# MODULE-3

## Dynamic Programming

Dynamic programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.

The idea is to simply store the results of subproblems, so that we donot have to re-compute them when needed

later. This simple optimization reduces time complexities from exponential to polynomial.

for example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity & if we optimize it by storing solutions of sub problems, time complexity reduces to linear.

## Principle of Optimality

A problem is said to satisfy the Principle of optimality if the subsolutions of an optimal solution of the problem are the themselves optimal solutions for their subproblems.

The principle of optimality is the basic principle of dynamic programming. which was developed by Richard Bellman: that an optimal path has the property that whatever the initial conditions & control variables (choices) over some initial period, the control (or decision variables) chosen over the remaining problem, with the state of resulting from the early decisions taken to be the initial condition.

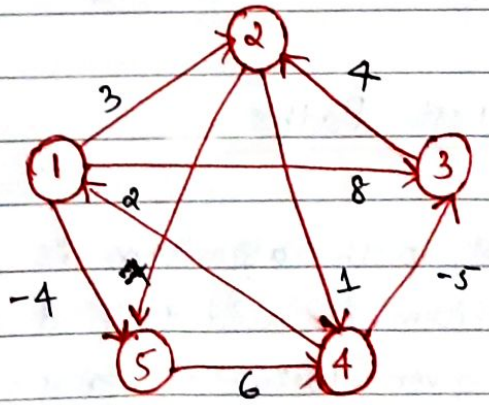
## - All - Pair Shortest Paths

The all pair shortest path algorithm is also known as floyd - Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

**Floyd-Warshall Algorithm** : use a different dynamic programming formulation to solve the all-pairs shortest path problem on a directed graph $G = (V, E)$. The resulting algorithm is known as the floyd-Warshall algorithm, runs in $O(V^3)$. Negative weight edges may be present.

Step 1 : START
Step 2 : floyd-Warshall (w)
Step 3 : n = w.rows
Step 4 : $D^{(0)} = W$
Step 5 : for k = 1 to n
Step 6 : let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new n x n matrix
Step 7 : for i = 1 to n
Step 8 : for j = 1 to n
Step 9 : $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
Step 10 : Return $D^{(n)}$
Step 11 : STOP

Eg:




$D^{(0)} =$

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 3 | 8 | ∞ | -4 |
| 2 | ∞ | 0 | ∞ | 1 | 7 |
| 3 | ∞ | 4 | 0 | ∞ | ∞ |
| 4 | 2 | ∞ | -5 | 0 | ∞ |
| 5 | ∞ | ∞ | ∞ | 6 | 0 |

$D^{(1)}$ =

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 3 | 8 | ∞ | -4 |
| 2 | ∞ | 0 | ∞ | 1 | 7 |
| 3 | ∞ | 4 | 0 | ∞ | ∞ |
| 4 | 2 | 5 | -5 | 0 | -2 |
| 5 | ∞ | ∞ | ∞ | 6 | 0 |

(2) $D$ =

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 3 | 8 | 4 | -4 |
| 2 | ∞ | 0 | ∞ | 1 | 7 |
| 3 | ∞ | 4 | 0 | 5 | 11 |
| 4 | 2 | 5 | -5 | 0 | -2 |
| 5 | ∞ | ∞ | ∞ | 6 | 0 |

(3) $D$ =

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 3 | 8 | 4 | -4 |
| 2 | ∞ | 0 | ∞ | 1 | 7 |
| 3 | ∞ | 4 | 0 | 5 | 11 |
| 4 | 2 | -1 | -5 | 0 | -2 |
| 5 | ∞ | ∞ | ∞ | 6 | 0 |

(4) $D$ =

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 3 | -1 | 4 | -4 |
| 2 | 3 | 0 | -4 | 1 | -1 |
| 3 | 7 | 4 | 0 | 5 | 3 |
| 4 | 2 | -1 | -5 | 0 | -2 |
| 5 | 8 | 5 | 1 | 6 | 0 |

(5) $D$ =

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | -3 | 2 | -4 |
| 2 | 3 | 0 | -4 | 1 | -1 |
| 3 | 7 | 4 | 0 | 5 | 3 |
| 4 | 2 | -1 | -5 | 0 | -2 |
| 5 | 8 | 5 | 1 | 6 | 0 |

Time Complexity : $\Theta(V^3)$

# Single Source Shortest Paths

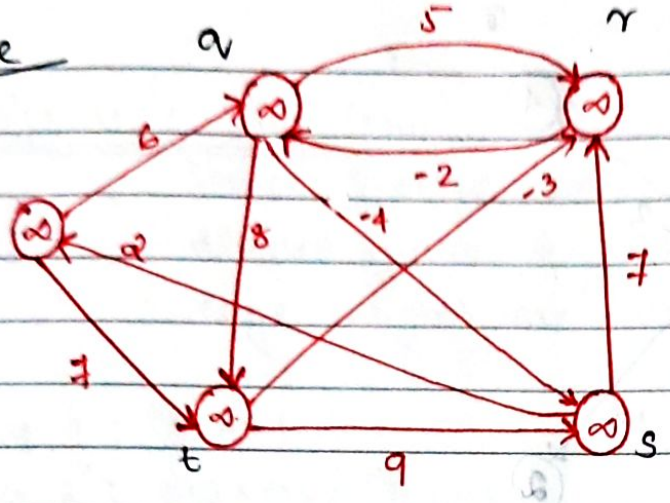
The single source shortest path is used to find Minimum distance from source vertex to any other vertex. There are 2 algorithms

1) Bellman - ford Algorithm
2) Dijkstra's Algorithm.

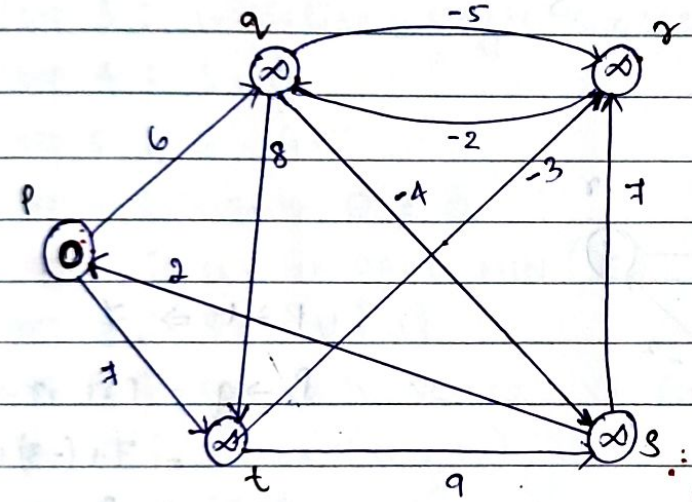
1) **Bellman - ford Algorithm** : used to solve the single source shortest path problem. Edge weights may be Negative. This algorithm returns a boolean value indicating whether or not there is a negative weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest path.

Step 1 : START
Step 2 : Bellman - ford (G, w, s)
Step 3 : Initialise single source (G, s)
Step 4 : for i = 1 to |G.V| - 1
Step 5 : for each edge (u, v) ∈ G.E
Step 6 : Relax (u, v, w)
Step 7 : for each edge (u.v) ∈ G.E
Step 8 : if v.d > u.d + w(u, v)
Step 9 : Return false
Step 10 : Return True
Step 11 : STOP

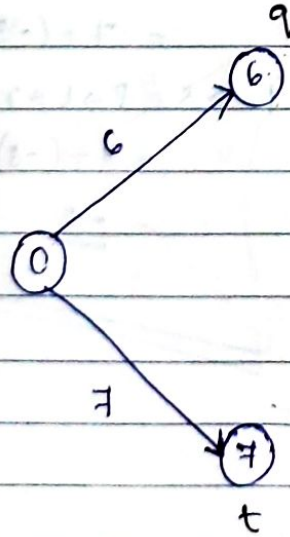
Example



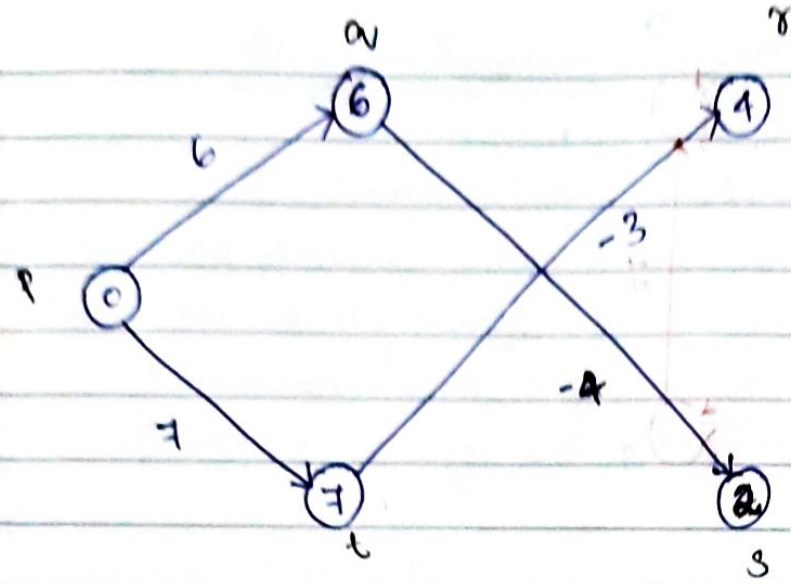

a)



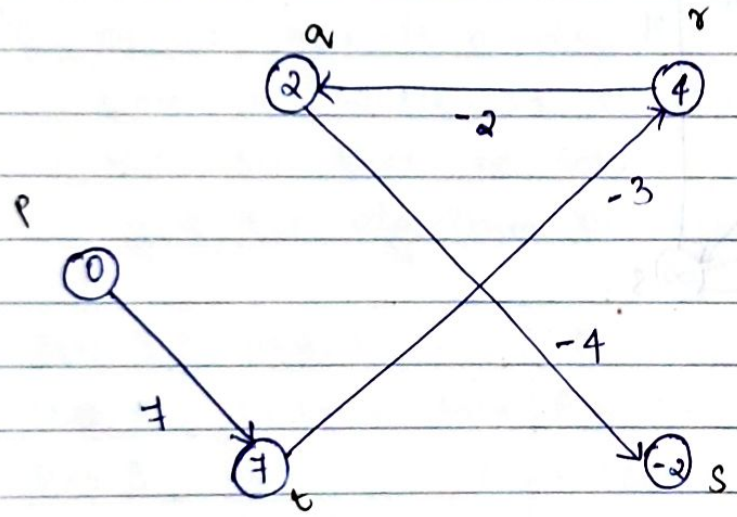

b)



c)




d)




$p \to t \Rightarrow \underline{7}$

$p \to q = p \to t \to r \to q$

$= 7 + (-3) + (-2)$

$= \underline{2}$

$p \to r = p \to t \to r$

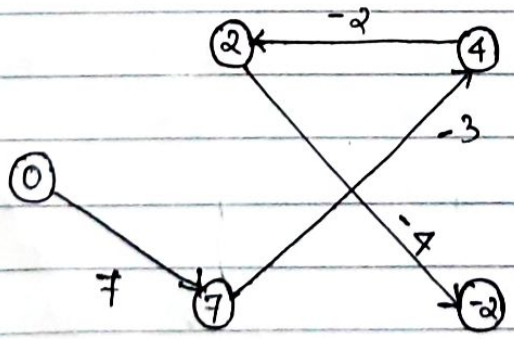
$= 7 + (-3) = \underline{4}$

$p \to s = p \to t \to r \to q \to s$

$= 7 + (-3) + (-2) + (-4)$

$= \underline{-2}$

final fig:

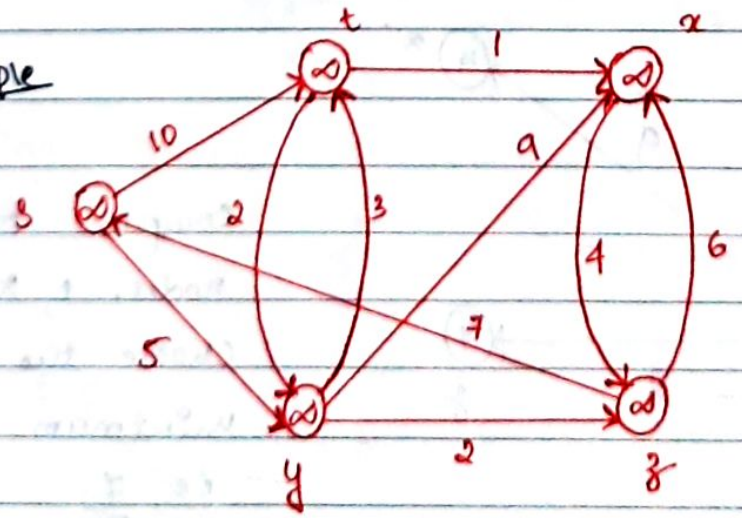



Run Time Complexity : $O(VE)$

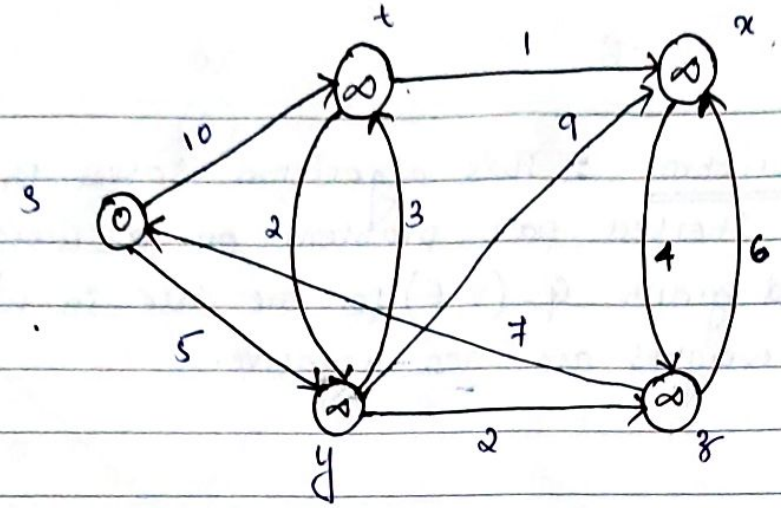
2) **Dijkstra's Algorithm** : This algorithm solves the single source shortest path problem on a weighted directed graph $G = (V, E)$ for the case in which all edge weights are non-negative.

Step 1 : START
Step 2 : DIJKSTRA (G, w, s)
Step 3 : Initialise_single_source (G, s)
Step 4 : $S = \phi$
Step 5 : $Q = G.V$
Step 6 : while $Q \neq \phi$
Step 7 : $u$ = EXTRACT_MIN (Q)
Step 8 : $S = S \cup \{u\}$
Step 9 : for each vertex $v \in G.adj[u]$
Step 10 : Relax (u, v, w)
Step 11 : STOP

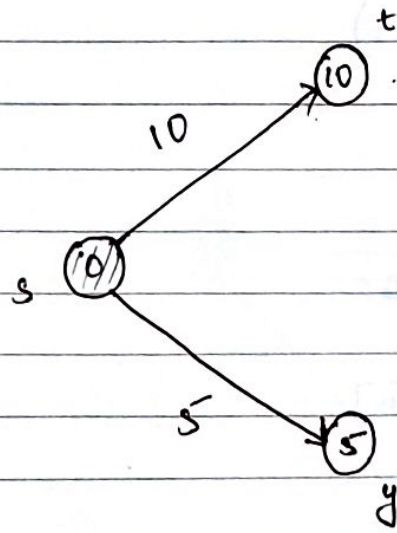
Example



a)

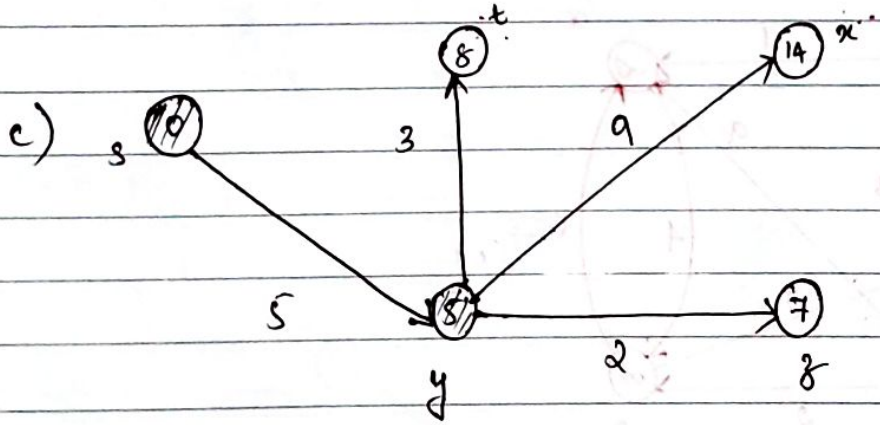



b)



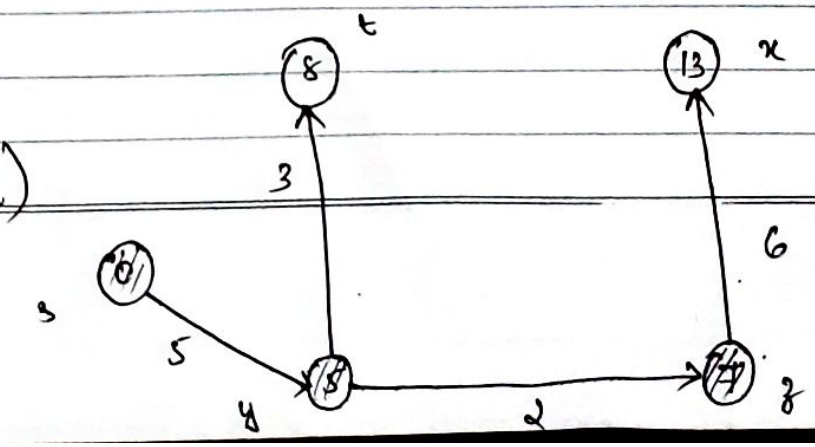

Compare the values of t & y ⇒ 10, 5
Choose the Minimum value. ie; 5.

c)



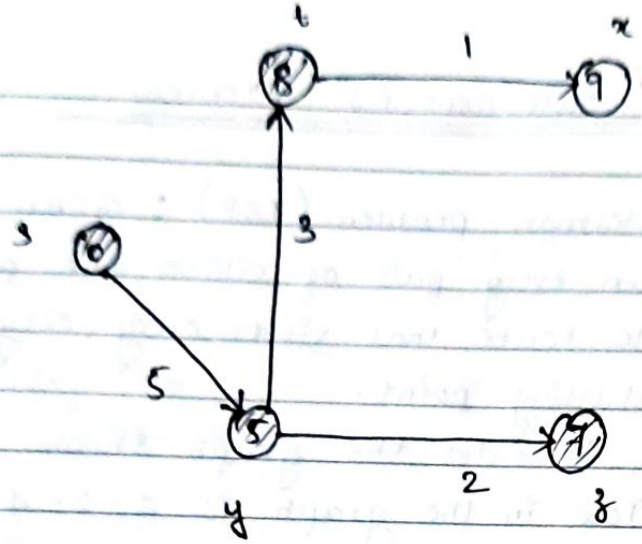

Compare other nodes. t, x, z.
Choose the minimum values.
ie; 7.

d)



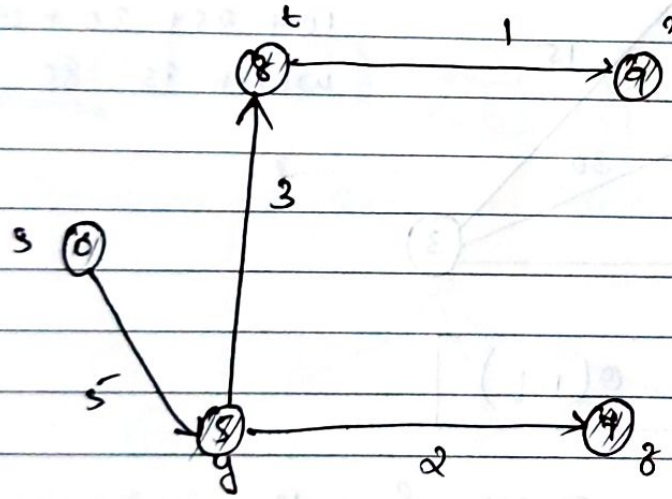

Compare t & x.
Minimum weight is 8. So select that node.

e)

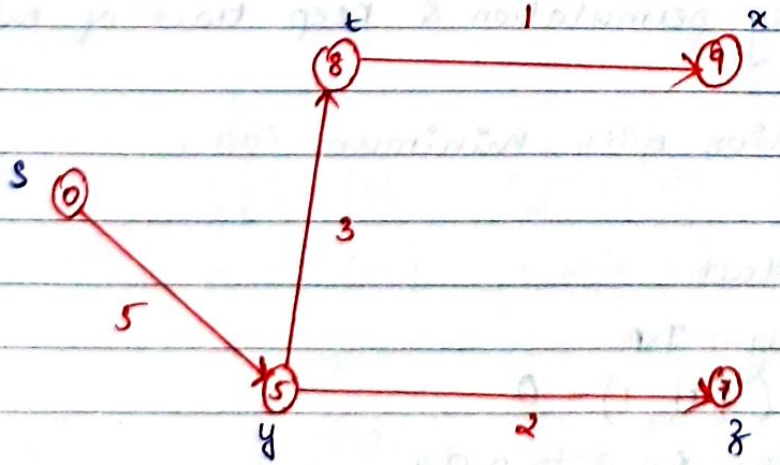



Choose the last node 'x'.

f)




Final figure




Shortest Path ⇒

$S - y \Rightarrow 5$

$S - t \Rightarrow S - y - t \Rightarrow 5 + 3 = 8$

$S - x \Rightarrow S - y - t - x \Rightarrow 5 + 3 + 1 = 9$

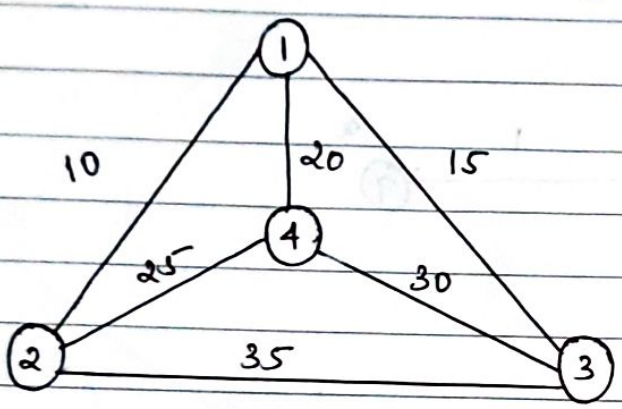
$S - z \Rightarrow S - y - z = 5 + 2 = 7$

Running Time Complexity → $O(v^2)$

# Travelling Salesperson's Problem

Travelling salesman problem (TSP) : Given a set of cities & distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once & returns to the starting point.

for example, consider the graph shown in figure on right side. A TSP Tour in the graph is 1-2-4-3-1.




The cost of the tour is
10 + 25 + 30 + 15
which is 80.

## Naive Solution

| Time Complexity : $O(n!)$ |
|---|

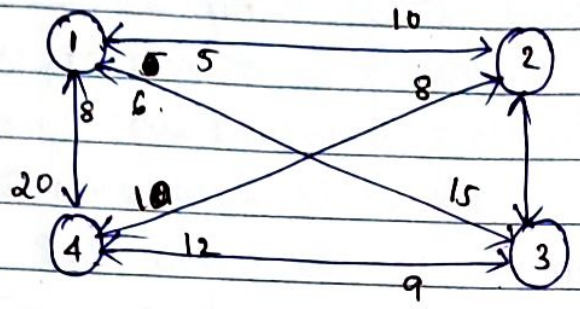
Step 1: Consider city 1 as the starting & ending point
Step 2: Generate all $(n-1)!$ permutations of cities
Step 3: Calculate cost of every permutation & keep track of minimum cost permutation.
Step 4: Return the permutation with minimum cost.

## Dynamic Method

Step 1: Start
Step 2: Algm TSP
Step 3: $c(\{1\}, 1) = 0$
Step 4: for $s = 2$ to $n$ do
Step 5: for all subsets $S \in \{1, 2, 3, \ldots n\}$ of size $s$
Step 6: $c(s, 1) = \infty$
Step 7: for all $j \in S$ & $j \neq 1$
Step 8: $c(s, j) = \min(c(s - \{j\}, i) + d(i, j)$ for $i \in S$ & $i \neq j\}$
Step 9: Return $\min\{c\{1,2,3\}, j) + d(1, i)$
Step 10: STOP

complexity: $O(2^n \cdot n^2)$

## Example



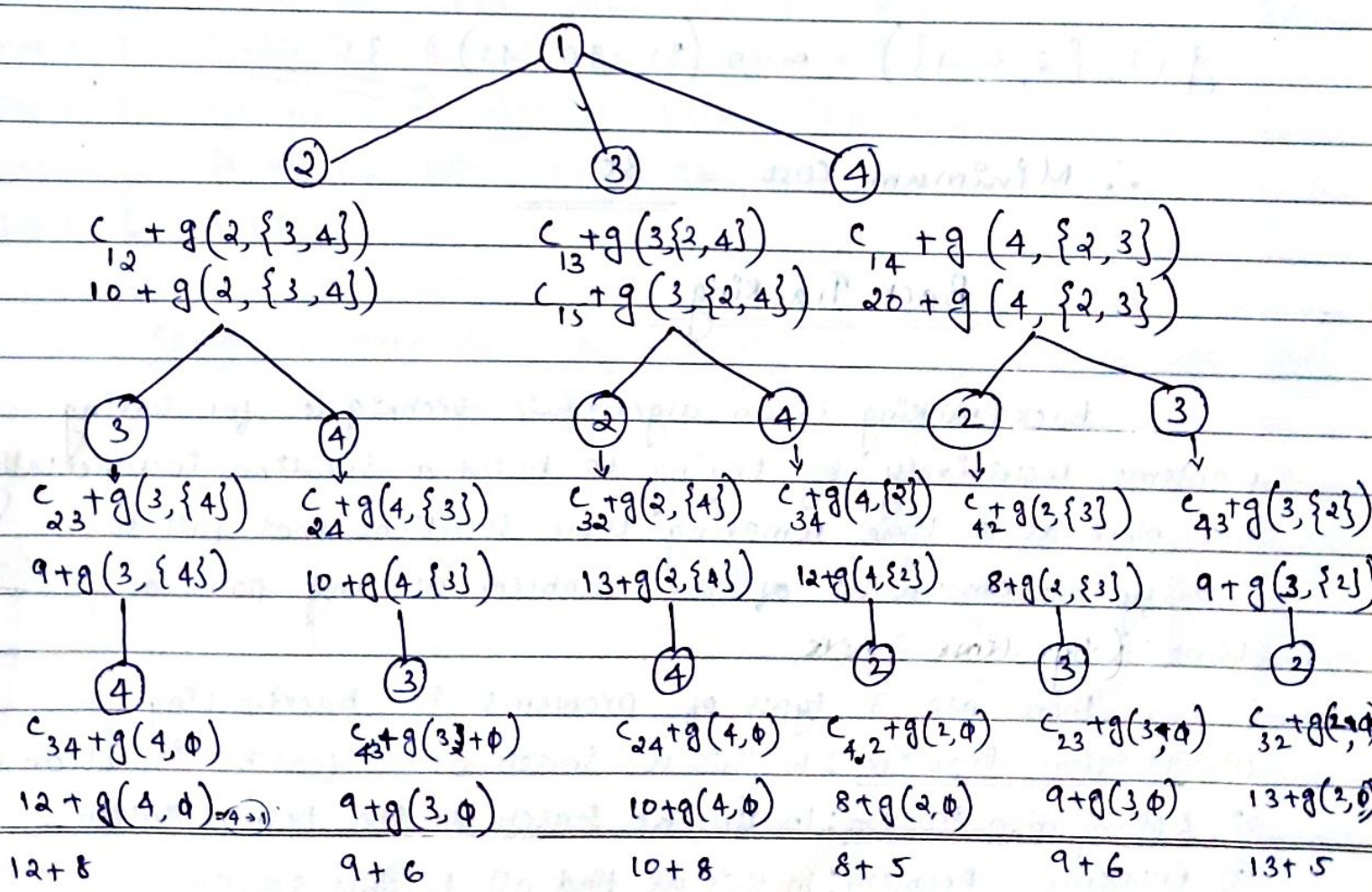

Cost adjacent Matrix

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 10 | 15 | 20 |
| 2 | 5 | 0 | 9 | 10 |
| 3 | 6 | 13 | 0 | 12 |
| 4 | 8 | 8 | 9 | 0 |

$$g(i,S) = \min_{k \in S} \{ c_{ik} + g(k, S - \{k\}) \}$$

$$g(1,\{2,3,4\}) = \min_{k \in \{2,3,4\}} \{ c_{1k} + g(k, \{2,3,4\} - \{k\}) \}$$

Generate a Recursive Tree



$g(2, \phi) = 5$

$g(3, \phi) = 6$

$g(4, \phi) = 8$

$g(2, \{3\}) = 9+6 = 15$

$g(2, \{4\}) = 18$

$g(3, \{4\}) = 12+8 = 20$

$g(4, \{2\}) = 8+5 = 13$

$g(4, \{3\}) = 15$

$g(3, \{2\}) = 18$

$g(2, \{3,4\}) = \min(29, 25) = 25$

$g(3, \{2,4\}) = \min(31, 25) = 25$

$g(4, \{2,3\}) = \min(23, 27) \quad 23$

$g(1, \{2,3,4\}) = \min(35, 40, 43) = 35$

$\therefore$ Minimum cost = 35.

## Back Tracking

Backtracking is an algoritmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time

There are 3 types of problems in backtracking.

① Decision Problem: In this, we search for a feasible solution.

② Optimization Problem: In this, we search for the best solution.

③ Enumeration Problem: In this, we find all feasible solutions.

# 8-QUEEN'S PROBLEM

* This is a typical eg of backtracking algorithm.
* Backtracking is a general algorithm for finding all solutions to some computational method.
* 8 Queen's problem is the problem of placing 8-chess queens on an 8x8 chessboard. so that no 2 queens threaten each other. Thus, a solution requires that no 2 queens share the same row, column or diagonal.

## Algorithm.

Step 1 : START
Step 2 : Algorithm (N Queen's problem)
Step 3 : Start with one queen at the first column, first row.
Step 4 : Continue with 2nd queen from 2nd column.
Step 5 : Go up until find a permissible situation.
Step 6 : Continue with next queen.
Step 7 : STOP.

Complete execution of 'N' queen backtracking algorithm is shown below.

a)

| | | | |
|---|---|---|---|
| x | | | x |
| x | | x | |
| x | x | | |
| Q | x | x | x |

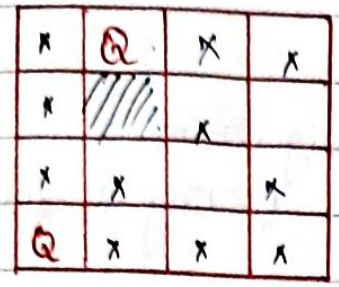
Place 1st Queen at left most bottom. choices available for 2nd are blank boxes.

b)

| | | | |
|---|---|---|---|
| x | x | x | x |
| x | Q | x | x |
| x | x | x | |
| Q | x | x | x |

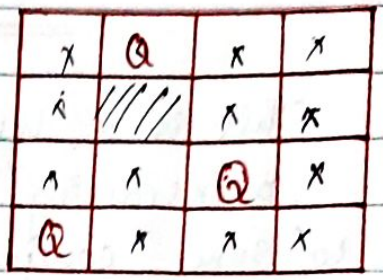
No way forward. only one option remaining.

c)



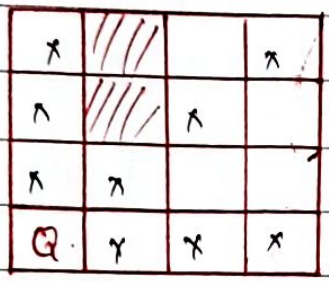
After Backtracking

d)



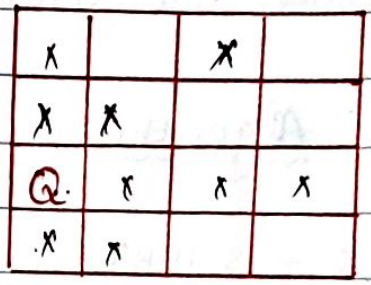
Place the 3rd queen.

e)



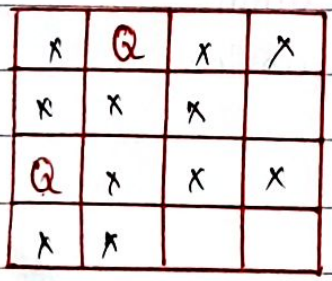
No way forward.
Apply backtracking.

f)



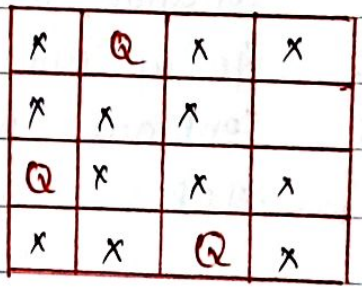
Replace the first queen

g)



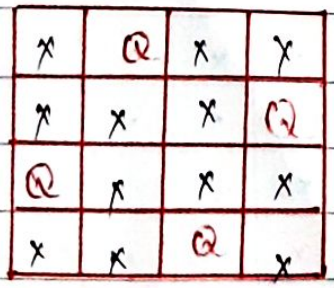
Place the 2nd queen without violating the rules.

h)



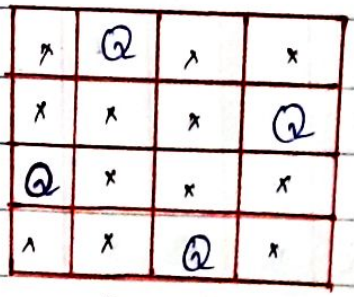
Place the 3rd queen without violating the rules.

i)



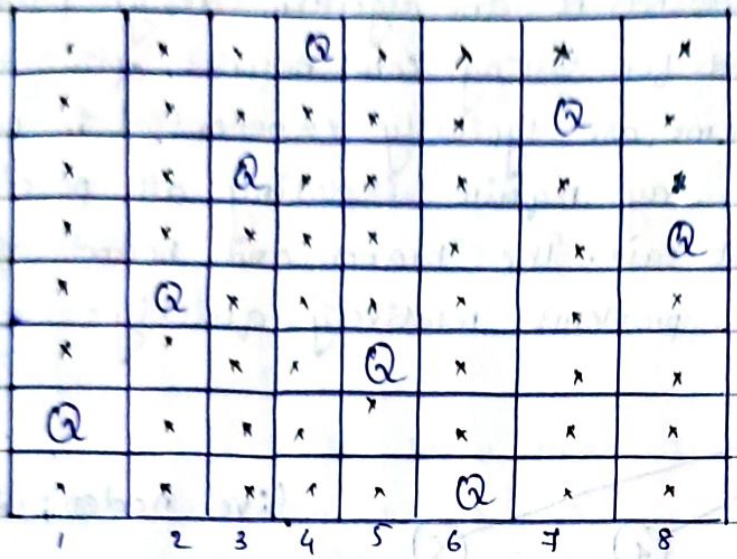
Place the 4th queen without violating the rules.

j)



final fig

## 8- Queens problem solution




## Implicit Constraints & Explicit Constraints

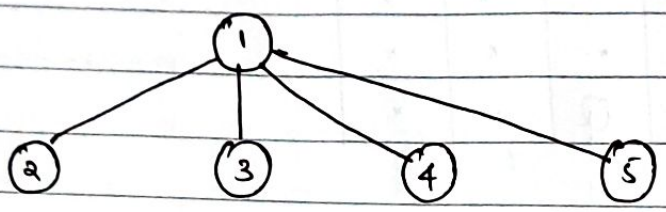
Backtracking algorithm solves the problem using two types of constraints.

1) Explicit constraint
2) Implicit constraint

* Explicit constraints: restrict the each element $x_i$ to take a value from given set, these conditions may or may not depends upon the particular instance of problem I, being solved. All the tuples which satisfy these explicit constraints define a solution space for a particular instance of the problem.
* Implicit constraints: determine which of the tuples in the solution space of I actually satisfy the criterion function, ie; implicit constraints specify how various $x_i$ related to one another.

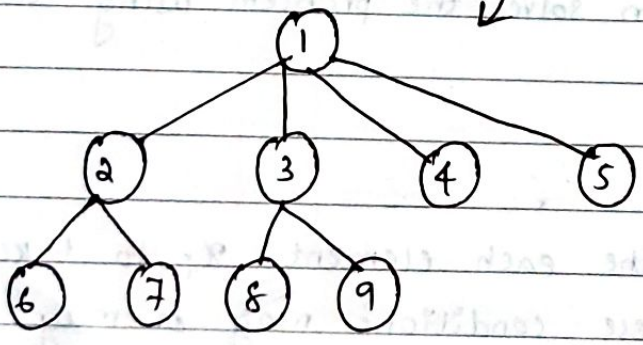
# Branch & Bound

Branch & bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The branch and bound algorithm technique solve these problems relatively quickly.

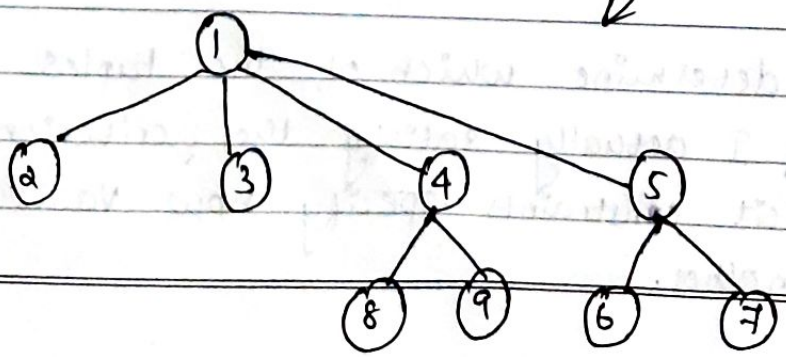



live nodes : 2, 3, 4, 5

* Children of E-node are inserted in a queue (FIFO branch and bound).

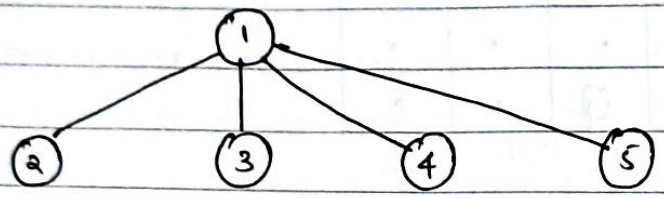



* Children of E-node are inserted in a stack (LIFO branch and bound)



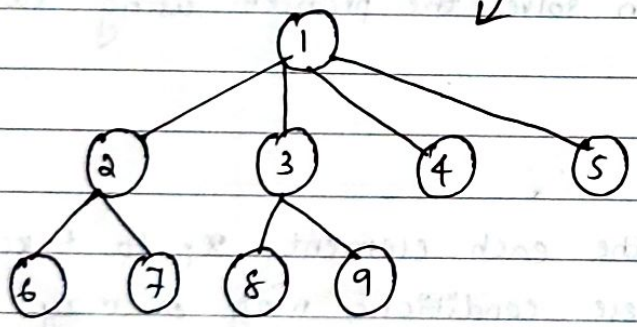
# Branch & Bound

Branch & bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The branch and bound algorithm technique solve these problems relatively quickly.

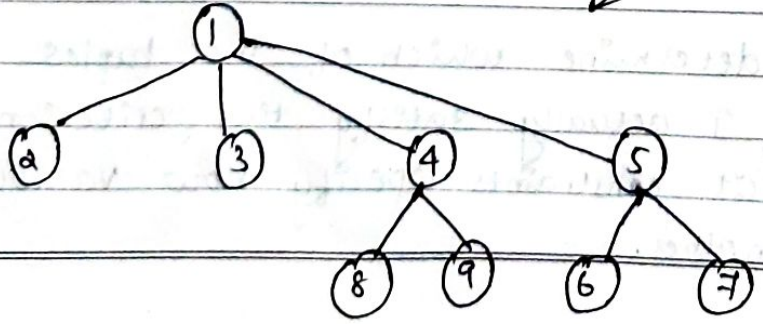



live nodes : 2, 3, 4, 5

* Children of E-node are inserted in a queue (FIFO branch and bound).




* Children of E-node are inserted in a stack (LIFO branch and bound)



# LC Search

In both LIFO and FIFO branch and bound, the selection rule doesnot give preference to nodes that will lead to answer quickly.

The search for an answer node can often be speeded by using an "intelligent" ranking function for live nodes. The next E-node is selected on the basis of this ranking function.

Let 'x' be a node in the state space tree and the cost function of x is

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

where $f(x)$ is the real cost of traversing from the root to x and $g(x)$ is an estimated cost from node x to an answer node.

eg: 15 - puzzle.

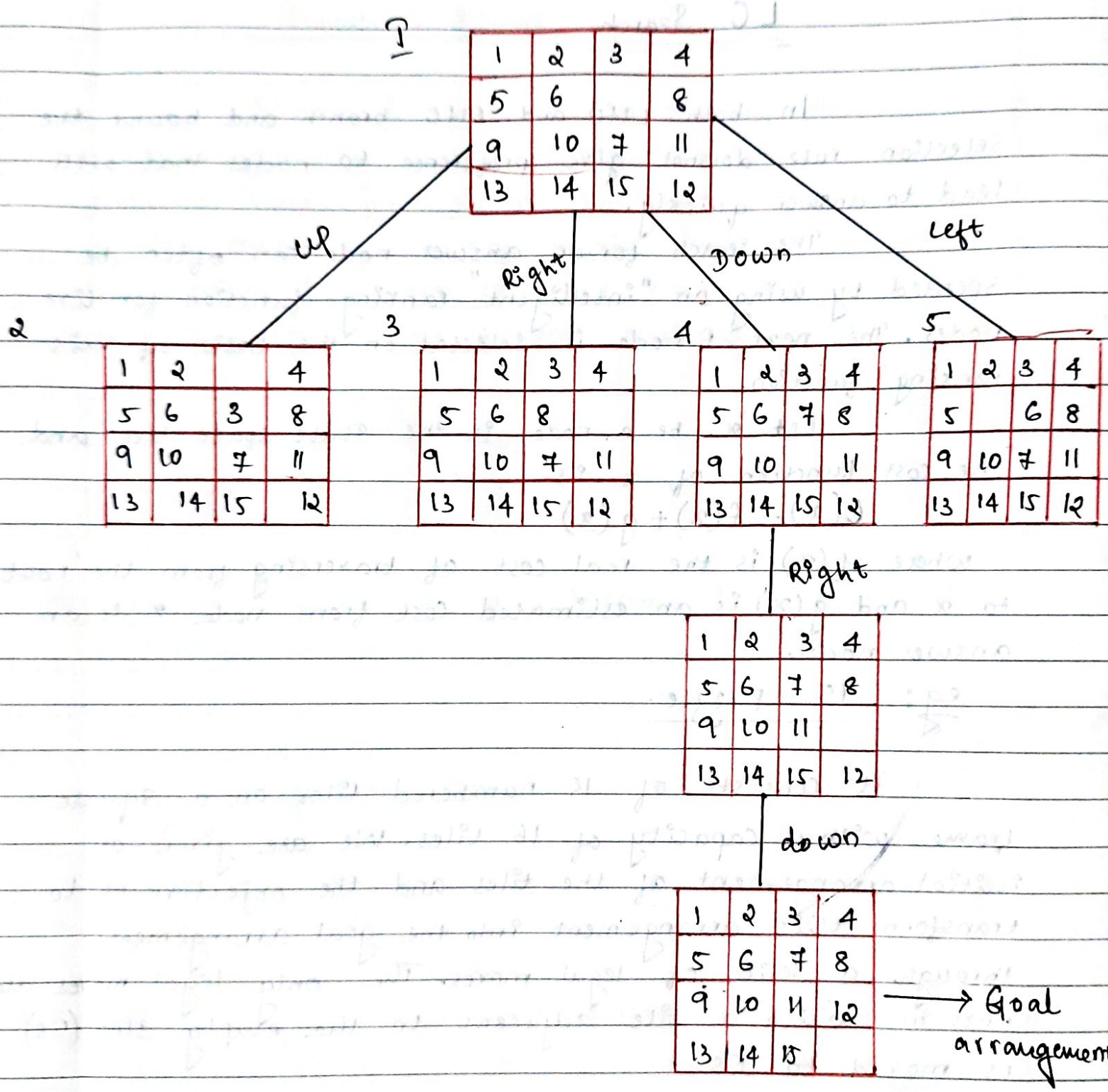
It consists of 15 numbered tiles on a square frame with a capacity of 16 tiles. We are given an initial arrangement of the tiles and the objective is to transform this arrangement into the goal arrangement through a series of legal moves. The only legal moves are ones in which a tile adjacent to the empty slot (Es) is moved to Es.

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

Initial arrangement

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

Goal arrangement


I
1 2 3 4
5 6 8
9 10 7 11
13 14 15 12
up
Right
Down
left
2
1 2 4
5 6 3 8
9 10 7 11
13 14 15 12
3
1 2 3 4
5 6 8
9 10 7 11
13 14 15 12
4
1 2 3 4
5 6 7 8
9 10 11
13 14 15 12
5
1 2 3 4
5 6 8
9 10 7 11
13 14 15 12
Right
1 2 3 4
5 6 7 8
9 10 11
13 14 15 12
down
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15
Goal arrangement

# LC - search algorithm

Step 1 : START
step 2 : Algorithm LC Search(t)
Step 3 : Search t for an answer node
step 4 : I/P : Root node of tree t.
Step 5 : O/P : Path from answer node to root.
Step 6 : if (*t is an answer node)
step 7 : {
Step 8 : print (*t)
Step 9 : Return;
step 10 : }
Step 11 : E=t; // E-node
Step 12 : Initialise the list of live nodes to be empty.
Step 13 : while (true)
Step 14 : {
Step 15 : for each child x of E
Step 16 : {
Step 17 : if x is an answer node
Step 18 : {
Step 19 : print the path from x to E.
step 20 : Return
Step 21 : }
Step 22 : Add (x); // add x to list of live nodes.
Step 23 : x → parent = E // pointer for path to root.
Step 24 : }
Step 25 : if there are no more live nodes
Step 26 : {
Step 27 : Print ("No answer node");
Step 28 : Return.
step 29 : }
step 30 : E = least();
Step 31 : }
Step 32 : END

## Live Node, E-node, Dead node

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

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

The End.