Problem Solving by Searching

Contents

- Definition, Problem as a state space search, Problem formulation, Well-defined problems, Solving Problems by Searching, Search Strategies, Performance evaluation of search techniques,
- Uninformed Search: Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Search, Bidirectional Search,
- Informed Search: Greedy Best first search, A* search, Hill Climbing, Simulated Annealing,
- Game playing, Adversarial search techniques, Mini-max Search, Alpha-Beta Pruning, Constraint Satisfaction Problems.

Why Search in AI?

- Search is an essential component of artificial intelligence (AI) because it enables machines to find solutions to complex problems within a vast search space.
- Search algorithms provide a systematic and efficient way for AI systems to navigate through this space and find optimal or satisfactory solutions. Here are some reasons why search is crucial in artificial intelligence:
    - Complex Problem Solving
    - Unpredictable Environments
    - Resource Efficiency
    - Game Playing
    - Planning and Decision Making
    - Optimization
    - Natural Language Processing
    - Robotics, Medical Diagnosis etc.

- Overall, search algorithms play a critical role in allowing AI systems to explore, evaluate, and select from a multitude of possibilities in various applications. They enable machines to make intelligent decisions and solve problems efficiently, even in scenarios with vast and intricate solution spaces.

Problem-Solving Agents

- The agent can follow this four-phase problem-solving process:
  - **Goal formulation:** The agent adopts the goal of reaching Bucharest. Goals organize behavior by limiting the objectives and hence the actions to be considered.
  - **Problem formulation:** The agent devises a description of the states and actions necessary to reach the goal—an abstract model of the relevant part of the world.
  - **Search:** Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called a **solution.**
  - **Execution:** The agent can now execute the actions in the solution, one at a time.

Search problems and solutions

- A search problem can be defined formally as follows:
  - A set of possible **states** that the environment can be in. We call this the **state space**.
  - The **initial state** that the agent starts in. For example: *Arad*
  - A set of one or more **goal states**.
  - The **actions** available to the agent. Given a state *s*, ACTIONS(*s*) returns a finite set of actions that can be executed in *s*. We say that each of these actions is applicable in *s*. An example: ACTIONS(*Arad*) = { *ToSibiu*, *ToTimisoara*, *ToZerind* }.
  - A **transition model**, which describes what each action does. RESULT(*s*, *a*) returns the Transition model state that results from doing action *a* in state *s*. For example, RESULT(*Arad*, *ToZerind*) = *Zerind* .
  - An **action cost function**, denoted by ACTION-COST(*s*, *a*, *s′*) when we are programming or *c*(*s*,*a*, *s′*) when we are doing math, that gives the numeric cost of applying action *a* in state *s* to reach state *s′*.

- A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.
- The state space can be represented as a **graph** in which the vertices are states and the directed edges between them are actions.
- **Example : Route Planning**

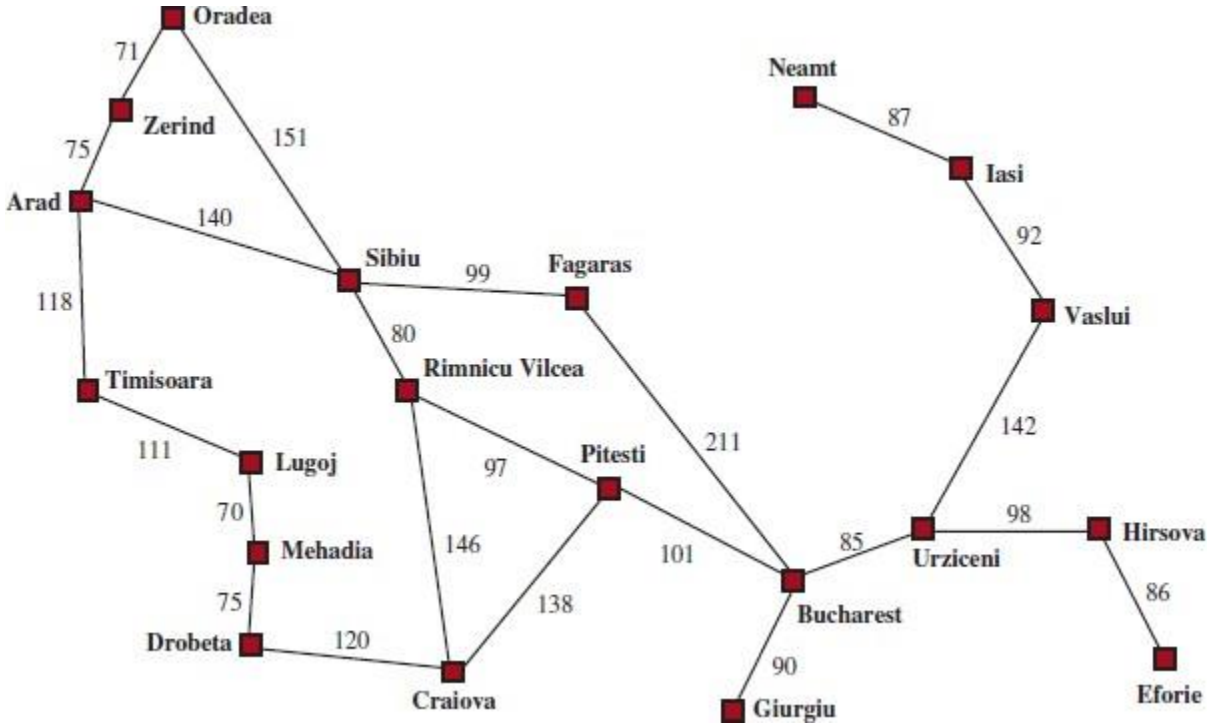


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Measuring problem-solving performance/ Properties of Search Algorithms

- We can evaluate an algorithm's performance in four ways:
  - **Completeness**: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
  - **Cost optimality:** Does it find a solution with the lowest path cost of all solutions?
  - **Time complexity:** How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.
  - **Space complexity:** How much memory is needed to perform the search?
- For an implicit state space, complexity can be measured in terms of
  - *d*, the depth or number of actions in an optimal solution;
  - *m*, the maximum number of actions in any path; and
  - *b*, the branching factor or number of successors of a node that need to be considered.

Example Problems

We list some of the best known here, distinguishing between *standardized* and *real-world* problems.

- A **standardized problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms.
- A **real-world problem**, such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data.

Standardized problems:

- A **grid world** problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell.

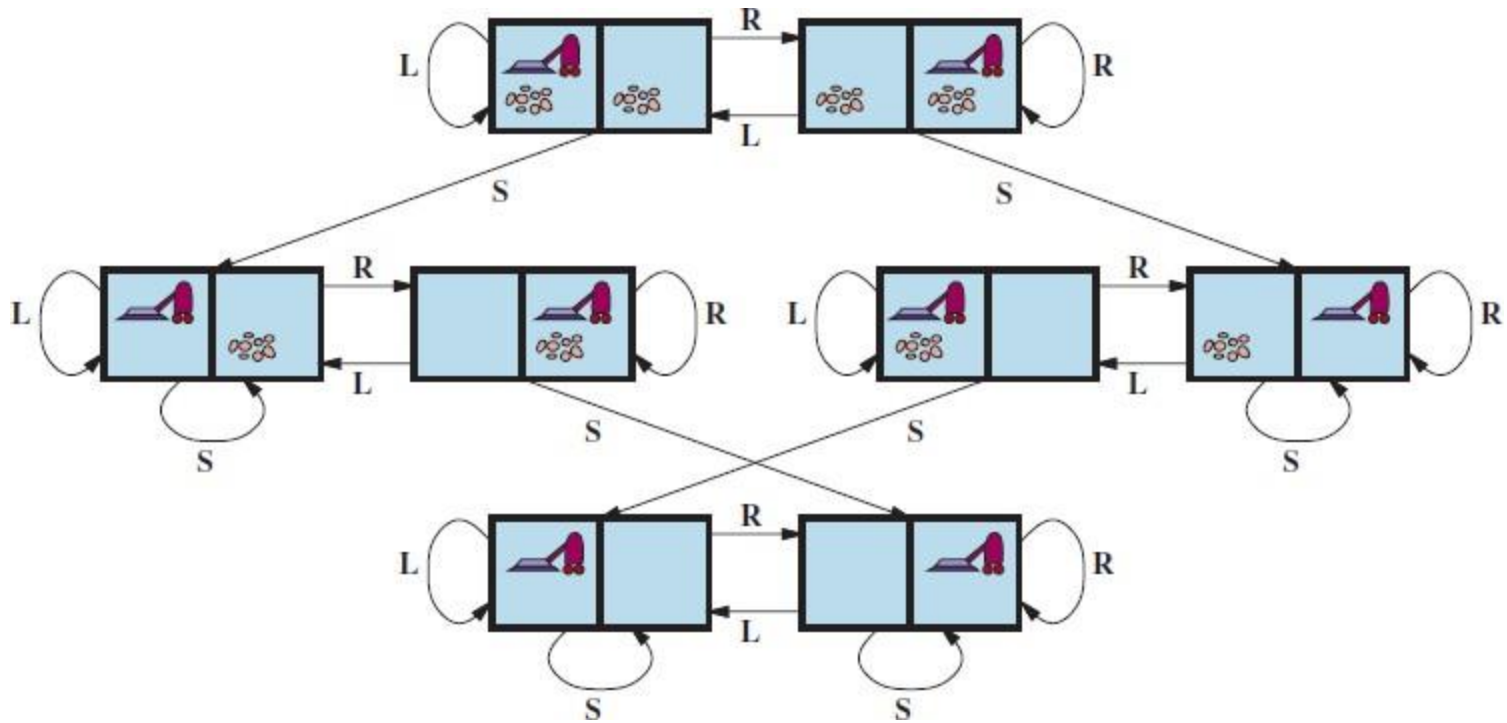


Figure 3.2: The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

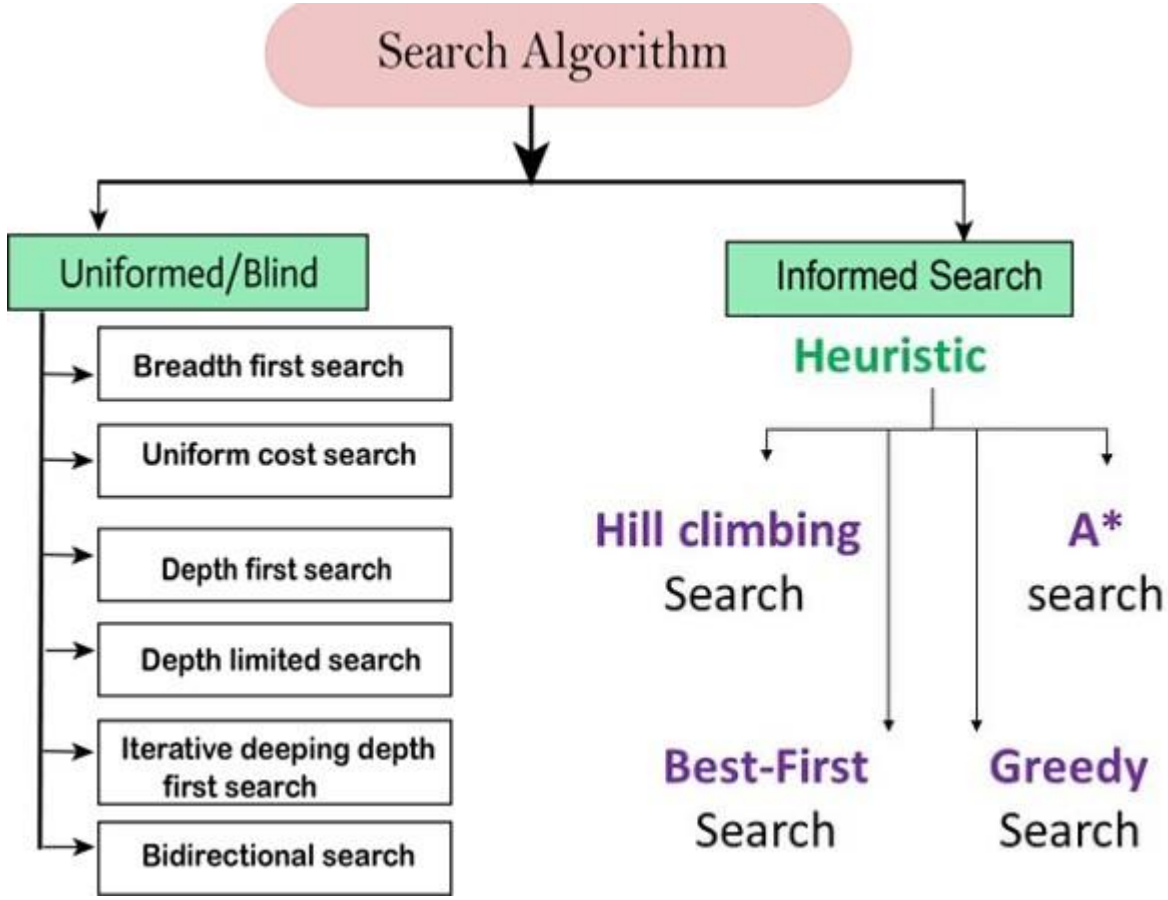
- The **vacuum world** can be formulated as a grid world problem as follows:
  - **States:** A state of the world says which objects are in which cells. For the vacuum world, the objects are the agent and any dirt. In the simple two-cell version, the agent can be in either of the two cells, and each call can either contain dirt or not, so there are $2 \cdot 2^2 = 8$ states. In general, a vacuum environment with n cells has $n \cdot 2^n$ states.
  - **Initial state:** Any state can be designated as the initial state.
  - **Actions:** In the two-cell world we defined three actions: *Suck*, move *Left*, and move *Right*. In a two-dimensional multi-cell world we need more movement actions. For example, *Forward*, *Backward*, *TurnRight*, and *TurnLeft*.
  - **Transition model:** *Suck* removes any dirt from the agent's cell.
  - **Goal states**: The states in which every cell is clean.
  - **Action cost:** Each action costs 1.

Real-world problems:

- Consider the **airline travel problems** that must be solved by a travel-planning Web site:
  - **States:** Each state obviously includes a location (e.g., an airport) and the current time.
  - **Initial state:** The user's home airport.
  - **Actions:** Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
  - **Transition model:** The state resulting from taking a flight will have the flight's destination as the new location and the flight's arrival time as the new time.
  - **Goal state:** A destination city. Sometimes the goal can be more complex, such as "arrive at the destination on a nonstop flight."
  - **Action cost:** A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, and so on.

Search Algorithms

- A search algorithm takes a search problem as input and returns a solution, or an indication of failure.
- Based on the search problems we can classify the search algorithms into **uninformed search (Blind search)** search and **informed search (Heuristic search)** algorithms.



Uninformed/Blind Search

- The uninformed search does not contain any domain knowledge such as closeness, the location of the goal.
- It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes.
- Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called **blind search.**
- It examines each node of the tree until it achieves the goal node.
- Divided into five main types:
  - Breadth-first search
  - Uniform cost search
  - Depth-first search
  - Iterative deepening depth-first search
  - Bidirectional Search

Breadth-first Search (BFS):

- 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.**
- In which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

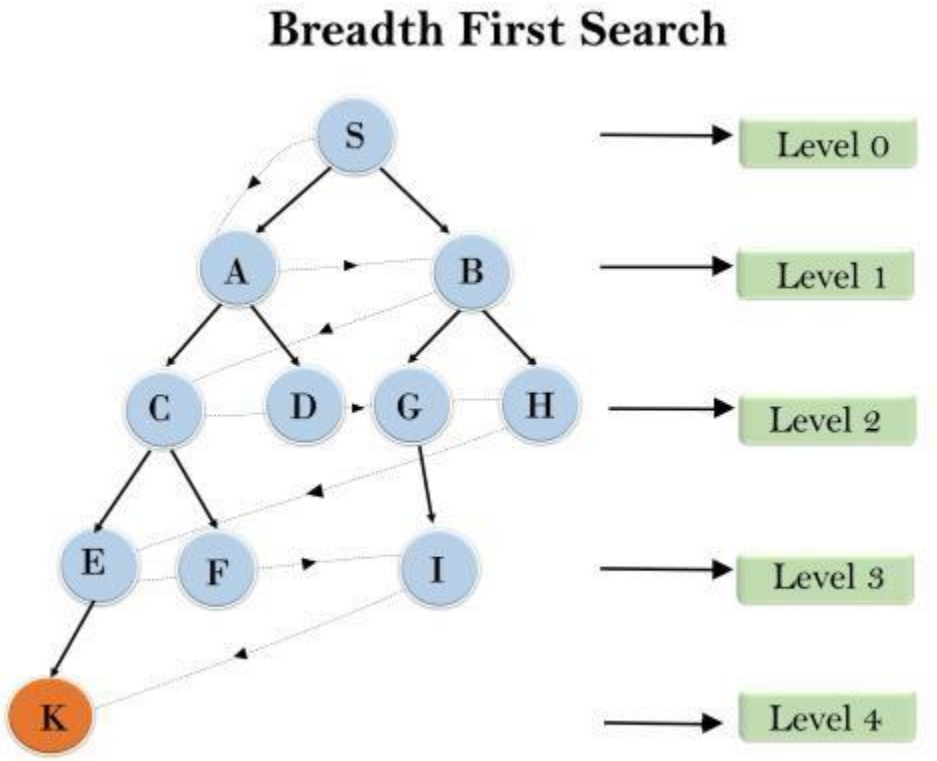
Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Example:

traversed path is

S--->A--->B---->C--->D---->G--->H--->E---->F---->I---->K



- **Time Complexity:** Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$$\mathbf{T(b) = 1+b^2+b^3+.....+ b^d= O(b^d)}$$

- **Space Complexity:** Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.
- **Completeness:** BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.
- **Optimality:** BFS is optimal if path cost is a non-decreasing function of the depth of the node.

Depth-first search (DFS):

- Depth-first search always expands the *deepest* node in the frontier first.
- 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:

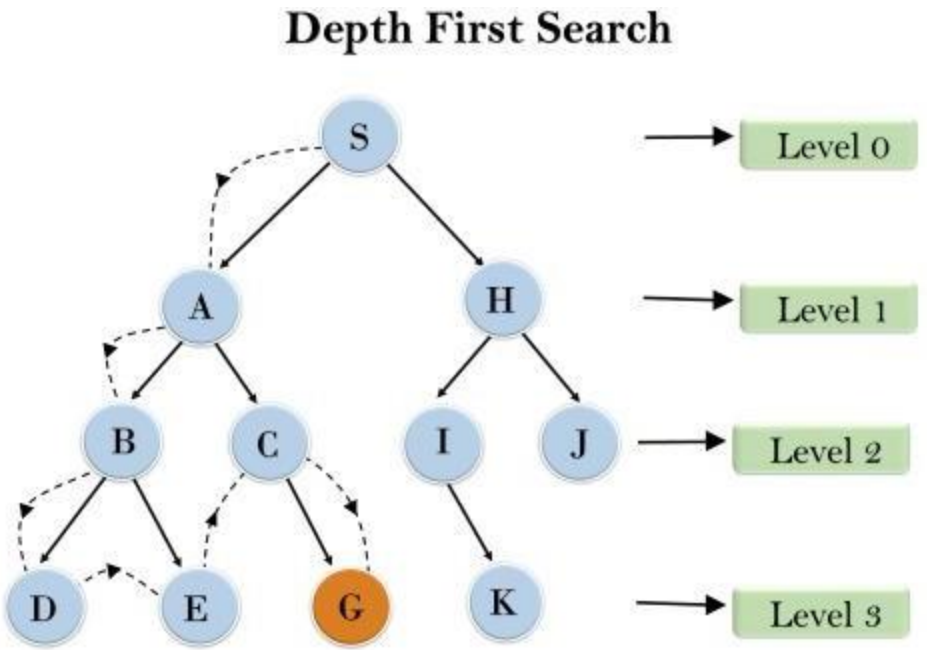
- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

Root node--->Left node ----> right node.



- **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- **Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by: **$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$ Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)**
- **Space Complexity:** DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
- **Optimal:** DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Depth-Limited Search (DLS) :

- A depth-limited search algorithm is similar to depth-first search with a predetermined limit.
- Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
- Depth-limited search can be terminated with two Conditions of failure:
    - Standard failure value: It indicates that problem does not have any solution.
    - Cutoff failure value: It defines no solution for the problem within a given depth limit.

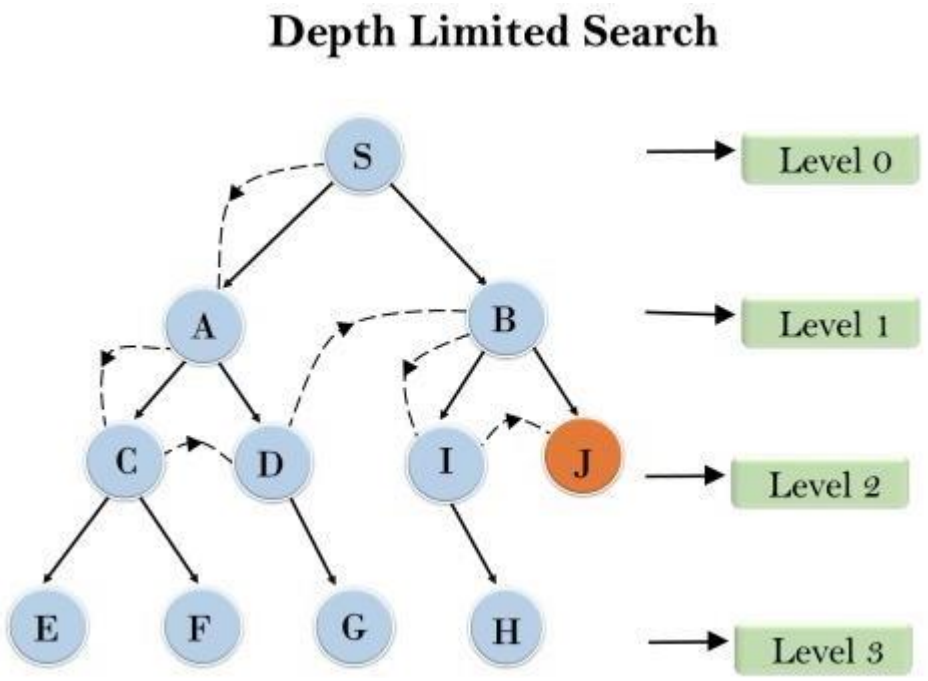
Advantages:

- Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example:



- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is **$O(b^{\ell})$**.
- **Space Complexity:** Space complexity of DLS algorithm is **$O(b \times \ell)$**.
- **Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

Uniform-cost Search (UCS):

- Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge.
- The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.
- Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand.
- A uniform-cost search algorithm is implemented by the **priority queue**. It gives maximum priority to the lowest cumulative cost.
- Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

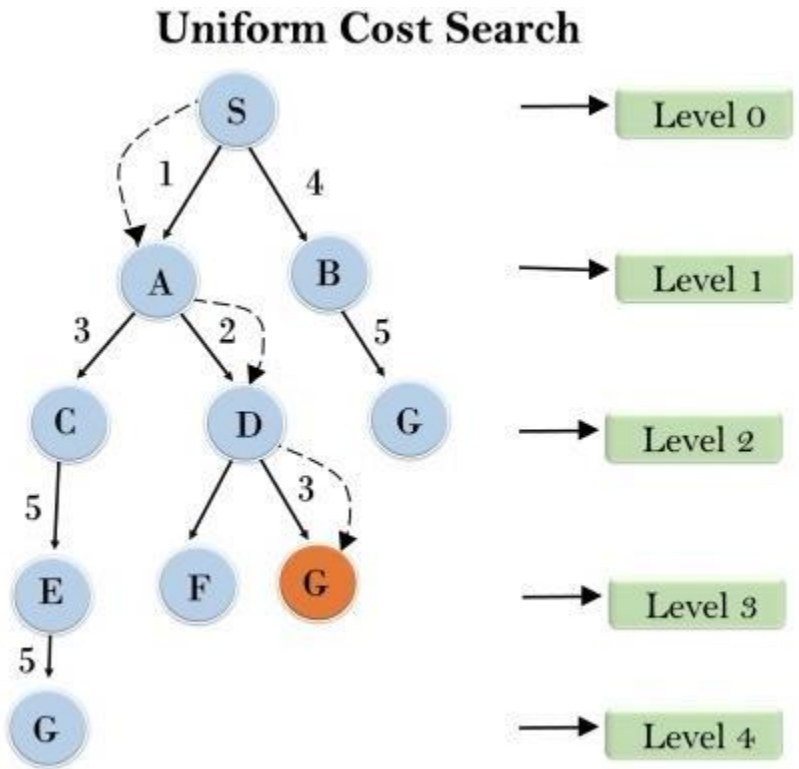
Advantages:

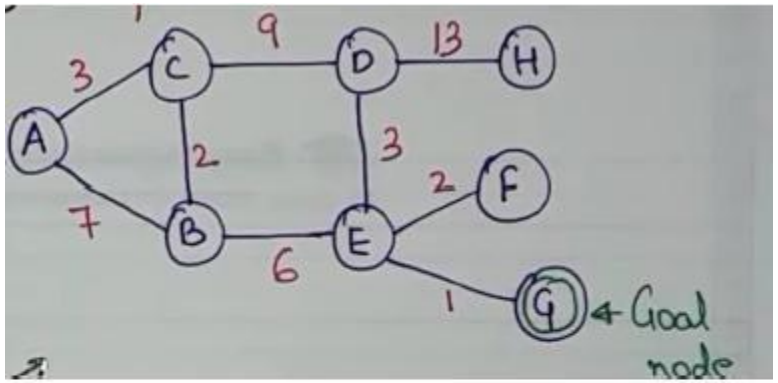
- Uniform cost search is optimal because at every state the path with the least cost is chosen.

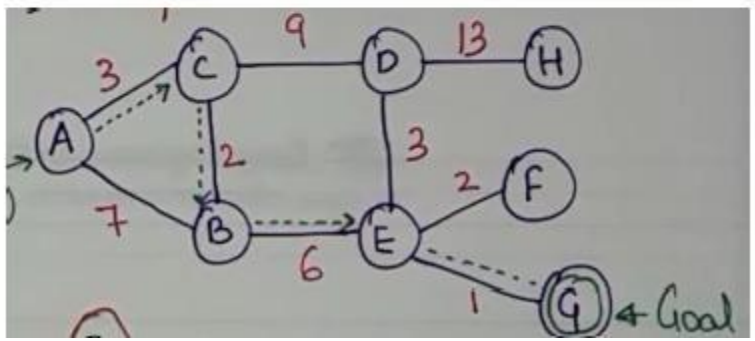
Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:


Uniform Cost Search
S
A
B
C
D
G
E
F
G
G
1
4
3
2
5
5
3
5
Level 0
Level 1
Level 2
Level 3
Level 4


A
C
D
H
B
E
F
G
3
9
13
7
2
3
2
6
1
Goal node


A
C
D
H
B
E
F
G
3
9
13
7
2
3
2
6
1
Goal

- **Completeness:** Uniform-cost search is complete, such as if there is a solution, UCS will find it.
- **Time Complexity:** Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε. Hence, the worst-case time complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$.
- **Space Complexity:** The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$.
- **Optimal:** Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Iterative deepening depth-first Search(IDDFS):

- The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

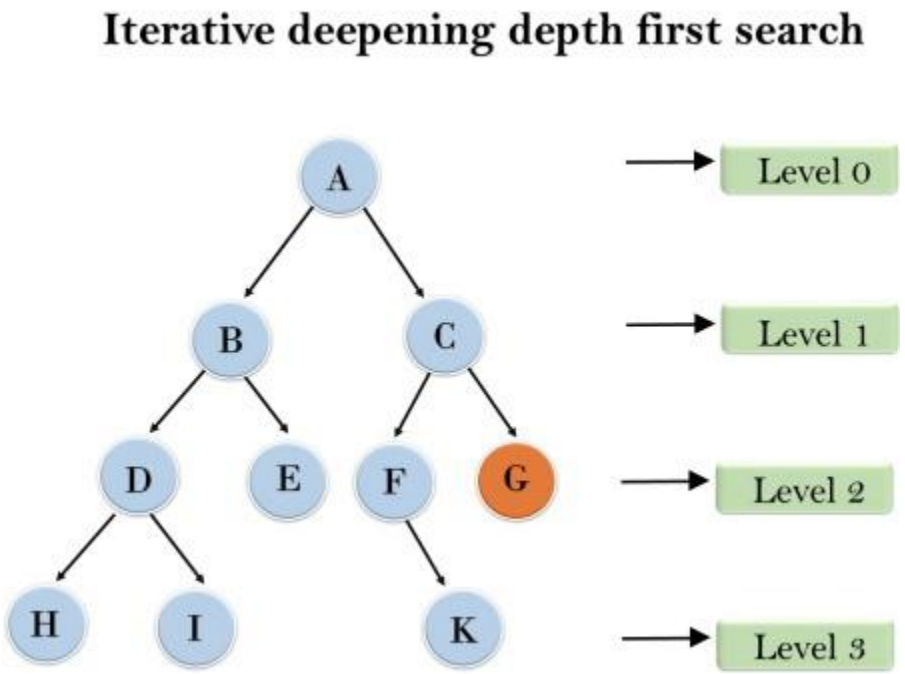
Advantages:

- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:



1'st Iteration-----> A

2'nd Iteration----> A, B, C

3'rd Iteration------>A, B, D, E, C, F, G

4'th Iteration------>A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

- **Completeness:** This algorithm is complete is ifthe branching factor is finite.
- **Time Complexity:** Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.
- **Space Complexity:** The space complexity of IDDFS will be **O(bd)**.
- **Optimal:** IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

Bidirectional search:

- Bidirectional search algorithm runs two simultaneous searches,
  - one form initial state called as **forward-search** and
  - other from goal node called as **backward-search**, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

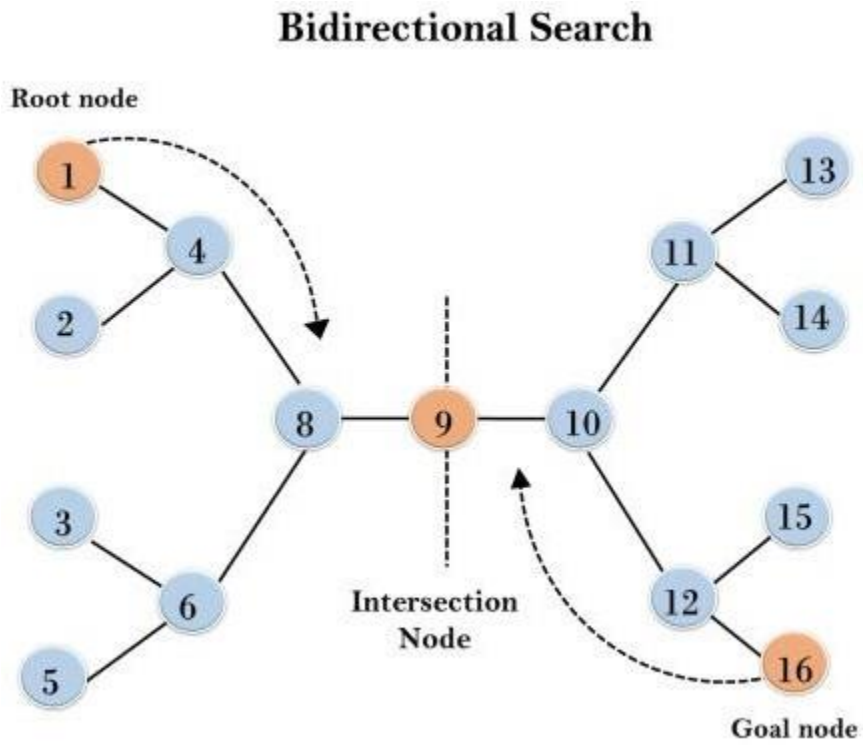
- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

Example:

The algorithm terminates at node 9 where two searches meet.



- **Completeness:** Bidirectional Search is complete if we use BFS in both searches.
- **Time Complexity:** Time complexity of bidirectional search using BFS is **$O(b^d)$**.
- **Space Complexity:** Space complexity of bidirectional search is **$O(b^d)$**.
- **Optimal:** Bidirectional search is Optimal.

Informed (Heuristic) Search

- Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy.
- A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.
- Informed search can solve much complex problem which could not be solved in another way.
- An example of informed search algorithms is a traveling salesman problem.
- Divided into as:
    - **Hill Climbing, Simulated Annealing**
    - **Greedy Search**
    - **Best first Search**
    - **A* Search**

Local Search and Optimization Problems

- **Local search** algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic—they might never explore a portion of the search space where a solution actually resides.
- Local search algorithms can also solve **optimization problems**, in which the aim is to find the best state according to an objective function.
- To understand local search, consider the states of a problem laid out in a **state-space landscape**, as shown in Figure below.
- Each point (state) in the landscape has an "elevation," de-fined by the value of the objective function. If elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**—and we call the process **hill climbing**. If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**—and we call it **gradient descent**.

Iterative Improvement Algorithm

1. Hill climbing search

2. Simulated Annealing search

Hill-climbing search:

- It keeps track of one current state and on each iteration moves to the neighboring state with highest value—that is, it heads in the direction that provides the **steepest ascent**. It terminates when it reaches a “peak” where no neighbor has a higher value.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia(Confusion).
- **Note** that one way to use hill-climbing search is to use the negative of a heuristic cost function as the objective function; that will climb locally to the state with smallest heuristic distance to the goal.

State Space Diagram for Hill Climbing

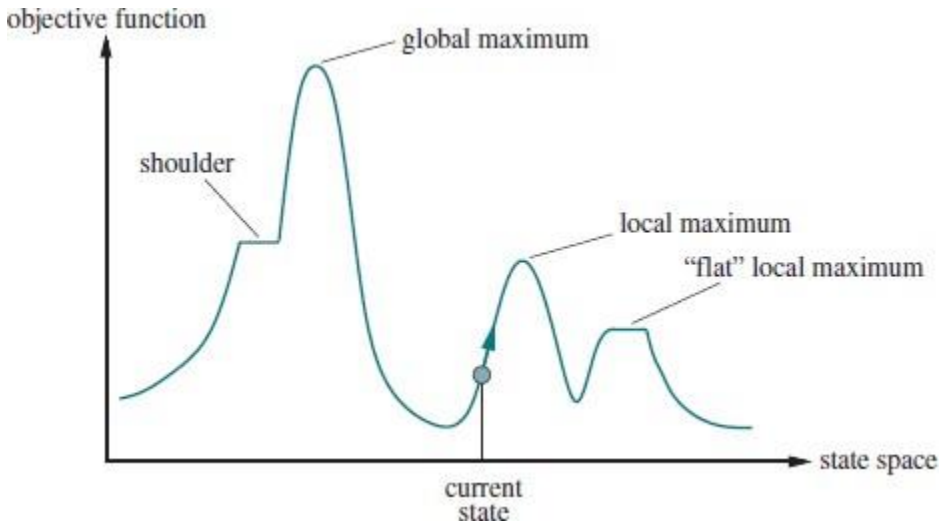


Figure: A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

- Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.

Algorithm:

function HILL-CLIMBING(problem) **returns** a state that is a local maximum

current ← problem. INITIAL

while true do

neighbor ←a highest-valued successor state of current

if VALUE(neighbor) ≤ VALUE(current) then return current

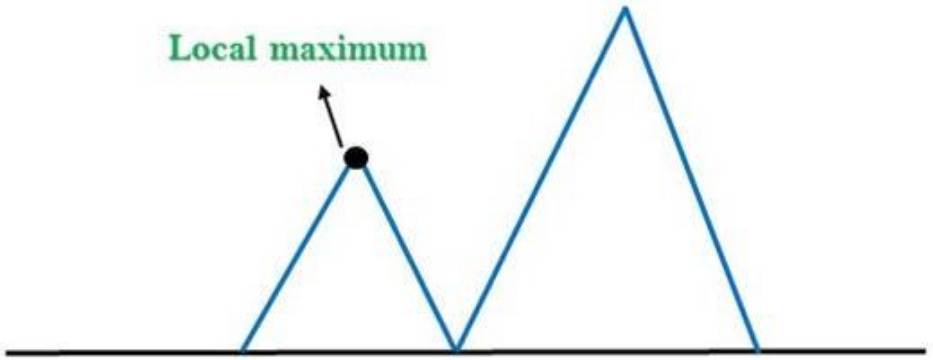
current ← neighbor

- It is the most basic local search technique. At each step the current node is replaced by the best neighbor.

Drawback of Hill Climbing Search

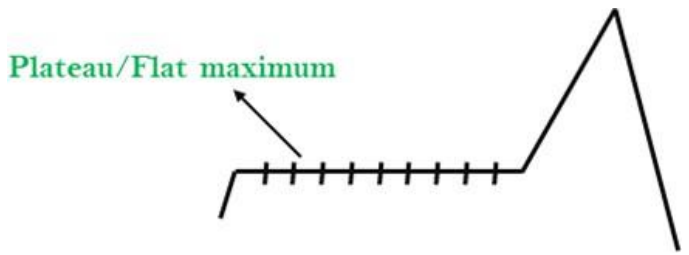
- **Local maxima:** A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.

Solution: Backtrack to some earlier node and try going to different direction.

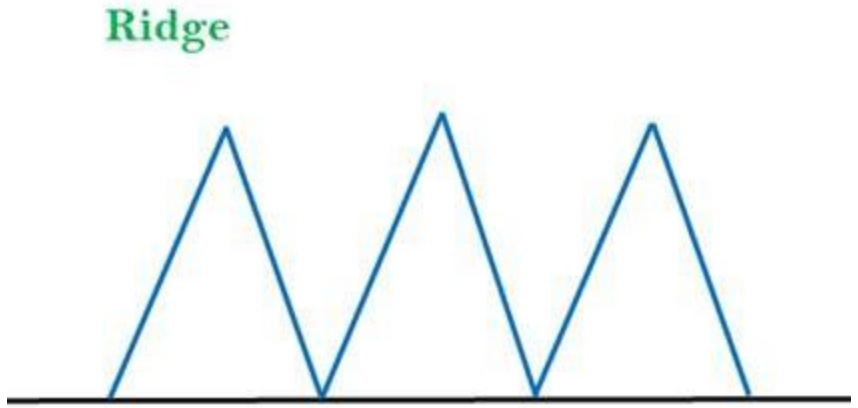


- **Plateaus:** A plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. Search might be unable to find its way of plateau.

Solution: Make a big jump in some direction to try to get a new section of the search space.



- **Ridges:** A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.



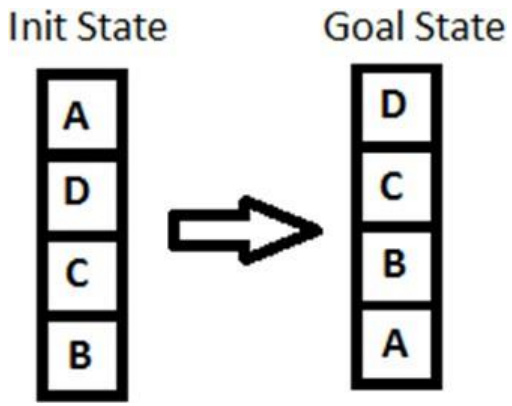
Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.

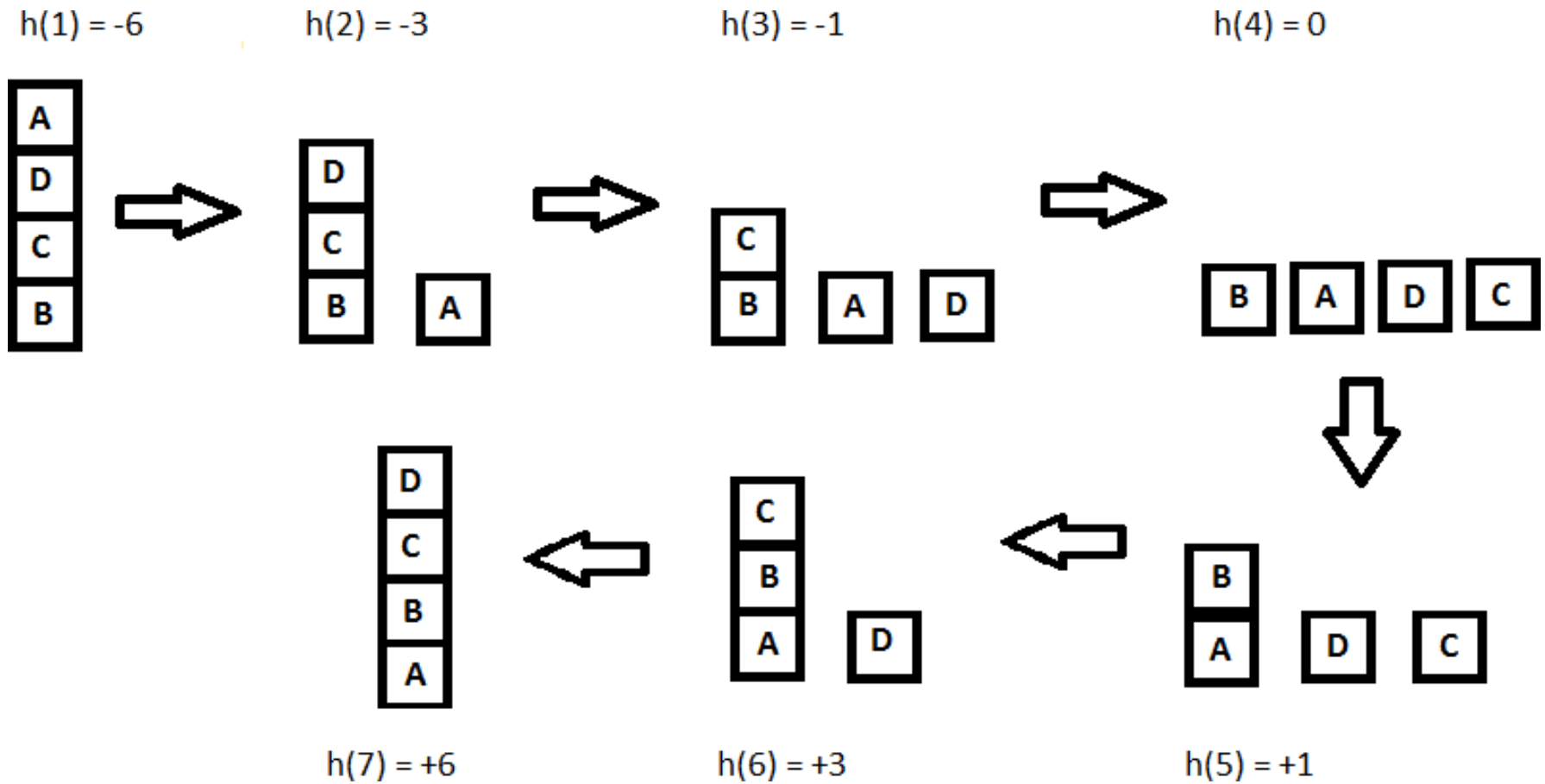
Example:

Key point while solving any hill-climbing problem is to choose an appropriate heuristic function.

Let’s define such function h:

- h(x) = +1 for all the blocks in the support structure if the block is correctly positioned.
- h(x) = -1 for all the blocks that has wrong support structure.




h(1) = -6
h(2) = -3
h(3) = -1
h(4) = 0
A
D
C
B
D
C
B
A
C
B
A
D
B
A
D
C
B
A
D
C
D
C
B
A
C
B
A
D
B
A
D
C
h(7) = +6
h(6) = +3
h(5) = +1

Simulated Annealing search:

- **Annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state.
- **Simulated annealing (SA)** is a probabilistic algorithm that finds approximate solutions to optimization problems.
- Instead of picking the best move, however, it picks a random move.
- If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.
- Probability of downward steps is controlled by **temperature parameter**.

- High temperature implies high chance of trying locally "bad" moves, allowing nondeterministic exploration.
- Low temperature makes search more deterministic (like hill-climbing).
- Temperature begins high and gradually decreases according to a predetermined annealing schedule.
- Initially we are willing to try out lots of possible paths, but over time we gradually settle in on the most promising path.
- If temperature is lowered slowly enough, an optimal solution will be found.
- In practice, this schedule is often too slow and we have to accept suboptimal solutions.

Applications:

- VLSI layout problem
- Factory scheduling
- Travelling salesman problem

Algorithm:

function SIMULATED-ANNEALING(problem, schedule) returns a solution state

Current ← problem. INITIAL

for t = 1 **to** ∞ **do**

T← schedule (t)

if T = 0 **then return** current

next ← a randomly selected successor of current

ΔE ← VALUE(current) – VALUE(next)

if **ΔE** > 0 **then** current ←next

else current ← next only with probability e^(ΔE /T)

- It is a version of stochastic hill climbing where some downhill moves are allowed. The schedule input determines the value of the “temperature” T as a function of time.

Greedy Best first search:

- The Best first search uses the concept of a Priority queue and heuristic search. To search the graph space, the best first search method uses two lists for tracking the traversal.
- An ‘Open’ list which keeps track of the current ‘immediate’ nodes available for traversal and ‘CLOSED’ list that keeps track of the nodes already traversed.
- Uses only a heuristic function h(n) to estimate the direct distance from the current node to the goal. Prioritizes nodes that seem closest to the goal.
- If traveling from city S to city E, Chooses the next city based purely on which seems closest to city E according to the heuristic.
- **Evaluation Function:** $f(n)=h(n)$ focuses only on heuristic distance. *h(n) = 0 for goal state*

Algorithm :

1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the ‘ordered’ OPEN list
3. Repeat the next steps until the GOAL node is reached .

- ✓ If the OPEN list is empty, then EXIT the loop returning ‘False’
- ✓ Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also, capture the information of the parent node.
- ✓ If N is a GOAL node, then move the node to the Closed list and exit the loop returning ‘True’. The solution can be found by backtracking the path.

4. If N is not the GOAL node, expand node N to generate the ‘immediate’ next nodes linked to node N and add all those to the OPEN list
5. Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n).

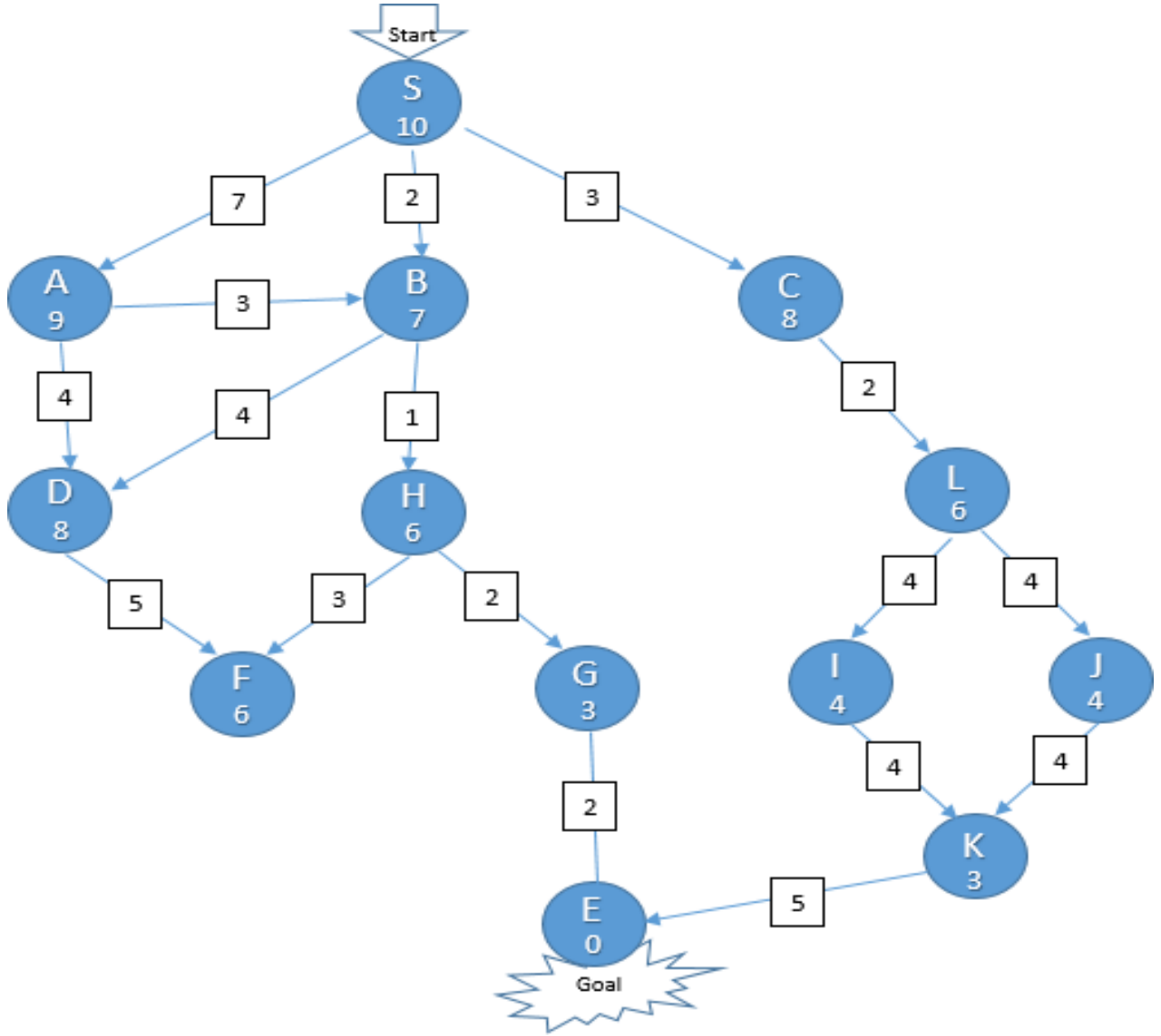
Advantages:

- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

Example:


Start
S
10
7
2
3
A
9
3
B
7
C
8
4
4
1
2
D
8
H
6
L
6
5
3
2
4
4
F
6
G
3
I
4
J
4
4
4
2
K
3
5
E
0
Goal

Greedy BFS with evaluation function f(n) = h(n)

Step 1 - Start by adding the start node (S) to the open list with the path distance as 0

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
S	10		

Repeat the next steps until the OPEN List is empty or the Goal node is moved to the CLOSED list

Step 2 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
A	9	S	
B	7		
C	8		

Step 2 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
B	7	S	
C	8		
A	9		

Step 3 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
C	8	S	
A	9	B	S
D	8		
H	6		

Step 3 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
H	6	S	
C	8	B	S
D	8		
A	9		

Step 4 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
C	8	S	
D	8	B	S
A	9	H	B
F	6		
G	3		

Step 4 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
G	3	S	
F	6	B	S
C	8	H	B
D	8		
A	9		

Step 5 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
E	0	S	
F	6	B	S
C	8	H	B
D	8	G	H
A	9		

Step 5 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
F	6	S	
C	8	B	S
D	8	H	B
A	9	G	H
E	0		

Step 6 (a) - Move the first node in the OPEN list to the CLOSED list and expand it's immediate successors by adding them to the OPEN list

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
F	6	S	
C	8	B	S
D	8	H	B
A	9	G	H
		E	G

EXIT returning 'True' as the Goal node (E) is moved to the CLOSED list. Backtrack the closed list to get the optimal path (E --> G --> H --> B --> S)

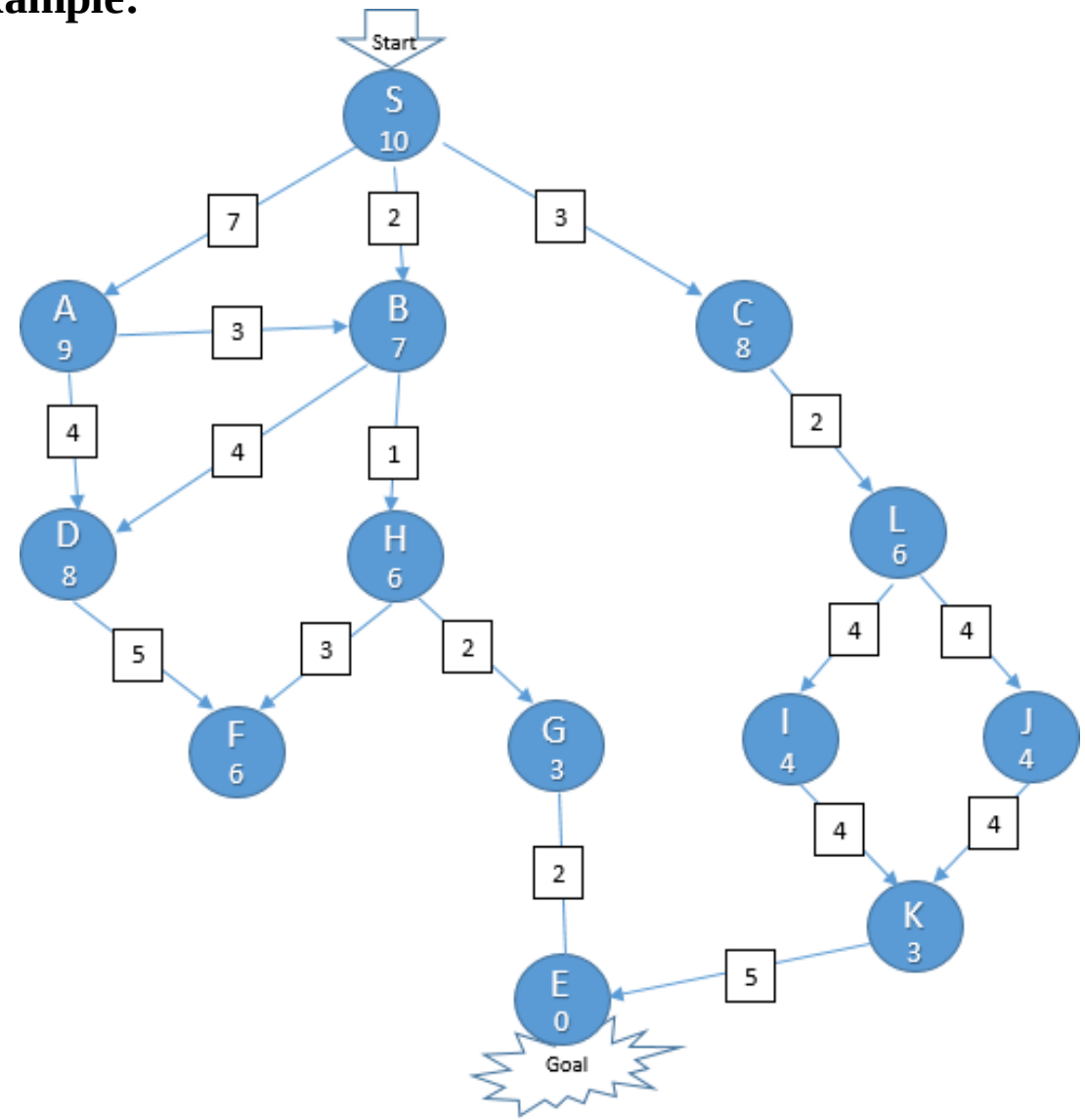
A* Search :

- A* Algorithm is one of the best and popular techniques used for path finding and graph traversals.
- A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.
- It is essentially a best first search algorithm.
- Working of A* Algorithm:
  - It maintains a tree of paths originating at the start node.
  - It extends those paths one edge at a time.
  - It continues until its termination criterion is satisfied.
- A* Algorithm extends the path that minimizes the following function- **f(n) = g(n) + h(n)**

  Here,

  g(n) = cost so far to reach n
  h(n) = estimated cost from n to goal
  f(n) = estimated total cost of path through n to goal

Example:


Start
S
10
7
2
3
A
9
3
B
7
C
8
4
4
1
2
D
8
H
6
L
6
5
3
2
4
4
F
6
G
3
I
4
J
4
4
4
2
K
3
5
E
0
Goal

A* BFS with evaluation function f(n) = h(n) + g(n)

Step 1 - Start by adding the start node (S) to the open list with the path distance as 0

OPEN				CLOSED	
Node	g(n)	h(n)	f(n)	Node	Parent Node
S	0	10	10		

Repeat the next steps until the OPEN List is empty or the Goal node is moved to the CLOSED list

Step 2 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN				CLOSED	
Node	g(n)	h(n)	f(n)	Node	Parent Node
B	2	7	9	S	
C	3	8	11		
A	7	9	16		

Step 2 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN				CLOSED	
Node	g(n)	h(n)	f(n)	Node	Parent Node
A	7	9	16	S	
B	2	7	9		
C	3	8	11		

Step 3 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN				CLOSED	
Node	g(n)	h(n)	f(n)	Node	Parent Node
C	3	8	11	S	
A	7	9	16	B	S
D	6	8	14		
H	3	6	9		

Step 3 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN				CLOSED	
Node	g(n)	h(n)	f(n)	Node	Parent Node
H	3	6	9	S	
C	3	8	11	B	S
D	6	8	14		
A	7	9	16		

Step 4 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN				CLOSED	
Node	g(n)	h(n)	f(n)	Node	Parent Node
G	5	3	8	S	
C	3	8	11	B	S
F	6	6	12	H	B
D	6	8	14		
A	7	9	16		

Step 4 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN				CLOSED	
Node	g(n)	h(n)	f(n)	Node	Parent Node
C	3	8	11	S	
D	6	8	14	B	S
A	7	9	16	H	B
F	6	6	12		
G	5	3	8		

Step 5 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN				CLOSED	
Node	g(n)	h(n)	f(n)	Node	Parent Node
C	3	8	11	S	
F	6	6	12	B	S
D	6	8	14	H	B
A	7	9	16	G	H
E	7	0	7		

Step 5 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN				CLOSED	
Node	g(n)	h(n)	f(n)	Node	Parent Node
E	7	0	7	S	
C	3	8	11	B	S
F	6	6	12	H	B
D	6	8	14	G	H
A	7	9	16		

Step 6 (a) - Move the first node in the OPEN list to the CLOSED list and expand it's immediate successors by adding them to the OPEN list

OPEN				CLOSED	
Node	g(n)	h(n)	f(n)	Node	Parent Node
C	3	8	11	S	
F	6	6	12	B	S
D	6	8	14	H	B
A	7	9	16	G	H
				E	G

EXIT returning 'True' as the Goal node (E) is moved to the CLOSED list. Backtrack the closed list to get the optimal path (E --> G --> H --> B --> S)

Adversarial search and games

- **Competitive environments,** in which two or more agents have conflicting goals are trying to explore the same search space for the solution**,** are called **adversarial searches,** often known as **Games**.
- The environment with more than one agent is termed as **multi-agent environment**, in which 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.
- In AI, games means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which utility values at the end of the game are always equal and opposite.
    - Example: **If first player wins, the other player necessarily loses**

Applications/ Role of Adversarial Search in AI:

1. Game Playing: Adversarial search is extensively used in AI for game playing. Classic examples include chess, checkers, tic-tac-toe, and Go. AI programs like Deep Blue and AlphaGo use adversarial search techniques to compete at a high level against human players.

2. Decision Making in Competitive Environments: Beyond board games, adversarial search is used in real-world applications where decisions involve competition, such as economic modeling, military strategy simulations, and negotiation systems.

- A game can be formally defined as a kind of search problem with the following components:
  - **Initial State**: This includes the board position and the starting player's turn.
  - **Players**: The players involved in the game, often referred to as MAX and MIN, where MAX aims to maximize the score and MIN aims to minimize it.
  - **Successor Function**: This function defines the set of legal moves a player can make and the resulting state of the game after each move.
  - **Terminal Test**: A test to determine if the game has reached an end state (also called a terminal state). For example, in chess, this would be checkmate or stalemate.
  - **Utility Function**: Also known as the payoff function, this function assigns a numerical value to each terminal state. In zero-sum games, one player's gain is another player's loss.
  - **Evaluation Function** (for non-terminal states in practical scenarios): This heuristic function estimates the utility of a given state, used when the game tree is too large to be searched exhaustively.

Types of Games in AI:

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

- **Perfect information:** A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
- **Imperfect information:** If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.

- **Deterministic games:** Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
- **Non-deterministic games:** Non-deterministic are those games which have various unpredictable events and has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games. Example: Backgammon, Monopoly, Poker, etc.

Minimax Algorithm:

- It is a **recursive algorithm** for choosing the next move in a n-player game, **usually a two player game**
- A value is associated with each position or state of the game
- The value is computed by means of a **position evaluation function** and **it indicates how good it would be for a player to reach the position.**
- The player then makes the move that maximizes the minimum value of the position from the opponents possible moves called **maximizing player** and other player minimize the maximum value of the position called **minimizing player**.
- It is applied in two players games like chess, checkers, tic-tac-toe, etc.

- The top node is the initial state, and MAX moves first, placing an X in an empty square.
- We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

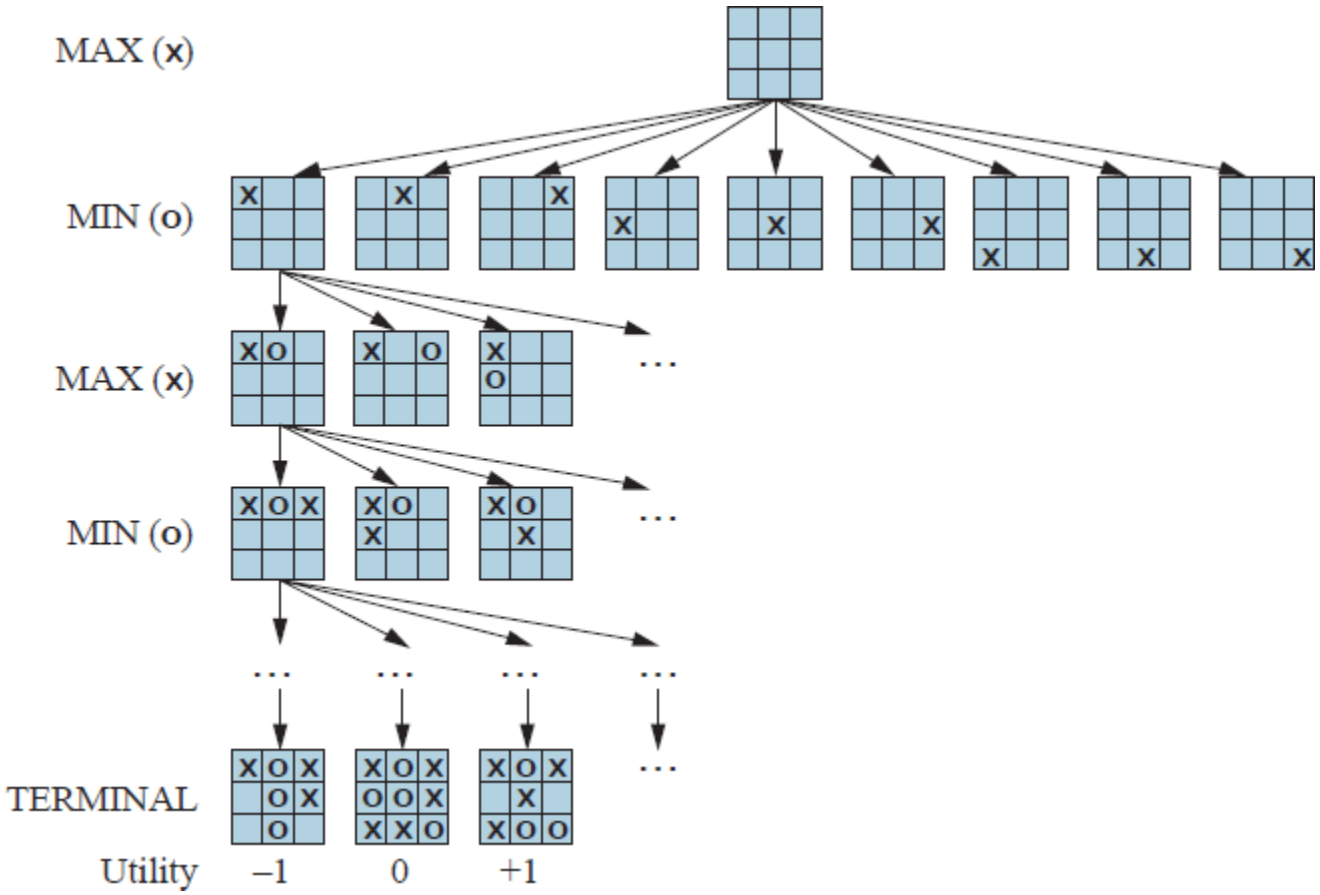


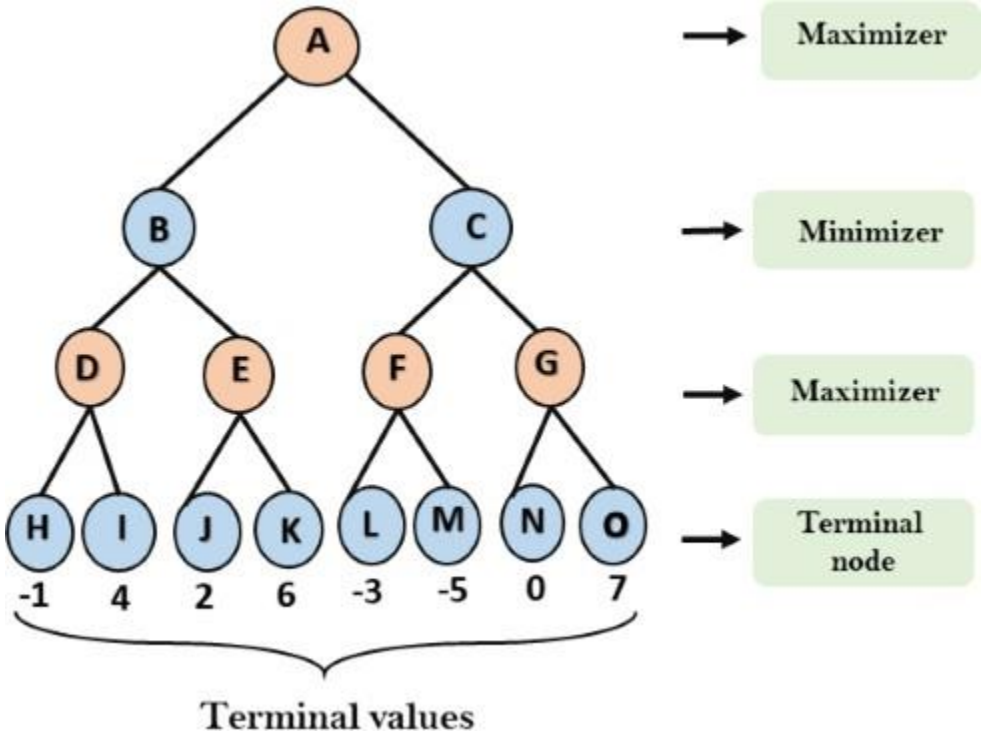
Figure : A (partial) game tree for the game of tic-tac-toe.

Pseudo-code for Min-Max Algorithm

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node
if MaximizingPlayer then  // for Maximizer Player
maxEva= -infinity
for each child of node do
eva= minimax(child, depth-1, false)
maxEva= max(maxEva,eva) //gives Maximum of the values
return maxEva
else    // for Minimizer player
minEva= +infinity
for each child of node do
eva= minimax(child, depth-1, true)
minEva= min(minEva, eva) //gives minimum of the values
return minEva
```

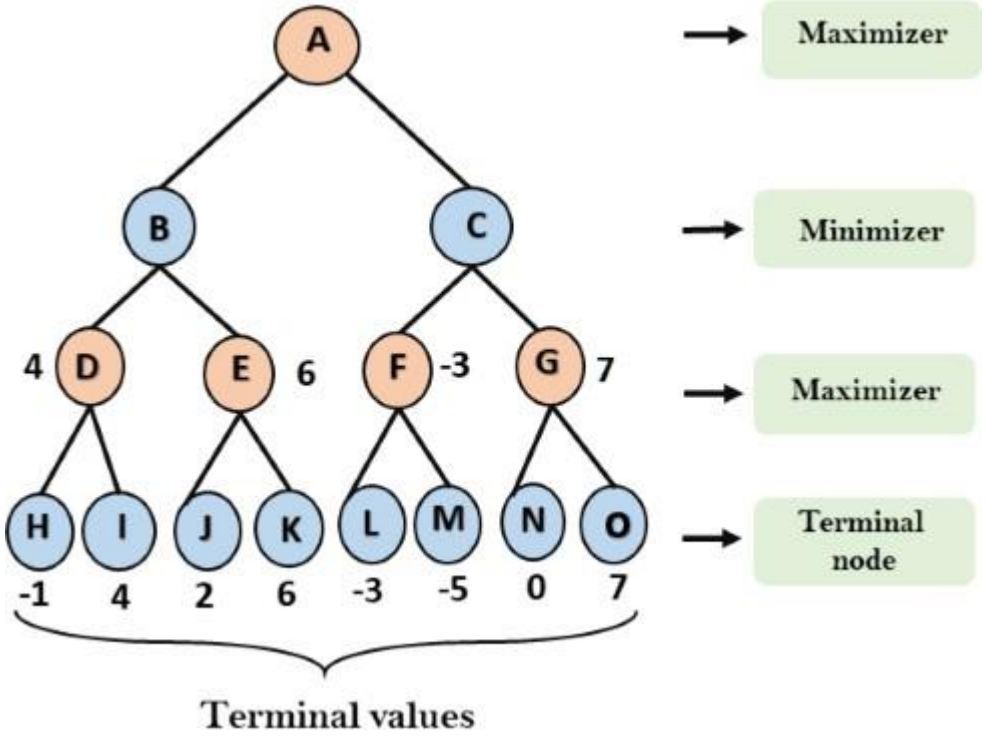
Example:

Step 1:



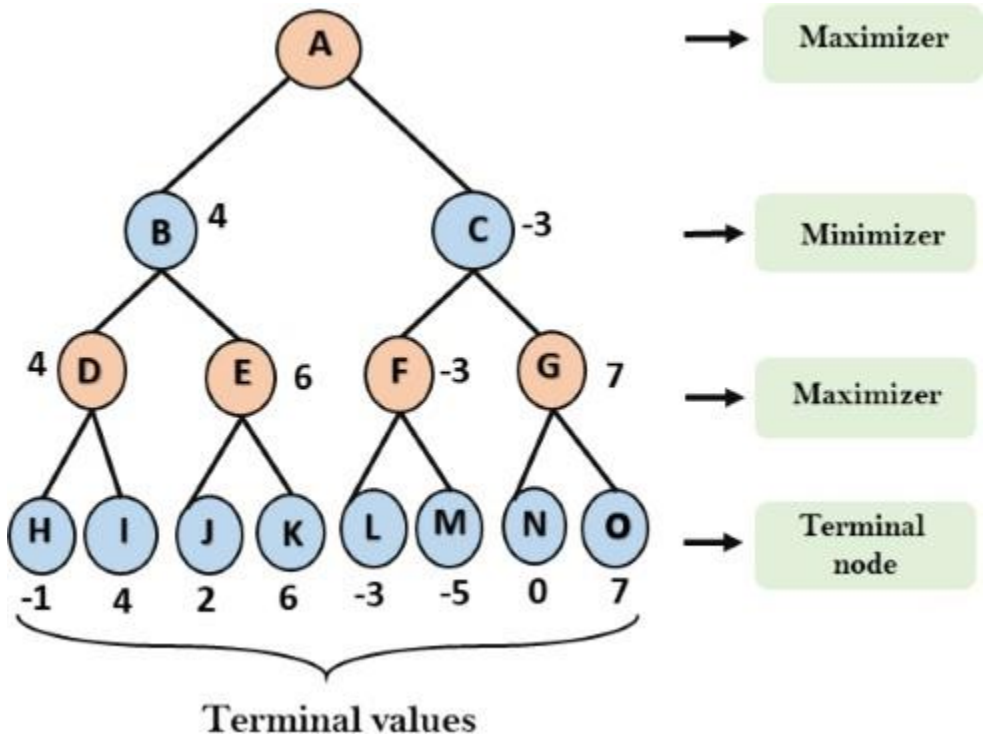
Step 2: Maximizer, its initial value is -∞, , so it will compare all nodes value with -∞.

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



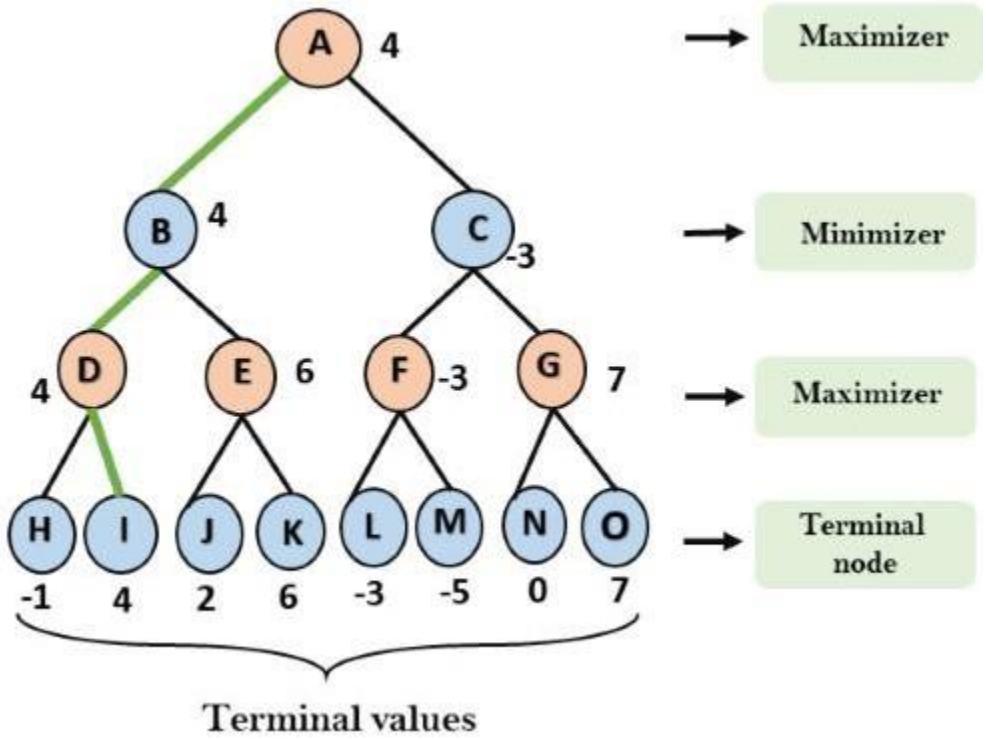
Step 3: minimizer, so it will compare all nodes value with +∞

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



Step 4:

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



Properties of Mini-Max algorithm

- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

Alpha–Beta Pruning:

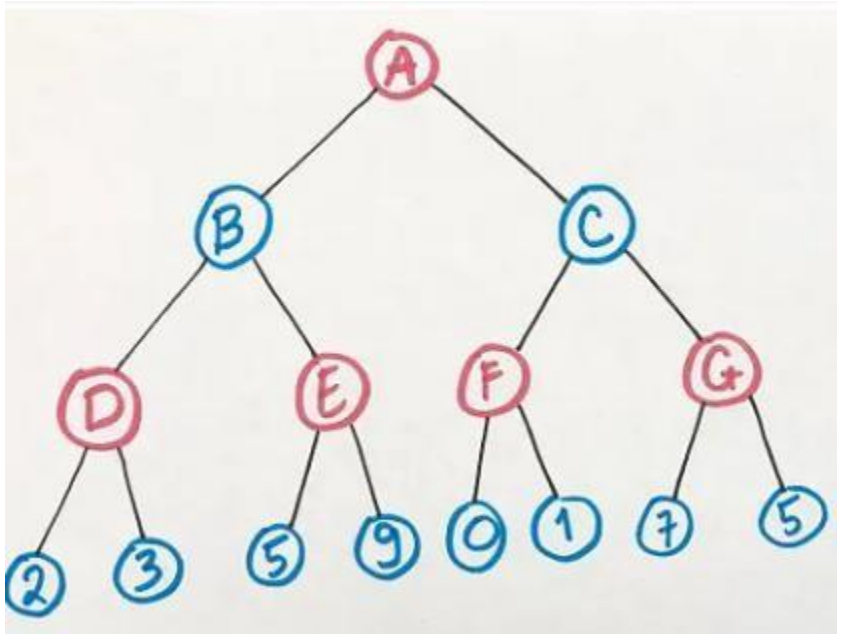
- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- The number of game states is exponential in the depth of the tree. No algorithm can completely eliminate the exponent, but we can sometimes cut it in half, computing the correct minimax decision without examining every state by **pruning** large parts of the tree that make no difference to the outcome. The particular technique we examine is called **alpha–beta pruning**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:

  Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.

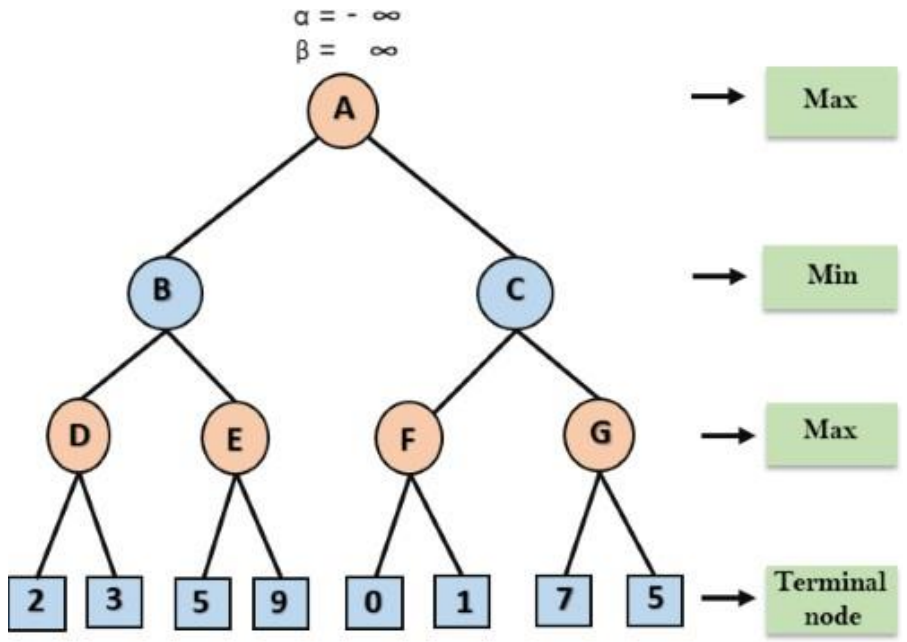
Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

Example:

Let's take an example of two-player search tree to understand the working of Alpha-beta pruning.

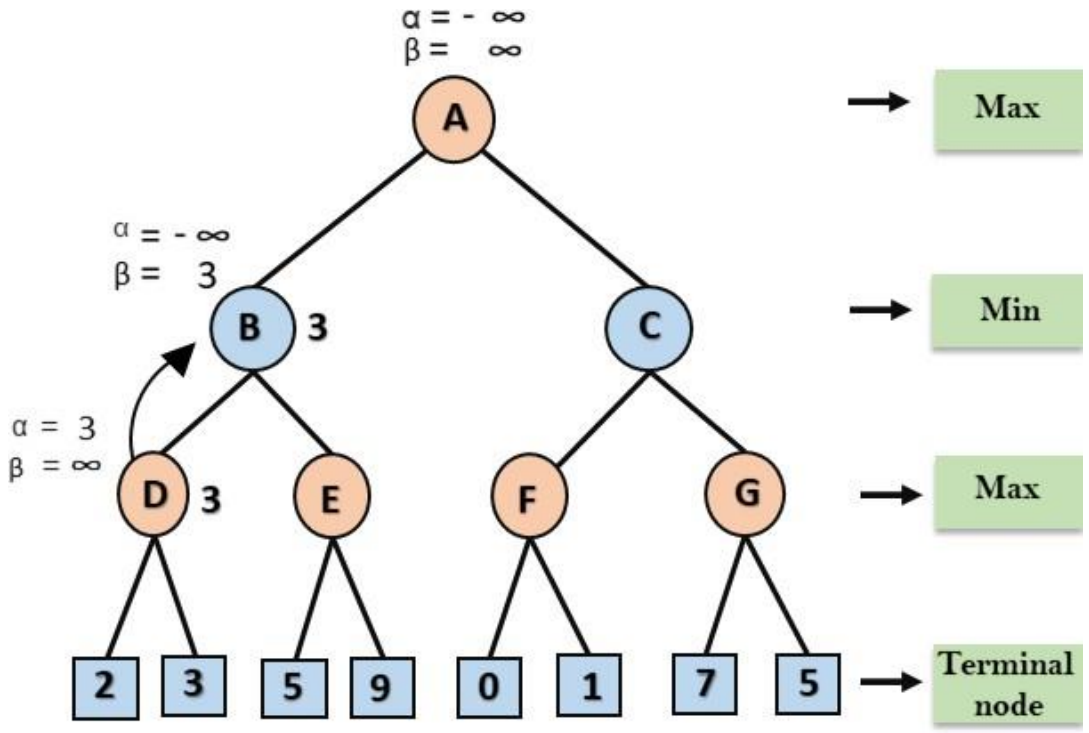


Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.



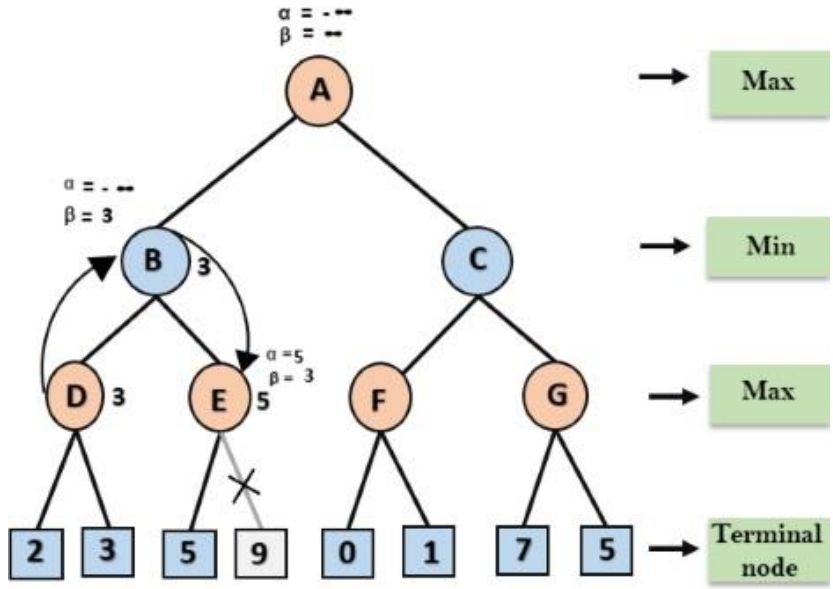
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

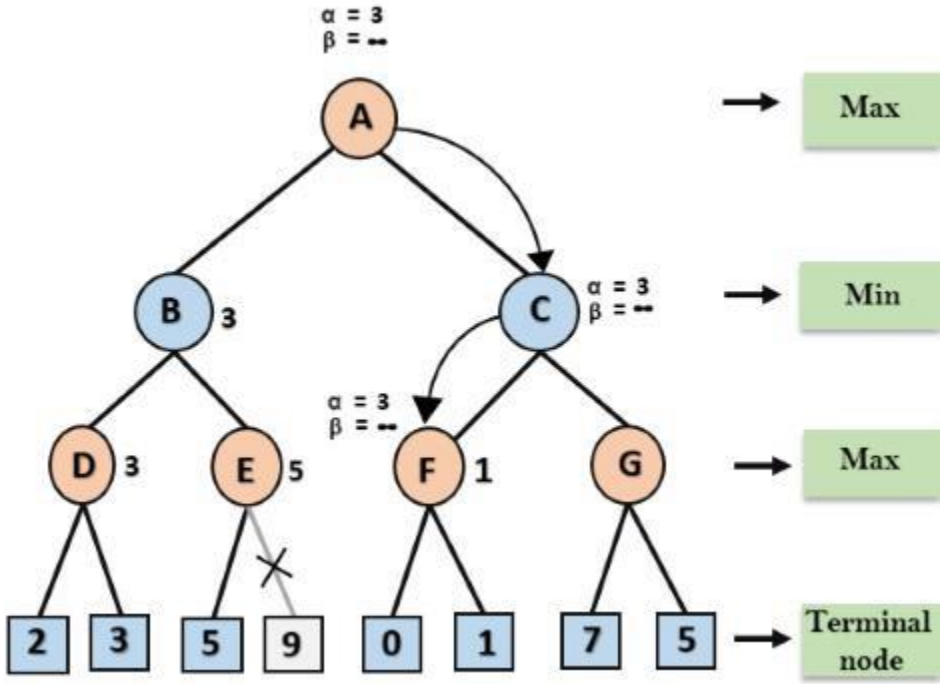
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α >=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



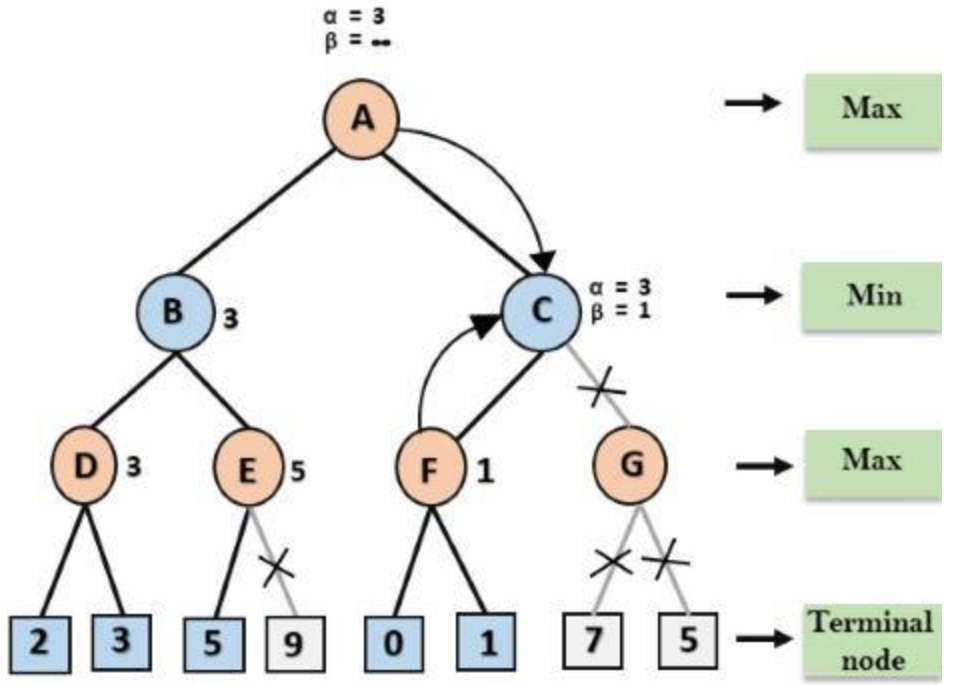
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

At node C, α=3 and β= +∞, and the same values will be passed on to node F.

Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.

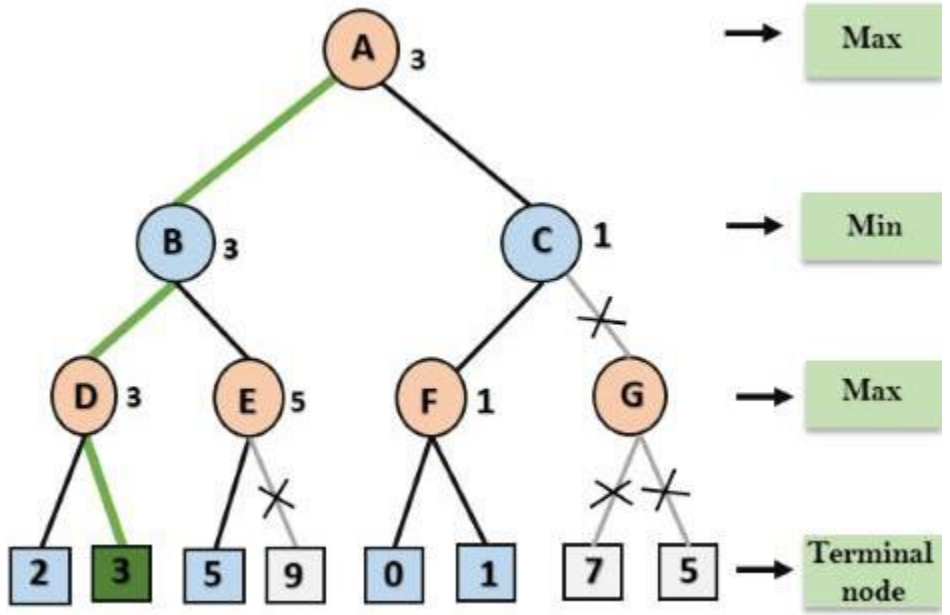


Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3.

Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



- **Complete:** Yes, for finite trees.
- **Optimal:** Yes, if the game tree has perfect information.
- **Time Complexity:** Best case $O(b^{m/2})$, Worst case $O(b^{m})$.
- **Space Complexity:** $O(m)$.

Constraint Satisfaction Problems(CSP)

- We break open the black box by using a **factored representation** for each state: a set of **variables**, each of which has a **value**. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or **CSP**.
- CSP search algorithms take advantage of the structure of states and use general rather than domain-specific heuristics to enable the solution of complex problems.
- The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.
- CSPs have the additional advantage that the actions and transition model can be deduced from the problem description

- A constraint satisfaction problem consists of three components, X,D, and C:
  - X is a set of variables, {X1, . . . ,Xn}.
  - D is a set of domains, {D1, . . . ,Dn}, one for each variable.
  - C is a set of constraints that specify allowable combinations of values.
- For example, if X1 and X2 both have the domain {1,2,3}, then the constraint saying that X1 must be greater than X2 can be written as ((X1,X2),{(3,1), (3,2), (2,1)}) or as ((X1,X2),X1 > X2).

Applications of CSP

- **Scheduling Problems**: Assigning time slots to exams or jobs without conflicts.
- **Game Solving**: Solving puzzles like Sudoku, crosswords, or N-Queens.
- **Resource Allocation**: Assigning limited resources (e.g., staff, machines) to tasks.
- **Natural Language Processing**: Parsing sentences where grammatical rules act as constraints.
- **Robotics**: Path planning under constraints.

Example problem: Map coloring

- Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories (Figure below(a)). We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions:

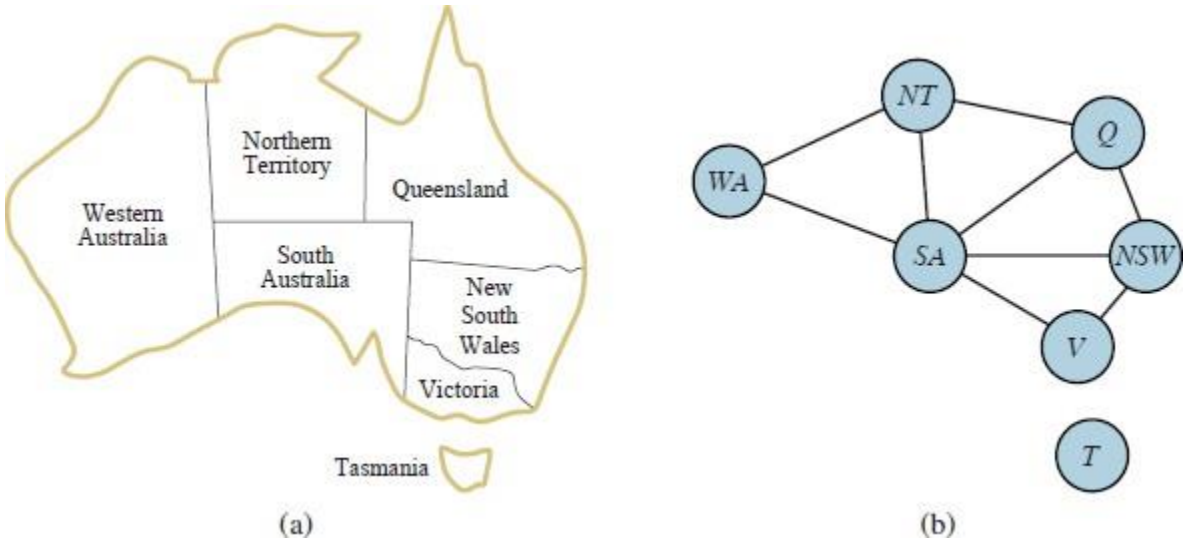
X = {WA, NT, Q, NSW, V, SA, T}.

The domain of every variable is the set Di ={red, green, blue}.

Since there are nine places where regions border, there are nine constraints: C = {SA ≠ WA, SA ≠ NT, SA ≠ Q,SA ≠ NSW,SA ≠ V, WA ≠ NT,NT ≠ Q,Q ≠ NSW,NSW ≠ V}.

Here we are using abbreviations; SA ≠ WA is a shortcut for ((SA,WA),SA ≠ WA), where SA ≠ WA can be fully enumerated in turn as {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}.

- There are many possible solutions to this problem, such as {WA=red, NT=green, Q=red, NSW=green, V =red, SA=blue, T =red }.
- It can be helpful to visualize a CSP as a **constraint graph**, as shown in Figure below (b). The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.



- Figure (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color.
- Figure (b) The map-coloring problem represented as a constraint graph.

Crypt-Arithmetic Problem:

- To solving cryptarithmetic problems in the realm of Artificial Intelligence, utilizing the **Constraint Satisfaction Problem (CSP)** methodology.
- A **Crypt-arithmetic puzzle**, also known as a cryptogram, is a type of mathematical puzzle in which we assign digits to alphabetical letters or symbols. The end goal is to find the unique digit assignment to each letter so that the given mathematical operation holds true.
- In simpler words, the crypt-arithmetic problem deals with the converting of the message from the readable plain text to the non-readable cipher text.
- The constraints which this problem follows during the conversion is as follows:
    - A number 0-9 is assigned to a particular alphabet.
    - Each different alphabet has a unique number.
    - All the same, alphabets have the same numbers.
    - The numbers should satisfy all the operations that any normal number does.

Example:

S E N D

+M O R E

- - - - - - - - -

M O N E Y

Here is a possible solution for SEND + MORE = MONEY:

✓ M = 1,O = 0, N = 6, E = 5, D = 7, R = 8, S = 9, Y = 2

Therefore, the values are: SEND (9567) + MORE (1085) = MONEY (10652).

```
  9 5 6 7
+ 1 0 8 5
 - - - - -
1 0 6 5 2
```

```
  B A S E
  B A L L
 --------
G A M E S
```

```
  2 4 6 1
  2 4 5 5
 ---------
0 4 9 1 6
```

```
  T W O
+ T W O
-------
F O U R
```

```
  928
+ 928
------
 1856
```

```
  FOUR
+FOUR
-----
EIGHT
```

```
  9235
+ 9235
------
 18470
```

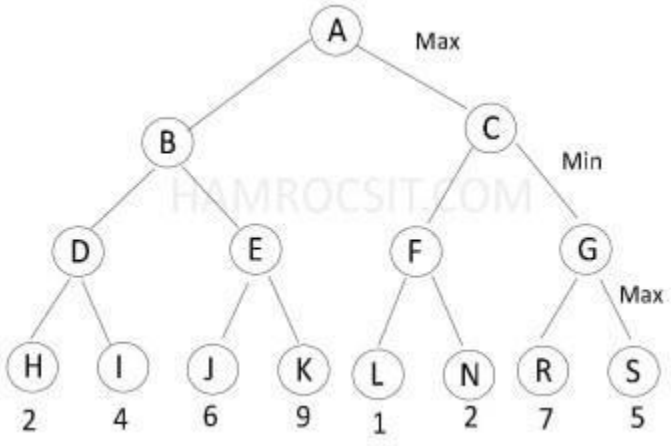
Questions

1. BASE + BALL = GAMES
2. FORTY + TEN + TEN = SIXTY
3. CROSS + ROADS = DANGER
4. APPLE + ORANGE = FRUITS
5. BRAIN + STORM = IDEAS
6. TWENTY + TWENTY + TEN = FIFTY
7. READ + WRITE = SKILL
8. FOUR + FIVE + SIX = FIFTEEN
9. EARTH + SEA + SKY = NATURE
11. SQUARE + ROOT = NUMBER
12. CHAIR + TABLE = FURNITURE
13. TRAIN + TRACK = JOURNEY
14. EIGHT + EIGHT = SIXTEEN
15. ALPHA + BETA = GAMMA
16. EIGHT + SEVEN = FIFTEEN

Assignments

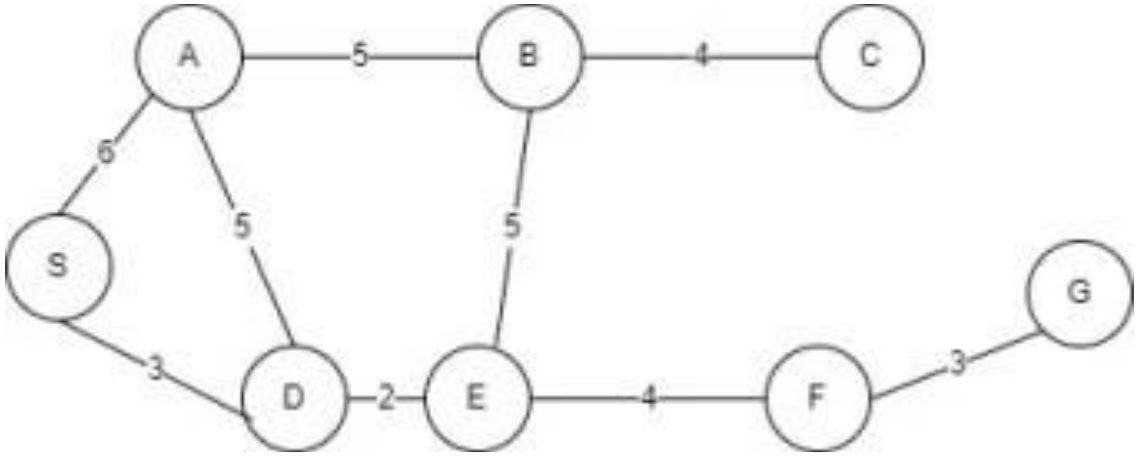
Q1. Explain different types of Hill Climbing Algorithms in detail.

Q2. Given following search space, determine if these exists any alpha and beta cutoffs.



Q3. What do you mean by heuristic search? Given following state space representation, show how greedy best first and A* search is used to find the goal state.

S is the start state and G is the goal state. The heuristics of the states are:
h(S) = 12, h(A)=8, h(D)=9, h(B)=7, h(E)= 4, h(C)=5, h(F)=2, h(G)=0.



Q4. How informed search are different than uniformed? Given following state space, illustrate how depth limited search and iterative depending search works? Use your own assumption for depth search.

Hence, A is start and K is goal.