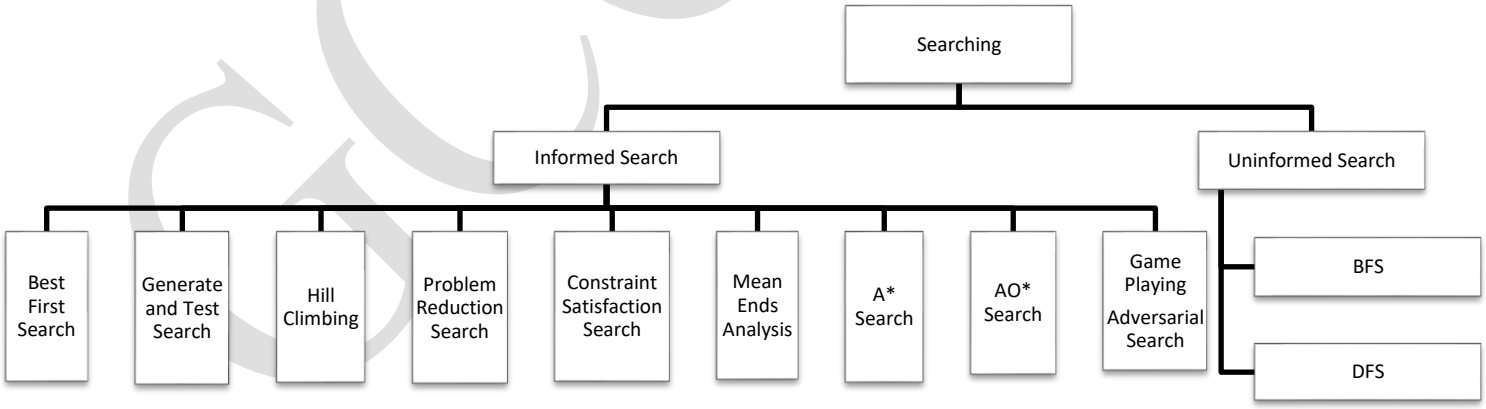
Unit -2 Searching and Reduction

- Problem solving in AI may be characterized as a systematic search through a range of possible actions in order to reach some predefined goal or solution
- **Search:** Searching is a step-by-step procedure to solve a search problem in a given search space.
- A **search algorithm** takes a problem as input and returns the solution in the form of an action sequence.
- **Search Space:** Search space represents a set of possible solutions, which a system may have.
- Once the solution is found, the actions it recommends can be carried out. This phase is called as the **execution phase**.
- A problem can be defined by 5 components.
    a. **Initial state (Start State):** It is a state from where the agent begins **the search**.
    b. **Goal State:** The goal state is the desired end condition that the search algorithm aims to achieve.
    c. **Current state**: The state at which the agent is present after starting from the initial state. It represents the state of the problem at any given point during the search process.
    d. **Successor function**: It is the description of possible actions and their outcomes.
    e. **Path cost:** It is a function that assigns a numeric cost to each path.
- **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.
- **Search tree:** A tree representation of a search problem is called a Search tree. The root of the search tree isthe root node which corresponds to the initial state.
- **Solution:** It is an action sequence that leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all the solutions.

Types of search algorithms

Based on the search problems we can classify the search algorithms into

1. Uninformed (Blind search) search
2. Informed search (Heuristic search) algorithms.



Uninformed/Blind Search:

- Uninformed search, also known as blind search, is a type of search strategy in artificial intelligence (AI) that explores the search space without any domain-specific knowledge or heuristics.
- It relies solely on the information in the problem definition and does not incorporate any guidance or estimation of the proximity to the goal state.
- Uninformed search algorithms systematically generate and evaluate all possible states to find a solution.
- Common uninformed search strategies include:
  - Breadth-first search
  - Depth-first search

Breadth-first Search

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor nodes at the current level before moving to nodes of the next level.
- It uses a queue data structure to keep track of nodes to be explored.

Advantages of BFS:

1. Shortest Path: BFS finds the shortest path from the root node to the target node.
2. Completeness: BFS will surely find a solution if one exists because it explores all nodes at the present depth level before moving to the next level.

Disadvantages of BFS:

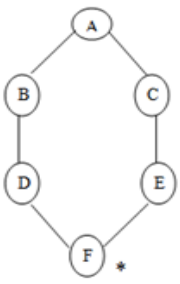
1. Memory Usage: BFS can consume a lot of memory since it stores all the nodes at the present depth level in the queue.
2. Time Complexity: The time complexity can be high as it needs to explore all nodes at each depth level.

Algorithm: Breadth-first Search

- Step 1: Place the root node inside the queue.
- Step 2: If the queue is empty then stop and return failure.
- Step 3: If the FRONT node of the queue is a goal node, then stop and return success.
- Step 4: Remove the FRONT node from the queue. Process it and find all its neighbors that are in ready state then place them inside the queue in any order.
- Step 5: Go to Step 3.
- Step 6: Exit.

Example: Let us implement the above algorithm of BFS by taking the following suitable example
Consider the graph in which let us take A as the starting node and F as the goal node (*)

Step 1: Place the root node inside the queue i.e. A, and mark it as visited.



A			

Step 2: Now the queue is not empty and also the FRONT node i.e. A is not our goal node. So, move to step 3.

Step 3: Remove the FRONT node from the queue(dequeue) i.e. A and enqueue the unvisited neighbours of A i.e. B and C and mark them as visited.

B	C		

Step 4: Now B is the FRONT node of the queue. Remove the FRONT node i.e. B from the queue(dequeue). Then process B and enqueue the unvisited neighbours of B i.e. D.

C	D		

Step 5: Now since C is the FRONT node of the queue, remove the FRONT node i.e. C from the queue(dequeue). Then, process C and enqueue the unvisited neighbours of C i.e. E

D	E		

Step 6:
Now D is the FRONT node. Remove the FRONT node i.e. D from the queue(dequeue). Then, process D and enqueue the unvisited neighbours of D i.e. F.

E	F		

Step 7:
Now E is the front node of the queue. Remove the FRONT node i.e. E from the queue(dequeue). The neighbour of E is F which is already added in the queue.

F			

Step 8:
Finally, F is the FRONT of the queue which is our goal node. So, exit.

A	B	C	D	E	F

Depth first Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Advantage:

- **Memory Efficiency:** DFS requires less memory than BFS as it doesn't need to store all nodes at a level but only the nodes along the current path.
- **Time Complexity:** DFS can be more efficient in terms of time complexity for deeper solutions,

Disadvantage:

- **Completeness:** DFS is not guaranteed to find the shortest path and may not find a solution in infinite graphs or if it goes down an infinitely deep branch.
- **Backtracking Cost:** If DFS goes down a deep path without finding the goal, it may need to backtrack significantly, which can be inefficient.

Algorithm: Depth first Search

Algorithm:

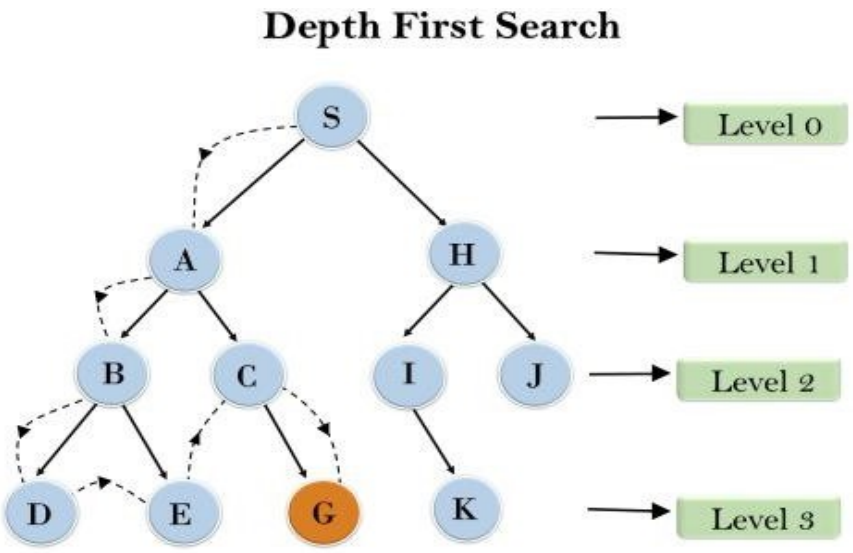
Step 1: PUSH the starting node into the stack.
Step 2: If the stack is empty then stop and return failure.
Step 3: If the top node of the stack is the goal node, then stop and return success.
Step 4: Else POP the top node from the stack and process it. Find all its neighbours that are in ready state and PUSH them into the stack in any order.
Step 5: Go to step 3.
Step 6: Exit.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:
Root node--->Left node > right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.

In the following example, S is the root node and G is the goal node.



Step 1: PUSH the starting node into the stack. i.e. S

S			

Step 2: Now the stack is not empty and the TOP of the stack S is not our goal. So, POP the TOP node from the stack i.e. S, and process it. PUSH the unvisited neighbours of S onto the stack. i.e. A, H

A	H		

Step 3: Now the TOP of the stack is A and it is not our goal. So, POP the TOP node from the stack i.e. A, and process it. PUSH the unvisited neighbours of A onto the stack. i.e. B, C

B	C	H	

Step 4: Now the TOP of the stack is B and it is not our goal. So, POP the TOP node from the stack i.e. B, and process it. PUSH the unvisited neighbours of B onto the stack. i.e. D, E

D	E	C	H

Step 5: Next the TOP of the stack is D and it is not our goal. So, POP the TOP node from the stack i.e. D, but since does not have any neighbour. So consider the next element in the stack. i.e E

E	C	H	

Step 6: Now the TOP of the stack is E and it is not our goal. So, POP the TOP node from the stack i.e. E, but since does not have any neighbour. So consider the next element in the stack. i.e C

C	H		

Step 6: Now the TOP of the stack is C and it is not our goal. So, POP the TOP node from the stack i.e. C, and process it. PUSH the unvisited neighbour of C onto the stack. i.e. G.

G	H		

Step 7 :Now since G is at the TOP node and it is the goal, so return success and exit.

Solution: S → A → B → D → E → C → G → H → I → K → J

Informed Search

- Informed search, also known as heuristic search, is a type of search strategy in artificial intelligence that uses additional information, beyond the basic problem definition, to find solutions more efficiently.
- Informed search strategies can find a solution more efficiently than an uninformed search strategy.
- **Use of Heuristics**: Informed search algorithms employ heuristic functions to estimate the cost or distance to the goal from a given state. These heuristics guide the search process, helping to prioritize which paths to explore. Hence, an Informed search is also called a Heuristic search.
- **Applications**: Informed search techniques are widely used in various applications, such as pathfinding in games, robotics navigation, route planning, and solving complex puzzle

1. Greedy Search
2. A* Search

Heuristic search techniques

- **Heuristics:** These are the General problem-solving methods that provide approximate solutions quickly by simplifying complex problems.
- The solutions may not be optimal but are sufficient in a given limited time frame.
- A heuristic is a way that might not always be guaranteed for best solutions but guaranteed to find agood solution in a reasonable time.
- **A Heuristic function,** often denoted as h(n) is a function that estimates the cost to reach the goal from a given node n in a search algorithm. It is used to guide the search process in finding the most promising path towards the goal.
- **Or**
- **Heuristic Function**: It is an estimation tool used in search algorithms to predict the cost from a current node to the goal, guiding the search process efficiently.
- **Heuristic Search**: Search strategies that employ heuristic functions to prioritize paths, reduce search space, and improve the efficiency of finding solutions.

Different heuristic search techniques are:

- Generate-and-test
- Hill climbing
- Best-first search
- Problem reduction
- Constraint satisfaction
- Mean-ends analysis.

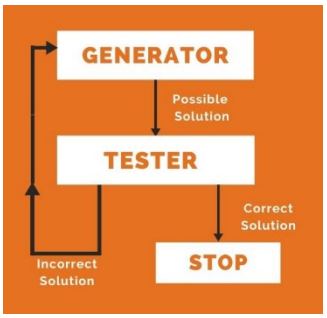
Generate-and-test

- Generate and Test is a very simple heuristic search technique.
- It is a search strategy used in problem-solving and optimization that involves generating possible solutions and testing them to see if they meet the desired criteria. This iterative process continues until an acceptable solution is found or all possible solutions are exhausted.
- It is based on Depth First Search with Backtracking.

Stepwise Algorithm for Generate and Test

1. **Generate Possible Solutions**: Create a possible solution for the problem.
2. **Test Each Solution**: Check the generated solution, if it satisfies the condition of the desired goal.
3. **Select Valid Solution**: If a solution satisfies the condition for the desired goal, it is considered valid.
4. **Terminate or Continue**: If a valid solution is found, terminate the search. If not, return to step 1.

The following diagram shows the Generate and Test Heuristic Search Algorithm



Advantages of Generate and Test Algorithm

- **Simplicity**: The algorithm is straightforward and easy to understand
- **Flexibility**: can be applied to various types of problems without domain knowledge
- **Completeness**: The algorithm will surely find a solution.

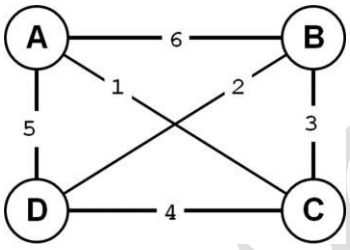
Disadvantages of Generate and Test Algorithm

- **Inefficiency**: The algorithm can be very slow and inefficient, especially if the search space is large, because it may generate and test a large number of invalid solutions before finding a valid one.
- **Exponential Growth**: For problems with a large search space, the number of solutions can grow rapidly.
- **Redundancy**: May generate and test redundant or duplicate solutions, increasing the computational cost.

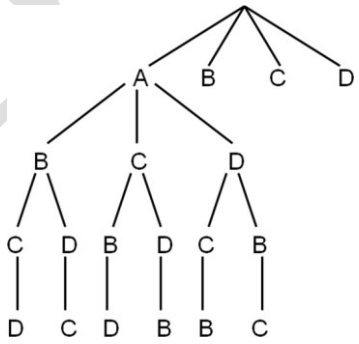
Example – Traveling Salesman Problem (TSP)

A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.

- Traveller needs to visit n cities.
- Know the distance between each pair of cities.
- Want to know the shortest route that visits all the cities once.



Search flow with Generate and Test



Finally, select the path whose length is less.

Search for	Path	Length of Path
1	ABCD	18
2	ABDC	13
3	ACBD	11
4	ACDB	13
5	ADBC	11
Continued		

Hill Climbing Algorithm

- Hill climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space.
- In hill climbing the basic idea is to always head towards a state better than the current one.
- Hill climbing algorithm is a local search algorithm that continuously moves in the direction of increasing elevation/value to find the peak of the mountain or the best solution to the problem.

- It terminates when it reaches a peak value where no neighbour has a higher value.
- It is also called greedy local search as it only looks to its good immediate neighbour state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state. Thus, it does not backtrack the search space, as it does not remember the previous states.
- Different regions in the state space landscape:
    - **Local Maximum:** Local maximum is a state which is better than its neighbour states, but there is also another state which is higher than it.
    - **Global Maximum:** Global maximum is the best possible state of state space landscape. It has the highest value of the objective function.

- **Types of Hill Climbing Algorithm:**
    - **Simple Hill Climbing**
    - **Steepest-Ascent Hill-Climbing:**

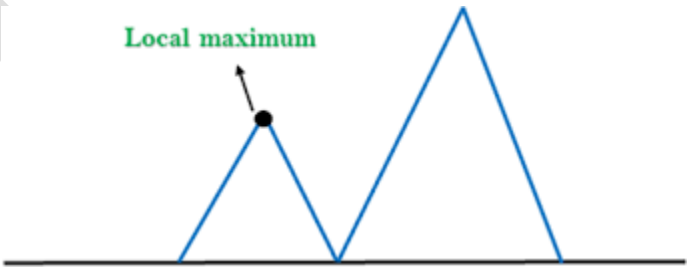
Simple Hill-Climbing Algorithm

Simple hill climbing is the simplest way to implement a hill-climbing algorithm. It only evaluates the neighbour node state at a time and selects the first one which optimizes the current cost and sets it as a current state. It only checks its one successor state, and if it finds better than the current state, then move else be in the same state.

Algorithm

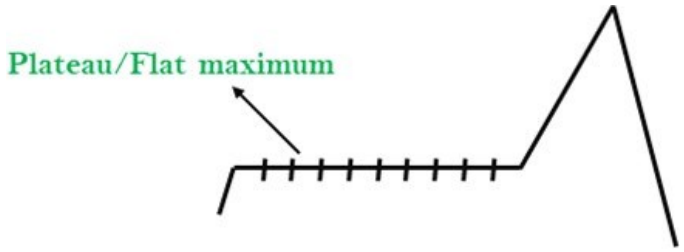
1. Evaluate the initial state.
   If it is also a goal state then return it, otherwise continue with the initial state as the current state.
2. Loop until the solution is found or until there are no new operators to be applied in the current state
    a. Select an operator that has not yet been applied to the current state and apply it to produce new state
    b. Evaluate the new state
        i. If it is a goal state then return it and quit
        ii. If it is not a goal state but it is better than the current state, then make it a current state
        iii. If it is not better than the current state, then continue in loop.

Problems in the Hill climbing algorithm or Drawbacks of Hill climbing algorithm
Local maximum



A local maximum is a state that is better than all its neighbours but it not better than some other states farther away.

Plateau

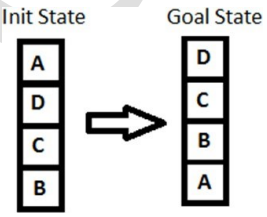


A plateau is a flat area of the search space in which a whole set of neighbouring states has the same value. In this, it is not possible to determine the best direction in which to move by making local comparisons.

Ridge

A ridge is a special kind of maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope.

Example:

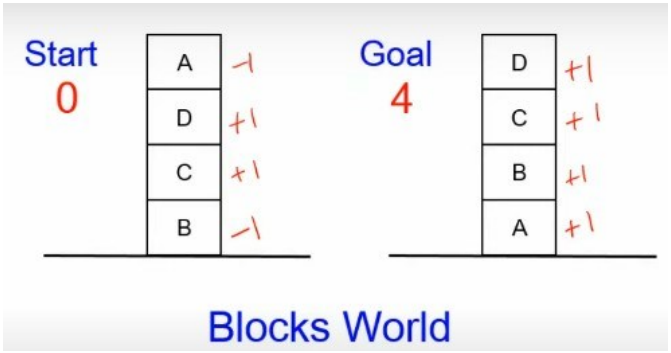


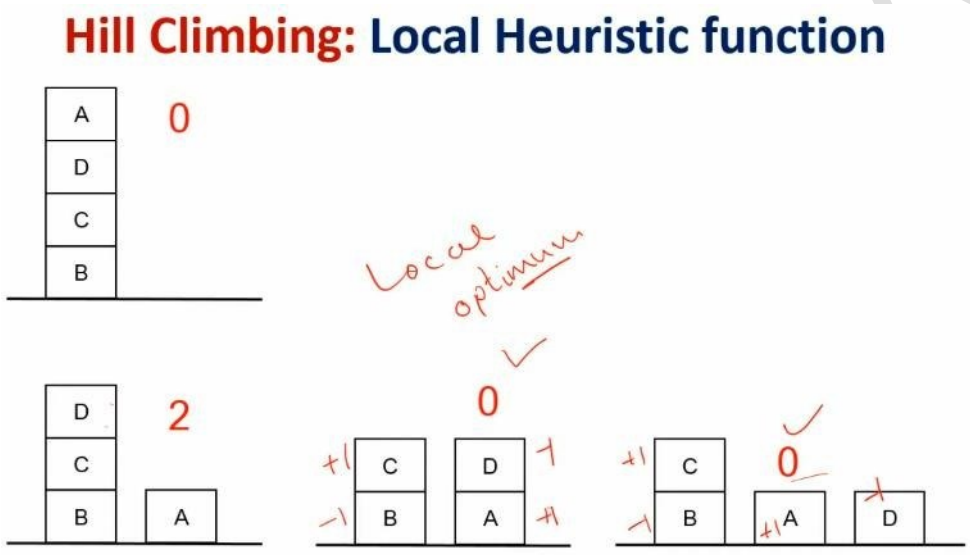
Simple Hill climbing using local heuristic function.

It uses local information- It decides what to do next by looking only at the immediate consequences of its choices. It will terminate when it is local optimum.

Let's define such function h: h(x)

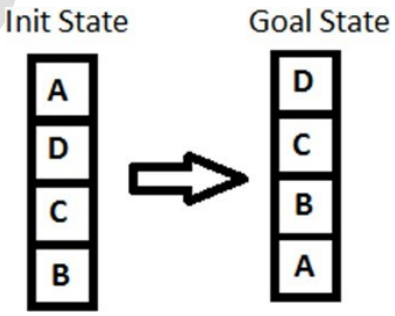
- **+1 for each block that is resting on the thing it is supposed to be resting on.**
- **-1 for each block that is resting on the wrong thing.**



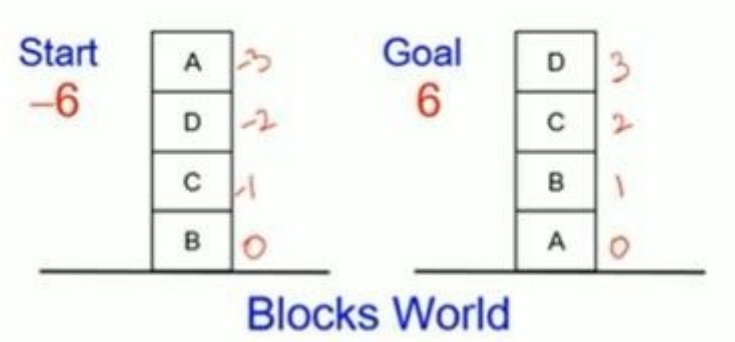


For the above example using local heuristic function you will not able to reach the goal state we reach here the Plateau problem arises.
So to reach the goal state we use the Global Heuristic function.

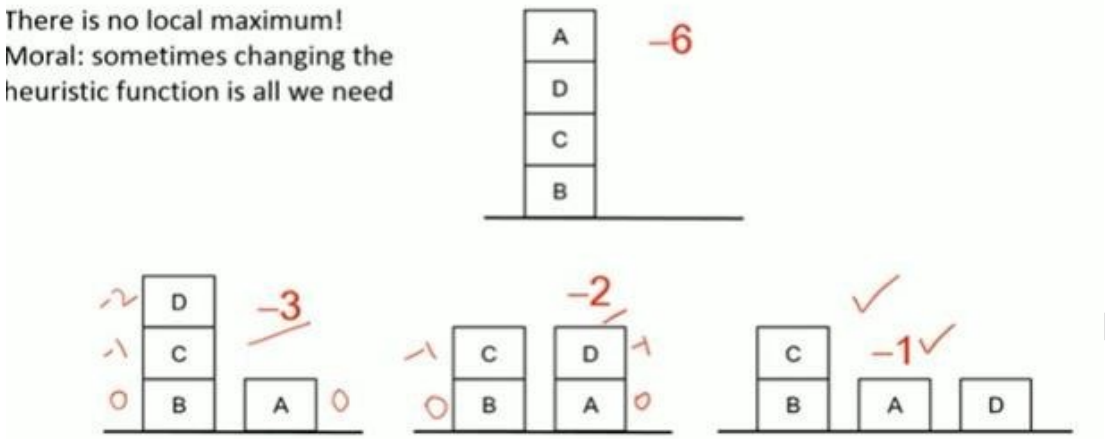
Simple Hill climbing using global heuristic function.



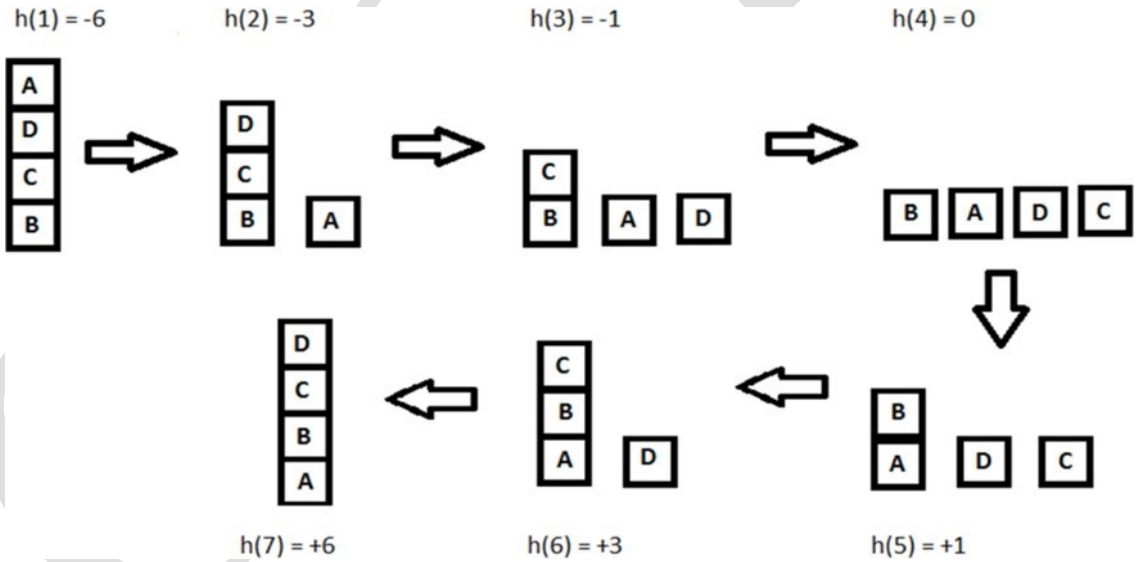
- For each block has the correct support structure: +1 to every block in the support structure.
- For each block has a wrong support structure: -1 to every block in the support structure.



Now we have to solve the problem. To reach from the Initial state to the goal state.



Solution of the problem



Steepest-Ascent Hill Climbing Search Algorithm

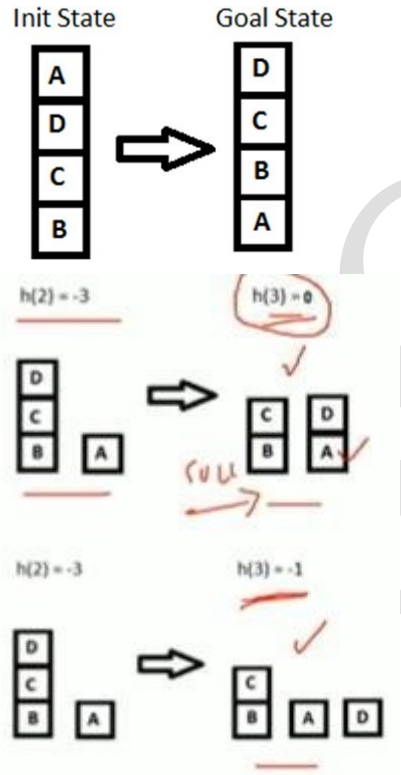
This is a variation of simple hill climbing. Instead of moving to the first state that is better, move to the best possible state that is one move away. Basic hill-climbing first applies one operator and gets a new state. If it is better that becomes the current state whereas the steepest climbing tests all possible solutions and chooses the best.

Algorithm

1. **Evaluate the initial state.**
   If it is also a goal state then return it and quit. Otherwise, continue with the initial state asthe current state.
2. **Loop until a solution is found or until a complete iteration produces no change to the currentstate:**
   a. **Let SUCC be a state such that any possible successor of the current state will be better than SUCC.**

b. For each operator that applies to the current state do:
i. Apply the operator and generate a new state.
ii. Evaluate the new state. If it is a goal state, then return it and quit.
If not, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.
c. IF the SUCC is better than the current state, then set the current state to SUCC.

Example:



It explores all the possible options from the current state.

Difference between Simple Hill Climbing and Steepest Hill Climbing

- Simple hill climbing just interacts with the first neighbour and if it is better, assumes it as the current state, and if not, it will give a result and exit.
- Steepest hill climbing is a greedy algorithm, and it evaluates all the possible moves from the current solution and selects the one that leads to the best improvement.
- In simple hill climbing, the first closer node is chosen, whereas in steepest ascent hill climbing all successors are compared and the closest to the solution is chosen.
- Steepest ascent hill climbing is similar to best-first search, which tries all possible extensions of the current path instead of only one.

Best-First Search (BFS)

- Best-First Search (BFS) combines the advantages of both depth-first search and breadth first search into a single method.
- It follows a single path at a time but switches paths whenever a competing path looks more promising than the current one.
- At each step of the Best First Search process; we select the most promising node from the generated list using an appropriate heuristic function
- Then expand the chosen node by using the rules to generate its successors.
- If one of them is a solution, then we can quit, else repeat the process until we search goal.

OR Graphs

BFS is implemented using the OR graph. This graph is called OR graph because each of its branches represents analternative problem-solving path.

To implement BFS search, we need two lists of nodes:

1. OPEN:
   - It is a priority queue.
   - Contains nodes that have been generated and have had the heuristic function applied to them, but which have not yet been examined.
   - The element with the most promising value of the heuristic function will have the highest priority.
2. CLOSED:
   - Contains nodes that have already been examined.
   - Nodes are retained in memory.
   - Referred by a newly generated node to confirm whether it has been generated before.

Algorithm:

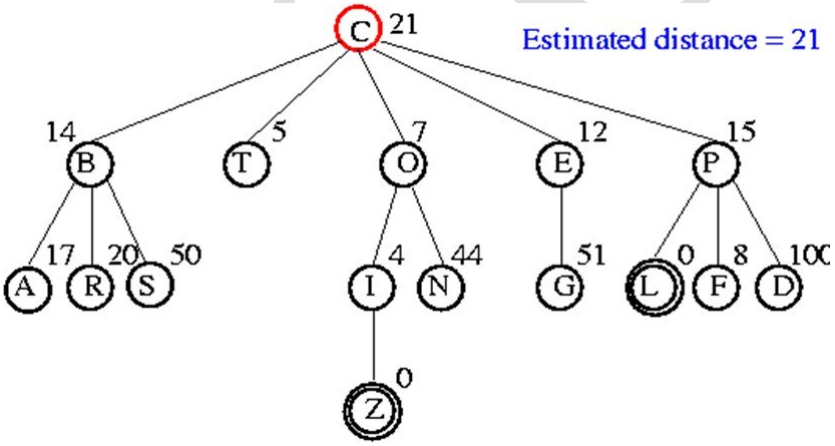
1. **Start with OPEN containing just the initial state**
2. **Until a goal is found or there are no nodes left on OPEN do:**
   a. Pick the best node on OPEN
   b. Generate its successors
   c. For each successor, do:
      i. If it is not been generated before, evaluate it, add it to OPEN, and record its parent.
      ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

Example: Best-First Search

Here **C** is the initial or source node and **L** and **Z** are goal nodes.

Open: C

Closed: —



Now, C is added to Closed, and B, T, O, E and P are added to Open.

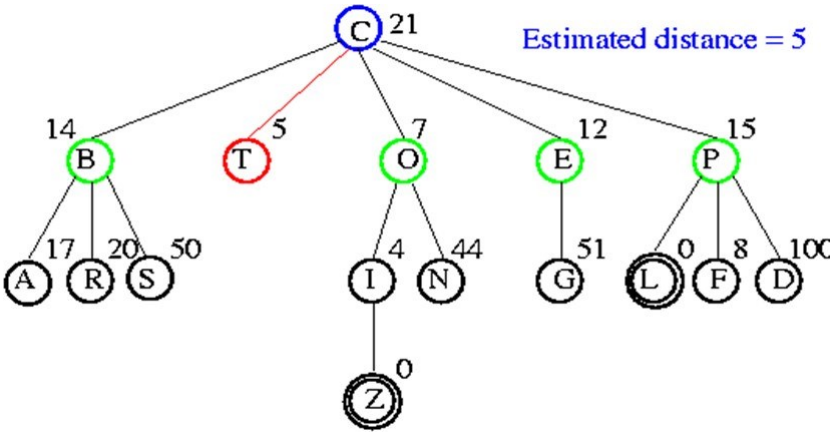
Open: T, O, E, B, P

Closed: C

Now, T has the least distance hence, T is added to Closed.

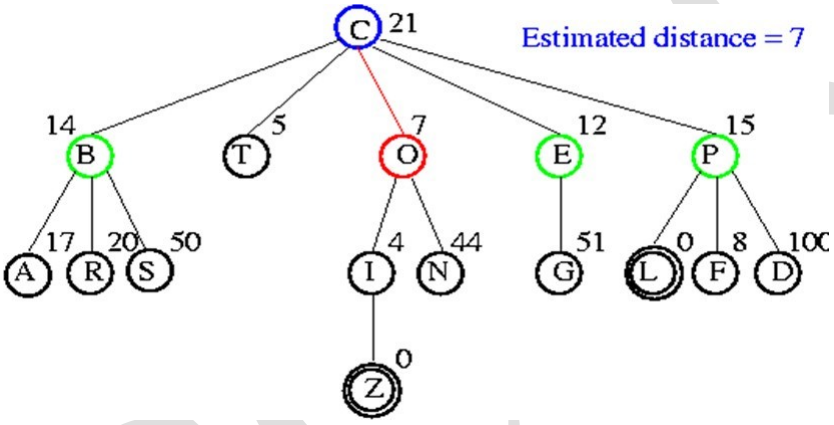
Open: O, E, B, P

Closed: C, T



As T does not have any successors, the next node from open that is O is removed from Open and added to closed.

Open: E, B, P

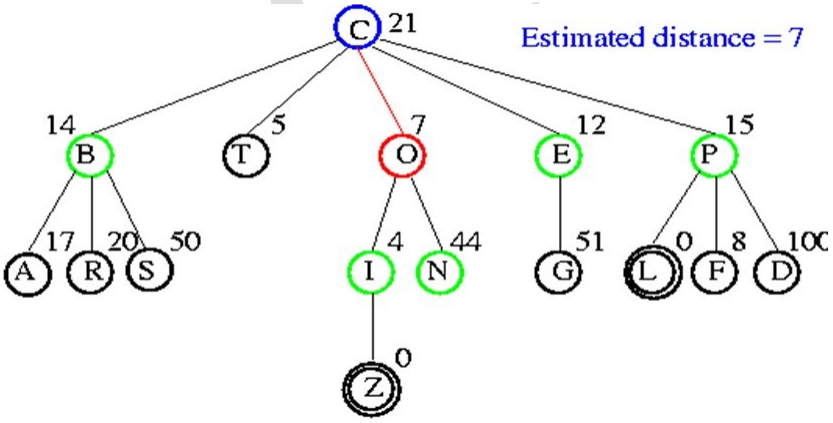


Closed: C, T, O

The successors of node O that is node I and N are added to Open.

Open: I, E, B, P, N

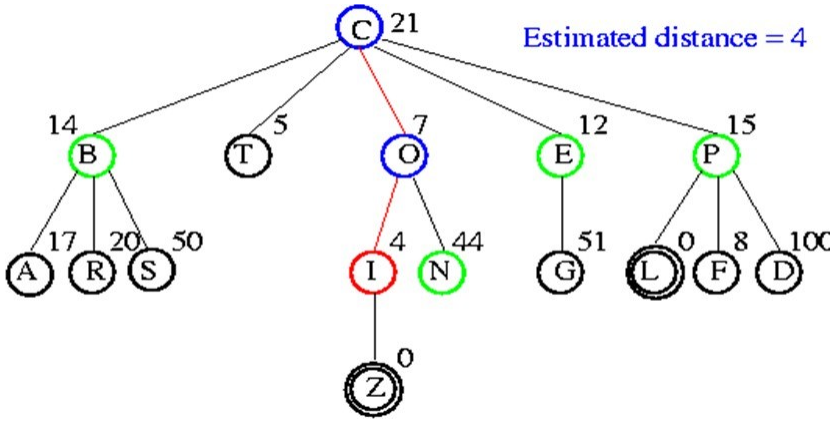
Closed: C, T, O



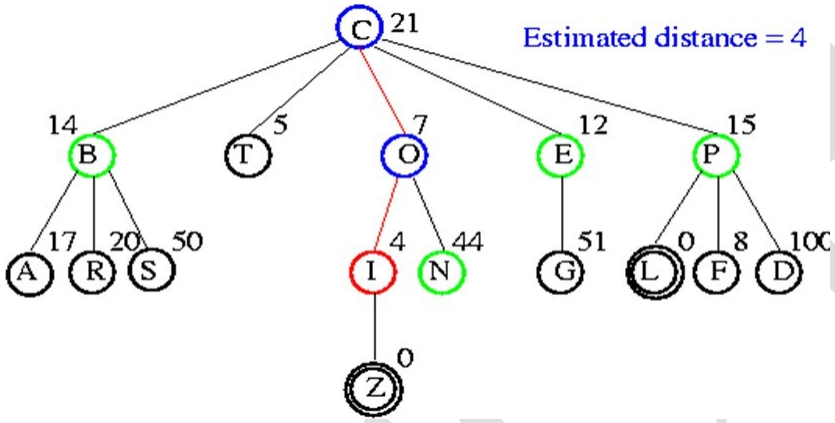
Now, node I is removed from Open and added to closed.

Open: E, B, P, N

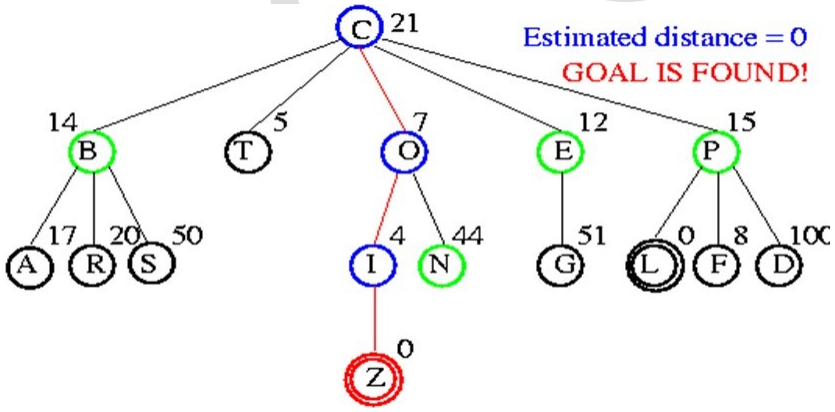
Closed: C, T, O, I



The successor of I that is Z is added to Open.
Open: Z, E, B, P, N
Closed: C, T, O, I



Now, node Z is removed from Open and added to closed.
Open: E, B, P, N
Closed: C, T, O, I, Z



The Goal is found. The final path is C – O – I – Z.

A* Algorithm

Developed in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael, the A* algorithm was designed as an extension and improvement of Dijkstra's algorithm, which is also known for finding the shortest path between nodes in a graph. Unlike Dijkstra's algorithm, which uniformly explores all directions around the starting node, A* uses heuristics to estimate the cost from a node to the goal, thereby optimizing the search process and reducing the computational load.

The core of the A* algorithm is based on cost functions and heuristics.
f (n) =g (n) +h (n),
It uses two main parameters:

g (n): cost of cheapest path from node initial state to node n
h (n): cost of cheapest path from node n to a goal state
***f (n)*:** cost of cheapest path from initial state to goal state

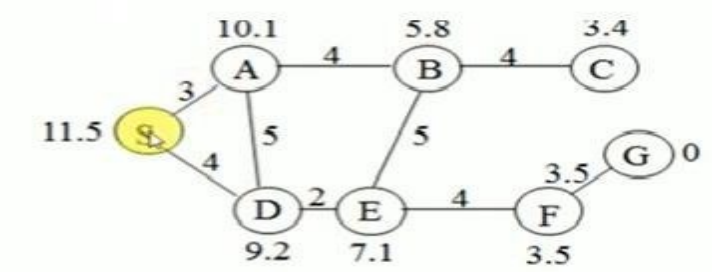
The A* algorithm functions by maintaining a priority queue (or open set) of all possible paths along the graph, prioritizing them based on their f (n) values. (calculated using above formula)
Algorithm A* selects the nodes to be explored based on the lowest value of f(n), preferring the nodes with the lowest estimated total cost to reach the goal.

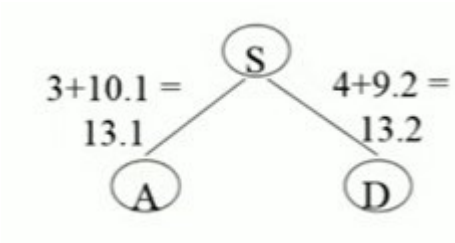
Algorithm

1. Create an open list of found but not explored nodes.
2. Create a closed list to hold already explored nodes.
3. Add a starting node to the open list with an initial value of g
4. Repeat the following steps until the open list is empty or you reach the target node:

a. Find the node with the smallest f-value (i.e., the node with the minor g(n) h(n)) in the open list.
b. Move the selected node from the open list to the closed list.
c. Create all valid descendants of the selected node.
d. For each successor, calculate its g-value as the sum of the current node's g value and the cost of moving from the current node to the successor node. Update the g-value of the tracker when a better path is found.
e. If the follower is not in the open list, add it with the calculated g-value and calculate its h-value. If it is already in the open list, update its g value if the new path is better.
f. Repeat the cycle. Algorithm A* terminates when the target node is reached or when the open list empties, indicating no paths from the start node to the target node.

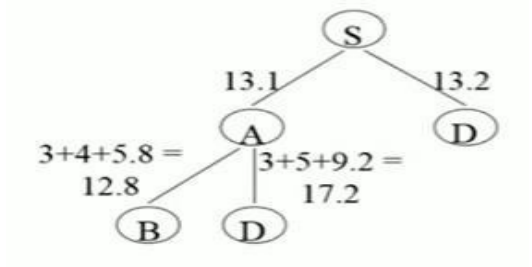
The A* search algorithm is widely used in various fields such as robotics, video games, network routing, and design problems because it is efficient and can find optimal paths in graphs or networks.
Example: Find the shortest path from source node S to goal node G which has its Heuristic value is 0.



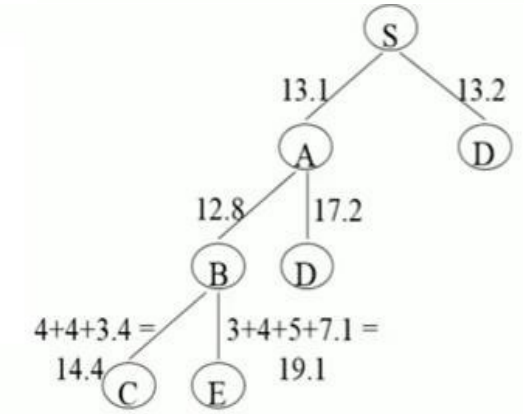


f(A)=g(A)+h(A) → 3+10.1=13.1
Distance from S to A is 3 and the Heuristic value of A is 10.1
f(D)=g(D)+h(D) → 4+9.2=13.2

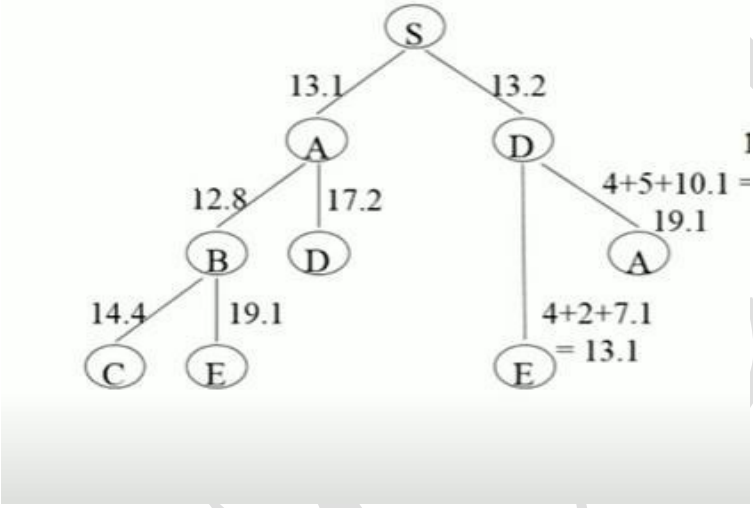
Here consider all leaf nodes, among that select the node which is having minimum f (n) value. Minimum f (n) among 13.1 and 13.2 and explore the node which has minimum f (n) value consider it as the current node and explore that node



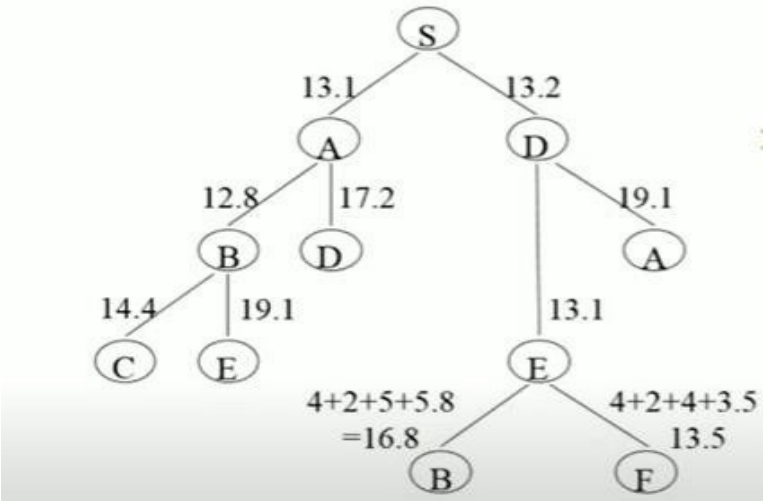
Here out of all leaf nodes, f (n) value of node B is the minimum explore B node next

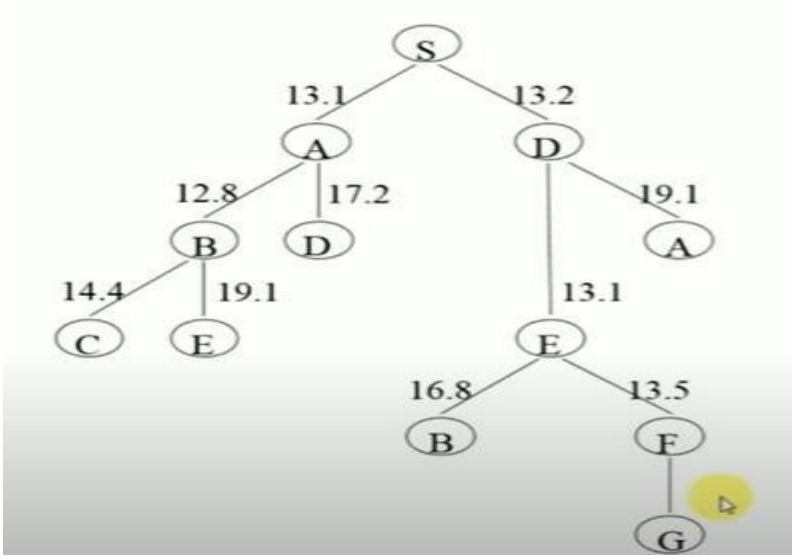


Here out of all leaf nodes C, E, and D, node C has a minimum f(n)value, next explore node D.



Here out of all leaf nodes C, E, D, A, node E has a minimum f(n) value. So next explore node E.





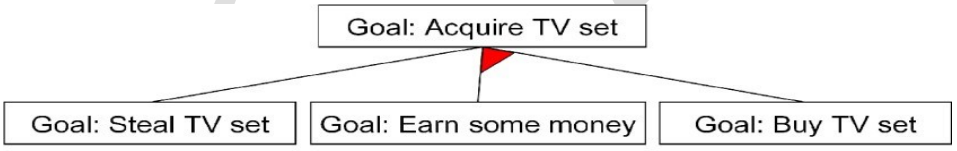
G is the goal node with the total distance from source node S to G is 13.5

Problem Reduction

In Problem Reduction technique, the problem is decomposed into smaller sub-problems. Each of these sub-problems are solved to get its sub-solution. These sub-solutions are then recombined to get a solution as a whole.

AND-OR graph or tree are used to represent the Problem Reduction solution. The decomposition of the problem/problem reduction generates AND arcs. One AND arc may point to any number of successor nodes and all these must be solved. One node may give rise to many arcs, indicating several possible solutions.

Example of AND-OR graph:



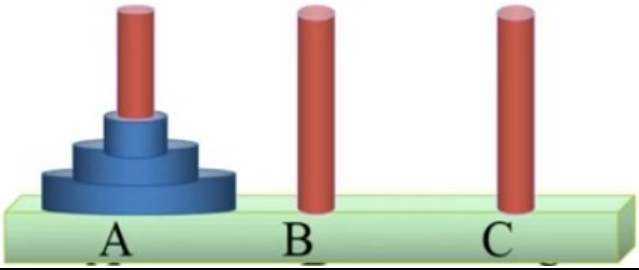
AND-OR Graphs

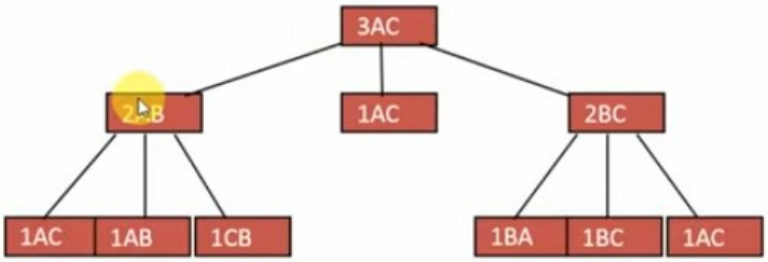
- In a problem reduction space, the **nodes** represent problems to be solved or goals to be achieved, and the **edges** represent the decomposition of the problem into subproblems.

Problem Reduction – Solved Example - AI

- The root node, labeled "3AC" represents the original problem of transferring all 3 disks from tower A to tower C.

- The root node, labeled "3AC" represents the original problem of transferring all 3 disks from tower A to tower C.
- The goal can be decomposed into three subgoals: 2AB, 1AC, 2BC. In order to achieve the goal, all 3 subgoals must be achieved.





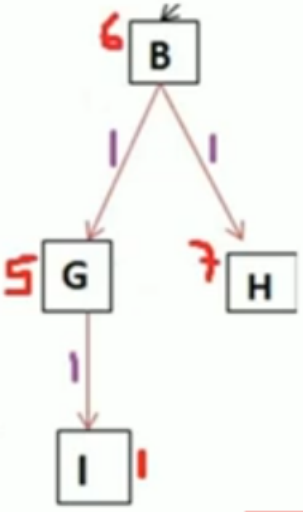
AO* Algorithm

AO* Search Algorithm in Artificial Intelligence

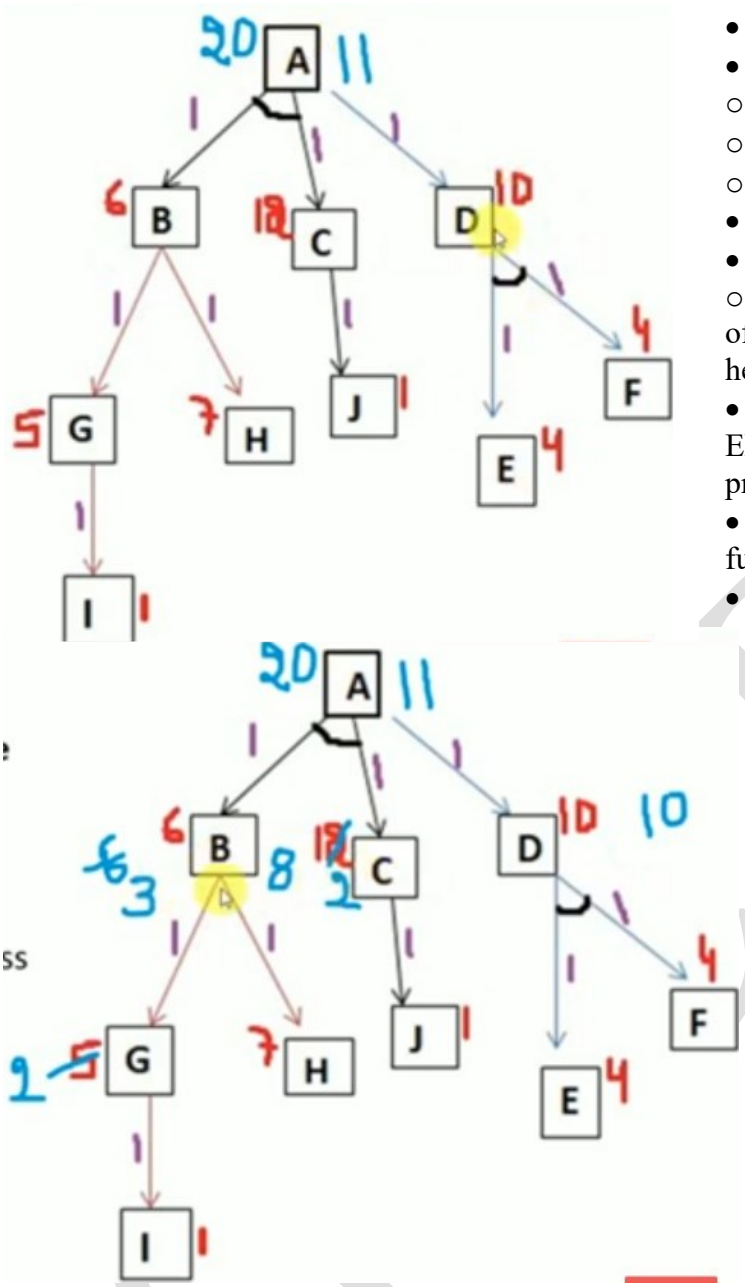
- AO* algorithm is a heuristic search algorithm in AI.
- AO* algorithm uses the concept of AND-OR graphs to decompose any complex problem given into smaller set of problems which are further solved.
- **Working of AO* algorithm:**
- The AO* algorithm works on the formula given below :

$$f(n) = g(n) + h(n)$$

- where,
- $g(n)$: The actual cost of traversal from initial state to the current state.
- $h(n)$: The estimated cost of traversal from the current state to the goal state
- $f(n)$: The actual cost of traversal from the initial state to the goal state.



Example: 1



- In diagram we have two ways from A (A to D or
- A to B-C).(Because of AND condition)
- Calculate the cost to select a path.
  - f(A-D) = 1+10=11
  - f(A-BC) = f(A-B)+f(A-C)=1+6+1+12=20.
  - As f(A-D) < f(A-BC) – Select f(A-D) path.
- From D we have one choice that is FE.
- Calculate cost of D-FE
  - f(D-FE) = 1+4+1+4=10. Thus, new cost of D =10 but the estimated cost is also10, hence no change.
- Calculate new cost of A-D =f(A-D-EF)) =10+1=11. It is similar to the previously calculated cost.
- Let's traverse f(A-BC) again by expanding further. From B we have two paths G and H.
- Let's calculate the cost from B.
  - f(B-G)=5+1=6 and f(B-H)=7+1=8
  - f(B-G) < f(B-H) So, select the path B-G
- Explore G for the next level. It has one node I.
- Calculate the cost from G to I
- f(G-I) = 1+1 =2. So new heuristic value/Cost of G is 2.
  - Thus; new cost of B-(G-I) =2+1=3 and is less than B-H=8, select B-G-I
- Explore C. It has one node J.
- Calculate cost of
  - F(C-J)=1+1=2.
  - Replace estimated cost of C which is 12 by 2.
- Calculate new cost of f(A-BC).
  - f(A-BC)=1+3+1+2=7. It is less than f(A-D) which is 11.
- Thus, A-BC is more cost effective than A-D.

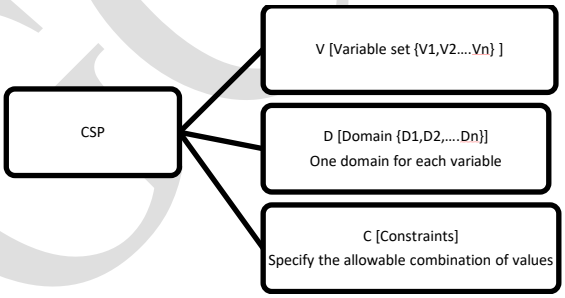
(Additional example is given in the PPT)

Difference between A* and AO* Algorithm

A*	AO*
It represents an OR graph algorithm that is used to find a single solution (either this or that). It is a computer algorithm which is used in path-finding and graph traversal. It is used in the process of plotting an efficiently directed path between a number of points called nodes. In this algorithm you traverse the tree in depth and keep moving and adding up the total cost of reaching the cost from the current state to the goal state and add it to the cost of reaching the current state.	It represents an AND-OR graph algorithm that is used to find more than one solution by ANDing more than one branch. In this algorithm you follow a similar procedure but there are constraints traversing specific paths. When you traverse those paths, cost of all the paths which originate from the preceding node are added till that level, where you find the goal state regardless of the fact whether they take you to the goal state or not.

Constraint Satisfaction

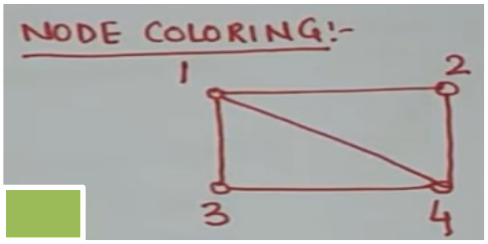
- It is a search procedure that operates in a space of constraint sets
- CSPs in AI have a goal of discovering some problem state that satisfies a given set of constraints.
- Constraint satisfaction problems are the problems which must be solved under some constraints.
- The focus must be on not to violate the constraint while solving such problems.
- **Process :**
  - Constraints are discovered and are propagated throughout the system
  - If still there is no solution, the search begins.
  - A guess is made about something and added as a new constraint.

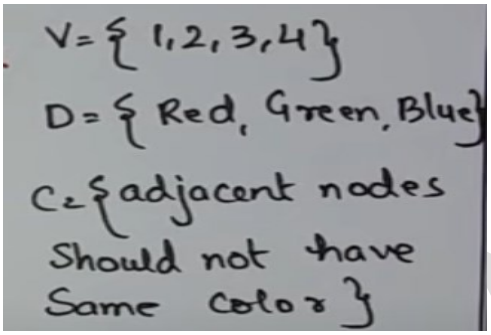


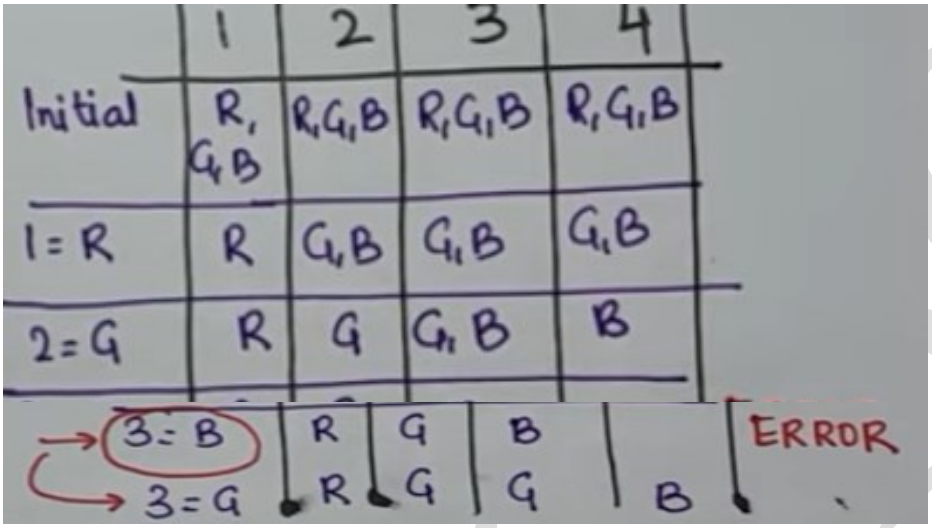
- Ci = (Scope, Relation)
- Scope : Set of variables that participate in a constraint.
- Relation : Defines values that a variable can take.

- Representation: Suppose V1 and V2 are variables with domains D1 and D2
- Constraint : Values of V1 and V2 can not be same.
- C1 = { (V1, V2), V1 <> V2}

- Intelligent Backtracking is used to solve CSP
- Ie Only Backtrack where conflict occurs.







Means-Ends Analysis

- AI has many search strategies which traverse either in forward or backward direction.
- The combination of the two is more appropriate to solve a complex and large problem.
- This mixed search algorithm will allow us to solve the major part of the problem first and then go back to solve the small problems that arise while combining the major parts.
- Such a technique of solving the problem is called Means-Ends Analysis [MEA].
- It concentrates on finding the difference between the current state and goal state and applying the operators to reduce this difference.

Working of Means-Ends Analysis Algorithm:

- To solve a given problem, we need to apply the MEA Algorithm recursively.
- Following are the major steps that describe the working principle of the MEA search technique to solve the problem.
    1. First, find the difference between the Initial (start) State and the Goal State.
    2. From the available set of operators, select an operator which can be applied to the current state to reduce the difference between the current state and the goal state.
    3. Apply the selected operator.
4. If an operator cannot be applied to the current state then divide the current state into sub-problems, and then apply an operator on sub-problems. Such type of analysis is called **Operator subgoaling**.

Algorithm

Let's assume the Current state as CURRENT and Goal State as GOAL, then the following are the steps of the MEA algorithm.

Step 1: Compare CURRENT to GOAL, if there are no differences between both then return Success and Exit.

Step 2: Else, select the most significant difference between CURRENT and GOAL and reduce it by doing the following steps until success or failure occurs.

a. Select a new operator **O** which is applicable for the current difference, if thereis no such operator, then signal failure.

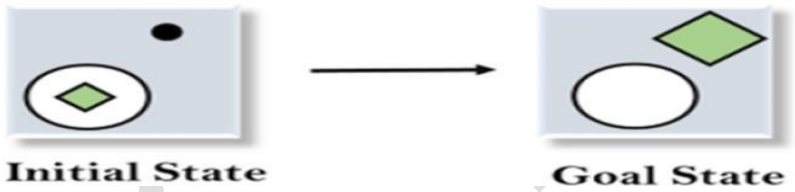
Attempt to apply operator **O** to CURRENT. Make a description of two states.

i. O-Start, a state in which O's preconditions are satisfied.
ii. O-Result, the state that would result if O were applied in O-start.

b. If
(First-Part <—— MEA (CURRENT, O-START) And
LAST-Part <—— MEA (O-Result, GOAL), are successful, then signal Success and return the result by combining FIRST-PART, O, and LAST-PART.

Example:

Let's assume the initial state and goal state as given below.



In this problem, we need to get the goal state from the initial state by finding differences between the initial state and the goal state and applying suitable operators.

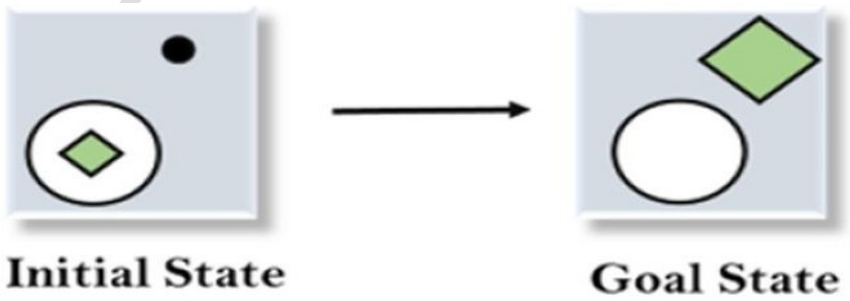
To solve the given problem, we need to find the differences between the initial state and the goal state, and for each of these differences, we will apply an operator to generate a new state.

The operators we have for this problem are:

- **Delete-** Delete the Black circle
- **Move –** Move the object diamond outside the circle
- **Expand –** Expand or increase the size of the diamond

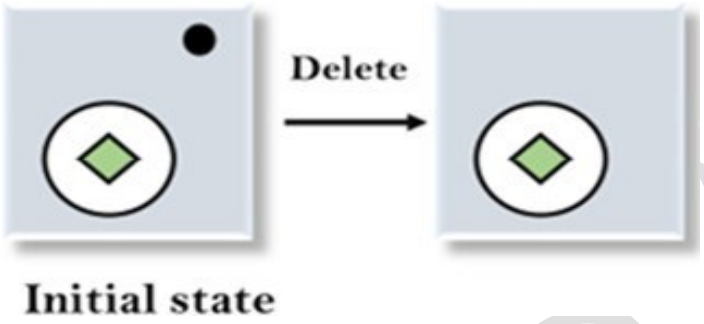
1. Evaluating the initial state:

In the first step, we will evaluate the initial state and will compare the initial state and goal state to find the differences between both states.



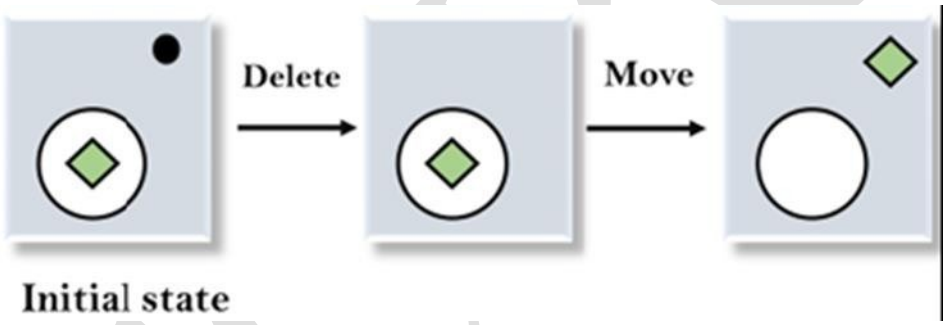
Applying Delete operator:

As we can check the first difference is that in the goal state there is no dot symbol that is present in the initial state, so, first, we will apply the Delete operator to remove this dot.



2. Applying Move Operator:

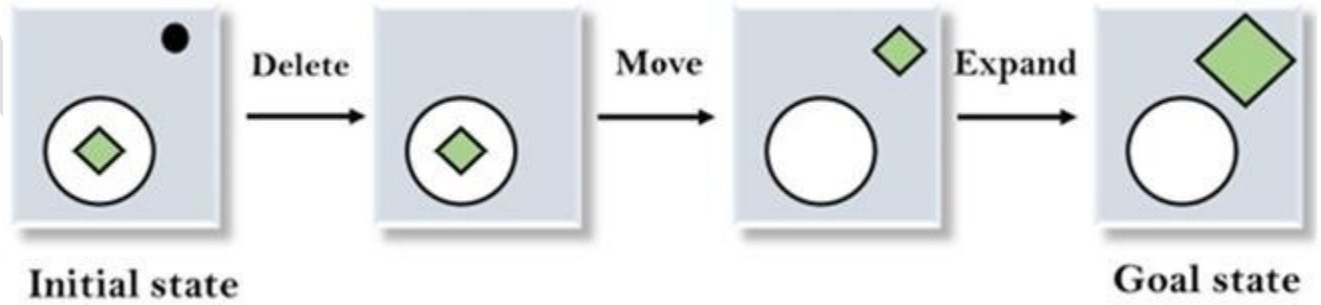
After applying the Delete operator, the new state occurs which we will again compare with the goal state. After comparing these states, there is another difference that is the square is outside the circle, so, we will apply the Move Operator.



3. Applying Expand Operator:

Now a new state is generated in the third step, and we will compare this state with the goal state.

After comparing the states there is still one difference which is the size of the square, so, we will apply Expand operator, and finally, it will generate the goal state.



Game Playing

- Game playing in AI refers to the field of artificial intelligence where algorithms and systems are developed to play games, often to achieve human-level or superhuman performance.
- Game-playing AI involves creating intelligent agents that can make decisions, strategize, and adapt

their gameplay based on the rules and objectives of the game.

- ❖ Game playing in AI involves creating algorithms that can play games against human opponents or other AI systems.
- ❖ These algorithms use a type of search called adversarial search because they must account for an opponent actively trying to counter their moves.
  Classic examples of such games include chess, checkers, and tic-tac-toe.
- ❖ **Game Tree**: A graphical representation of all possible moves in a game. Each node represents a game state, and each edge represents a move from one state to another.
- ❖ The initial state of the game is represented by the root
- ❖ Root node's successors are the positions that the first player can reach in one move;
- ❖ Their successors are the positions resulting from the second player's move and so on.
- ❖ Terminal or leaf nodes are represented by WIN(+1), LOSS(-1) OR DRAW(0).

- There might be some situations where more than one agent is searching for the solutionin the same search space, and this situation usually occurs in game playing.
- The environment with more than one agent is termed as **multi-agent environment**, inwhich each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
- So,.
- Games are modelled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

A game can be defined as a type of search in AI which can be formalized of the following elements:

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called a terminal state.
- **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called a payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

Adversial Search- Vaishali Kulkarni

- Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution are called adversarial searches, often known as Games

Mini-Max Algorithm

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The mini-max algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The mini-max algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Working of Min-Max Algorithm:

- The working of the mini-max algorithm can be easily described using an example. Below we have taken an example of game-tree which represents the two-player game.
- In this example, there are two players one is called Maximizer and the other is called Minimizer.
- The maximizer will try to get the Maximum possible score, and the Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs.
- Following are the main steps involved in solving the two-player game tree:

(Algorithm : Not So imp)

Given a game tree, the optimal strategy can be determined from the ***mini-max value*** *of each node, which we write as MINIMAX(n). The mini-max value of a node is the utility (for MAX) of being in the corresponding state,* ***assuming that both players play optimally*** *from there to the end of the game. Obviously, the mini-max value of a terminal state is just its utility. Furthermore, given a choice, MAX prefers to move to astate of maximum value, whereas MIN prefers a state of minimum value*

Algorithm: MINIMAX(Position, Depth, Player)

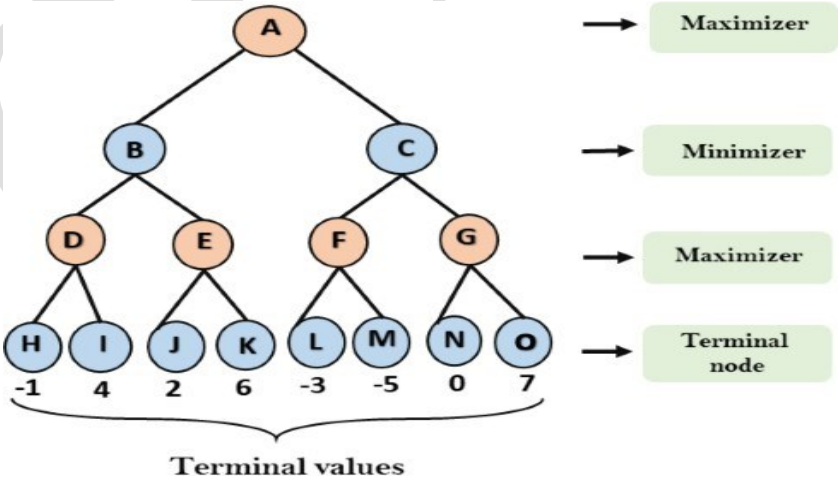
1. If DEEP-ENOUGH(*Position, Depth*), then return the structure

   VALUE = *STATIC*(*Position, Player*);
   PATH = nil

   This indicates that there is no path from this node and that its value is that determined by the static evaluation function.
2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(Position Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned if DEEP-ENOUGH had returned true.
4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.
   Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the hest score that can be achieved by an element of SUCCESSORS.
   For each element SUCC of SUCCESSORS, do the following:
   (a) Set RESULT-SUCC to
   MINIMAX(SUCC, *Depth* + 1, OPPOSITE(*Player*))
   This recursive call to MINIMAX will actually carry out the exploration of SUCC.
   (b) Set NEW-VALUE to - VALUE(RESULT-SUCC). This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level.
   (c) If NEW-VALUE > BEST-SCORE, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:
   (i) Set BEST-SCORE to NEW-VALUE.
   (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined hy the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).
5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So return the structure
   VALUE = BEST-SCORE
   PATH = BEST-PATH

Step-1:

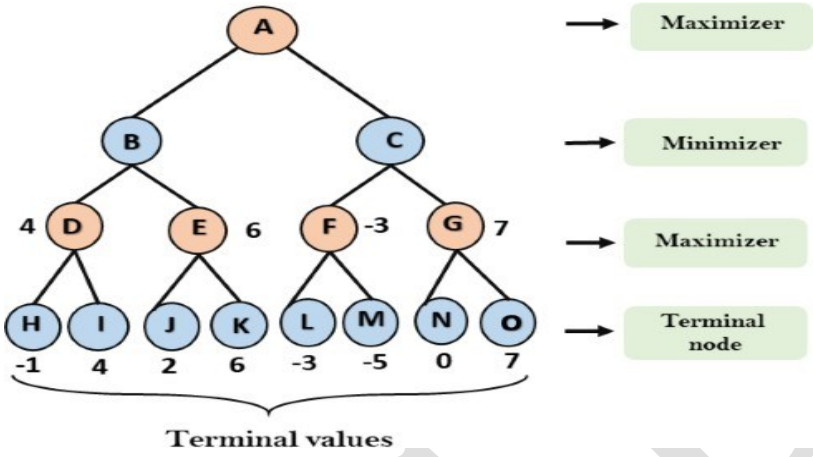
In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states.



Step 2:

Now, first, we find the utility value for the Maximizer,

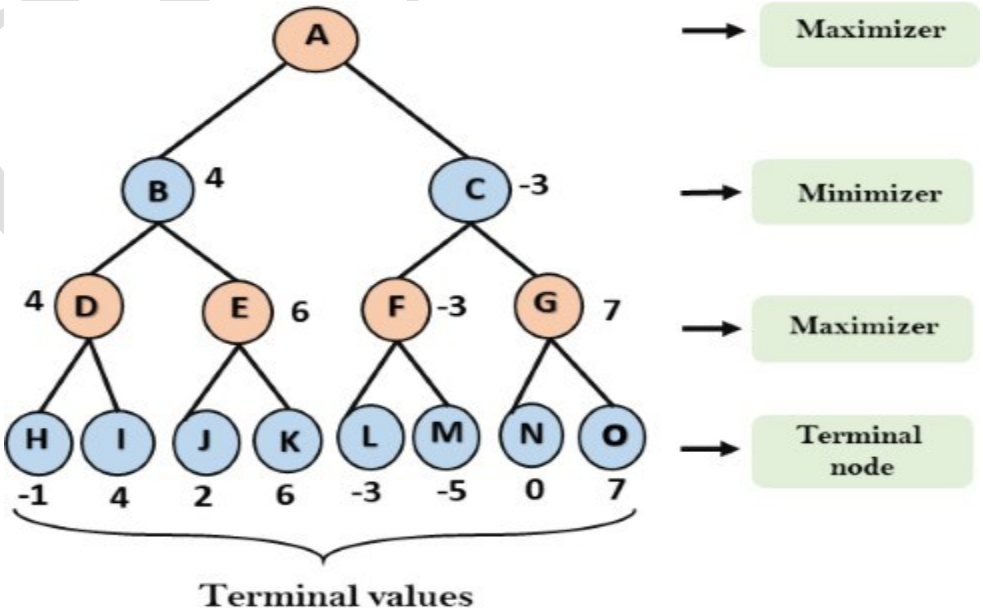
- For node D max(-1,4)= 4
- For Node E max(2, 6)= 6
- For Node F max(-3,-5) = -3
- For node G max(0, 7) = 7



Step 3:

In the next step, it's a turn for the minimizer, so it will fetch the minimum node values from the leaf nodes.

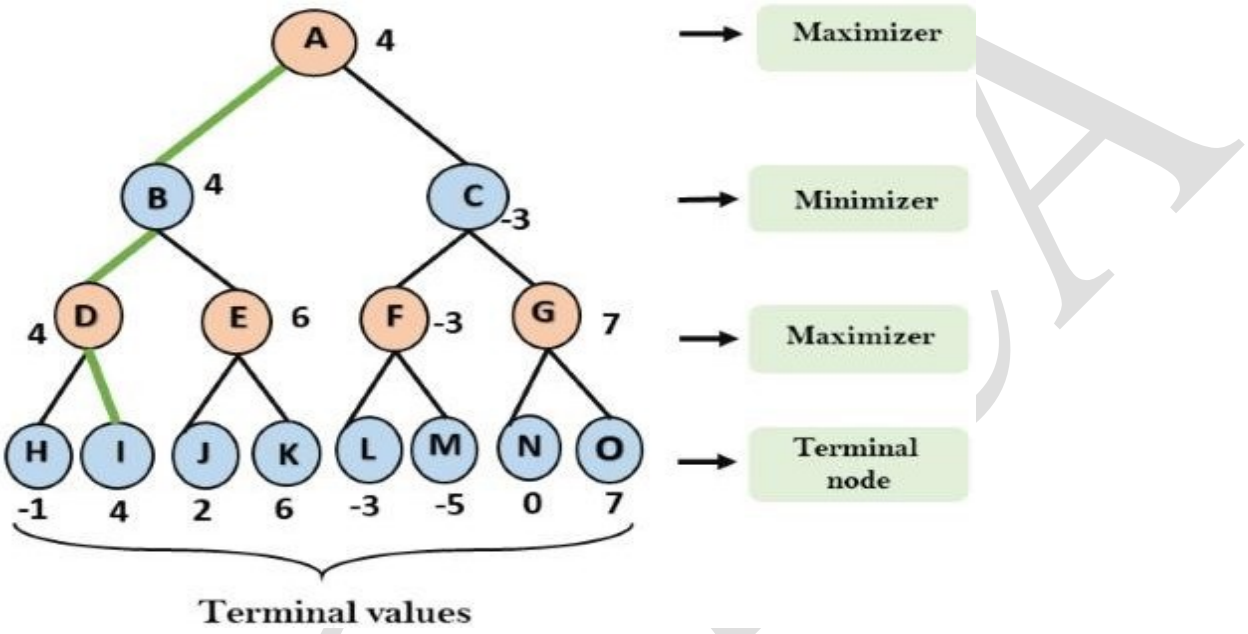
- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



Step 4:

Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node.

- For node A max(4, -3)= 4



The path that will lead the player to victory is **A - B – D - I**

Summary of MINIMAX Algorithm

- **Goal**: To determine the best move for a player assuming that the opponent also plays optimally.
- **Process**:
  - Generate the game tree from the current state.
  - Evaluate the leaf nodes using an evaluation function.
  - Backpropagate the values up the tree, choosing maximum or minimum values depending on whose turn it is.
  - The move corresponding to the highest value at the root node is chosen as the best move.
- **Advantages:**
  - Provides a clear strategy for decision-making in competitive environments.
  - Guarantees the best possible outcome against an optimal opponent.
- **Limitations:**
  - Computationally expensive, especially for games with large branching factors and deeper game trees.
  - Assumes that the opponent will always play optimally, which might not be true in real-world scenarios.

HEURISTIC SEARCH STRATEGIES & FUNCTIONS

Heuristic strategies and heuristic functions are essential concepts in artificial intelligence (AI), particularly in areas like search algorithms, optimization problems, and decision-making processes.

HEURISTIC STRATEGIES

Heuristic strategies refer to approaches or methods used to make decisions or solve problems more efficiently when traditional methods are too slow or fail to find an exact solution.

These strategies aim to produce good enough solutions within a reasonable time frame, often by simplifying the problem or focusing on the most promising areas

HEURISTIC FUNCTIONS

A heuristic function (often denoted as h(n) is a function used to rank alternatives in search algorithms at each step based on available information to estimate the best path to a goal. It provides a way to evaluate which node (or state) in a search tree should be explored next.

Characteristics of Heuristic Functions

• **Guidance:** Heuristic functions provide guidance to the search process by estimating the cost to reach the goal from a given state. They help prioritize which paths to explore.

• **Accuracy:** The accuracy of a heuristic refers to how closely it estimates the true cost to reach the goal. More accurate heuristics lead to more efficient searches

. • **Domain-specific**: Heuristics are often tailored to specific problems or domains. Effective heuristics leverage the structure and properties of the particular problem being solved.

• **Informedness**: A good heuristic is informed, meaning it uses domain-specific knowledge to make educated guesses about which paths are more likely to lead to a solution.

• **Efficiency**: Heuristics aim to reduce the search space, making the search process more efficient by focusing on promising areas and ignoring less likely paths.

• **Admissibility:** An admissible heuristic never overestimates the cost to reach the goal. This ensures that the heuristic is optimistic and leads to optimal solutions in algorithms like A*.

Application of Heuristic functions

• Navigation Systems
• Game Playing
• Search Engines
• Robot Path Planning
• Recommendation Systems
• One of the most illustrative real-life examples of heuristic values in AI can be found in the field of navigation and route planning, such as those used by GPS systems or map applications like Google Maps.