UNIT- I

PROBLEM SOLVING

1. WATER JUGS PROBLEM :

You are given two jugs, a 4 gallon one and a 3 gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

Here the initial state is (0,0)

The goal state is (2,n) for any value of n.

State Space Representation.

We will represent a state of the problem as a tuple (x,y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3 gallon jug.

Note that $0 \leq x \leq 4$ and $0 \leq y \leq 3$.

S.NO	Current State	Next State	description
1.	(x,y) if $x<4$	$(4,y)$	Fill the 4 gallon jug
2.	(x,y) if $y<3$	$(x,3)$	Fill the 3 gallon jug
3.	(x,y) if $x>0$	$(x-d,y)$	Pour some water out of the 4 gallon jug
4.	(x,y) if $y>0$	$(x,y-d)$	Pour some water out of the 3 gallon jug
5.	(x,y) if $x>0$	$(0,y)$	Empty the 4 gallon jug
6.	(x,y) if $y>0$	$(x,0)$	Empty the 3 gallon jug on the ground
7.	(x,y) if $x+y>=4$ and $y>0$	$(4,y-(4-x))$	Pour water from the 3 gallon jug into the 4 gallon jug until the 4 gallon jug is full.
8.	(x,y) if $x+y>=3$ and $x>0$	$(x-(3-y),3)$	Pour water from the 4 gallon jug into the 3 gallon jug until the 3 gallon jug is full
9.	(x,y) if $x+y<=4$ and $y>0$	$(x+y,0)$	Pour all the water fro 3 gallon jug into the 4 gallon jug.

To solve this we have to make some assumptions not mentioned in the problem

They are :-

* We can fill a jug from the Pump
* We can Pour water out of a jug to the ground
* We can Pour water from one jug to another
* There is no measuring device available.

Operators → We must define a set of operators that will take us from one state to another

10.	(x, y) if $x + y <= 3$ and $x > 0$	$(0, x + y)$	Pour all the water from the 4 gallon jug into the 3 gallon jug
11.	$(0, 2)$	$(2, 0)$	Pour the 2 gallons from 3 gallon jug into the 4 gallon jug
12.	$(2, y)$	$(0, y)$	Empty the 2 gallons in the 4 gallon jug on the ground

There are several sequence of operations that will solve the Problem

One of the Possible solution is given as:

Gallons in the 4 gallon jug	Gallons in the 3 gallon jug	Rule applied
0	0	2
0	3	9
3	0	2
3	3	7
4	2	5 or 12
0	2	9 or 11
2	0	—

2. # Hill Climbing

* Hill climbing is a simple optimization algorithm used in AI to find the best possible solutions for a given problem.

* It belongs to the family of local search algorithms and is often used in optimization problems where the goal is to find the best solution from a set of possible solutions.

Algorithm for hill climbing

i) Evaluate the initial state. If it is goal state quit, otherwise make current state as initial state.

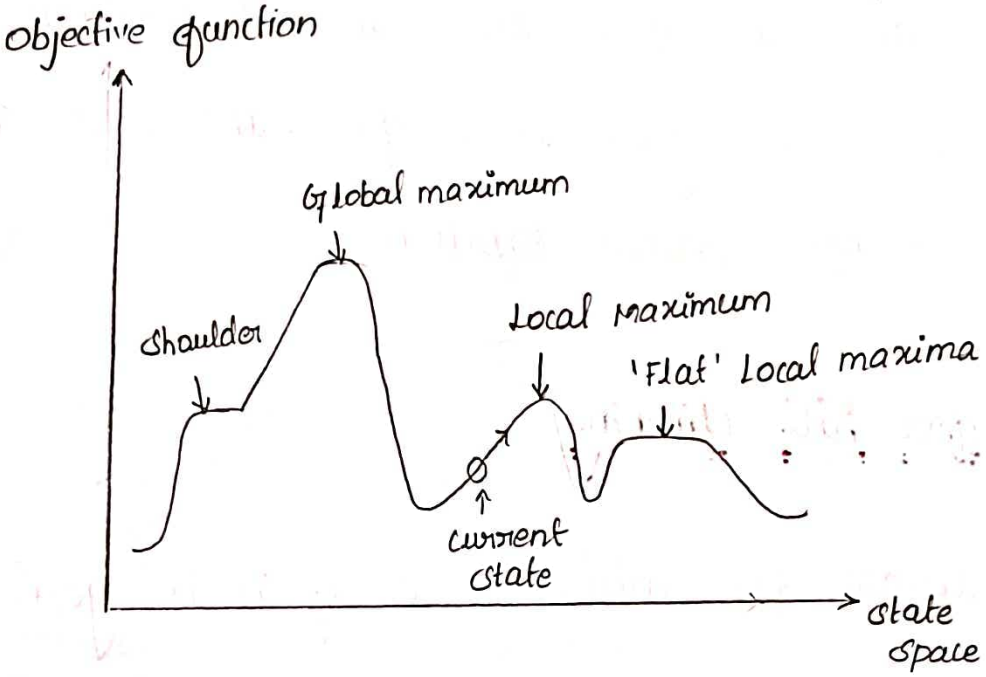
ii) Select a new operator that could be applied to this state and generate a new state.

iii) Evaluate the new state. If this new state is closer to the goal state than current state make the new state as the current state.

iv) If the current state is goal state or no new operators are available, quit. Otherwise repeat from 2.

Problems with Hill climbing

1. Local maxima - can't see higher peak
2. Shoulder - can't see the way out

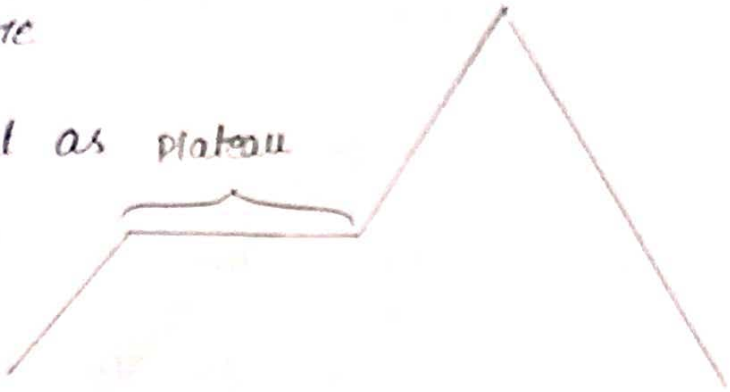


Local Maxima

It is a state where we have climbed to the top of the hill, and missied on better solution.

Plateau

It is a state where everything around is about as good as where we are currently



Ridges

In this state we are on a ridge leading up, but we can't directly apply an operator to improve the situation, so we have to apply more than one operator to get there.

Solving Problems Associated With hill climbing

All the above discussed problems could be solved using methods like backtracking, making big jumps (to handle plateaus or poor local maxima), applying multiple rules before testing (helps with ridges) etc.

Example for local search

Consider the 8-queens problems:

A complete-state formulation is used for local search algorithms. In 8-queens problem, each state has 8-queens on the board one per column. There are two functions related with 8-queens.

1. **The successor function:**

It is function which returns all possible states which are generated by a single queen move to another cell in the same column. The total successor of the each state $8 \times 7 = 56$.

2. **The heuristic cost function:**

It is a function 'h' which hold the number of attacking pair of queens to each other either directly or indirectly. The value is zero for the global minimum of the function which occurs only at perfect solutions.

Advantages of Hill climbing

* Hill climbing is an optimization technique for solving computationally hard problems.

* It is best used in problems with the property that the "state description itself contains all the information needed for a solution.

* The algorithm is memory efficient since it does not maintain a search tree. It looks only at the current state and immediate future states.

Variations of Hill climbing

1. Stochastic hill climbing:

Chooses at random from among the uphill moves, the probability of selections can vary with the steepness of the uphill move.

2. First choice hill climbing

Implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (eg. thousands) of successors.

3. Steepest ascent hill climbing

This algorithm differs from the basic hill climbing algorithm by choosing the best successor rather than the first successor that is better. This indicates that it has elements of the breadth first algorithm.

3. Uninformed Search Strategies

1. Breadth First Search (B.F.S)

The Procedure:

In BFS root node is expanded first, then all the successor of root node are expanded and then their successor and so on

The implementation:

BFS can be implemented using first, in first out queue data structure where fringe will be stored and processed. As soon as node is visited it is added to queue.

The Performance evaluation

1. Completeness

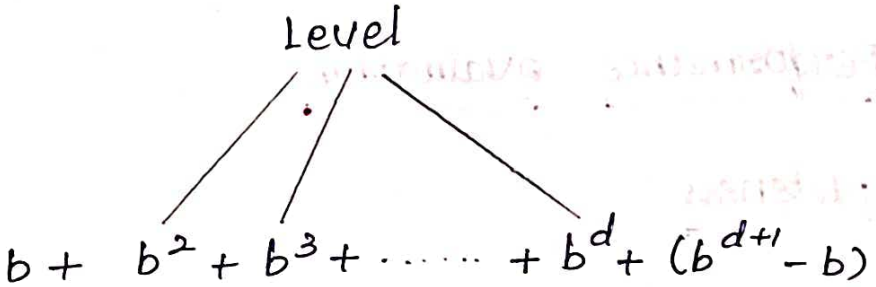
BFS is complete because if the shallowest goal node is at some finite depth d, BFS will eventually find it and will generate solution.

2. Optimality

The shaw shallowest goal node is not necessarily optimal. BFS will yield optimal solution only when all actions have the same cost, let it be at any depth.

3. Time and space complexity

As the level of search tree grows more time is incurred. In general if search tree is at level d then $O(b^{d+1})$ time is required, where b is the Number of nodes generated from each node, starting at root node.



$$b + b^2 + b^3 + \cdots + b^d + (b^{d+1} - b)$$

$$= O(b^{d+1})$$

Algorithm BFS (G, n)

```
// Breadth first search of G
{
    for i := 1 to n do
        Visited [i] := 0;
    for i := 1 to n do
        if (Visited [i] = 0) then BFS (i);
}
```

BFS

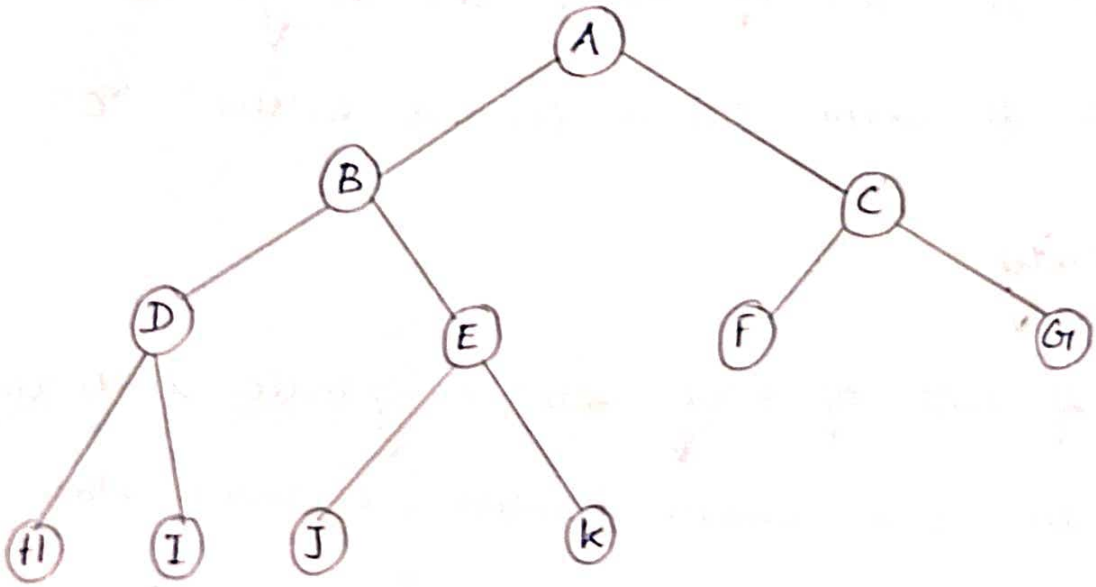
Time complexity - $O(b^{d+1})$

Space complexity - $O(b^{d+1})$

BFS

D and J are goal state

[D is found through the Path] → (A-B-D) (Shallowest goal)



Uniform cost search

The Procedure :

Uniform cost search expands the node 'n' with the lowest path cost.

The implementation:

We can use queue data structure for storing fringe as in BFS. The major difference will be while adding the node to queue, we will give priority to the node with lowest pathcost.

The Performance evaluation

1. completeness :

Uniform cost search guarantees completeness provided the cost of every step is greater than or equal to some small positive constant "c".

2. Optimality

If cost of every step is greater than or equal to some small positive constant then uniform cost search will yeild optimal solution, by

reaching the goal state which has lowest path cost.

3. Time and space complexity

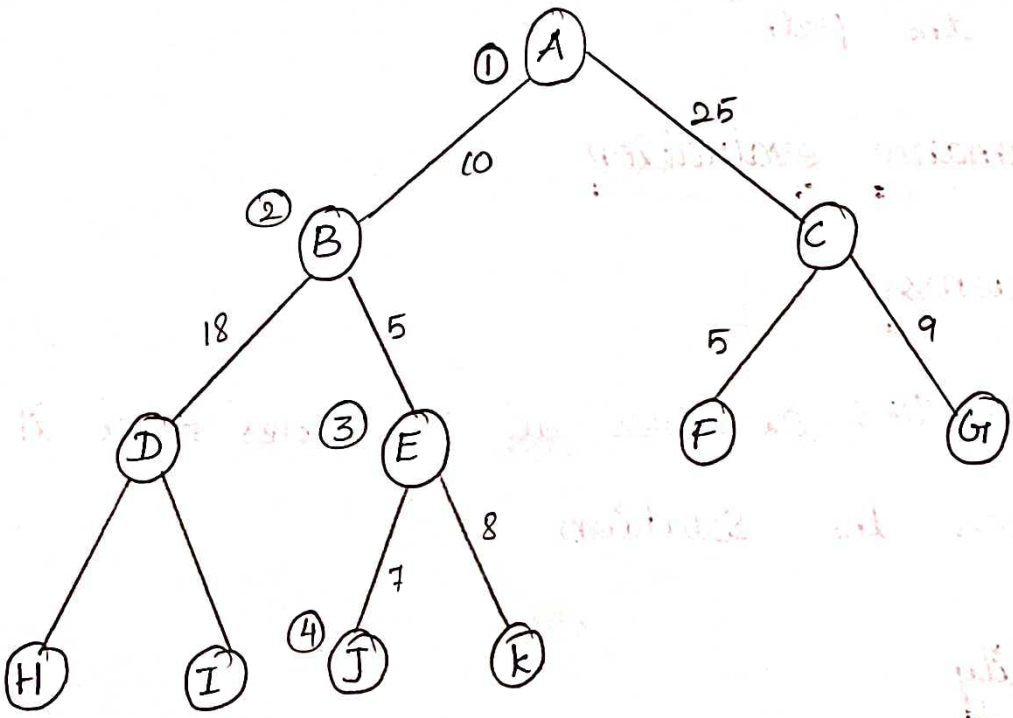
Uniform-cost search does not care about the number of steps a path has, but only about their total cost.

Uniform cost search

Time complexity - $O(b^{c/\epsilon})$

Space complexity - $O(b^{c/\epsilon})$

Where - c is the cost of optimal solution



Assuming J and D are the both goal state

A - B - E - J

Depth first search

The Procedure

Depth first search always expands the deepest node in the current unexplored node set (fringe) of the search tree. The search goes in to depth until there is no more successor node.

The implementation

DFS can be implemented with stack (LIFO) data structure which will explore the latest added node first, suspending exploration of all Previous nodes on the path.

The Performance evaluation

1. completeness

As DFS explores all the nodes hence it guareentees the solution.

2. optimality

As DFS reach to deepest node first, it may ignore some shallow node which can be goal state.

3. Time and Space complexity

DFS requires some moderate amount of memory as it needs to store single path from root to some node to a particular level, along with unexpanded siblings.

Algorithm

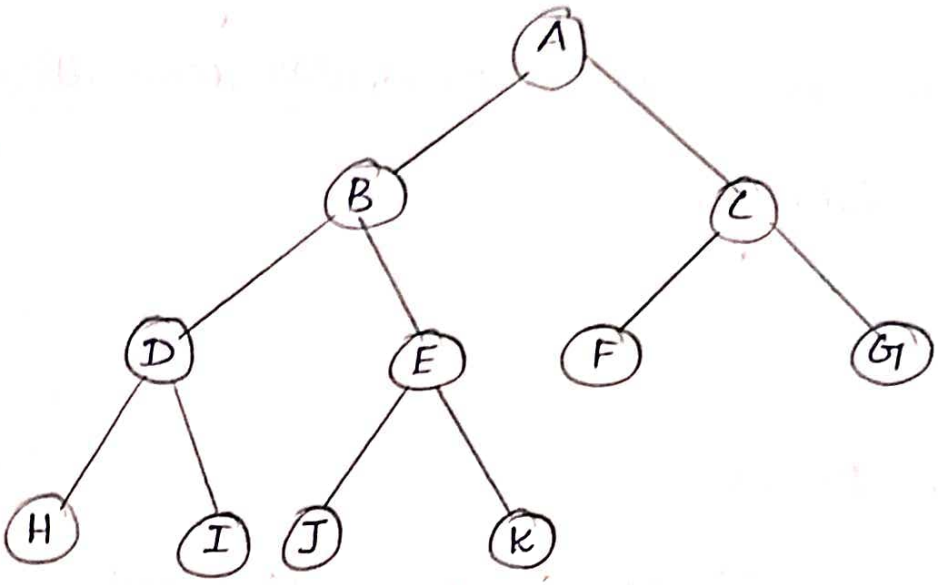
```
Algorithm DFS(v)
// Given an undirected (or directed) graph G = (V, E) with
// n vertices and an array visited initially set
// to zero. this algorithm visits all vertices
// reachable from v. G and visited [] are global
{
        visited[v] := 1;
        for each vertex w adjacent from v do
        {
            if (visited[w] = 0 then DFS(w);
        }
}
```

Time complexity - $O(b^d)$

Space complexity - $O(b^d + 1)$

Goal state path → (A - B - D)

Where D is a goal state



Drawback of DFS

DFS can make a wrong choice and get stuck going down a very long (even infinite) path when a different choice would lead to a solution near the root of the search tree

4. Depth - Limited Search (DLS)

The Procedure:

If we can limit the depth of search tree to a certain level then searching will be more efficient. This removes the problem of unbounded trees

The implementation

Same as DFS but limited to depth level l.

DLS will terminate with two kinds of failure.

The Performance evaluation

1. completeness

DLS suffers from completeness because if we choose l (levels to be searched) < d (actual levels), when shallowest goal is beyond the depth limit l. It generally happens when 'd' is unknown

2. optimality

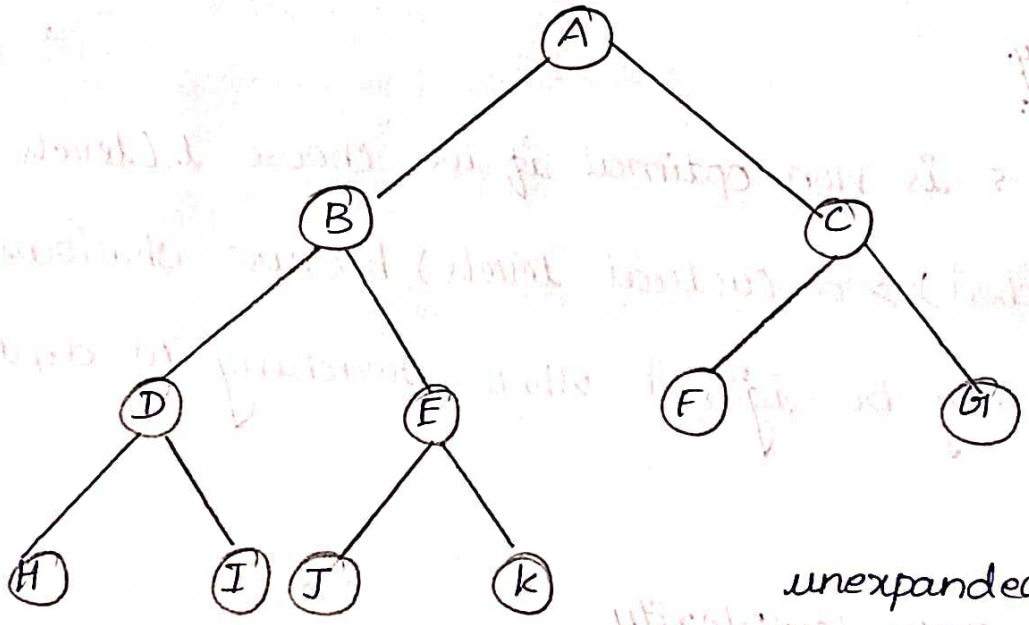
DLS is non-optimal if we choose l (levels to be searched) > d (actual levels) because shallowest goal state may be ignored while reaching to depth level l.

3. Time and space complexity

Its time complexity is $O(b^l)$ and space complexity is $O(bl)$

Algorithm for Depth Limited Search

```
DLS (node, goal, depth)
{
    if (depth >= 0)
    {
        if (node == goal)
            return node
        for each child expand (node)
            DLS (child, goal, depth-1)
    }
}
```



unexpanded node

Iterative deepening depth first search (IDDFS)

The Procedure

In iterative deepening depth-first search, dfs is applied along with the best depth limit. In each step gradually it increases the depth limit until the goal is found.

The implementation

IDDFS can be implemented similar to BFS (Where queue is used for storing fringe) because it explores a complete layer of new nodes at each iteration before going on to the next layer

The Performance evaluation

1. **completeness**

It guarantees completeness as the search does not stop until goal node is found.

2. **Optimality**

As iterative deepening depth first search stop when the first goal node is reached, it is not

necessary that it is the optimal

3. Time and space complexity

* IDDFS has very moderate space complex which is O(bd)

* Its time complexity depends on branching factor (b) and the bottom most level that is depth (d). It is $O(b^d)$

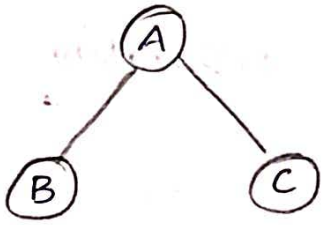
Algorithm for IDDFS

```
// In IDDFS algorithm we are using DLS algorithm
 from earlier section IDDFS (root, goal)
{
    depth = 0
    while (no solution)
    {
        solution = DLS (root, goal, depth)
        depth = depth + 1
    }
    return solution
}
```

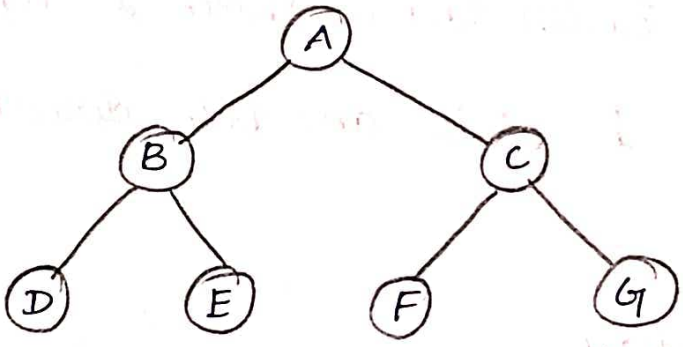
Example - IDDFS

Limit = 0

Limit = 1



Limit = 2



Bidirectional Search

The Procedure:

As the name suggests bi-directional that is two directional searches are made in this searching technique

The implementation:

Bidirectional search is implemented by having one or both of the searches check, each node before it is expanded, is examined to see

if it is in the fringe of the other search tree... If so solution is found. Fringe can be maintained in queue data structure, like BFS.

The Performance evaluation

1. completeness

Bidirectional search is complete if branching factor b is finite and both directions searches use BFS

2. Optimality

It is optimal as the goal is being searched from both directions. so guaranteed to find optimal (best) goal state.

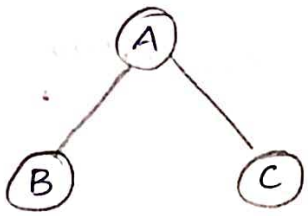
3. Time and Space complexity

Bidirectional search has time complexity as $O(b^{d/2})$, where 'b' is branching factor.

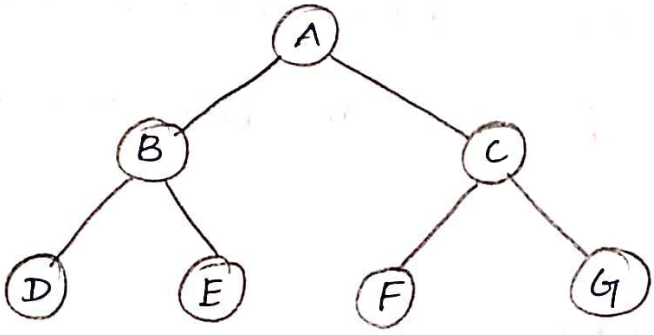
Example - IDDFS

Limit = 0

Limit = 1



Limit = 2



Bidirectional Search

The Procedure:

As the name suggests bi-directional that is two directional searches are made in this searching technique

The implementation:

Bidirectional search is implemented by having one or both of the searches check, each node before it is expanded, is examined to see