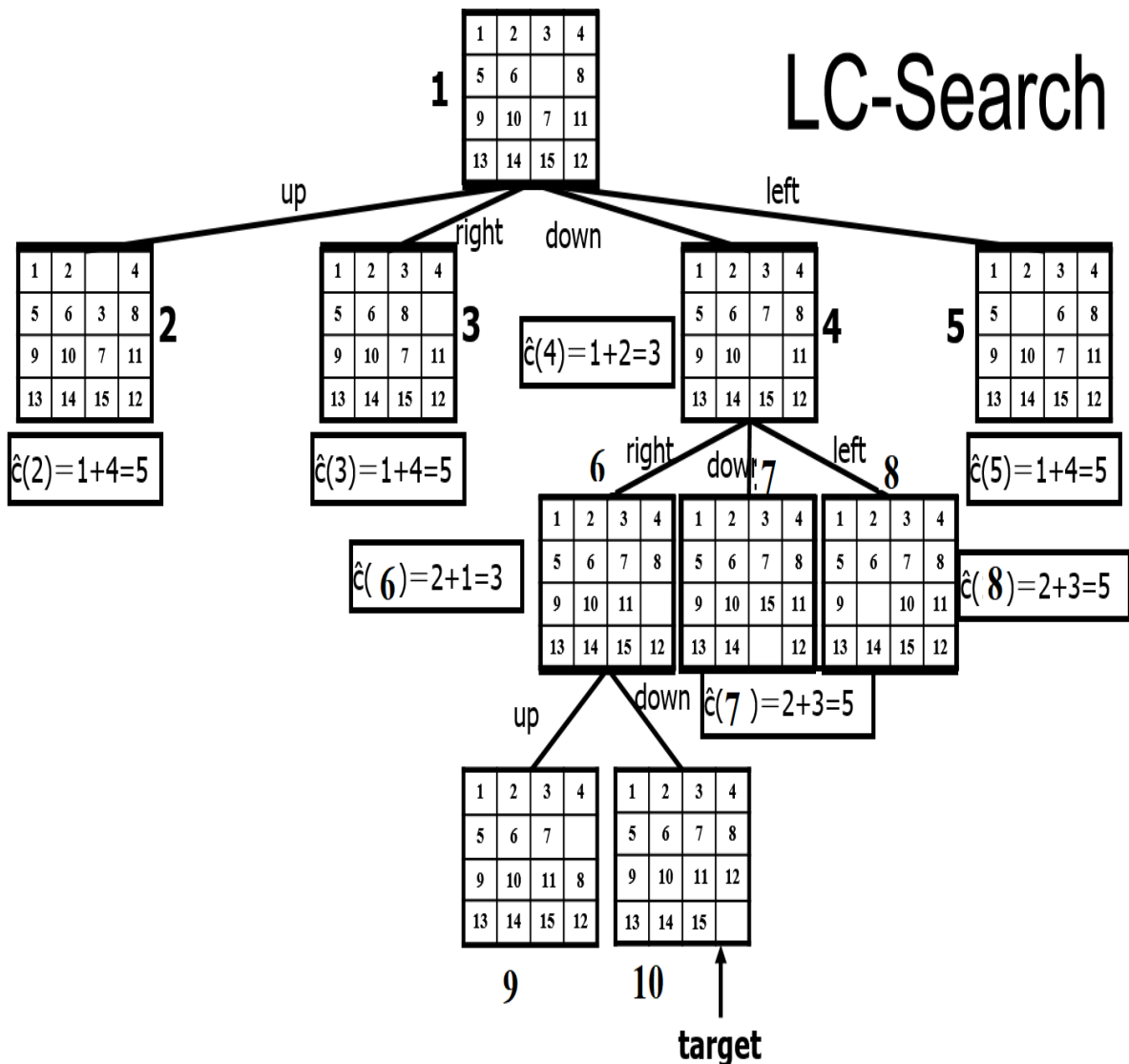
When tile moved towards Up, Cost (2) = 1 + 4 = 5

When tile moved towards right, Cost (3) = 1 + 4 = 5

When tile moved towards down, (4) = 1 + 2 = 5

When tile moved towards left (5) = 1 + 4 = 4

The next node to become the E-node is a live node with least cost
Node 4 becomes the E-node and its children are generated.



The possible moves at E Node 4 are right, down and left only

When tile moved towards right, Cost (6) = 2 + 1 = 3

When tile moved towards down, (7) = 2 + 3 = 5

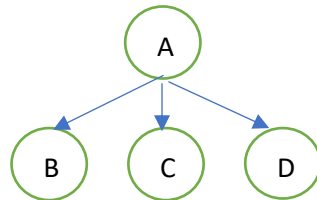
When tile moved towards left (8) = 2 + 3 = 5

Now E-Node will be node 6 having least cost

New nodes 9 and 10 are derived in which the 10 is the goal state.

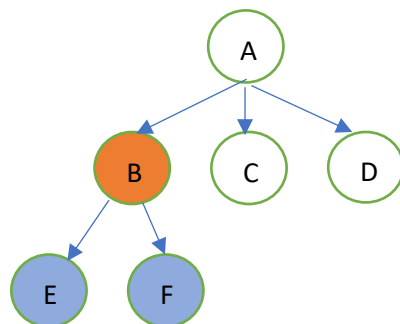
FIFO Branch and Bound

In The First-In-First-Out approach we follow the **queue** mechanism to create the state-space tree which is similar to breadth first search. the elements at a particular level are all searched, and then the elements of the next level are searched. i.e., if Nodes at level 1 are explored from left to right then only we move to level 2 and repeat same order.



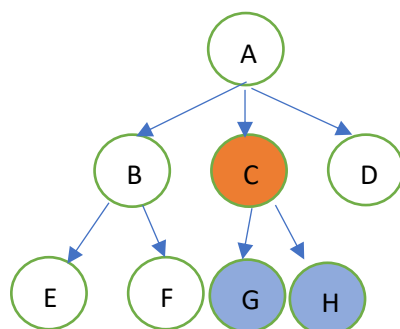
B	C	D	
---	---	---	--

Here once the root node A is explored, it generates the child nodes called to be B, C and D. In FIFO approach we explore the node which reached first into the queue i.e., here B will be selected when it derives new child nodes they will be inserted into the the queue



B	C	D	E	F	
---	---	---	---	---	--

As B is explored now next node in queue is C, and it will be explored



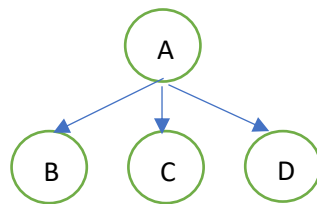
B	C	D	E	F	G	H	
---	---	---	---	---	---	---	--

LIFO Branch and Bound :

In LIFO Branch and Bound, Stack mechanism is used to explore the nodes i.e., the last inserted child node will be explored first.

LC Branch and Bound :

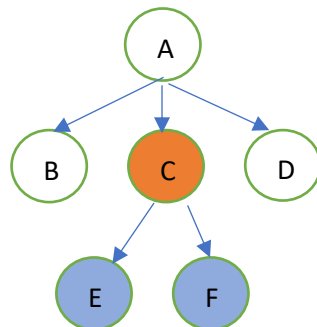
In Least Cost Branch and Bound approach, Once the root node is explored and its child nodes are maintained in a list (unlike queue) as live nodes. Now cost of each live node will be calculated which ever node has the least cost that node will be explored first (its child nodes will be inserted into existing list) the same process will be repeated until a feasible solution is obtained.



Live Nodes

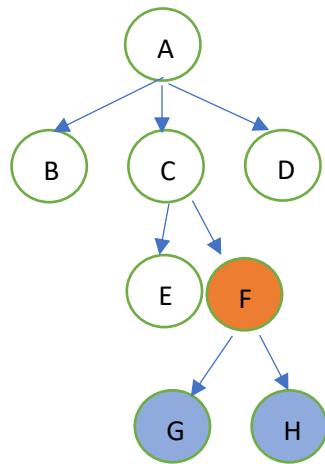
B	C	D	
---	---	---	--

Now from the live nodes B, C and D, say Node C has the least cost. Then node C will be explored and its child nodes will be inserted into the list of existing live nodes and C will be called as dead node.



B	C	D	E	F	
---	---	---	---	---	--

As C is already explored, now whichever node has least cost among all the live nodes i.e., B, D, E and F that node will be selected to explore. Let say F has least cost among them then explore F and add its child nodes to list again, and process repeats until feasible solution is obtained.



B	C	D	E	F	G	H	
---	---	---	---	---	---	---	--

0/1 Knapsack using LC Branch and Bound

The 0/1 problem is a maximization problem, whereas the Branch and Bound method is for minimization problems. Hence, the values will be multiplied by -1 so that this problem gets converted into a minimization problem.

the procedure to solve the problem is as follows are:

- Set initial global upper bound $G = \infty$
- Calculate the cost function $c(x)$ and the Upper bound $u(x)$ for each node.

Here, the $(i + 1)^{\text{th}}$ level indicates whether the i^{th} object is to be included or not.
- If the $u(x)$ value of any node is smaller than G , then set $G = u(x)$
- If The cost function $c(x)$ for a node is greater than G , then the node need not be explored further. Hence, we can kill this node.
- The next node to be checked after reaching all nodes in a particular level will be the one with the least cost function value among the unexplored nodes.
- While including an object, one needs to check whether the adding the object crossed the Maximum knapsack size. If it crosses when should not consider that path.

Solve the following 0/1 knapsack problem using LCBB, where $n = 4$, profits = {10, 10, 12, 18}, weights = {2, 4, 6, 9} and $M = 15$.

Negate the profits so problem turns to minimization from maximization

$$\text{Profits} = \{-10, -10, -12, -18\}$$

Initially Set global upper bound $G = \infty$

Calculating $U(x)$ for first node

Place first item in knapsack. Remaining weight of knapsack is $15 - 2 = 13$.

Place next item w_2 in knapsack and the remaining weight of knapsack is $13 - 4 = 9$.

Place next item w_3 in knapsack then the remaining weight of knapsack is $9 - 6 = 3$.

No fractions are allowed in calculation of upper bound so w_4 cannot be placed in knapsack.

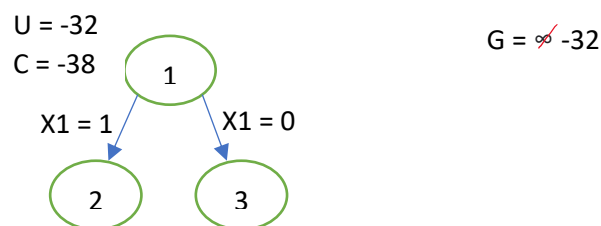
$$U(x) = P_1 + P_2 + P_3 = -10 - 10 - 12$$

$$\text{Also set } G = u(x) = -32$$

Calculating Cost $c(x)$

To calculate cost we should place w_4 in knapsack, since fractions are allowed in calculation of cost.

$$C(x) = -10 - 10 - 12 + ((3/9) \times -18) = -32 - 6 = -38$$



Now we will calculate upper bound and cost for nodes 2, 3.

For node 2, $x_1 = 1$, means we must place first item in the knapsack.

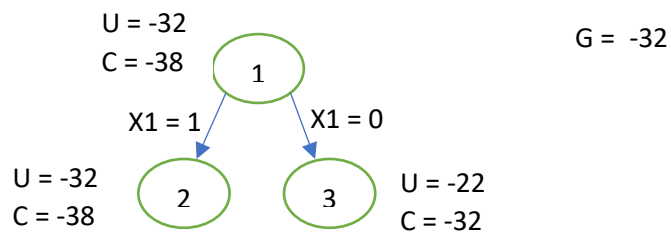
$$U(x) = -10 - 10 - 12 = -32,$$

$$C(x) = -10 - 10 - 12 + ((3/9) \times -18) = -32 - 6 = -38$$

For node 3, $x_1 = 0$, means we should not place first item in the knapsack.

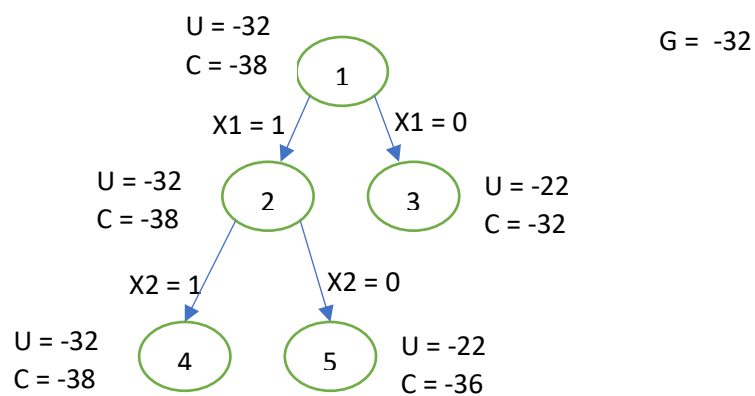
$$U(x) = -10 - 12 = -22,$$

$$C(x) = -10 - 12 + (5/9) \times -18 = -10 - 12 - 10 = -32$$

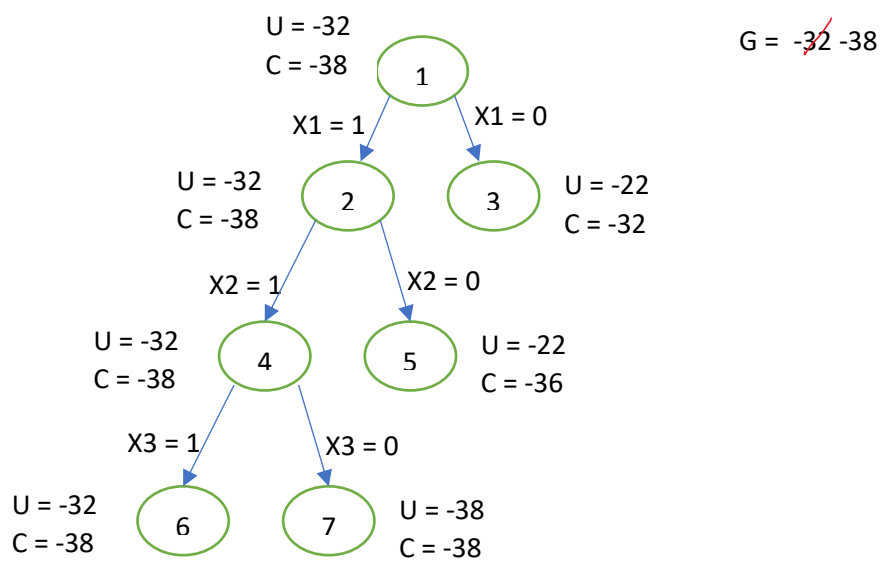


Now among Nodes 2 and 3, node 2 has least cost, so explore node 2

Calculate upper bound and cost for the child nodes



Now Explore Node 4 which has least cost among the live nodes 3,4,5 and calculate u and c value for its child nodes

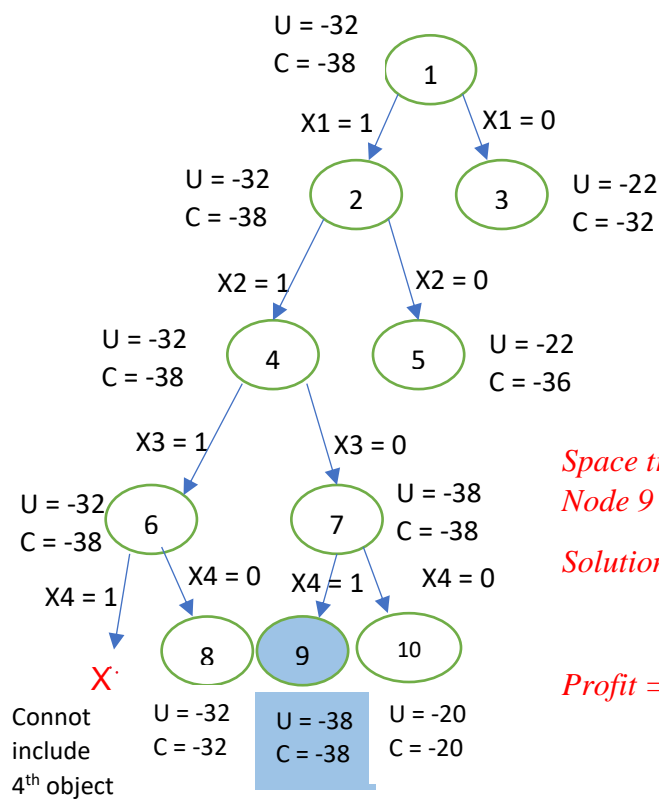


For node 7, upper bound is smaller than Global upper bound G, So set $G = -38$

Now Cost of nodes 3 and 5 are greater than G, So kill the nodes 3 and 5

If we explore node 6 choosing $x_4 = 1$, it is not a valid solution because the weight of all 4 objects together(21) will be greater than knapsack weight(15)

Now explore node 7 and calculate upper bound and cost of child nodes



$G = -38$

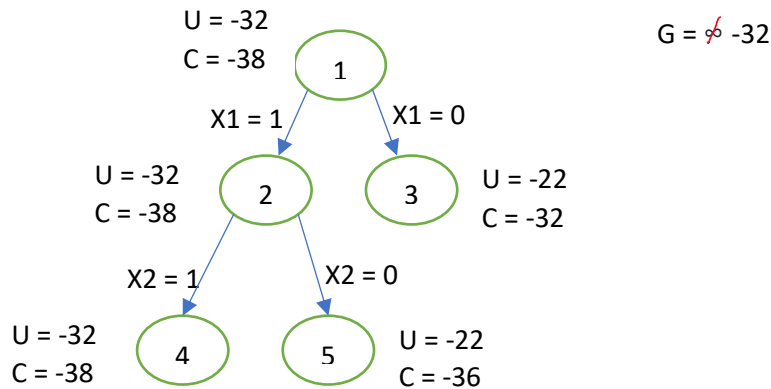
Space tree is drawn for all four objects and Node 9 having least cost is the solution node.

*Solution = $\{X_1 = 1, X_2 = 1, X_3 = 0, X_4 = 1\}$
= $\{1, 1, 0, 1\}$*

Profit = 38

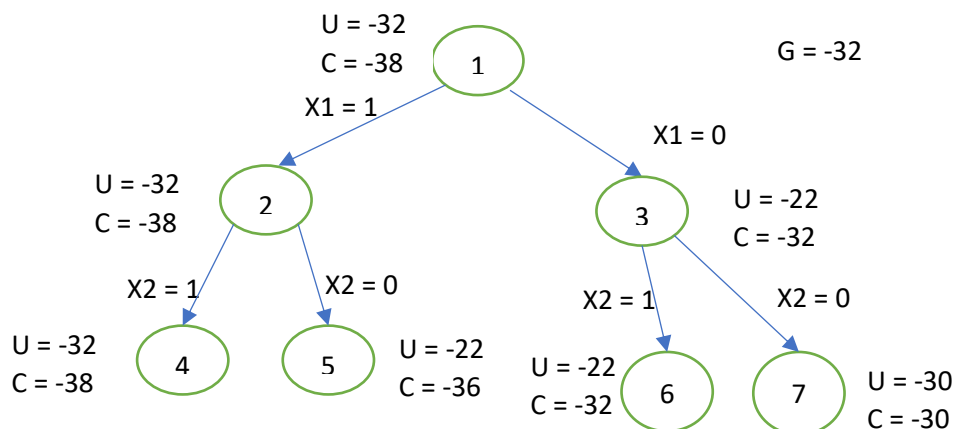
For 0/1 Knapsack using FIFO Branch and Bound

unlike LCBB the nodes will be explored in Queue mechanism. The first arrived live node will be explored first instead of node having least cost.



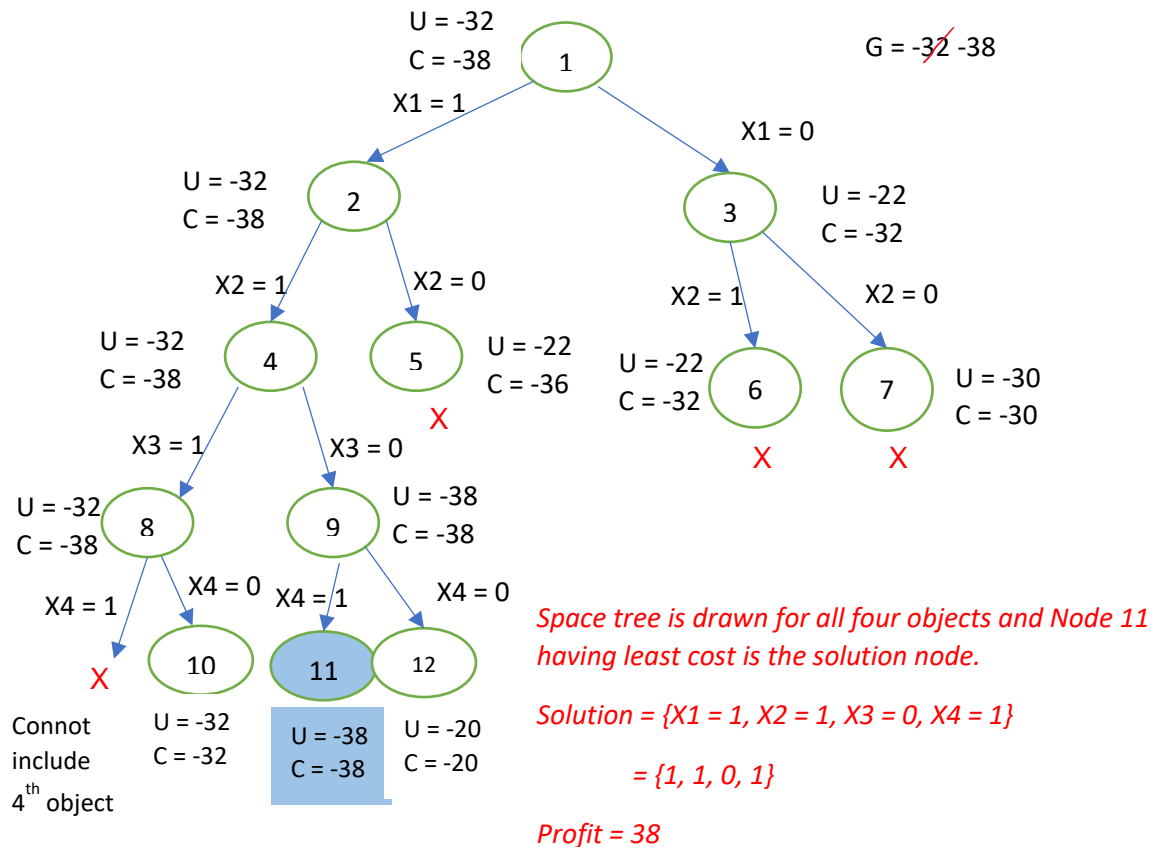
Here the live nodes in queue are 3 4 5.

Where 3 will be explored before than 4 and 5



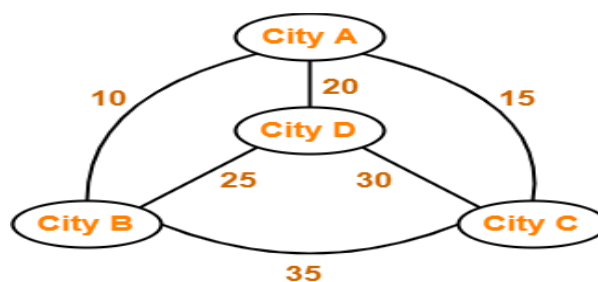
Now Among the live nodes 4, 5, 6 and 7

4 will be explored first and calculate the cost and upper bound for those nodes



Travelling Salesman Problem

Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.



Travelling Salesman Problem

If A is the starting city, then a TSP solution for the graph is-

$$A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$$

$$\text{Cost of the tour} = 10 + 25 + 30 + 15$$

Procedure for solving traveling sale person problem:

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:

- a) *Row reduction:* Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.
- b) Find the sum of elements, which were subtracted from rows.
- c) Apply column reductions for the matrix obtained after row reduction.

Column reduction: Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.

- d) Find the sum of elements, which were subtracted from columns.
- e) Obtain the cumulative sum of row wise reduction and column wise reduction.

Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.

Associate the cumulative reduced sum to the starting state as lower bound and ∞ as upper bound.

2. Calculate the reduced cost matrix for every node R. Let A is the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge $\langle i, j \rangle$ in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:

- a) Change all entries in row i and column j of A to ∞ .
- b) Set A (j, 1) to ∞ .
- c) Reduce all rows and columns in the resulting matrix except for rows and column containing only ∞ . Let r is the total amount subtracted to reduce the matrix.
- c) Find $\bar{c}(S) = \bar{c}(R) + A(i, j) + r$, where 'r' is the total amount subtracted to reduce the matrix, $\bar{c}(R)$ indicates the lower bound of the i^{th} node in (i, j) path and $\bar{c}(S)$ is called the cost function.

3. Repeat step 2 until all nodes are visited.

Travelling Salesperson Problem

Solve the Travelling Salesperson Problem given the adjacency Matrix

	A	B	C	D
A	∞	4	12	7
B	5	∞	∞	18
C	11	∞	∞	6
D	10	2	3	∞

A is the Root Node

Calculate reduction Cost = Row Reduction + column Reduction

Row Reduction

	A	B	C	D
A	∞	4	12	7
B	5	∞	∞	18
C	11	∞	∞	6
D	10	2	3	∞

Row Reduction = 4 + 5 + 6 + 2 = 17

	A	B	C	D
A	∞	0	8	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	1	∞

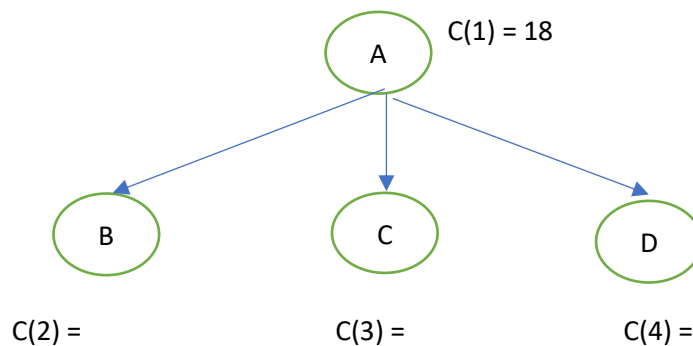
Column Reduction

	A	B	C	D
A	∞	0	8	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	1	∞

Column Reduction = 1

Reduction Cost = 17 + 1 = 18

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞



A - B

Parent Node is A, its Reduced Matrix is

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

Step 1 : Set Outdegree of Vertex A to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

Step 2 : Set Indegree of Vertex B to ∞


	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	5	∞	∞	0
D	8	∞	0	∞

Step 3 : set B \rightarrow A as ∞

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	13
C	5	∞	∞	0
D	8	∞	0	∞

Row Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	13
C	5	∞	∞	0
D	8	∞	0	∞




	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	5	∞	∞	0
D	8	∞	0	∞

Row Reduction = 13

Column Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	5	∞	∞	0
D	8	∞	0	∞



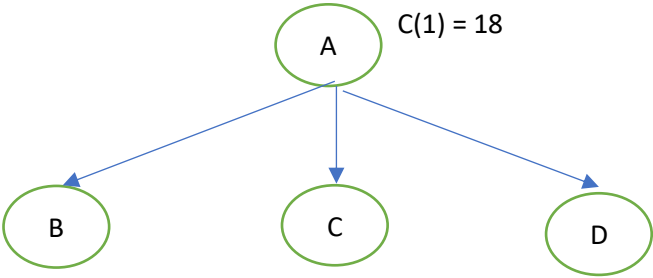
	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	0	∞	∞	0
D	3	∞	0	∞

Column Reduction = 5

Reduction Cost = 13 + 5 = 18

$\text{Cost} = G[A,B] + \text{Cost}(1) + \text{Reduction Cost} = 0 + 18 + 18 = 36$

(Parent A) (Parent A)



C(2) = 36

C(3) =

C(4) =

A - C

Parent Node is A, its Reduced Matrix is

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

Step 1 : Set Outdegree of Vertex A to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

Step 2 : Set Indegree of Vertex C to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	∞	∞

Step 3 : set C \rightarrow A as ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

Row Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞



	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

Row Reduction = 0

Column Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞



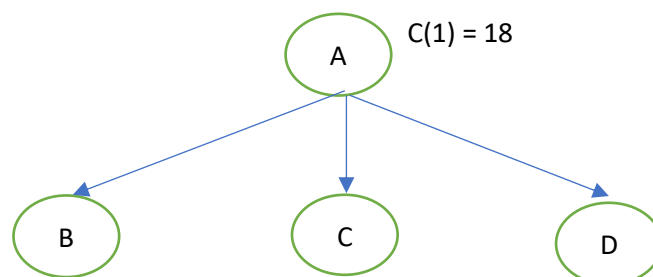
	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

Column Reduction = 0

Reduction Cost = 0 + 0 = 0

$$\text{Cost} = G[A, C] + \text{Cost}(1) + \text{Reduction Cost} = 7 + 18 + 0 = 25$$

(Parent A) (Parent A)



C(2) = 36

C(3) = 25

C(4) =

A - D

Parent Node is A, its Reduced Matrix is

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

Step 1 : Set Outdegree of Vertex A to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

Step 2 : Set Indegree of Vertex D to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	5	∞	∞	∞
D	8	0	0	∞

Step 3 : set D \rightarrow A as ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	5	∞	∞	∞
D	∞	0	0	∞

Row Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	5	∞	∞	∞
D	∞	0	0	∞

Row Reduction = 5



	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	0	∞	∞	∞
D	∞	0	0	∞

Column Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	0	∞	∞	∞
D	∞	0	0	∞

Column Reduction = 0

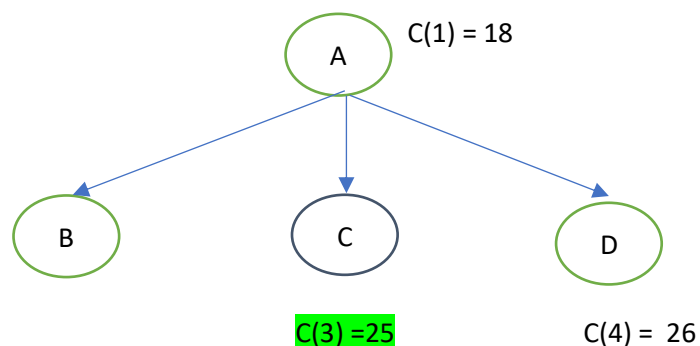


	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	0	∞	∞	∞
D	∞	0	0	∞

Reduction Cost = 5 + 0 = 5

$$\text{Cost} = G[A,D] + \text{Cost}(1) + \text{Reduction Cost} = 3 + 18 + 5 = 26$$

(Parent A) (Parent A)



Node with Least Cost

A – C -B

Parent Node is A-C, its Reduced Matrix is

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

Step 1 : Set Outdegree of Vertex C to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	∞
D	8	0	∞	∞

Step 2 : Set Indegree of Vertex B to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	∞
D	8	∞	∞	∞

Step 3 : set B -> A as ∞

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	13
C	∞	∞	∞	∞
D	8	∞	∞	∞

Row Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	13
C	∞	∞	∞	∞
D	8	∞	∞	∞

Row Reduction = 13+8

Column Reduction

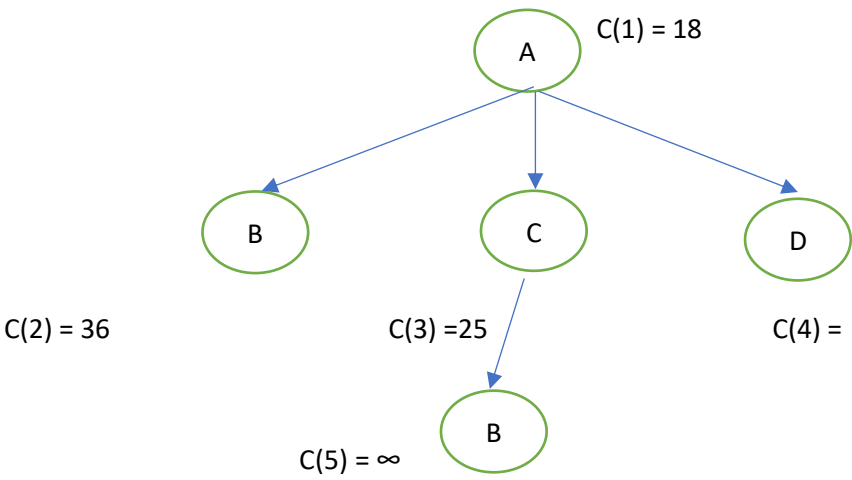
	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	∞	∞	∞	∞
D	8	∞	∞	∞

Column Reduction = 0

Reduction Cost = 21 + 0 = 21

$$\text{Cost} = G[C,B] + \text{Cost}(3) + \text{Reduction Cost} = \infty + 25 + 21 = \infty$$

(Parent A) (Parent A)



A – C -D

Parent Node is A-C, its Reduced Matrix is

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

Step 1 : Set Outdegree of Vertex C to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	∞
D	8	0	∞	∞

Step 2 : Set Indegree of Vertex D to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	8	0	∞	∞

Step 3 : set D -> A as ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	∞	0	∞	∞

Row Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	∞	0	∞	∞

Row Reduction = 0

Column Reduction

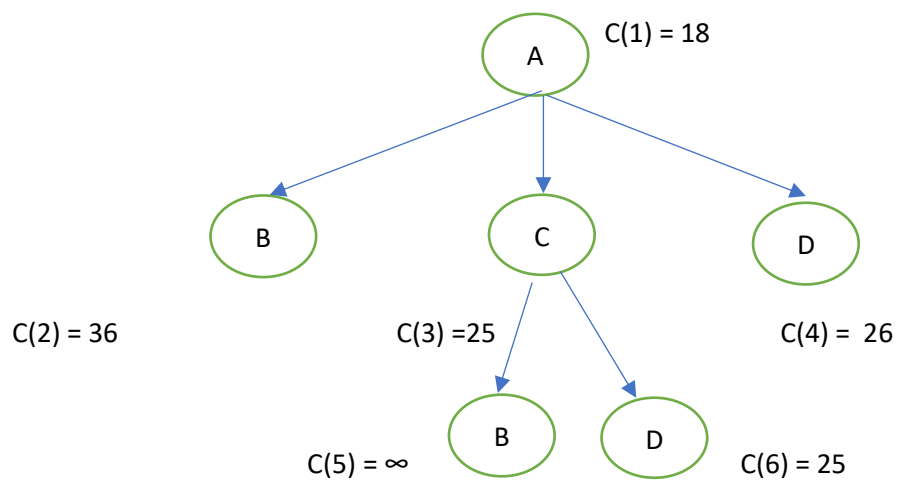
	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	∞	∞	∞	∞
D	8	∞	∞	∞

Column Reduction = 0

Reduction Cost = 0 + 0 = 0

$$\text{Cost} = G[C,D] + \text{Cost}(3) + \text{Reduction Cost} = 0 + 25 + 0 = 0$$

(Parent) (Parent)



A – C -D - B

Parent Node is A-C-D, its Reduced Matrix is

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	∞	∞	∞	∞
D	8	∞	∞	∞

Step 1 : Set Outdegree of Vertex D to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	∞	∞	∞	∞
D	∞	∞	∞	∞

Step 2 : Set Indegree of Vertex B to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	∞	∞	∞	∞
D	∞	∞	∞	∞

Step 3 : set B -> A as ∞

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	∞	∞	∞	∞
D	∞	∞	∞	∞

Row Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	∞	∞	∞	∞
D	∞	∞	∞	∞

Row Reduction = 0

Column Reduction

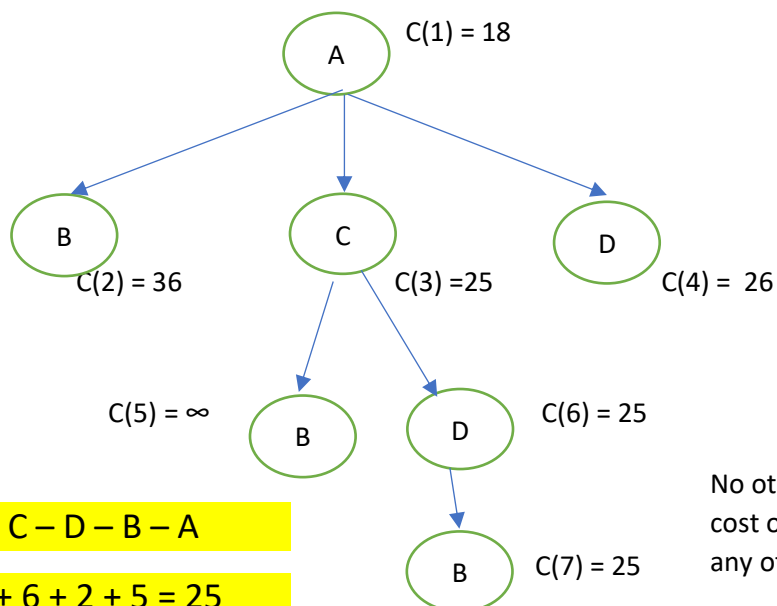
	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	∞	∞	∞	∞
D	∞	∞	∞	∞

Column Reduction = 0

Reduction Cost = 0 + 0 = 0

$$\text{Cost} = G[D,B] + \text{Cost}(5) + \text{Reduction Cost} = 0 + 25 + 0 = 0$$

(Parent) (Parent)



Path = A - C - D - B - A

Cost = 12 + 6 + 2 + 5 = 25

No other nodes has the cost less than cost of node B, so no need to explore any other node

P, NP, NP-Complete and NP-Hard

class P - Problems

- The class P consists of those problems that can be solved and verified in polynomial time.
- More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k ,
- For P Class problems, Deterministic algorithms can be written
- P Problems are subset of NP Problems
- If problems belongs to class P, it is easy to find the solution

Example

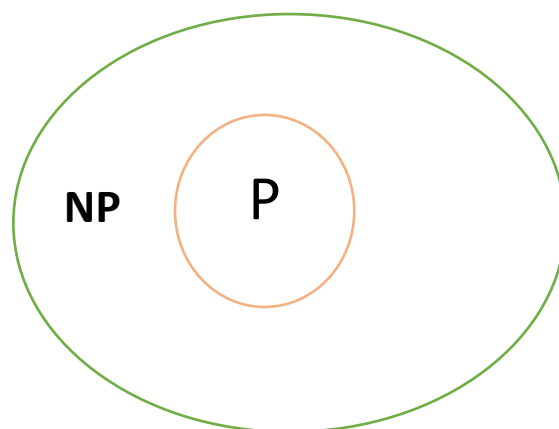
Linear search, Binary Search, matrix multiplication

NP - Non-Deterministic polynomial time

- Solution to NP Problems cannot be obtained in polynomial time, but if the solution is given it can be verified in polynomial time.
- It is the collection of decision problems that can be solved by a non-deterministic algorithm in polynomial time.
- NP problems are super set of P Problems

Example

Travelling Salesperson, knapsack problem



Deterministic Algorithms :

For the Deterministic algorithms, given a particular input it will always produce the same output. In Deterministic algorithm the path of execution for algorithm is same in every execution

Non-Deterministic Algorithms :

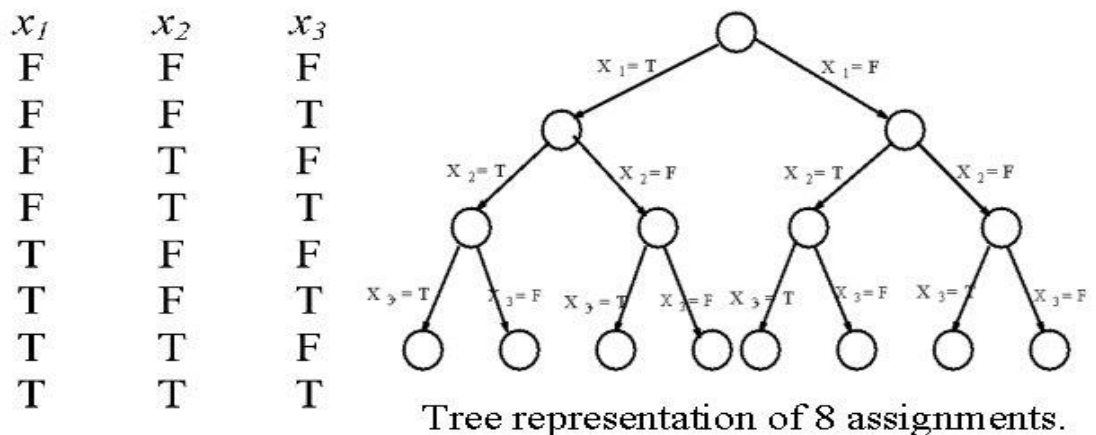
In Non-deterministic algorithms, the path of execution is not same in every execution. The outcomes are inconsistent

Reducibility : Let $L1$ and $L2$ be two problems. Problem $L1$ reduces to $L2$ also written as $L1 \propto L2$. It means if we have a polynomial time algorithm for $L2$, then we can solve $L1$ in polynomial time. Here \propto is transitive relation

If $L1 \propto L2$ and $L2 \propto L3$ then $L1 \propto L3$

Satisfiability – Satisfiability is a problem where given a boolean expression, determining if there exists a truth assignment to its variables

Satisfiability problem



If there are n variables x_1, x_2, \dots, x_n , then there are 2^n possible assignments.

NP – Hard :

A problem L is NP-hard if and only if satisfiability reduces to L.

To show that a problem L2 is NP-hard, it is adequate to show $L1 \leq L2$, where L1 is some problem already known to be NP-hard.

Example

Halting problem, vertex cover problem.

NP – Complete :

A problem is NP-complete if it is both NP and NP-hard.

If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time. For an NP-Complete problem there should be a non deterministic algorithm.

Example

satisfiability problems, clique problem.

