Search Algorithm - Step by step procedure to solve a search-problem in a given search space. A search prob. can have 3 main factors -

① Search Space - represents a set of possible solutions which a system may have.

② Start state - It is a state where agent begins.

③ Goal state - It is a function which observe the current state and returns whether the goal state is achived or not.

Essential properties of search algorithm to compare the efficiency of these algo -

Completeness : Said to be complete if it guarantees to return a solution if at least one solution exists for any random i/p.

Optimality : If the solution found for an algo is guaranteed to be the best solution (lowest path cost) among all other solution, then such solution is said to be an optimal solution.

Time Complexity : Measure of time for an algo to complete its task.

Space Complexity : Maximum storage space required at any point during search, as the complexity of a problem.

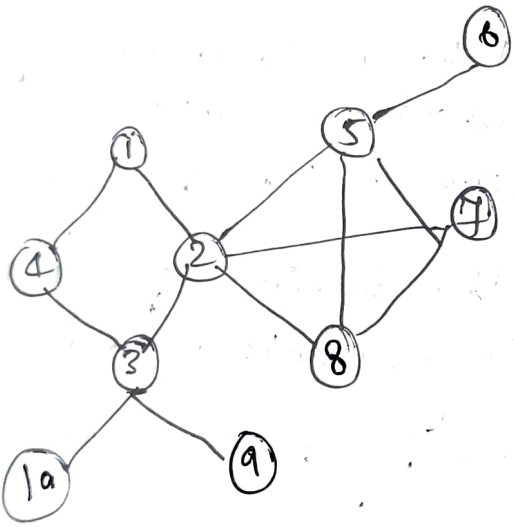
Search Algorithm

① Uniformed / Blind
- i) Breadth First Search
- ii) Depth first search
- iii) Uniform Cost Search
- iv) Depth Limited Search
- v) Bidirectional Search

② Informed
- i) Best first Search
- ii) A* Search.

{Also called Heuristic Search}

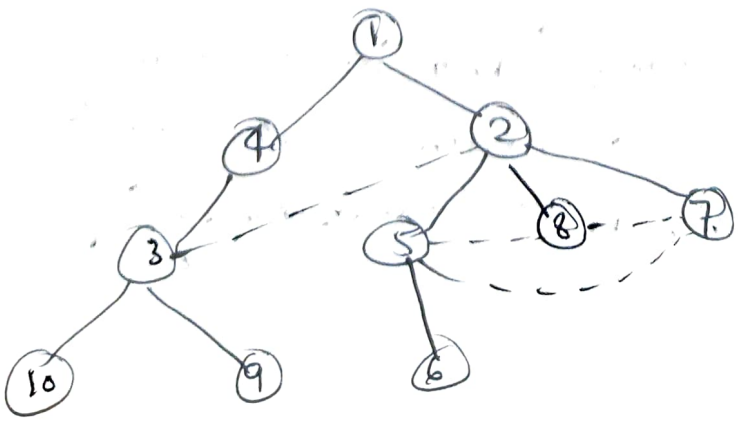
i) Breadth First Search - Queue (FIFO)



Rule -
1. start from point
2. Select adjacent node
3. Use queue to move further.
4. Do not repeat the same node.

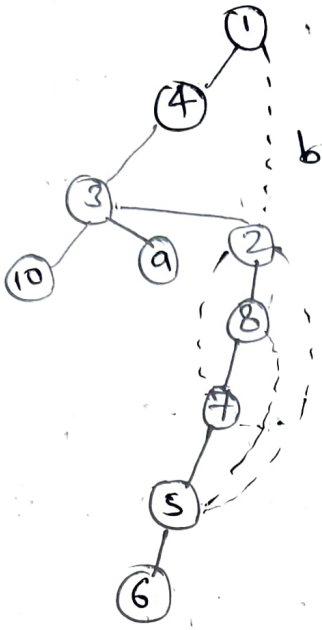
1, 4, 2, 3, 5, 8, 7, 10, 9, 6

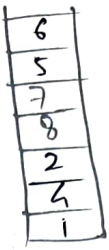
BFS Spanning Tree -



(ii) Depth first Search — Stack. (LIFO)

1, 4, 3, 10, 9, 2, 8, 7, 5, 6





Recursive algo for traversing a tree/graph

BFS -

① Will provide solution if any
② Require lot of memory
③ Lot of time if far away from N.
④ Find shortest path to soln.

Time Complexity - $O(b^d)$

Space Complexity - $O(b^d)$

Completeness - Is Complete

Optimality - Optimal

① No guarantee
② Require less memory
③ Require less time
④ Sometime infinite loop
⑤ Doesn't find shortest path

Time complexity - $O(n^m)$

Space - $O(b.m)$

m = max depth of node

Completeness - Complete $!= \infty$

Optimal - Non-optimal

(iii) Depth Limited Search — DFS with predetermined limit.

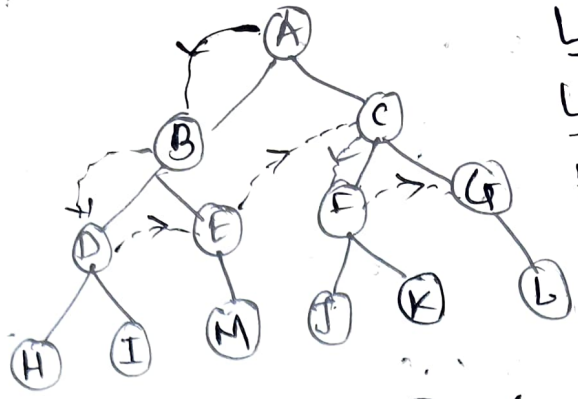
Limit the search.

No addressal for loop (∞)

May not be optimal for more than 1 soln.

Stand failure value — Problem doesn't have any solution

Cutoff failure value — No soln for the problem within given depth limit.



A, B, D, E, C, F, G

Time Complexity - $O(b^l)$ b is the branching factor & l is the depth limit

Space Complexity - $O(b \times l)$

Completeness - Complete if soln" above l.

Optimality - Also not optimal even $l > d$.

⇒ Use stack and Array of visited Nodes.

Uniform Cost Search Algo - Traversing a weighted tree of graph.

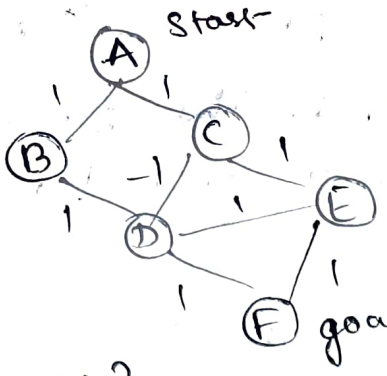
Dijkshatra

Goal - find a path to a goal with lowest cumulative cost. Implemented by Priority Queue (PQ). Max priority to lowest cumulative cost. It is equivalent to BFS if cost is equal.

↳ May stuck in ∞ loop

↳ Always complete if a Solution is available.

↳ Always optimal.



① $PQ = \{ A_0^{NA} \}$

② $PQ = \{ B_1^A, C_1^A \}$ $V = \{ A_0^{NA} \}$

$g(B^A) = g(A^{NA}) + g(A \to B) = 0 + 1 = 1 = g(C^A)$

③ $PQ = \{ C_1^A, D_2^B \}$ $V = \{ A_0^{NA}, B_1^A \}$

④ $PQ = \{ D_0^C, E_2^C \}$ $V = \{ A_0^{NA}, B_1^A, C_1^A \}$

⑤ $PQ = \{ E_1^D, F_1^D, E_2^C \}$ $V = \{ A_0^{NA}, B_1^A, C_1^A, D_0^C \}$

⑥ $V = \{ A_0^{NA}, B_1^A, C_1^A, D_0^C, F_1^D \}$

$F_1^D \to D_0^C \to C_1^A \to A_0^{NA}$

Completeness — Yes Optimal — Yes

Time Complexity - $O(b^{(1+\epsilon)+1})$

Space Complexity - $O(b^{(1+\epsilon)+1})$

Reach to the goal node with the optimal cost c, if every edge/action have minimal cost ϵ, computed depth of the goal node -

$$d = \left[\frac{c}{\epsilon}\right] + 1$$

(v) Iterative Deepening - DFS — (DFS + BFS)

This search algo finds the Breadth First Search's fast search and DFS's memory efficiency.

When search space is large & depth is unknown.

Steps - ① Set the depth limit to 0.

② Perform DFS for depth limit

③ If the goal state is found return.

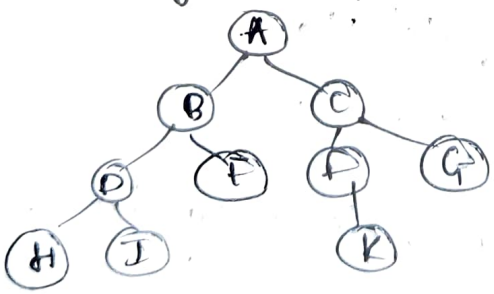
④ If goal state is not found & depth limit not reached, increment the limit & repeat 2-4

⑤ If the depth limit is reached and goal state is not found then return failure.

→ Guarentees to find optimal solution

→ Always find a soln if one exist.

Disadvantage - Repeat all the previous work.



Completeness - Yes

Time Complex - $O(b^d)$

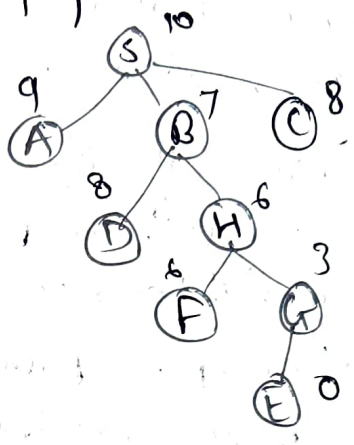
Space Comp - $O(b \times d)$

Optimal - Yes.

1. A
2. A, B, C
3. A, B, D, E, C, F, G
4. A, B, D, H, I, E, C, F, K, G

Best first Search - (Heuristic search)

① Create 2 empty list
② Start from initial node and put it in the 'ordered' Open list
③ Repeat the next step until the goal node is searched or Open list is empty.



(Open ordered) Close

S - 10

B(7), C(8), A(9) → S(10)

H(6), D(8), C(8), A(9) → B^S(7)

G(3), F(6), D(8), C(8), A(9) → H^B(6)

E(0), F(6), D(8), C(8), A(9) → G^H(3)

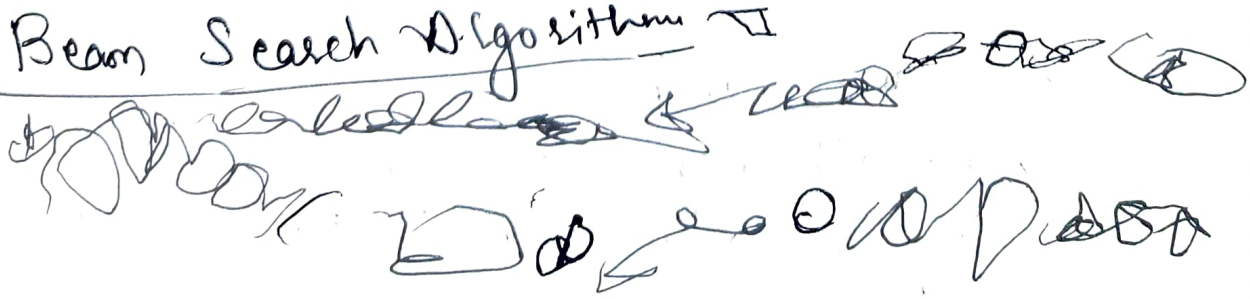
F(6), D(8), C(8), A(9) → E^G(0)

Path: E → G → H → B → S

Time Complexity - $O(b^d)$
Space Complexity - $O(b^d)$

Completeness - Yes
Optimal - Yes.

Beam Search Algorithm



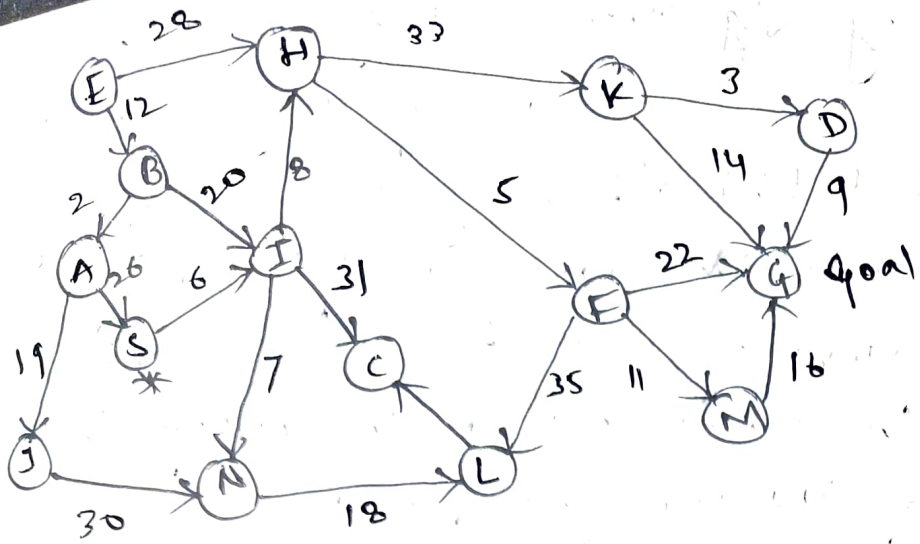
Beam Search Algorithm — (Informed)

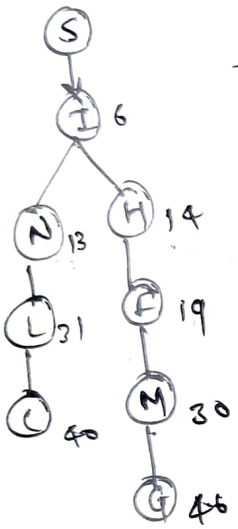
Is a heuristic search algo that explores a graph by evaluating a set numbers of the most promising node at each level of the search tree. This limits the number of node explored which narrows the search space and makes algo. fast & memory efficient.

Variant of BFS that searches a weighted graph for an optimal path from some start point. Only difference is, only top K candidates are choosen for further exploration. Here β is known as Beam width.

Steps -

1. Start from source s. This is level zero in the tree
2. Add s to the closed list and evaluate adjacent to s node.
3. Select top β nodes ~~in open list~~ add them as child node for s. Disregard all other node.
4. Add the selected nodes to open list for exploration.
5. Remove all nodes from open list & add them to the closed list
6. Evaluate all adjacent node and sort them according to their respective parent node
7. Select top β node & add them to tree, connecting them to the respetive parent node
8. Repeat the process.





Node (cost)	Closed List
I [6]	S
N [13]	S
H [14]	I
C [37]	
~~M [30]~~ F [19]	S
~~C [40]~~ L [31]	I
~~G [41]~~ K [47]	N
~~L [54]~~	H
M [30]	S I N H F L
C [40]	
G [41]	
L [54]	
G [46]	S I N H F L M C G

Space Complexity - $O(2^d)$

Completeness → No

Optimality → No

Time Complexity (w^d)

Here space complexity b/w $O(w \cdot d) \rightarrow O(w^d)$

Hill Climbing Algorithm - Algo used to solve optimization problem

Idea behind hill climbing is to continuously move towards the direction of increasing value (or decreasing cost) in order to find the peak (local maxima) or the valley (local minima) of the objective function.

Types of Hill Climbing -

① Simple Hill Climbing -
- → evaluate neighbors one by one.
- → As soon as it finds a neighbor that improves the current state, it moves to the neighbor.
- → If none of the neighbor improves it stops.

② Steepest-Ascent Hill Climbing -
- → evaluate all neighbors and choose that the one offers the greatest improvement in the objective function
- → It ensures that each step is a move towards the best possible neighbor, improving the search direction.

③ Stochastic/Random Hill Climbing -
- → select random neighbor and moves to it if offers an improvement.
- → This is more exploratory than deterministic

Time Complexity - O(N) (Worst Case)
Space Complexity - O(1) (current state & neighbour)

→ It can be simple, fast and effective but in prone to getting stuck at local optima or plateaus.

Problem in Hill Climbing -

① Local Maxima - 

Sol - Backtracking Technique

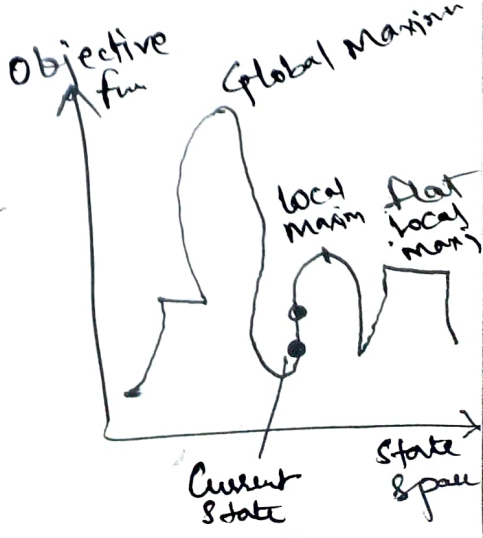
② Plateau - Is the flat area of the search space, did not find the best direction to move. Algo might be lost in such situation 

Sol - Randomly select a state which is far away from the current state so the algo will be able to find non-plateau region.

③ Ridge: Special form of local maxima. It has an area which is higher than its surrounding area and itself has a slop and can't be reached in a single move.

Sol - Use Bidirectional Search or by moving in different direction. 

→ No Backtracking
→ No Guarantee of optimal Solution



Algorithm –

① Start the initial solution within a search space

② Evaluation – Evaluate the quality of current solution by making small modification to the current solution using an objective function

③ Neighbor generation – Generate neighboring solution by making small modification to the curr. solution

④ Selection – Choose the best neighboring solution based on objective function value.

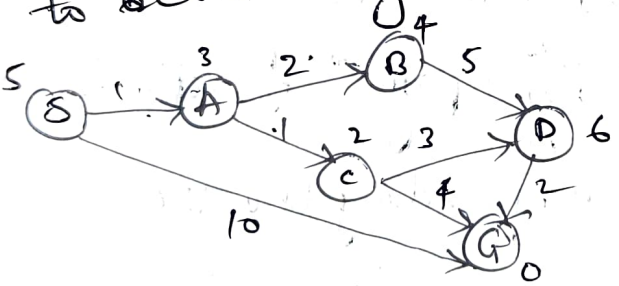
⑤ Comparision – Compare the objetive function value of bast neighboring solution with the current solution.

⑥ Iteration – Repeat the solution until a termination condition is met.

A* Algorithm - Widely used pathfinding algorithm in AI. designed to find most efficient route b/w two points. It improves on dijkstra algo by incorporating a heuristic that eliminates the cost of reaching the goal from any given node, making it more efficient in terms of computational time.

Why more efficient - Unlike BFS or DFS, which explore all nodes indiscriminately A* prioritizes nodes that are likely to lead to the goal node by balancing two factors -

1. g(n). The actual cost of reaching node n from start.
2. h(n). The heuristic estimation of the cost to reach the goal node n.



Open	Closed
A(4), G(10)	S
B(3+4=7), C(2+2=4), G(10)	S, A
G(6+0), D(5+6), B(7), G(10)	S, A, C
G(6), B(7), D(11), G(10)	S, A, C

G → C → A → S

6

Mini - Max Algorithm - Recursive / Backtrack

→ In this algo, two players play the game, one is called MAX and other is called MIN

→ Oponent gets the min benifit while they get the maximum benifit.

→ MAX will select the maximum value while MIN will select the minimum value

↳ The minimax algorithm performs a DFS search algo for the exploration of the complete game tree.

↳ The minimax algo goes all the way down to the terminal node of the tree & back-track the tree as the recursion.

Steps -

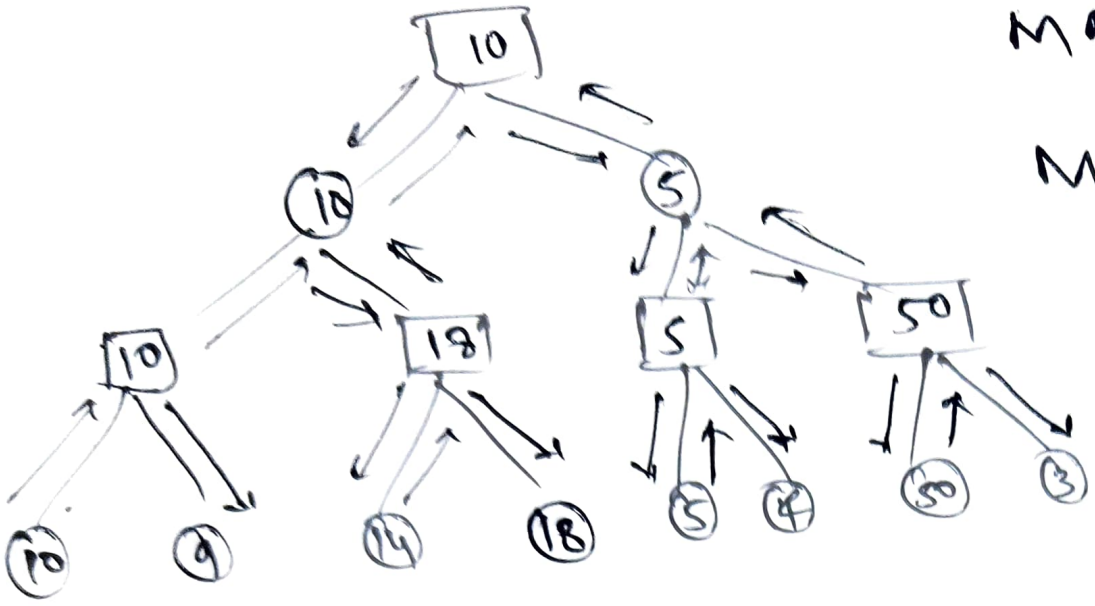
① Generate the game tree

② Apply utility (payoff) function to leaves

③ Use DFS to expanding the tree.

④ Backup values from leaves toward the root
- a Max node will compute the maximum value from its child values
- a Min node computes the minimum value from its child values

⑤ When value reaches the root optimal move is determined.



Constraint Satisfaction Problem (CSP) —