Search Algorithms in AI
AI

1. Searching for Solutions(CO2)

- ***Artificial Intelligence*** is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.
- A search problem consists of:
  - **A State Space.** Set of all possible states where you can be.
  - **A Start State.** The state from where the search begins.
  - **A Goal Test.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

1.1 Search Algorithm

- Search algorithms help find the correct sequence of actions in a search space, to reach a goal state. The sequence of actions might be:
  - Sequence in which cities are to be visited to travel from a source to a destination under a given cost function (shortest path, cheapest fare etc.)
  - Sequence in which an agent should play moves in a game (chess, tic tac toe, pacman etc.) to win a board game
  - Sequence in which a robot arm should solder components on a PCB under a given cost function (e.g. shortest time)
- They are mostly implemented as graphs, where the node once visited is not expanded agAI(KCS-071)n if revisited during traversal (as agAI(KCS-071)nst tree, where there is a possibility of a node getting expanded repeatedly if revisited during traversal - leading to recursion).

1.2 Search Algorithm Terminologies

- **Search:** Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three mAI(KCS-071)n factors:
  - **Search Space:** Search space represents a set of possible solutions, which a system may have.
  - **Start State:** It is a state from where agent begins **the search**.
  - **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 search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the avAI(KCS-071)lable actions to the agent.

- **Transition model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

1.3 Properties of Search Algorithms

- **Completeness:** A search algorithm is sAI(KCS-071)d to be complete if it guarantees to return a solution if at least any solution exists for any random input.
- **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is sAI(KCS-071)d to be an optimal solution.
- **Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.
- **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

1.4 Types of Search Algorithms

- There are two types of search algorithms.
  - ***Uninformed*** : Uninformed search algorithms do not have any domAI(KCS-071)n knowledge. It works in a brute force manner and hence also called brute force algorithms. It has no knowledge about how far the goal node is, it only knows the way to traverse and to distinguish between a leaf node and goal node. It examines every node without any prior knowledge hence also called blind search algorithms
  - ***Informed*** : Informed search algorithms have domAI(KCS-071)n knowledge. It contAI(KCS-071)ns the problem description as well as extra information like how far is the goal node. It is also called the Heuristic search algorithm. It might not give the optimal solution always but it will definitely give a good solution in a reasonable time. It can solve complex problems more easily than uninformed.

Topic Objective

- **Topic**:

  Types of search algorithms

- **Topic Objective**:

  To Learn about informed and uninformed search algorithms

  To study the applications of all search algorithms

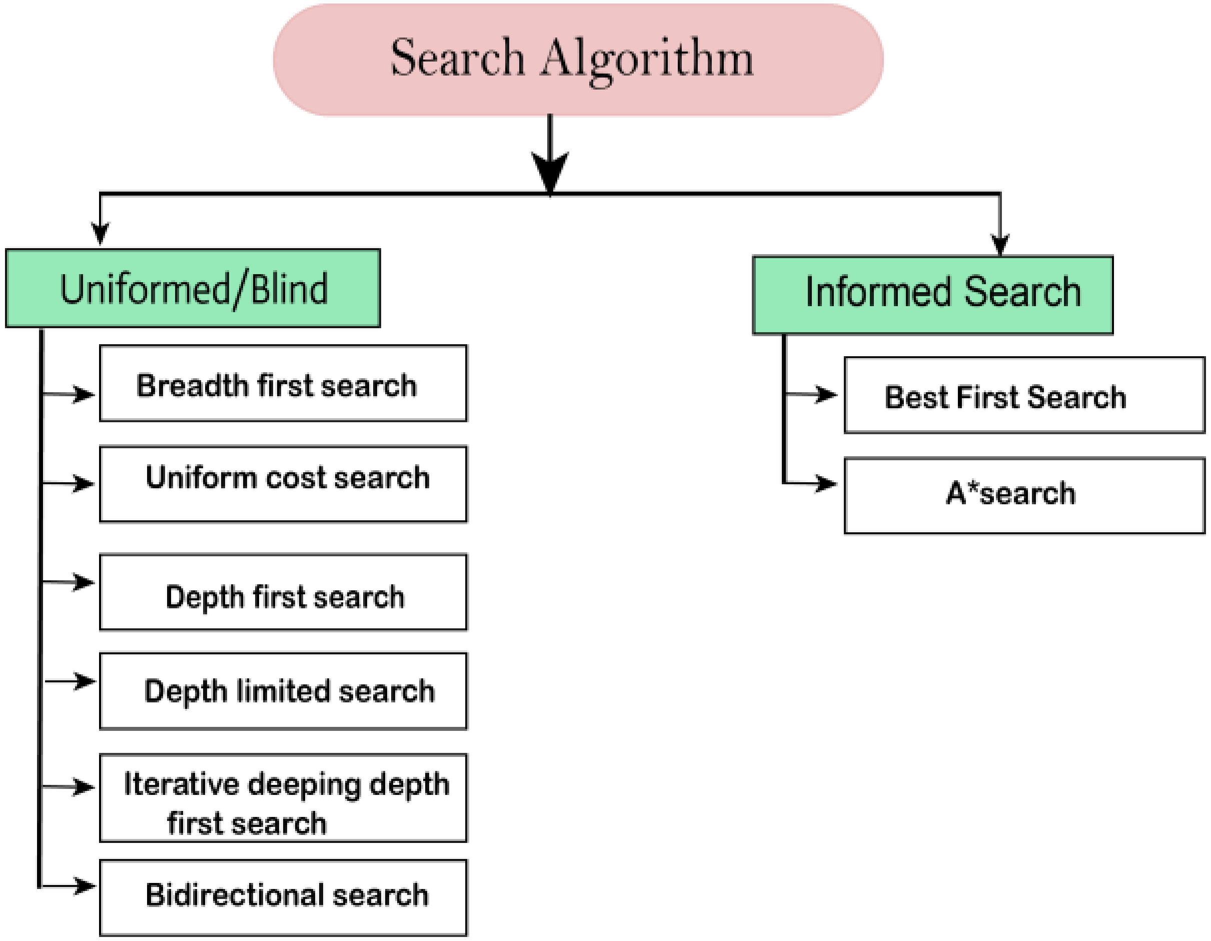
Prerequisites & Recap

- **Prerequisites:**
  - Understanding of basic concepts of artificial intelligence
  - Good knowledge of Python, LISP or PROLOG.
  - Good foundation in calculus, linear algebra, statistics and discrete mathematics.

- **Recap:**
  - Learned the meaning of a search Algorithm
  - Discussed types & Properties of search algorithms

2. Types of Search Algorithms(CO2)

- Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



2.1 Breadth-first Search

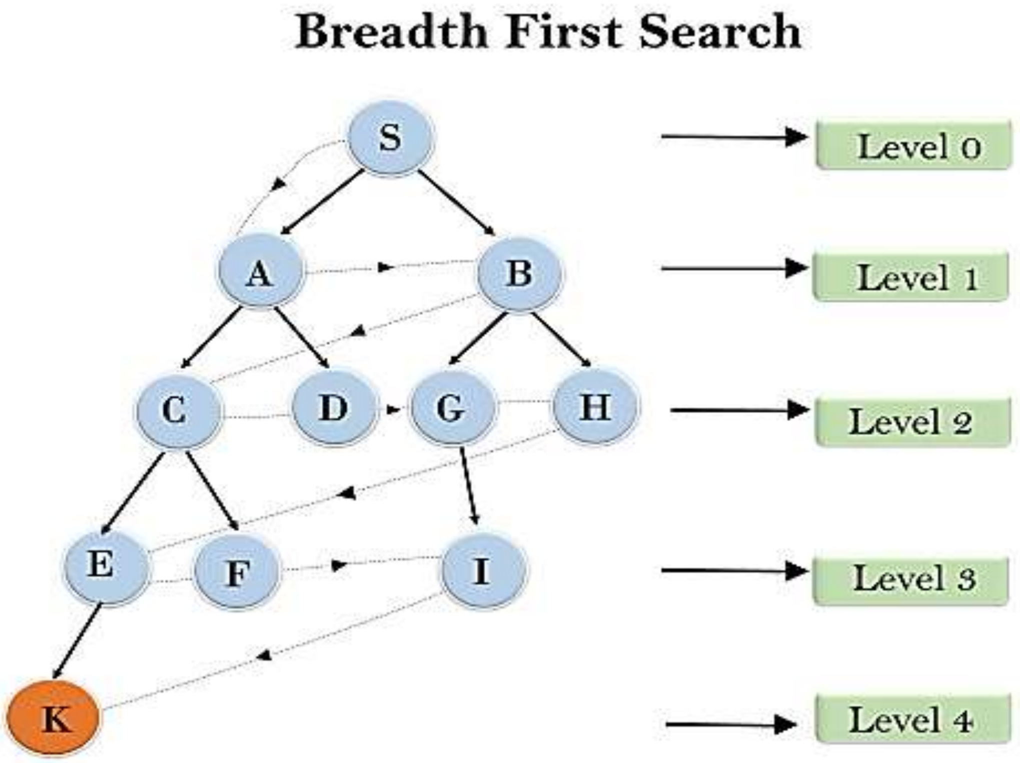
- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages/Disadvantages of BFS

- **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: BFS

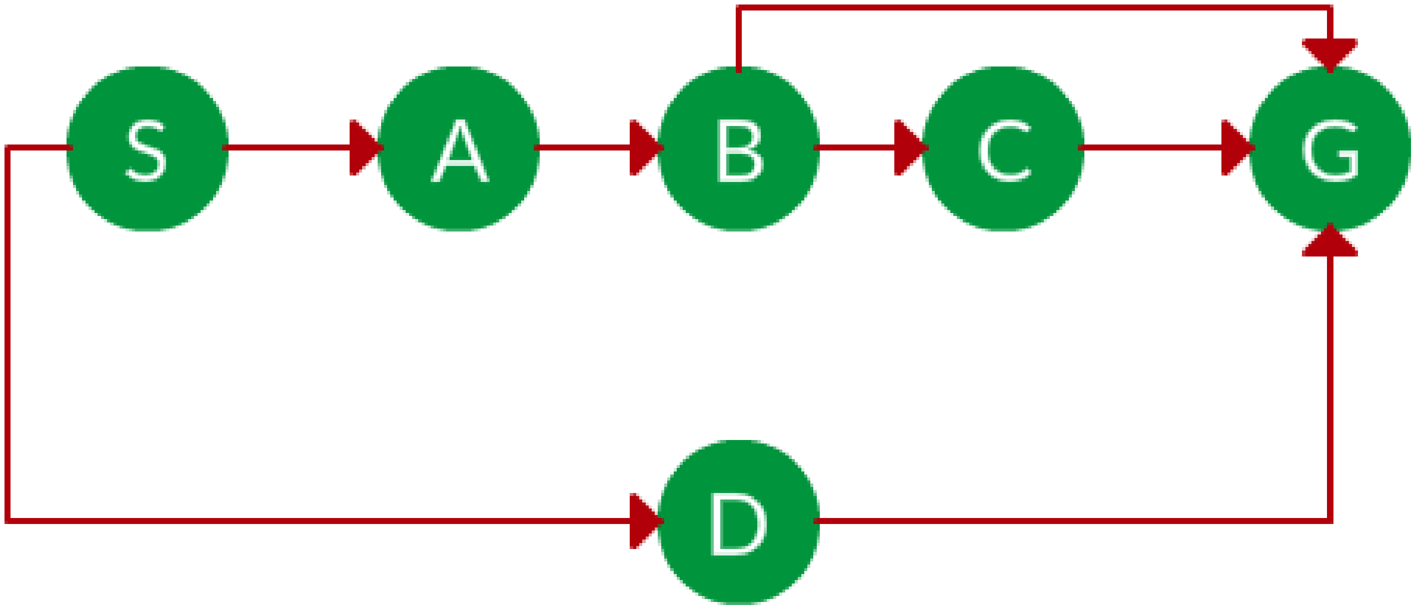
- In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:
- S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



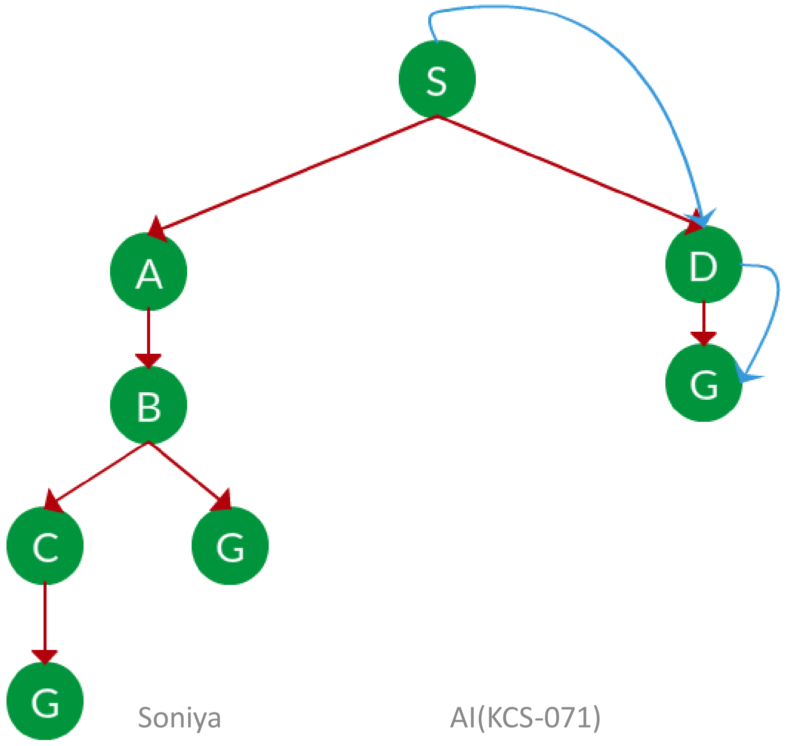
Properties of BFS

- **Time Complexity:** Time Complexity of BFS algorithm can be obtAI(KCS-071)ned 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.
- **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.

- **Question.** Which solution would BFS find to move from node S to node G if run on the graph below?



- **Solution.** The equivalent search tree for the above graph is as follows. As BFS traverses the tree “shallowest node first”, it would always pick the shallower branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown in blue arrows.



- **Path:** S -> D -> G
- Let *S* = the depth of the shallowest solution.
  n^i = number of nodes in level *i*.
- **Time complexity:** Equivalent to the number of nodes traversed in BFS until the shallowest solution.

$$T(n) = 1 + n^2 + n^3 + ... + n^s = O(n^s)$$

- **Space complexity:** Equivalent to how large can the fringe get.

$$S(n) = O(n^s)$$

2.2 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.

Advantages/Disadvantages of DFS

- **Advantages:**
  - 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).
- **Disadvantages:**
  - 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.

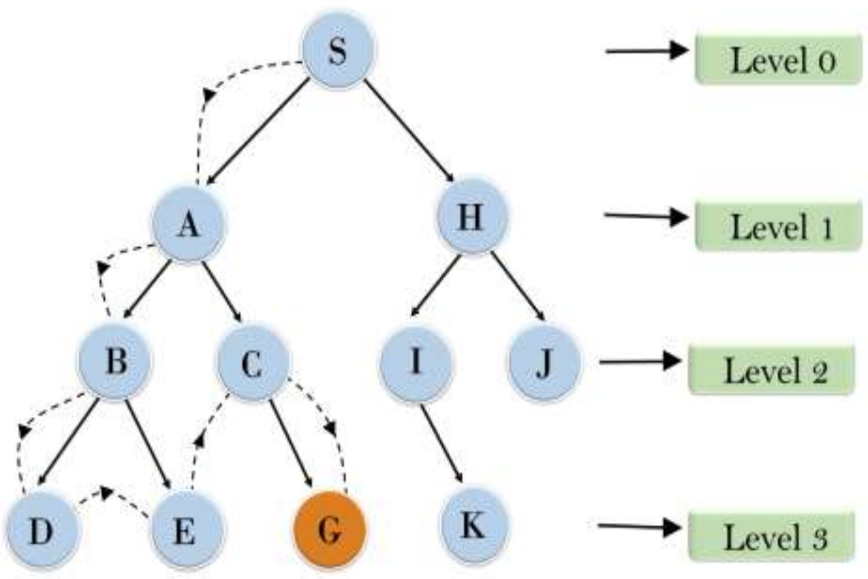
Example: DFS

- 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 & 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.

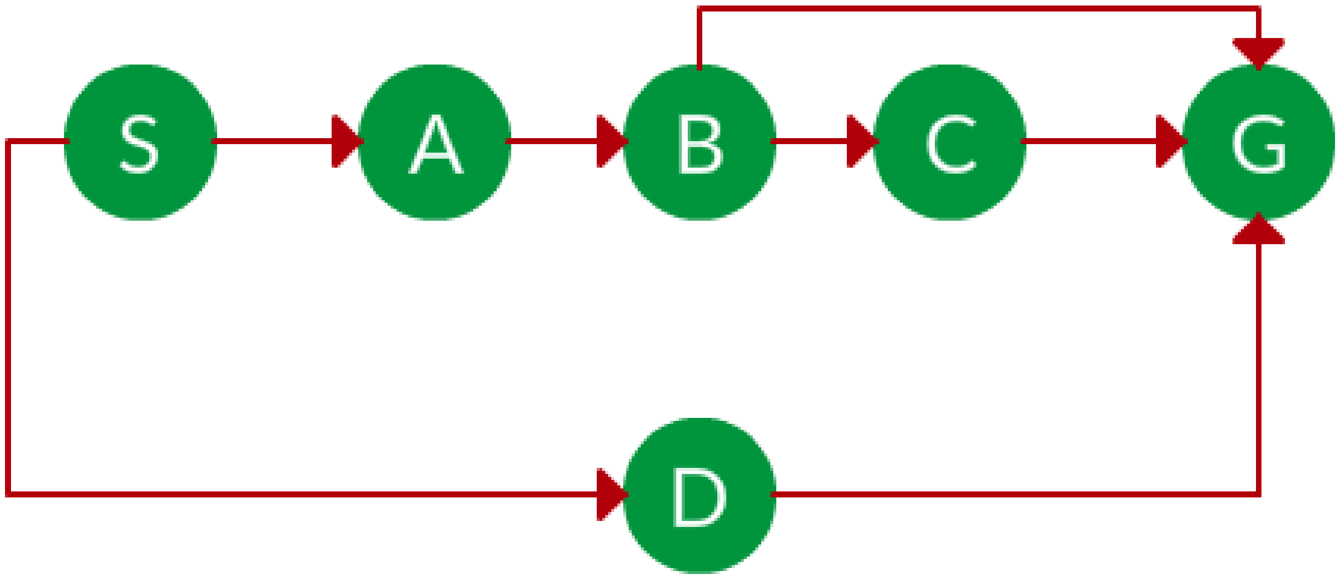
Depth First Search



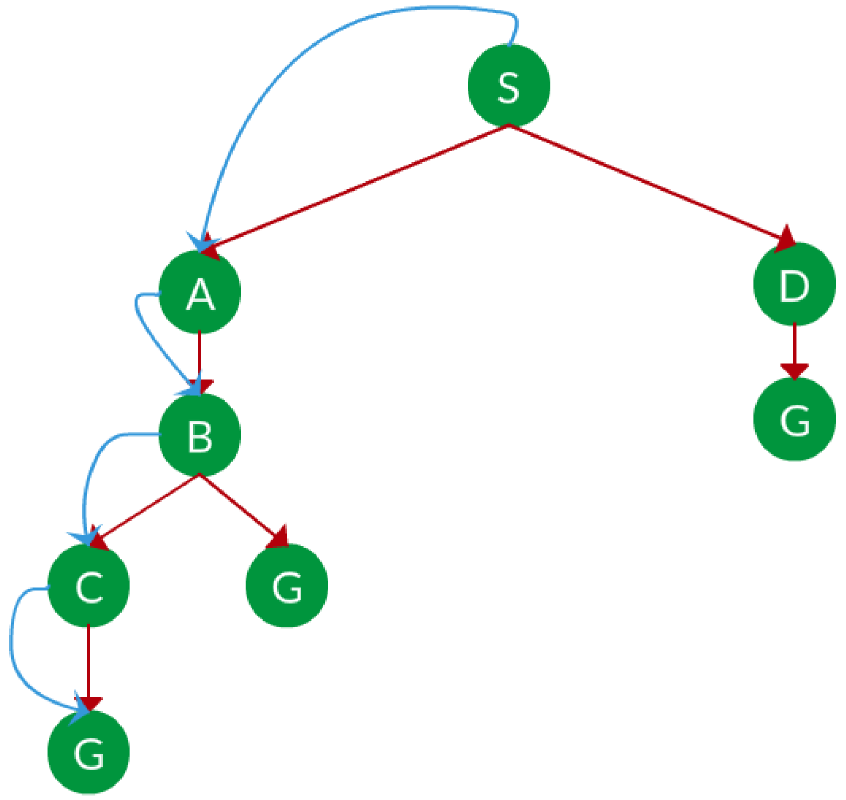
Properties of DFS

- **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.

- **Question.** Which solution would DFS find to move from node S to node G if run on the graph below?



- The equivalent search tree for the above graph is as follows. As DFS traverses the tree "deepest node first", it would always pick the deeper branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown in blue arrows.



- **Path:** S -> A -> B -> C -> G
- Let *d* = the depth of the search tree = number of levels of the search tree.
  n^i = number of nodes in level *i*.
- **Time complexity:** Equivalent to the number of nodes traversed in DFS.

$$T(n) = 1 + n^2 + n^3 + ... + n^d = O(n^d)$$

- **Space complexity:** Equivalent to how large can the fringe get.

$$S(n) = O(n \times d)$$

Difference between BFS and DFS

1.	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
4.	BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from source.

5.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.
6.	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.
7.	Here, siblings are visited before the children	Here, children are visited before the siblings

2.3 Depth-Limited Search

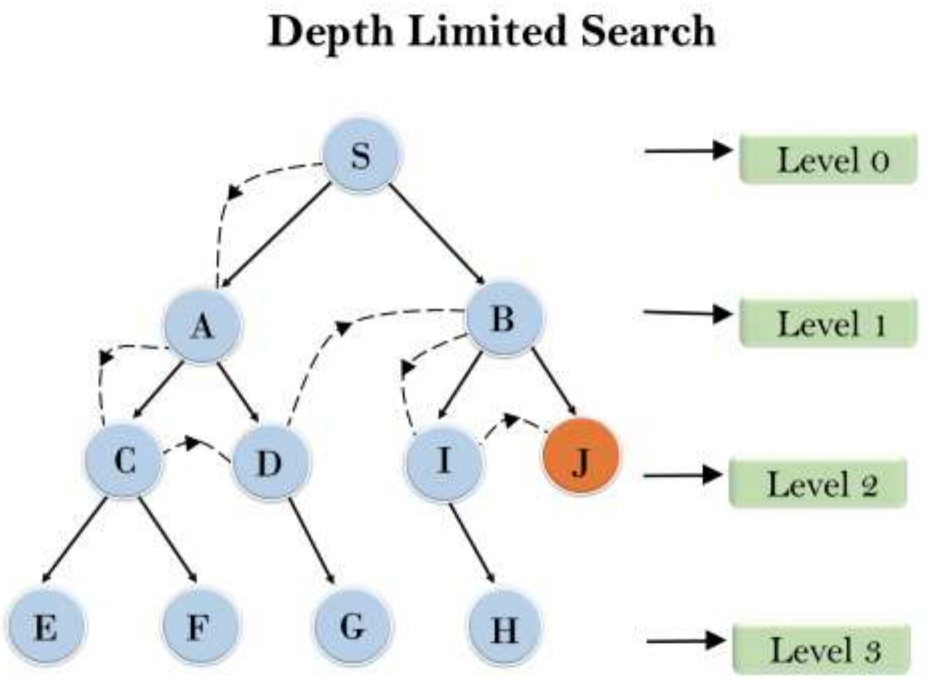
- A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
- Depth-limited search can be terminated with two Conditions of fAI(KCS-071)lure:
- Standard fAI(KCS-071)lure value: It indicates that problem does not have any solution.
- Cutoff fAI(KCS-071)lure value: It defines no solution for the problem within a given depth limit.

Advantages/Disadvantages

- **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

- 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.



Properties

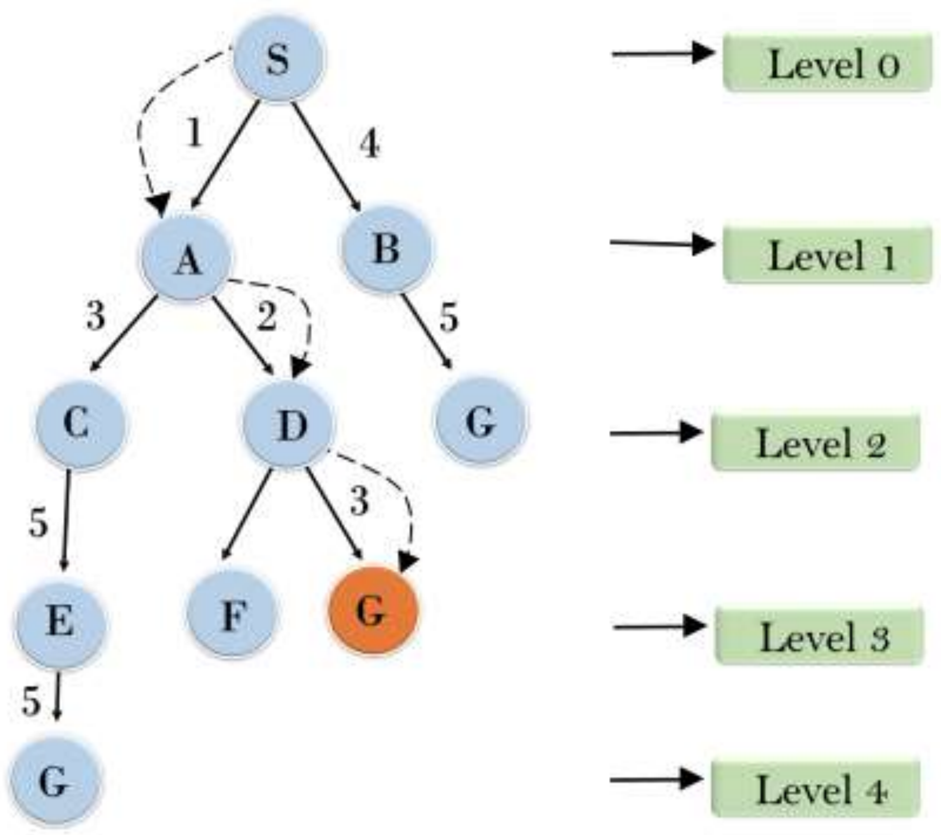
- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is **$O(b^{\ell})$**.
- **Space Complexity:** Space complexity of DLS algorithm is O**($b \times \ell$)**.
- **Optimal:** Depth-limited search can be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

2.4 Uniform-cost Search

- 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 avAI(KCS-071)lable 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.

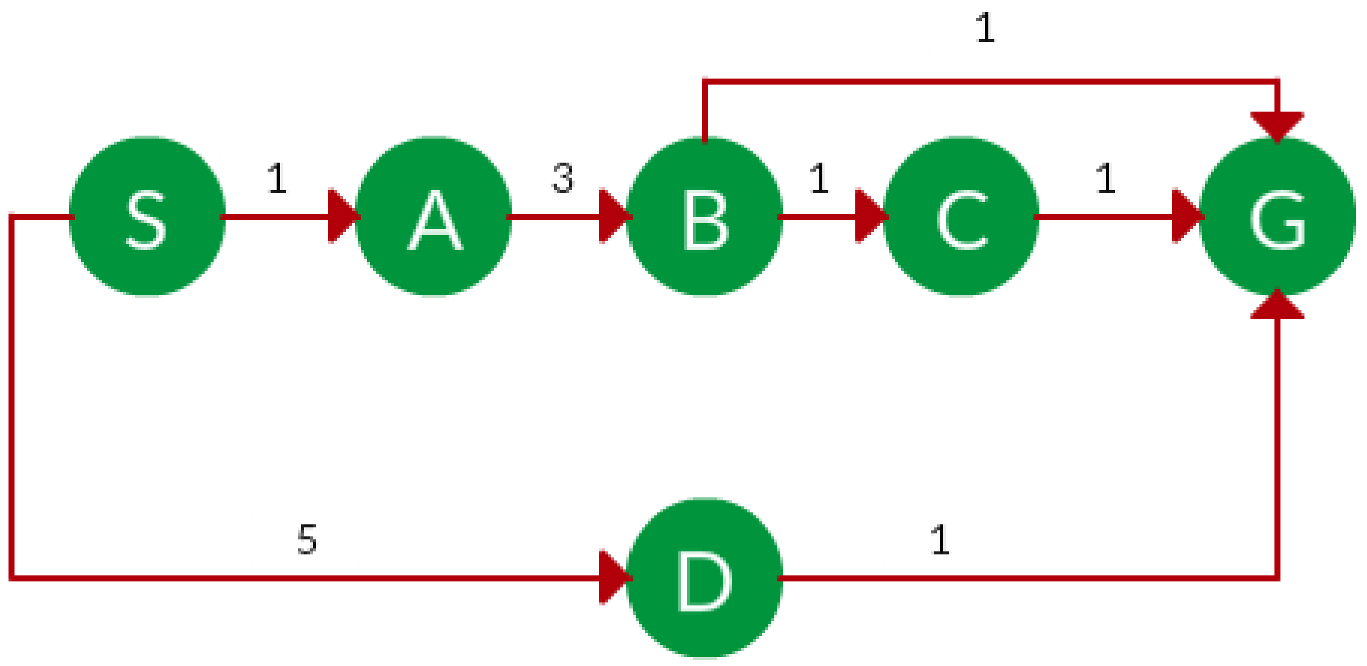
Uniform Cost Search



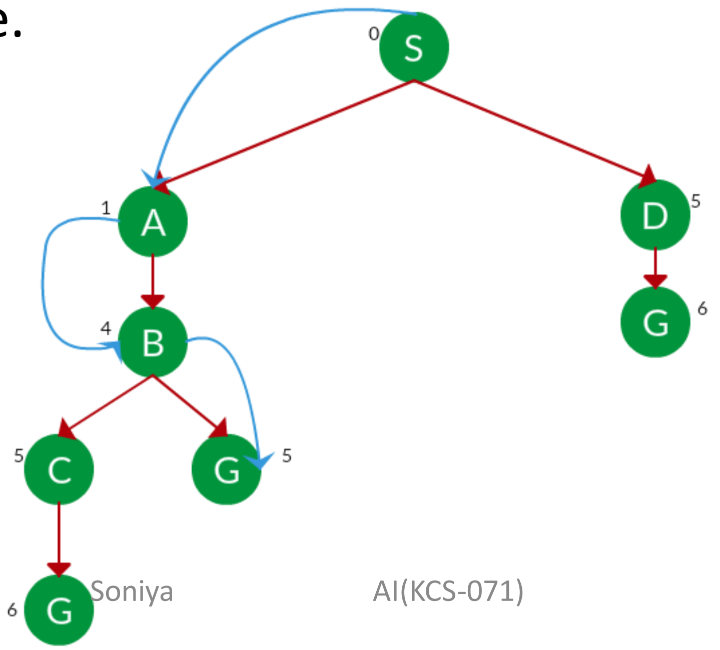
Properties

- **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**$O(b^{1 + [C^*/\varepsilon]})$/**.
- **Space Complexity:** The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **$O(b^{1 + [C^*/\varepsilon]})$**.
- **Optimal:** Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

- **Question.** Which solution would UCS find to move from node S to node G if run on the graph below?



- **Solution.** The equivalent search tree for the above graph is as follows. Cost of each node is the cumulative cost of reaching that node from the root. Based on UCS strategy, the path with least cumulative cost is chosen. Note that due to the many options in the fringe, the algorithm explores most of them so long as their cost is low, and discards them when a lower cost path is found; these discarded traversals are not shown below. The actual traversal is shown in blue.



- **Path:** S -> A -> B -> G
  Cost: 5
- Let *C* = cost of solution.
  e = arcs cost.
- Then *C/e* = effective depth
- **Time complexity:** $T(n) = O(n^{C/\varepsilon})$
- **Space complexity:** $S(n) = O(n^{C/\varepsilon})$

2.5 Iterative deepening depth-first Search

- 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 certAI(KCS-071)n "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/Disadvantages

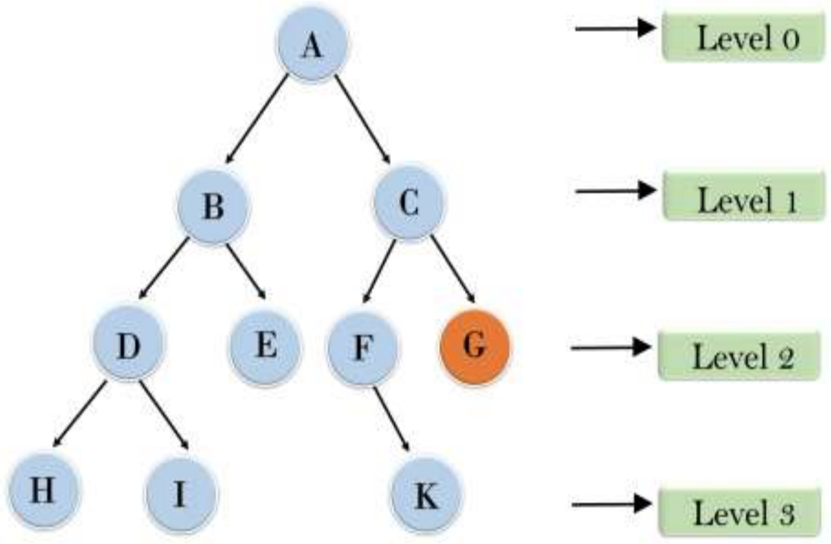
- **Advantages:**
  - It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.
- **Disadvantages:**
  - The mAI(KCS-071)n drawback of IDDFS is that it repeats all the work of the previous phase.

Example

- Example: Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

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

Iterative deepening depth first search



Properties

- **Completeness:** This algorithm is complete is if the 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.

2.6 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 sub graphs 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.

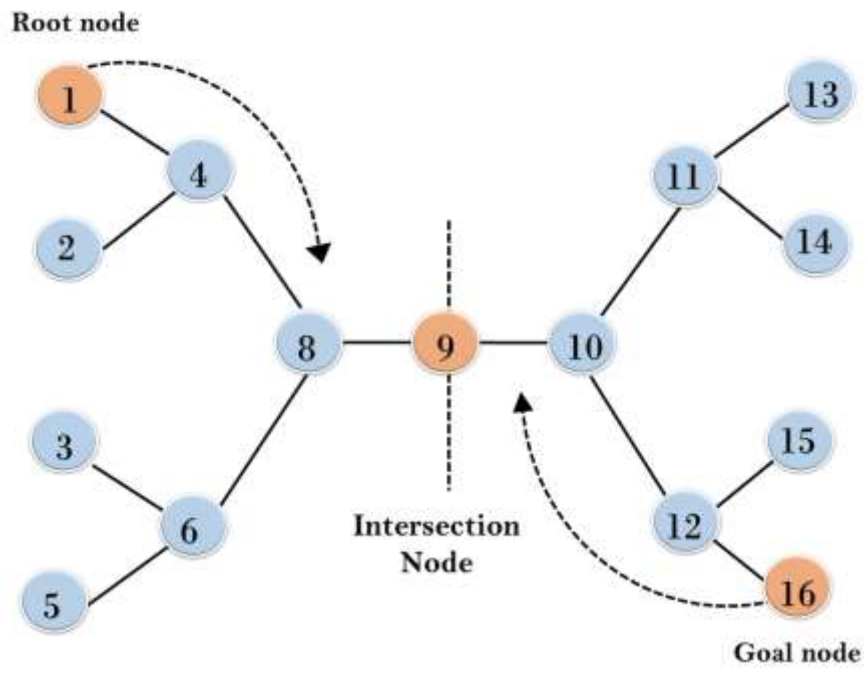
Advantages/Disadvantages

- **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

- In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.
- The algorithm terminates at node 9 where two searches meet.

Bidirectional Search



Properties

- **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.

Heuristic Function

- Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.
- **Heuristics function:** Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.
- The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.
- Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pAI(KCS-071)r of states. The value of the heuristic function is always positive.

Heuristic Function

- Admissibility of the heuristic function is given as:

  h(n) <= h*(n)
- Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.
- Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It mAI(KCS-071)ntAI(KCS-071)ns two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

3.1 Best-first Search(Greedy Search)(CO2)

- Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

  f(n)= g(n).
- Were, h(n)= estimated cost from node n to the goal.
- The greedy best first algorithm is implemented by the priority queue.

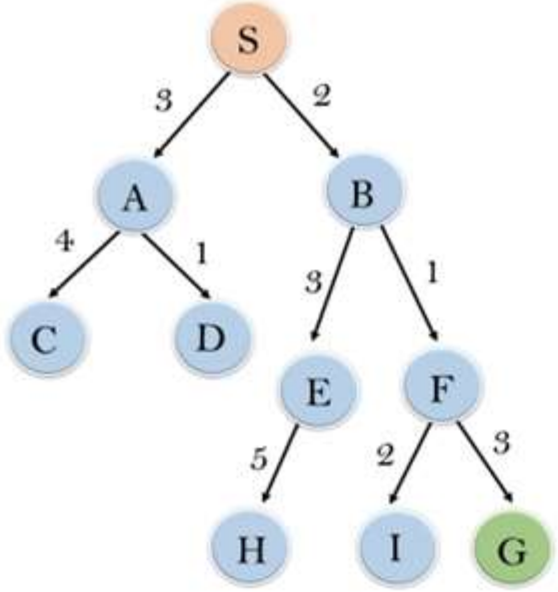
- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return fAI(KCS-071)lure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

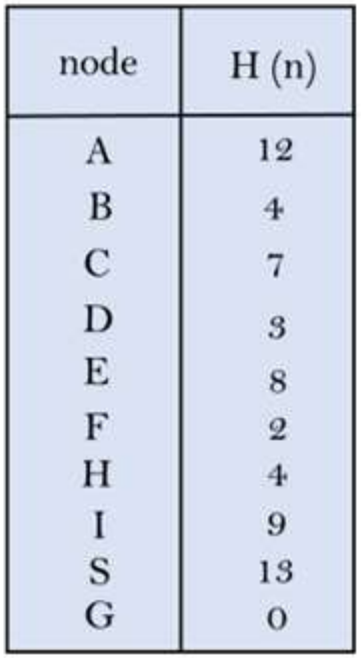
Advantages/Disadvantages

- **Advantages**:
  - Best first search can switch between BFS and DFS by gAI(KCS-071)ning 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

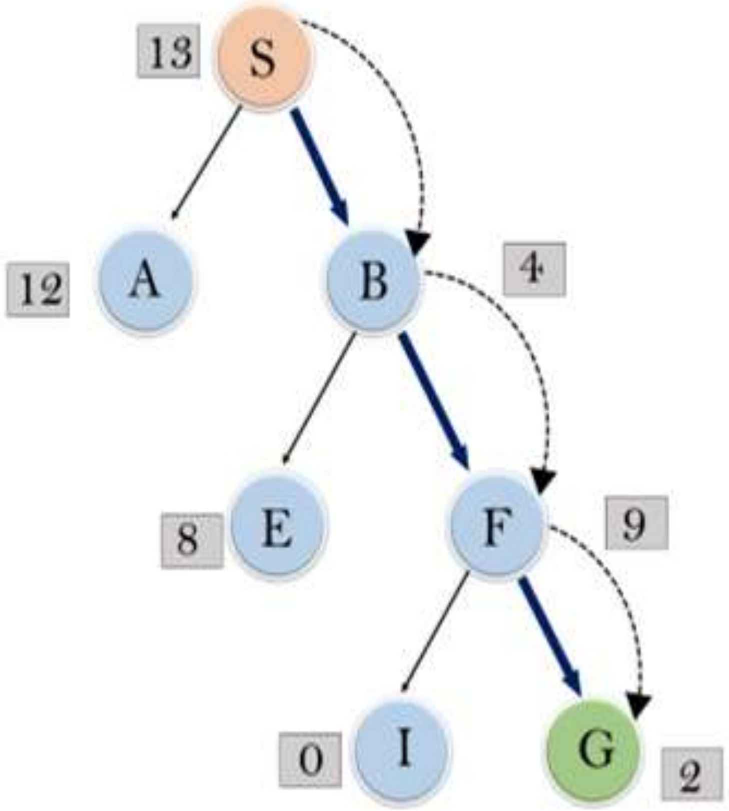
- Example: Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.





node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

- In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.
- **Expand the nodes of S and put in the CLOSED list**
- **Initialization:** Open [A, B], Closed [S]
- **Iteration 1:** Open [A], Closed [S, B]
- **Iteration 2:** Open [E, F, A], Closed [S, B]
  : Open [E, A], Closed [S, B, F]
- **Iteration 3:** Open [I, G, E, A], Closed [S, B, F]
  : Open [I, E, A], Closed [S, B, F, G]
- Hence the final solution path will be: **S----> B----->F----> G**

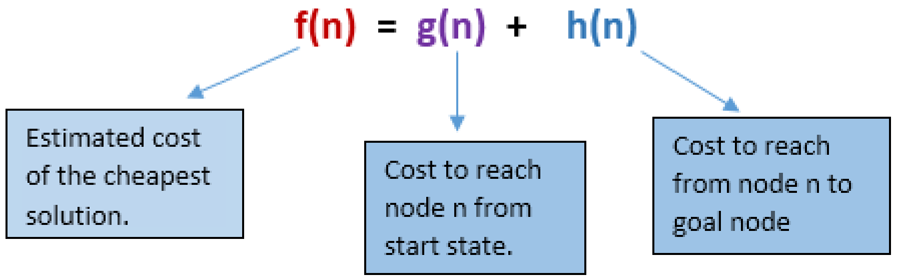


Properties

- **Time Complexity:** The worst case time complexity of Greedy best first search is O(b^m).
- **Space Complexity:** The worst case space complexity of Greedy best first search is O(b^m). Where, m is the maximum depth of the search space.
- **Complete:** Greedy best-first search is also incomplete, even if the given state space is finite.
- **Optimal:** Greedy best first search algorithm is not optimal.

3.2 A* Search Algorithm

- A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently.
- A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster.
- A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

- In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.

Algorithm

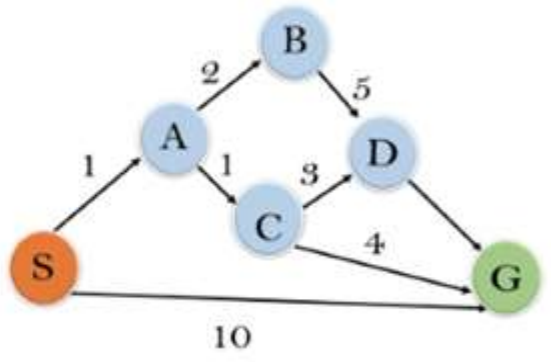
- **Step1:** Place the starting node in the OPEN list.
- **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return fAI(KCS-071)lure and stops.
- **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
- **Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
- **Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- **Step 6:** Return to **Step 2**.

Advantages/Disadvantages

- **Advantages**:
  - A* search algorithm is the best algorithm than other search algorithms.
  - A* search algorithm is optimal and complete.
  - This algorithm can solve very complex problems.
- **Disadvantages**:
  - It does not always produce the shortest path as it mostly based on heuristics and approximation.
  - A* search algorithm has some complexity issues.
  - The mAI(KCS-071)n drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example

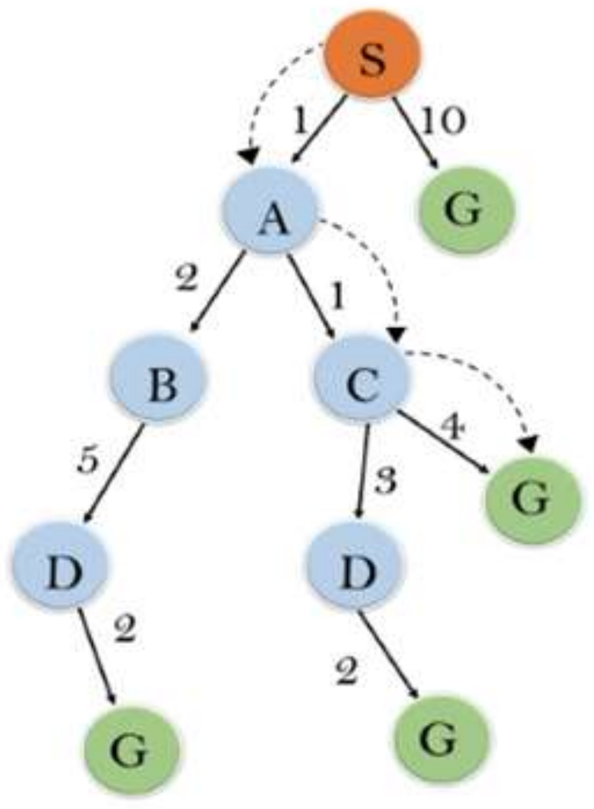
- Example: In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.
- Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution

- **Initialization:** {(S, 5)}
- **Iteration1:** {(S--> A, 4), (S-->G, 10)}
- **Iteration2:** {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
- **Iteration3:** {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}
- **Iteration 4** will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.



Properties

- **Complete:** A* algorithm is complete as long as:
  - Branching factor is finite.
  - Cost at every action is fixed.
- **Optimal:** A* search algorithm is optimal if it follows below two conditions:
  - **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
  - **Consistency:** Second required condition is consistency for only A* graph-search.
- If the heuristic function is admissible, then A* tree search will always find the least cost path.

- **Time Complexity:** The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.
- **Space Complexity:** The space complexity of A* search algorithm is **O(b^d)**

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remAI(KCS-071)ning paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)

Informed Search vs. Uninformed Search

INFORMED SEARCH	UNINFORMED SEARCH
It uses knowledge for the searching process.	It doesn’t use knowledge for searching process.
It finds solution more quickly.	It finds solution slow as compared to informed search.
It is highly efficient.	It is mandatory efficient.
Cost is low.	Cost is high.
It consumes less time.	It consumes moderate time.
It provides the direction regarding the solution.	No suggestion is given regarding the solution in it.
It is less lengthy while implementation.	It is more lengthy while implementation.
Greedy Search, A* Search, Graph Search	Depth First Search, Breadth First Search

Topic Objective

- **Topic**:

  Local search algorithms

- **Topic Objective**:

  To understand the working and applications of Hill Climbing Algorithm and Simulated Annealing approach

Prerequisites & Recap

- **Prerequisites:**
  - Understanding of basic concepts of artificial intelligence
  - Good knowledge of Python, LISP or PROLOG.

- **Recap:**

  Learned about uninformed search techniques like BFS, DFS, UCS

  Learned about informed search algorithm techniquesms like A* algorithm

  Discussed the applications of all search algorithms

4. Local search algorithms(CO2)

- In many optimization problems, the state space is the space of all possible complete solutions.
- We have an objective function that tells us how “good” a given state is, and we want to find the solution (goal) by minimizing or maximizing the value of this function.
- The start state may not be specified.
- The path to the goal doesn’t matter

- Types of Local Search Algorithms
  - Hill Climbing
  - Simulated Annealing

DAI(KCS-071)ly Quiz

- Which search strategy is also called as blind search?
  a) Uninformed search
  b) Informed search
  c) Simple reflex search
  d) All of the mentioned
- What is the other name of informed search strategy?
  a) Simple search
  b) Heuristic search
  c) Online search
  d) None of the mentioned
- A heuristic is a way of trying ____________
  a) To discover something or an idea embedded in a program
  b) To search and measure how far a node in a search tree seems to be from a goal
  c) To compare two nodes in a search tree to see if one is better than another
  d) All of the mentioned

4.1 Hill Climbing Algorithm

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountAI(KCS-071)n or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is avAI(KCS-071)lable.
- In this algorithm, we don't need to mAI(KCS-071)ntAI(KCS-071)n and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.

- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.

- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing

- The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.

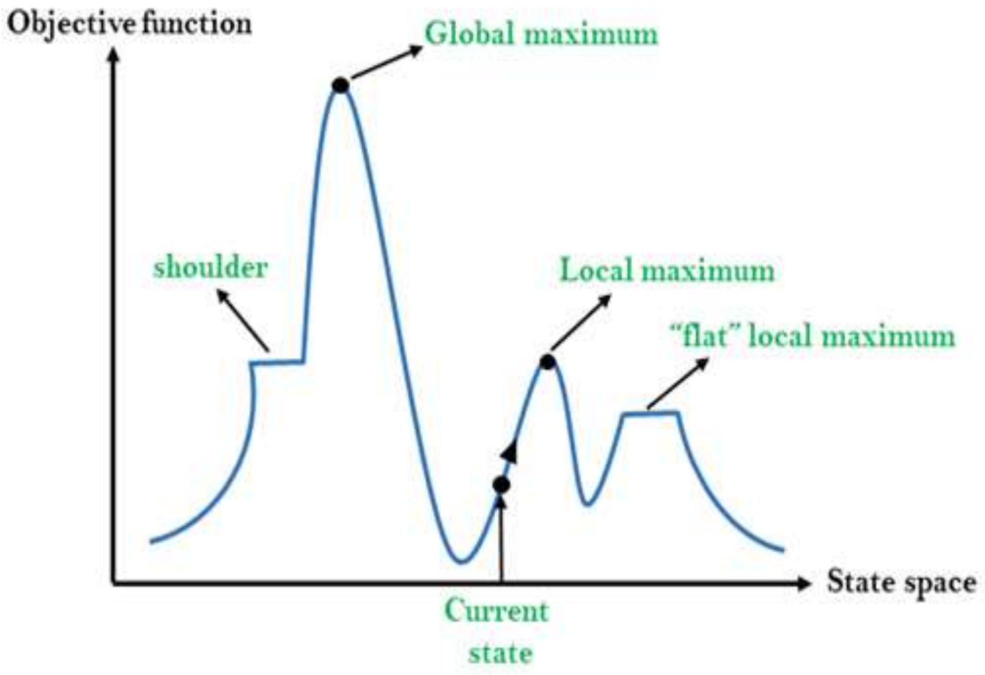
- On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis.

- If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum.

- If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.

Different regions in the state space landscape

- **Local Maximum:** Local maximum is a state which is better than its neighbor 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 objective function.
- **Current state:** It is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum:** It is a flat space in the landscape where all the neighbor states of current states have the same value.
- **Shoulder:** It is a plateau region which has an uphill edge.



Types of Hill Climbing Algorithm

- **Simple hill Climbing:** This is the simplest way to implement a hill climbing algorithm.
- It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state.
- It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state.
- This algorithm has the following features:
  - Less time consuming
  - Less optimal solution and the solution is not guaranteed

Types of Hill Climbing Algorithm

- **Steepest-Ascent hill-climbing**: This algorithm is a variation of simple hill climbing algorithm.
- This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state.
- This algorithm consumes more time as it searches for multiple neighbors

- **Stochastic hill Climbing:** It does not examine for all its neighbor before moving.
- Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Algorithm for Simple Hill Climbing

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
  - If it is goal state, then return success and quit.
  - Else if it is better than the current state then assign new state as a current state.
  - Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

Algorithm for Steepest-Ascent hill climbing

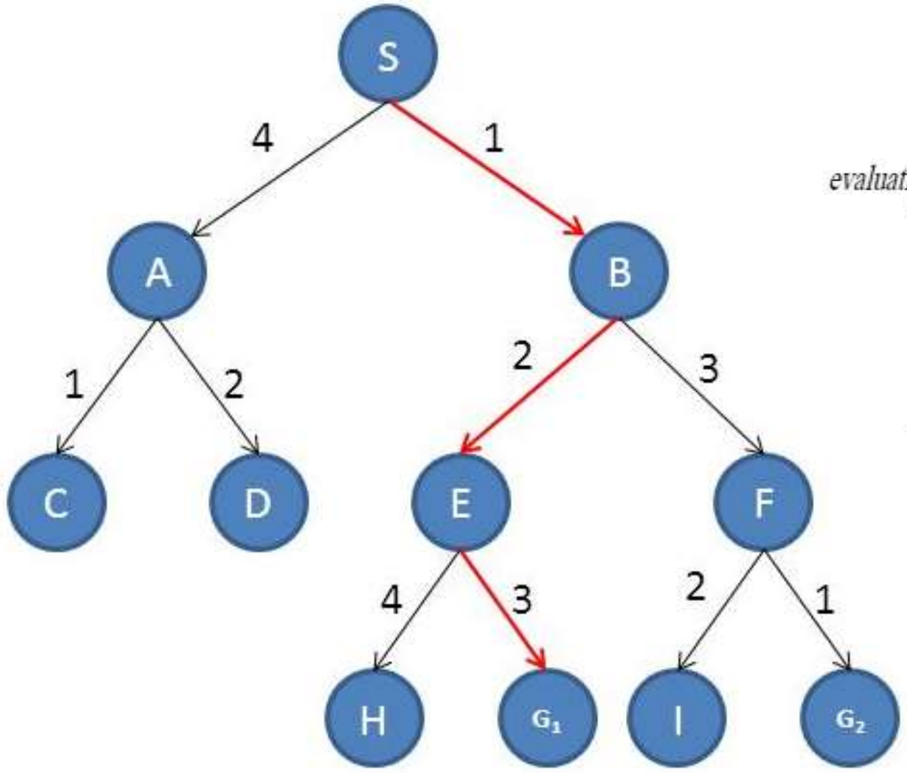
- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
- **Step 3:** Let SUCC be a state such that any successor of the current state will be better than it.
- **Step 4:** For each operator that applies to the current state:
    - Apply the new operator and generate a new state.
    - Evaluate the new state.
    - If it is goal state, then return it and quit, else compare it to the SUCC.
    - If it is better than SUCC, then set new state as SUCC.
    - If the SUCC is better than the current state, then set current state to SUCC.
- **Step 5:** Exit.

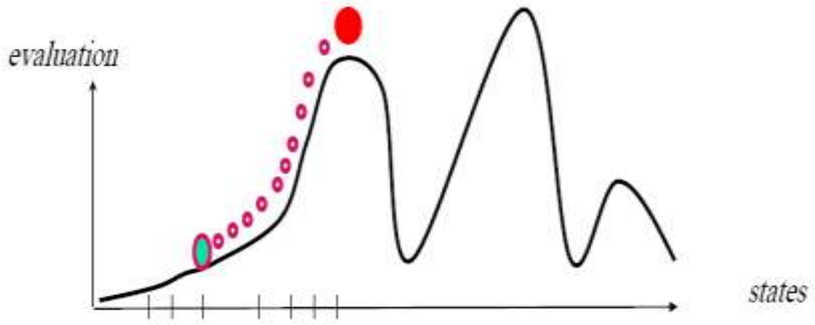
Example

S → B → E → G1

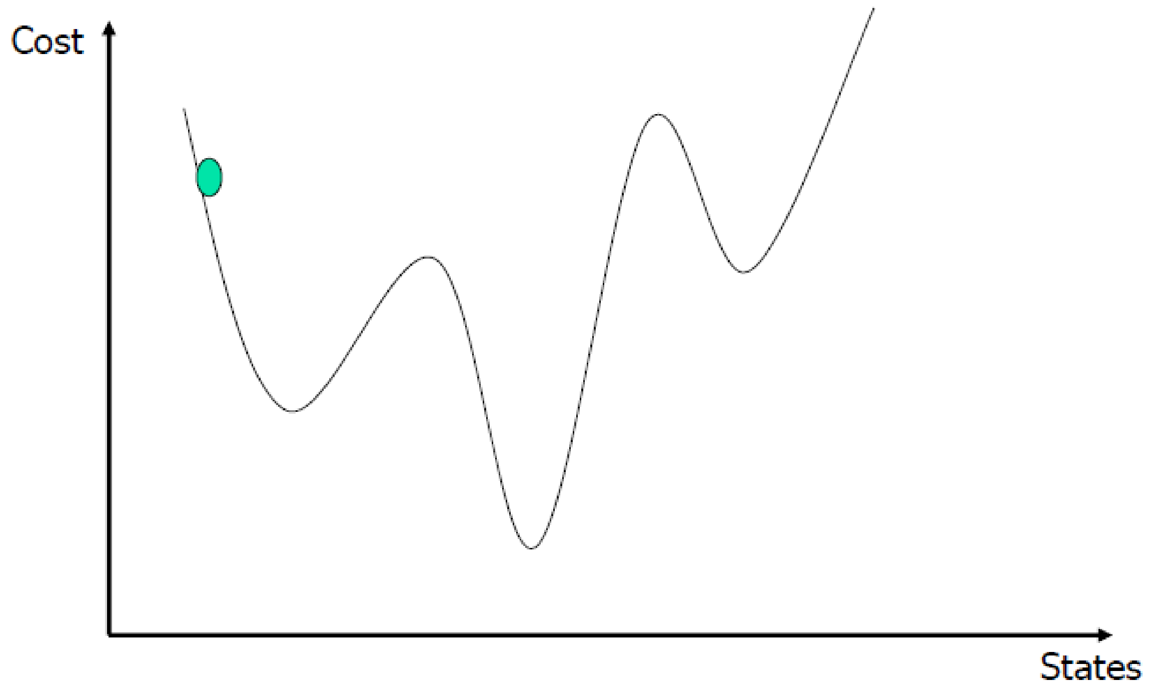
SEARCH PATH = [S_0, B_1, E_2, $G1_3$]

Cost = 1+2+3=6

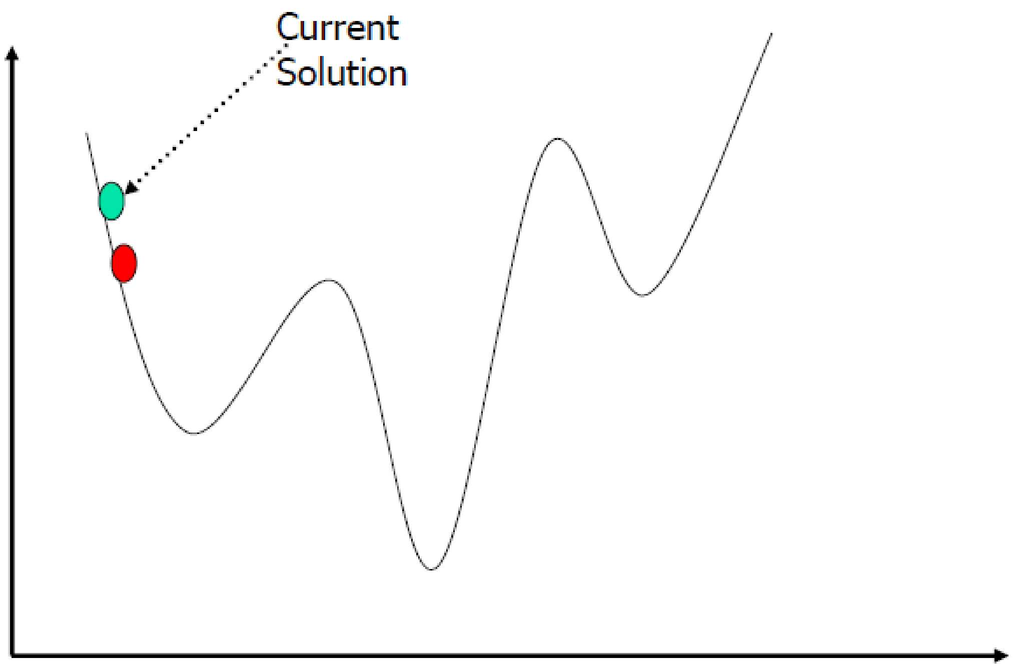




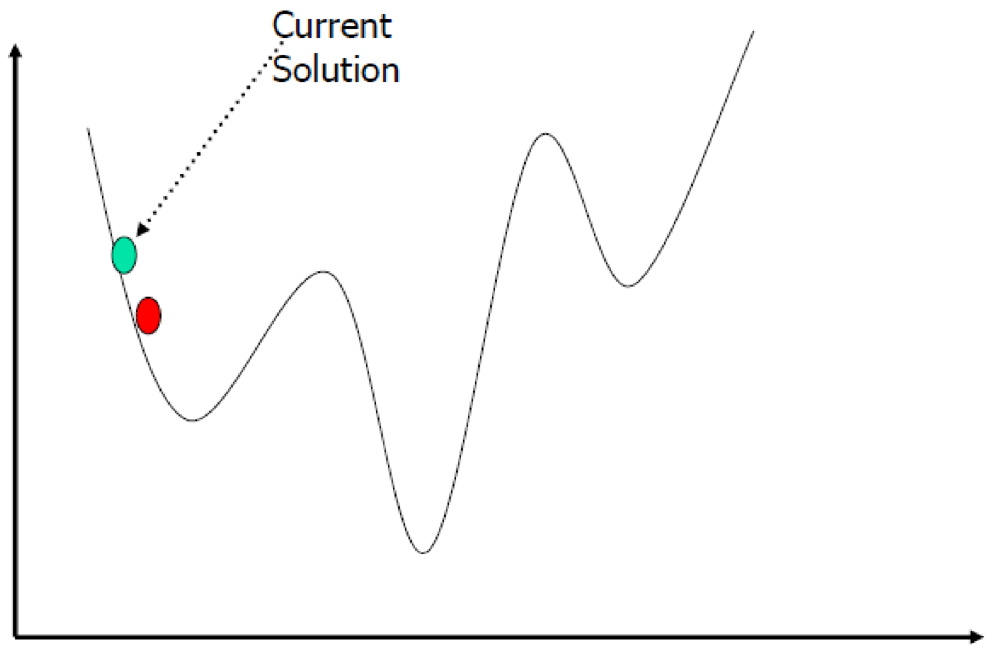
Example



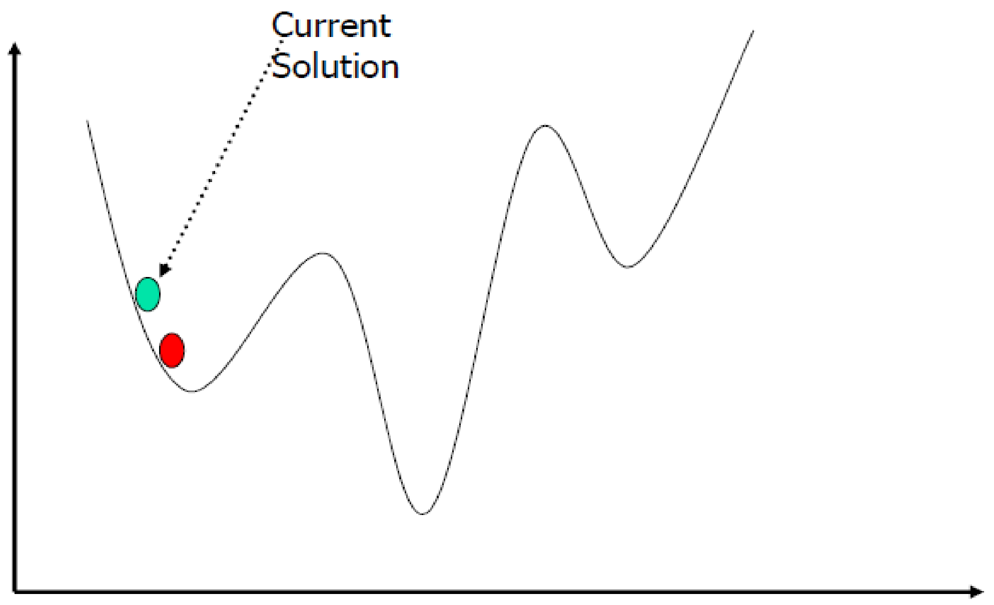
Example



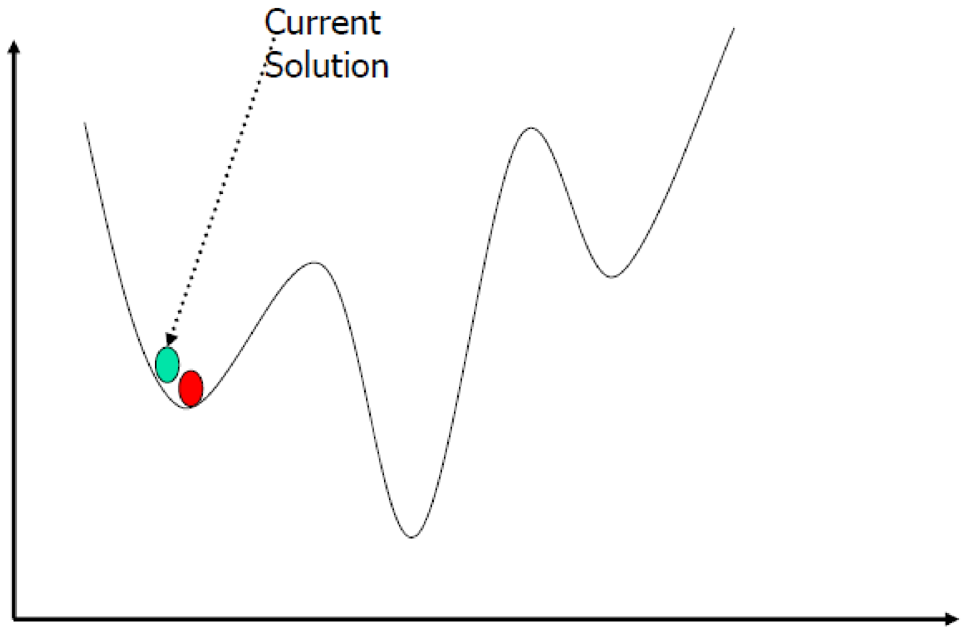
Example



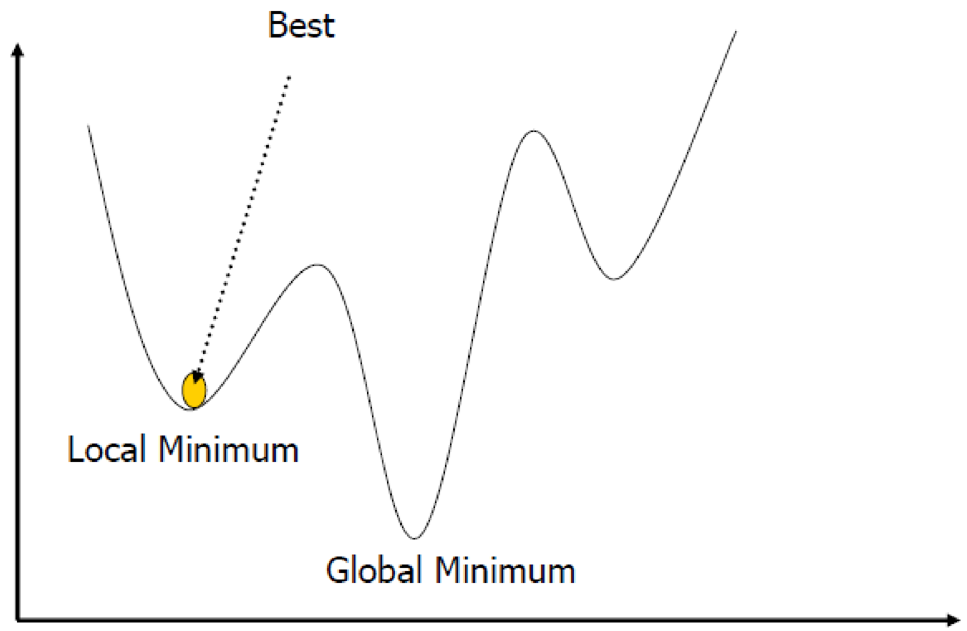
Example



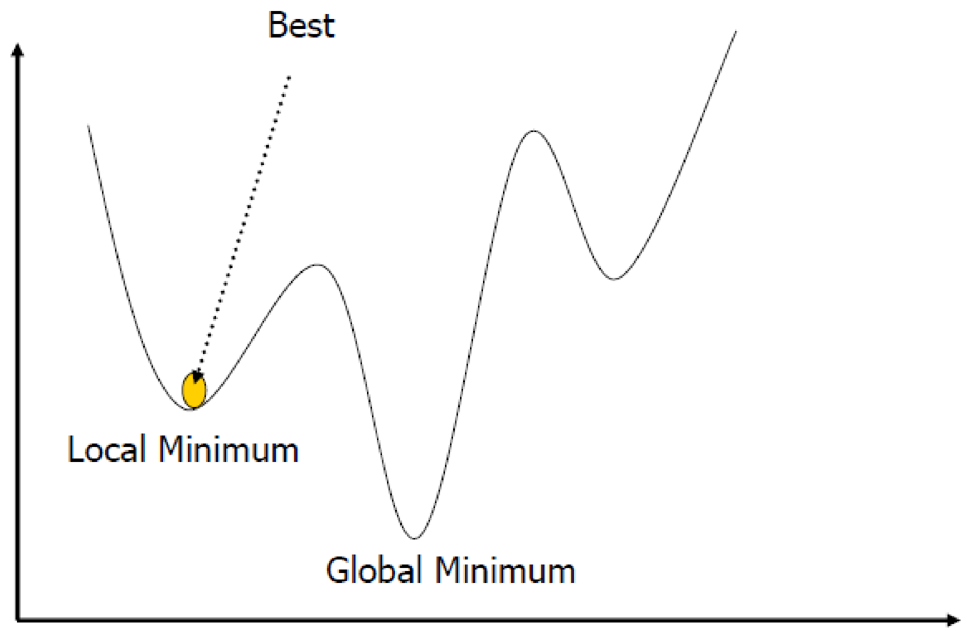
Example



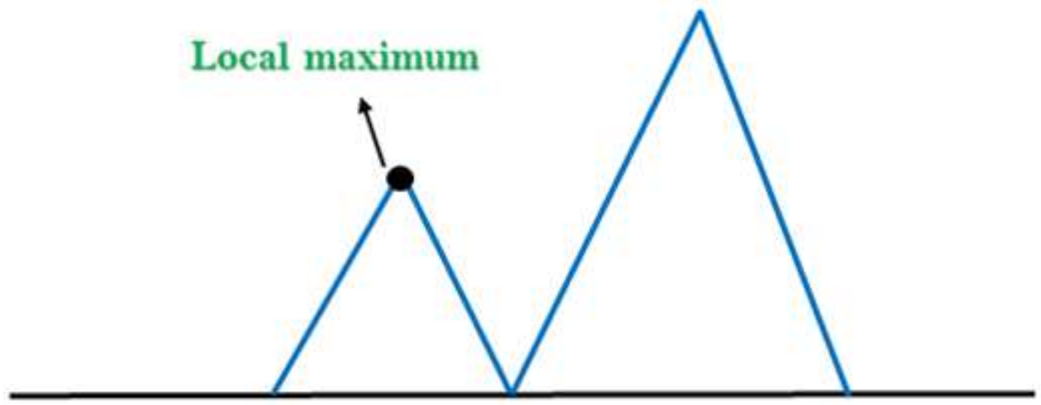
Example



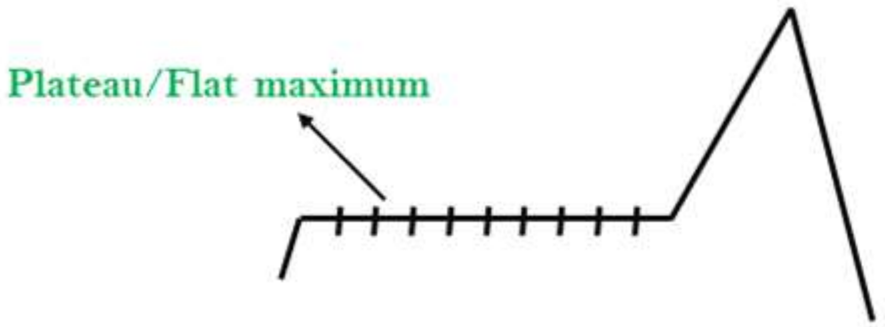
Example



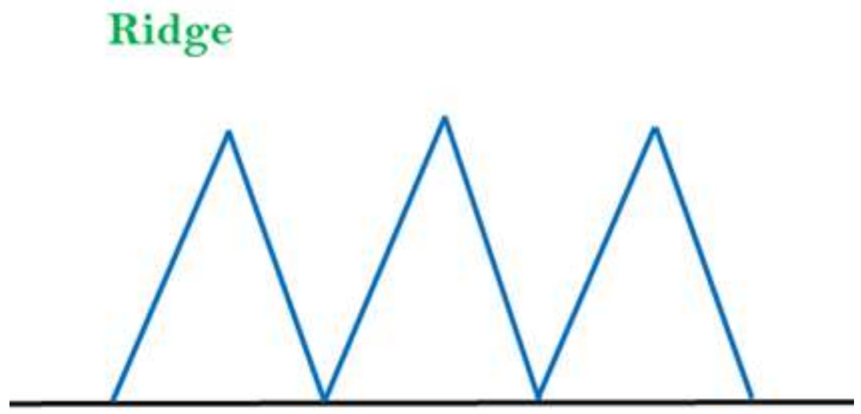
- **Local Maximum:** A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.
- **Solution:** Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



- **Plateau:** A plateau is the flat area of the search space in which all the neighbor states of the current state contAI(KCS-071)ns the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.
- **Solution:** The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



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



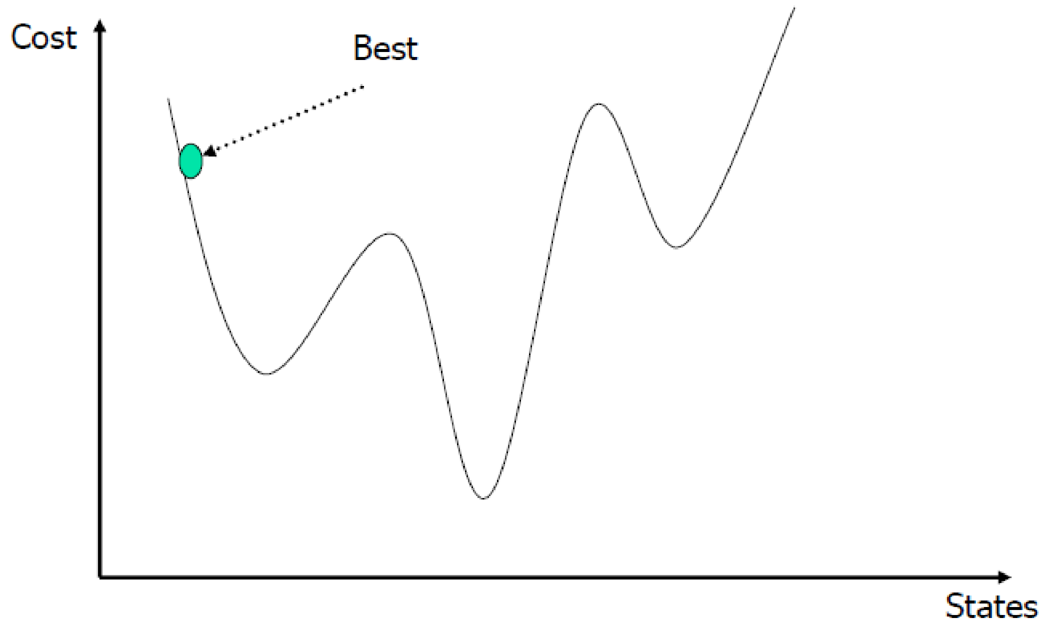
4.2 Simulated Annealing

- A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient.
- **Simulated Annealing** is an algorithm which yields both efficiency and completeness.
- In mechanical term **Annealing** is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state.
- The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

- **Basic inspiration**: What is annealing?
  - In metallurgy, annealing is the physical process used to temperature or harden metals or glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to coalesce into a low energy crystalline state.
  - **Heating then slowly cooling a substance to obtAI(KCS-071)n a strong crystalline structure.**
- **Key idea:** Simulated Annealing combines Hill Climbing with a random walk in some way that yields both efficiency and completeness.

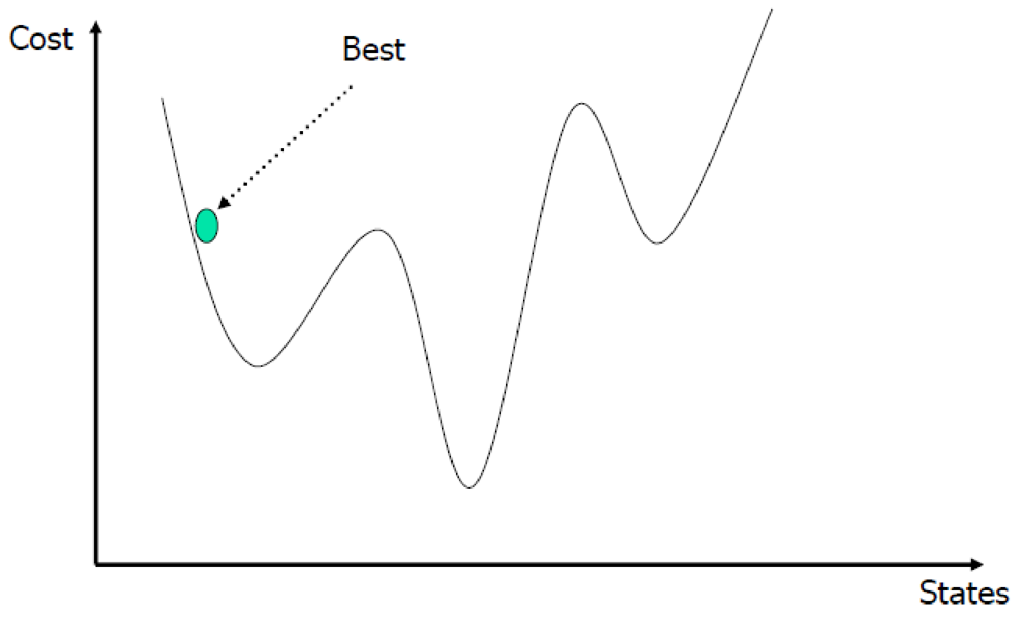
Simulated Annealing in action

•



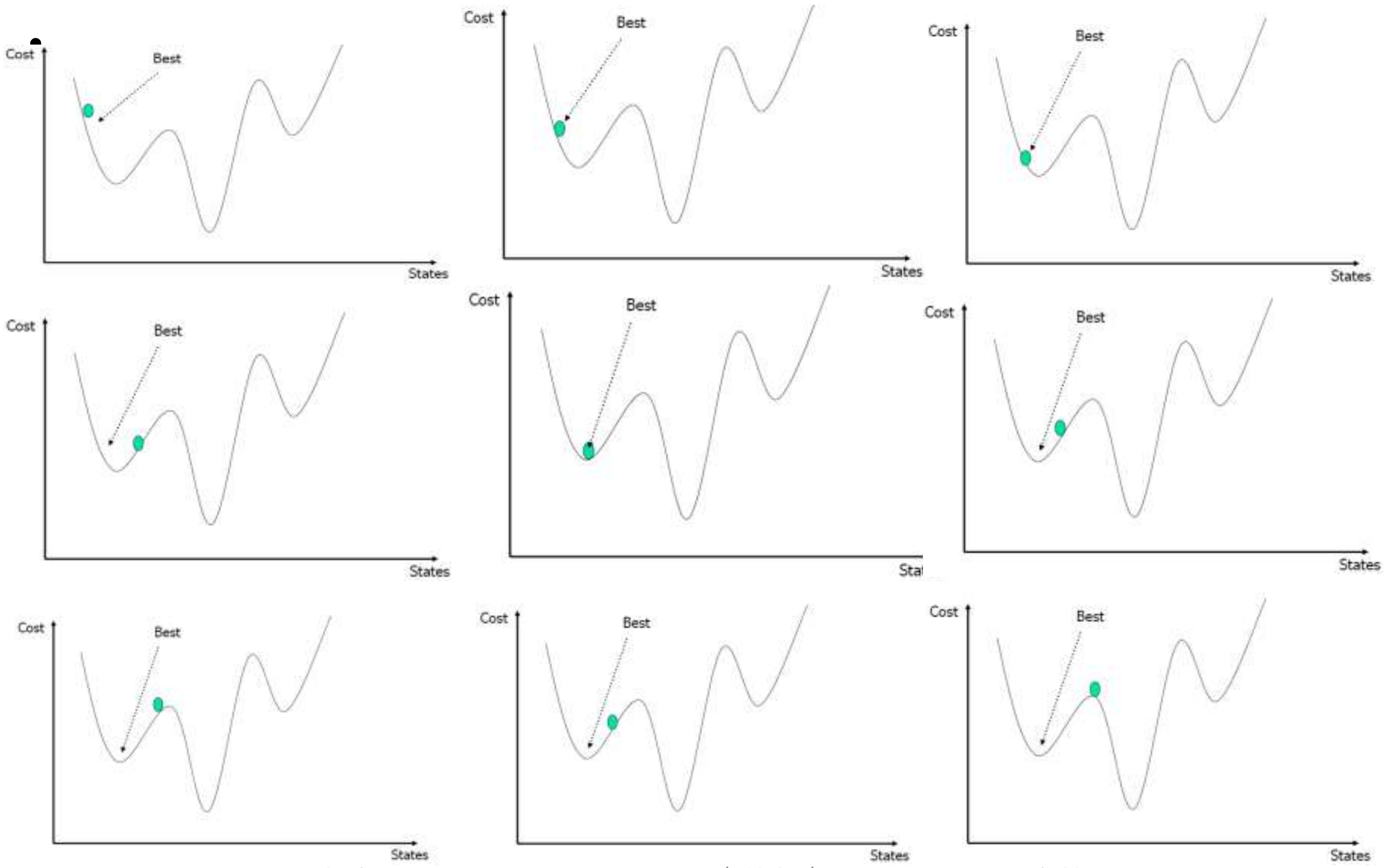
Simulated Annealing in action

•

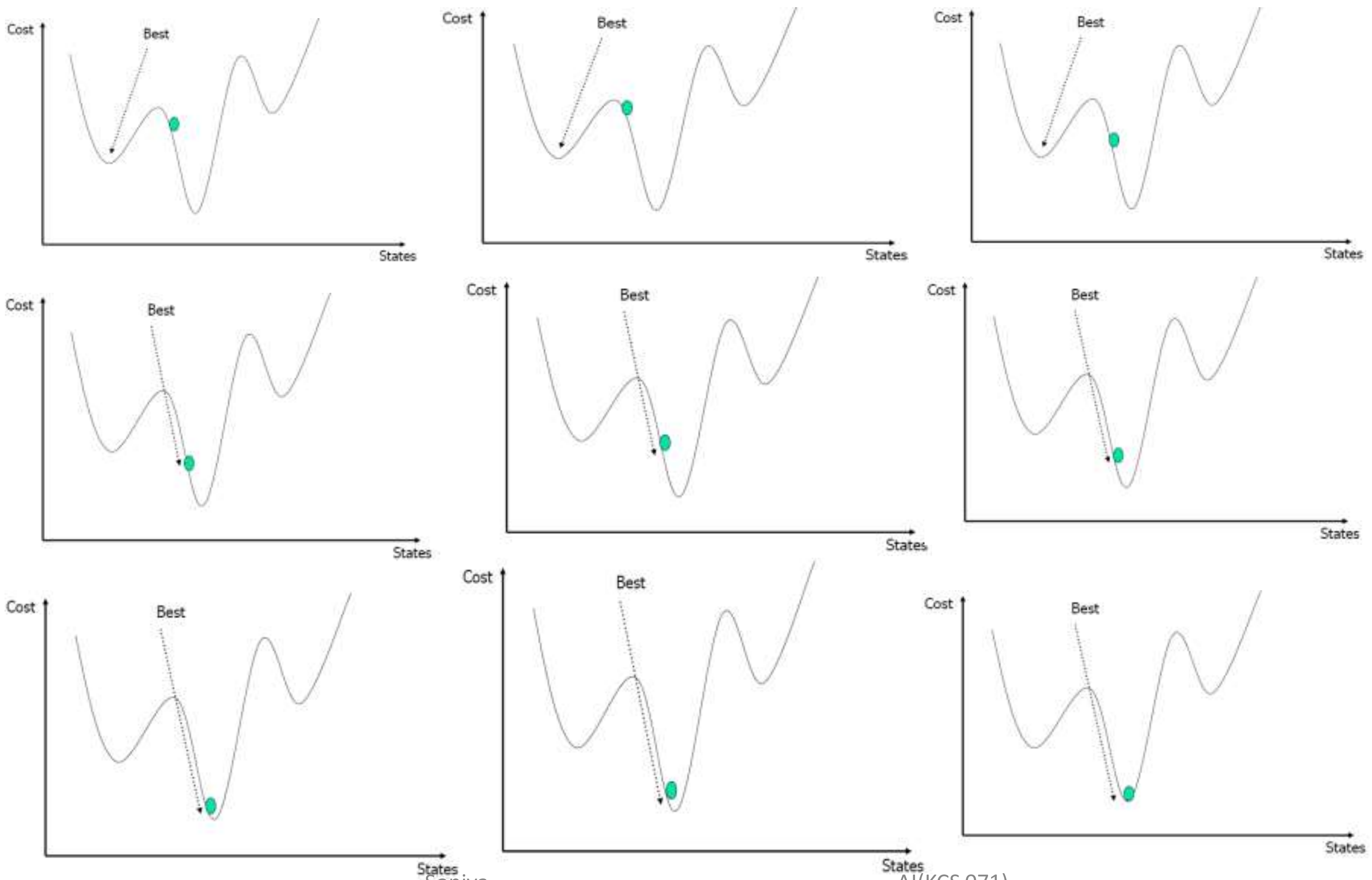


Simulated Annealing in action

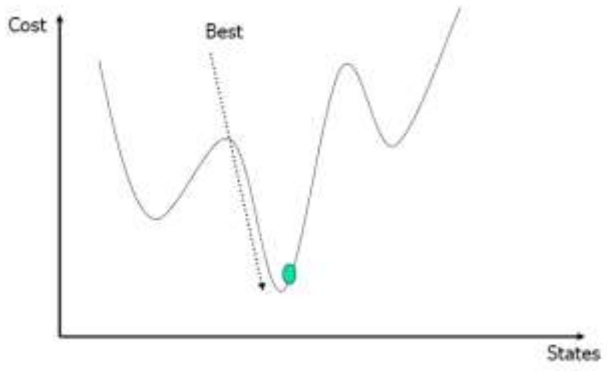
•

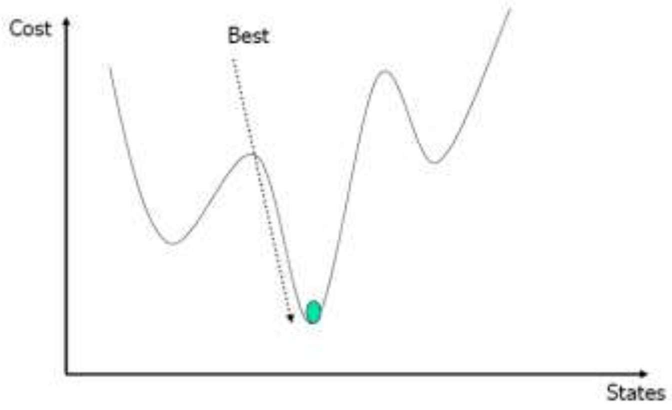


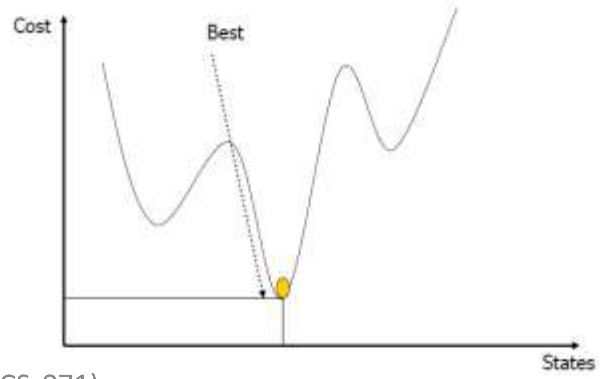
Simulated Annealing in action



Simulated Annealing in action







Topic Objective

- **Topic**:

  Adversarial Search

- **Topic Objective**:

  To understand Game Theory

Prerequisites & Recap

- **Prerequisites:**
  - Understanding of basic concepts of artificial intelligence
  - Basic Knowledge of Computer Games like chess and tic-tac-toe
  - Good knowledge of Python, LISP or PROLOG.

- **Recap:**

  Learned about uninformed search techniques like BFS, DFS, UCS

  Learned about informed search algorithm techniques like A* algorithm

  Discussed the working and applications of Hill Climbing Algorithm and Simulated Annealing approach

DAI(KCS-071)ly Quiz

- What is the space complexity of Depth-first search?
  a) O(b)
  b) O(bl)
  c) O(m)
  d) O(bm)

- Zero sum game has to be a _______ game.
  a) Single player
  b) Two player
  c) Multiplayer
  d) Three player

5. Adversarial Search

- Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning agAI(KCS-071)nst us.
  - In previous topics, we have studied the search strategies which are only associated with a single agent that AI(KCS-071)ms to find the solution which often expressed in the form of a sequence of actions.

5. Adversarial Search (CO2)

- But, there might be some situations where more than one agent is searching for the solution in 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, in which each agent is an opponent of other agent and playing agAI(KCS-071)nst each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
- So, 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.
- Games are modelled as a Search problem and heuristic evaluation function, and these are the two mAI(KCS-071)n factors which help to model and solve games in AI(KCS-071).

Introduction

- Mathematically, this search is based on the concept of **‘Game Theory.’** ***According to game theory, a game is played between two players. To complete the game, one has to win the game and the other looses automatically.’***

My Interest
“I Win”
Your Interest
“You Loose”
We are opponents- I win, you loose.

Types of Games in AI(KCS-071)

- **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.

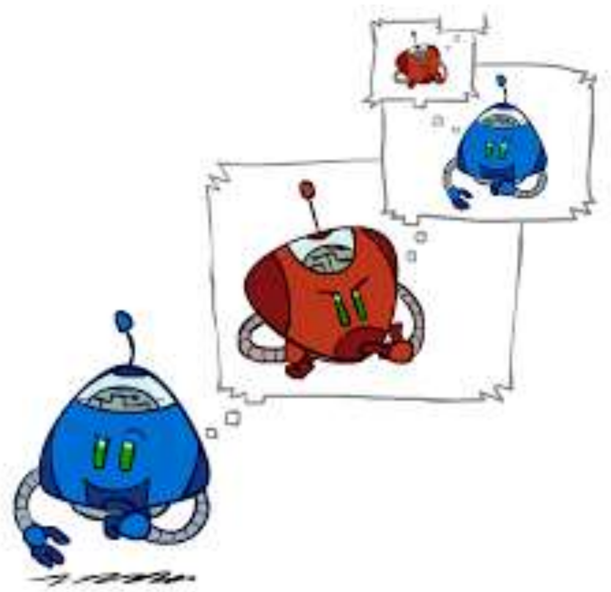
Types of Games in AI(KCS-071)

- **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.

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

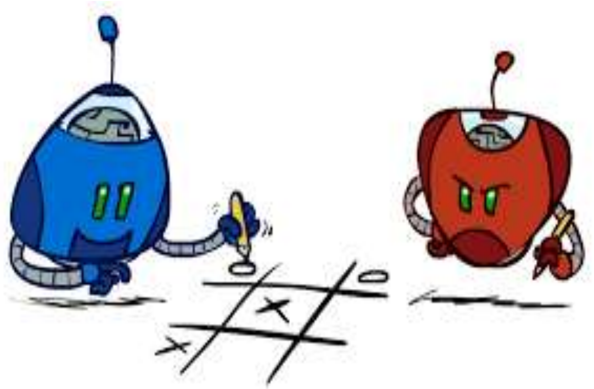
Zero-Sum Game

- Zero-sum games are adversarial search which involves pure competition.
- In Zero-sum game each agent's gAI(KCS-071)n of utility is exactly balanced by the losses or gAI(KCS-071)ns of utility of another agent.
- One player of the game try to maximize one single value, while other player tries to minimize it.
- Each move by one player in the game is called as ply.
- Chess and tic-tac-toe are examples of a Zero-sum game.



Zero-sum game: Embedded thinking

- The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:
  - What to do.
  - How to decide the move
  - The opponent also thinks what to do
  - Needs to think about his opponent as well



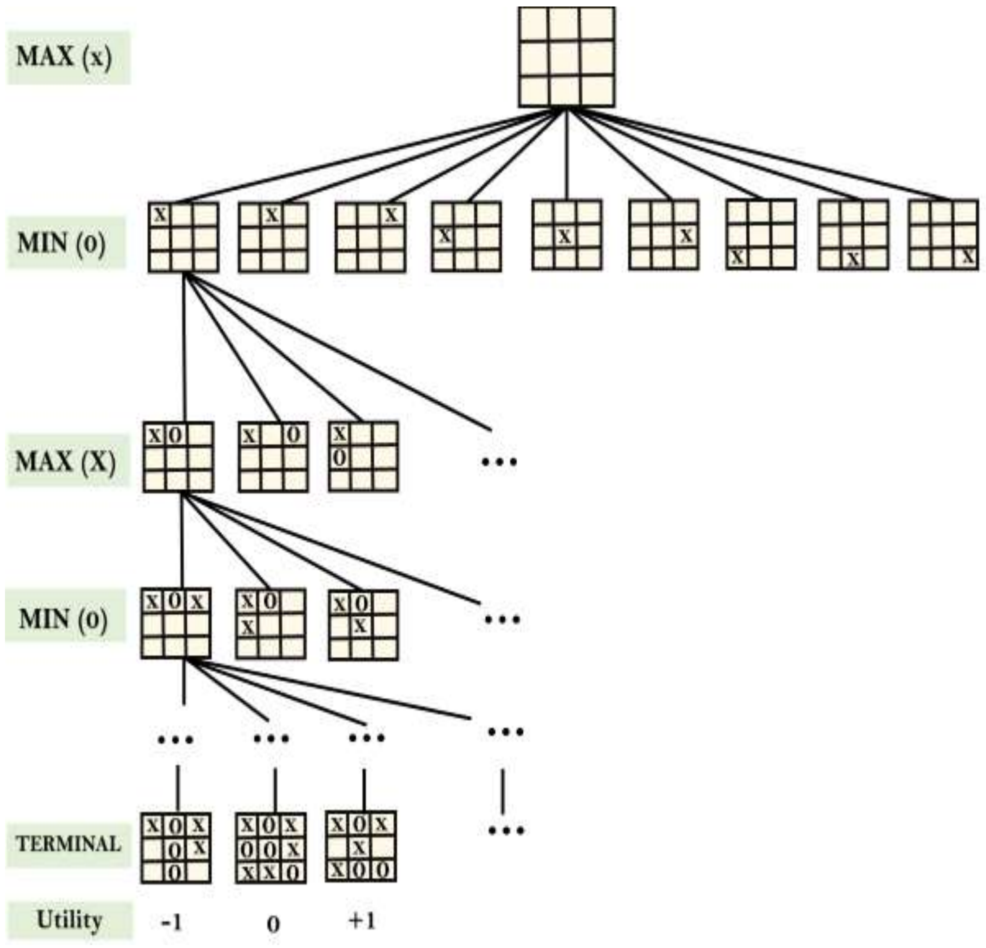
- Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI(KCS-071).

Elements of Game Playing search

- **A game can be defined as a type of search in AI(KCS-071) 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 terminal states.
  - **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 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.

Game tree

- A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.
- **Eg: Tic-Tac-Toe game tree:**
  - There are two players MAX and MIN.
  - Players have an alternate turn and start with MAX.
  - MAX maximizes the result of the game tree
  - MIN minimizes the result.



Example Explanation:

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility agAI(KCS-071)nst an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting agAI(KCS-071)nst Max in the game.
- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as **Ply**. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Adversarial Search for the minimax procedure

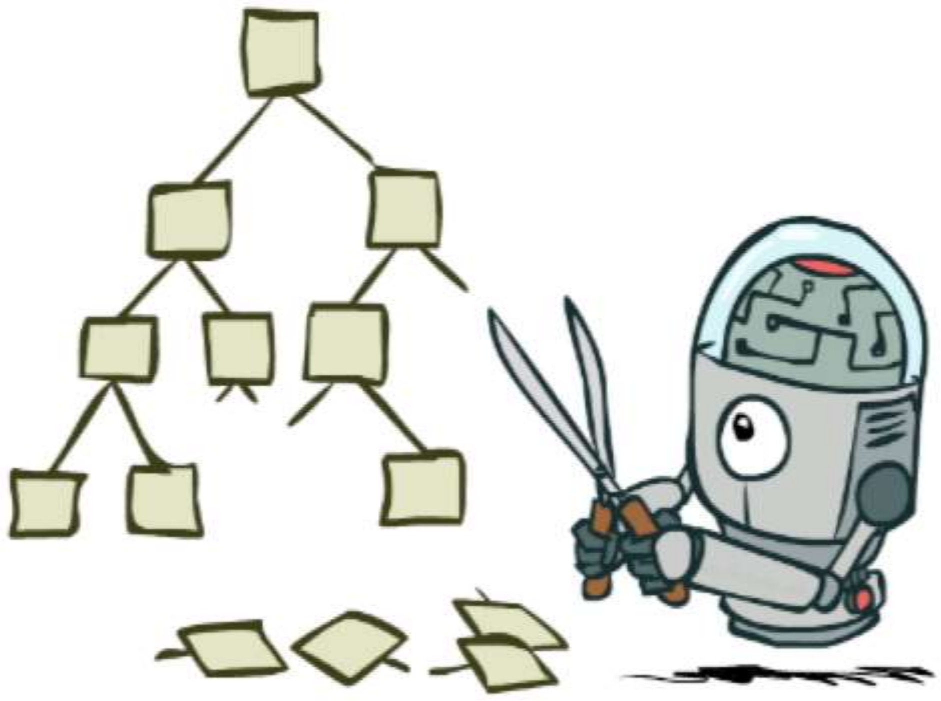
- It AI(KCS-071)ms to find the optimal strategy for MAX to win the game.
- It follows the approach of Depth-first search.
- In the game tree, optimal leaf node could appear at any depth of the tree.
- Propagate the minimax values up to the tree until the terminal node discovered.
- In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefe :

For a state S MINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN}. \end{cases}$$

Types of adversarial search

- There are following types of adversarial search:
  - **Minimax Algorithm**
  - **Alpha-beta Pruning.**

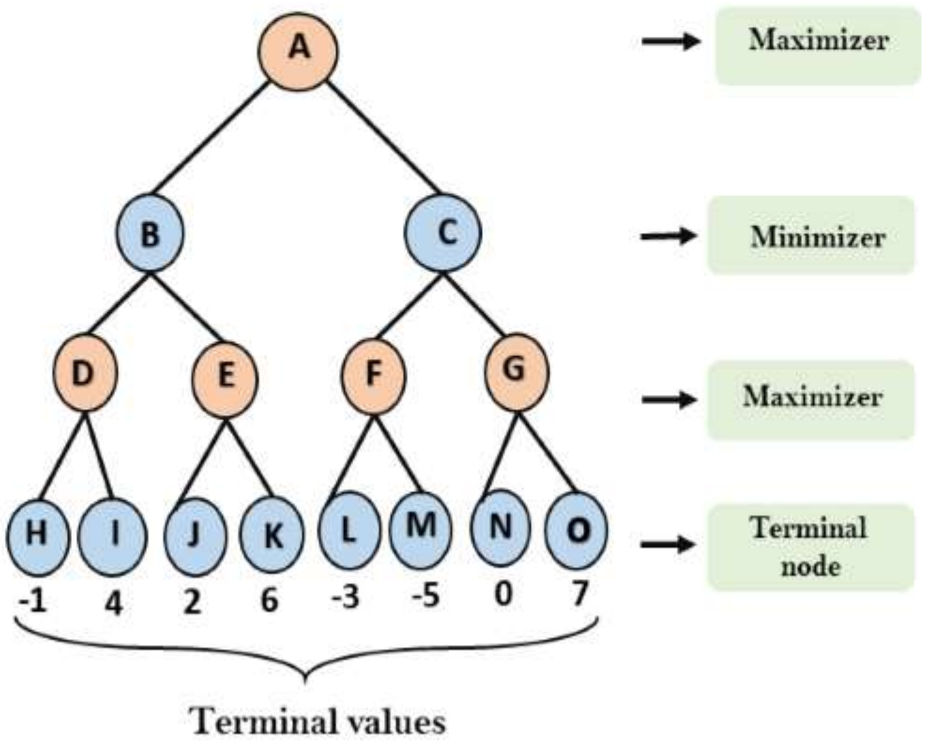


Mini-Max Algorithm

- 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.
- It is mostly used for game playing in AI(KCS-071). This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- It performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax 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:

- **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. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



Working of Min-Max Algorithm:

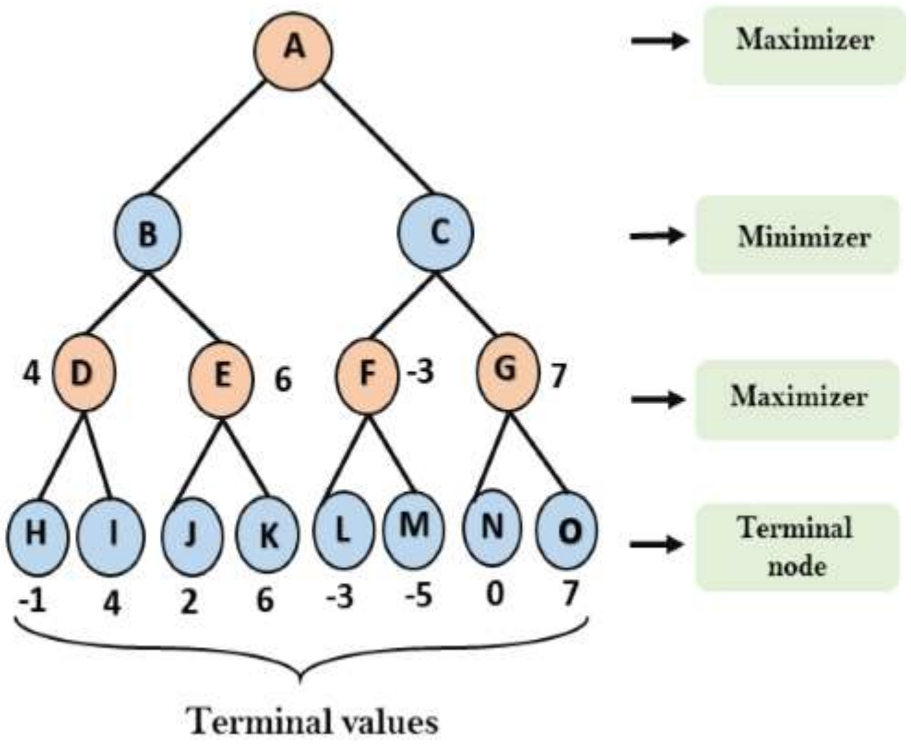
- **Step 2:** Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare eacl value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.
  - For node D

    max(-1,- -∞) => max(-1,4)= 4
  - For Node E

    max(2, -∞) => max(2, 6)= 6
  - For Node F

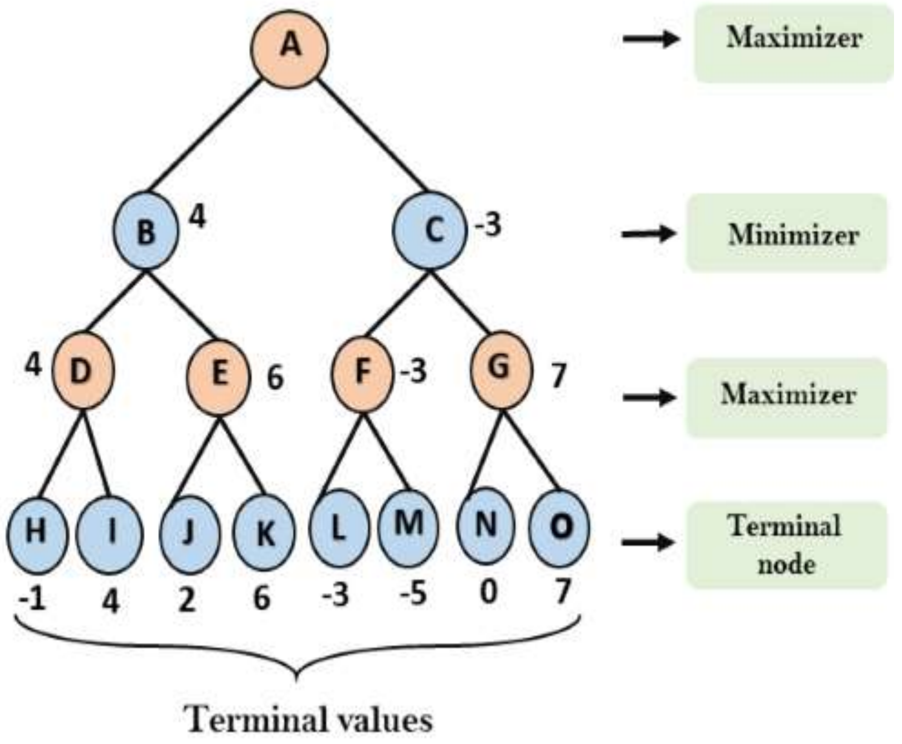
    max(-3, -∞) => max(-3,-5) = -3
  - For node G

    max(0, -∞) = max(0, 7) = 7



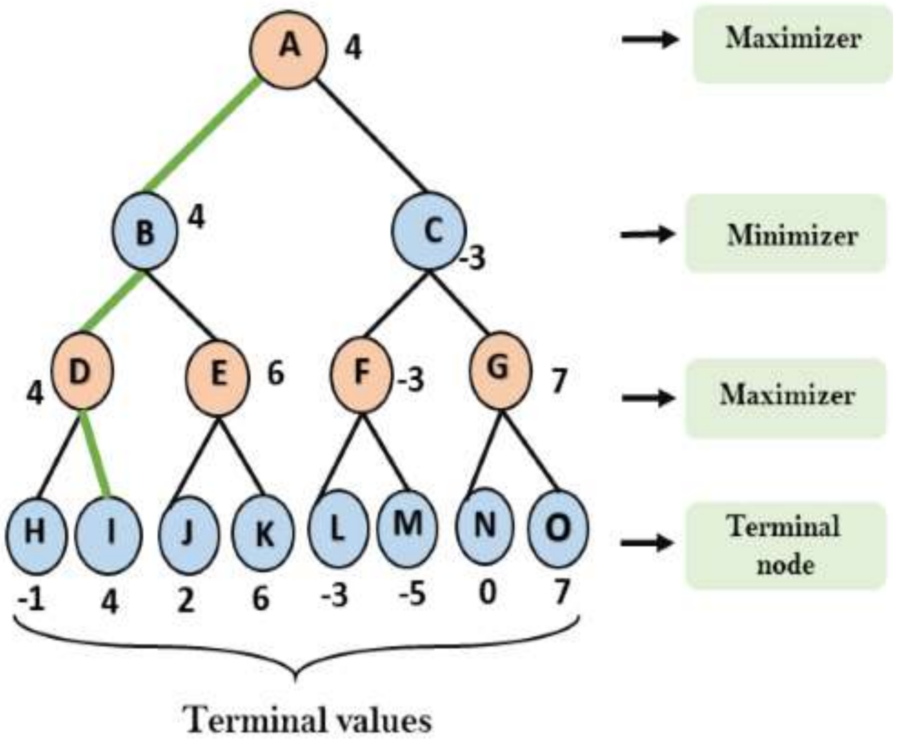
Working of Min-Max Algorithm:

- **Step 3:** In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.
  - For node B= min(4,6) = 4
  - For node C= min (-3, 7) = -3



Working of Min-Max Algorithm:

- **Step 4:** Now it's a turn for Maximizer, and it will agAI(KCS-071)n choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.
  - 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)**.

Topic Objective

- **Topic**:

  Alpha-Beta Pruning

- **Topic Objective**:

  To understand the concept of Pruning

  To learn the working of alpha-beta pruning technique

Prerequisites & Recap

- **Prerequisites:**
  - Understanding of basic concepts of artificial intelligence
  - Good knowledge of Python, LISP or PROLOG.

- **Recap:**

  Discussed the working and applications of Hill Climbing Algorithm and Simulated Annealing approach

  Learned about Game Theory in AI(KCS-071)

DAI(KCS-071)ly Quiz

- General algorithm applied on game tree for making decision of win/lose is _____________
  a) DFS/BFS Search Algorithms
  b) Heuristic Search Algorithms
  c) Greedy Search Algorithms
  d) MIN/MAX Algorithms

- What is the complexity of minimax algorithm?
  a) Same as of DFS
  b) Space – bm and time – bm
  c) Time – bm and space – bm
  d) Same as BFS

6. Alpha-Beta Pruning(CO2)

- Alpha-beta pruning is an optimization technique for the minimax algorithm.
- There is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**.
- This involves two threshold parameter Alpha and beta for future expansion, so it 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 **+∞**.

6. Alpha-Beta Pruning(CO2)

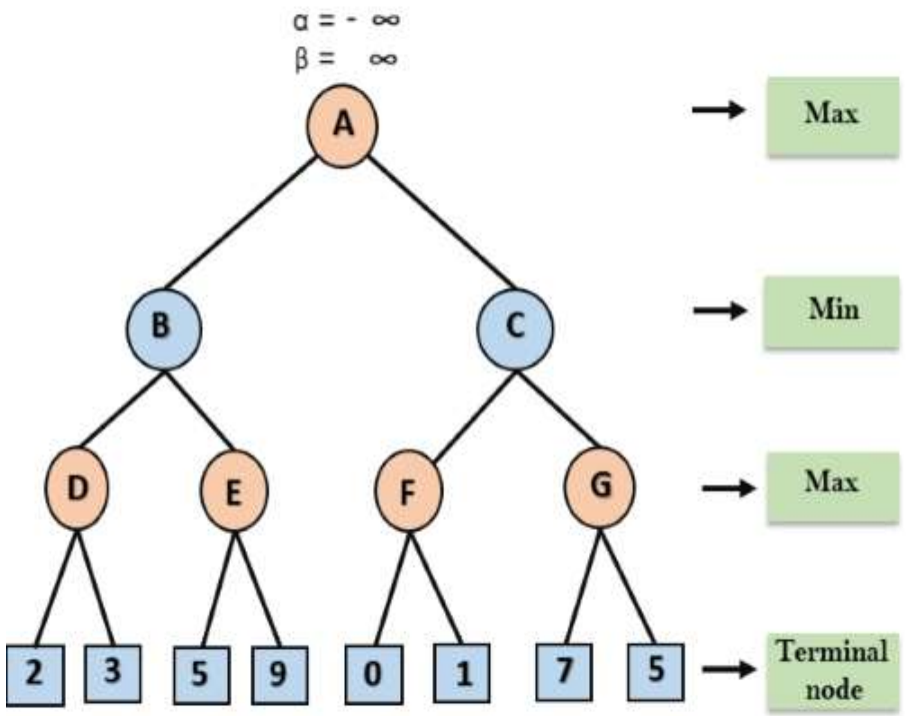
- Condition for Alpha-beta pruning:
  - The mAI(KCS-071)n condition which required for alpha-beta pruning is:

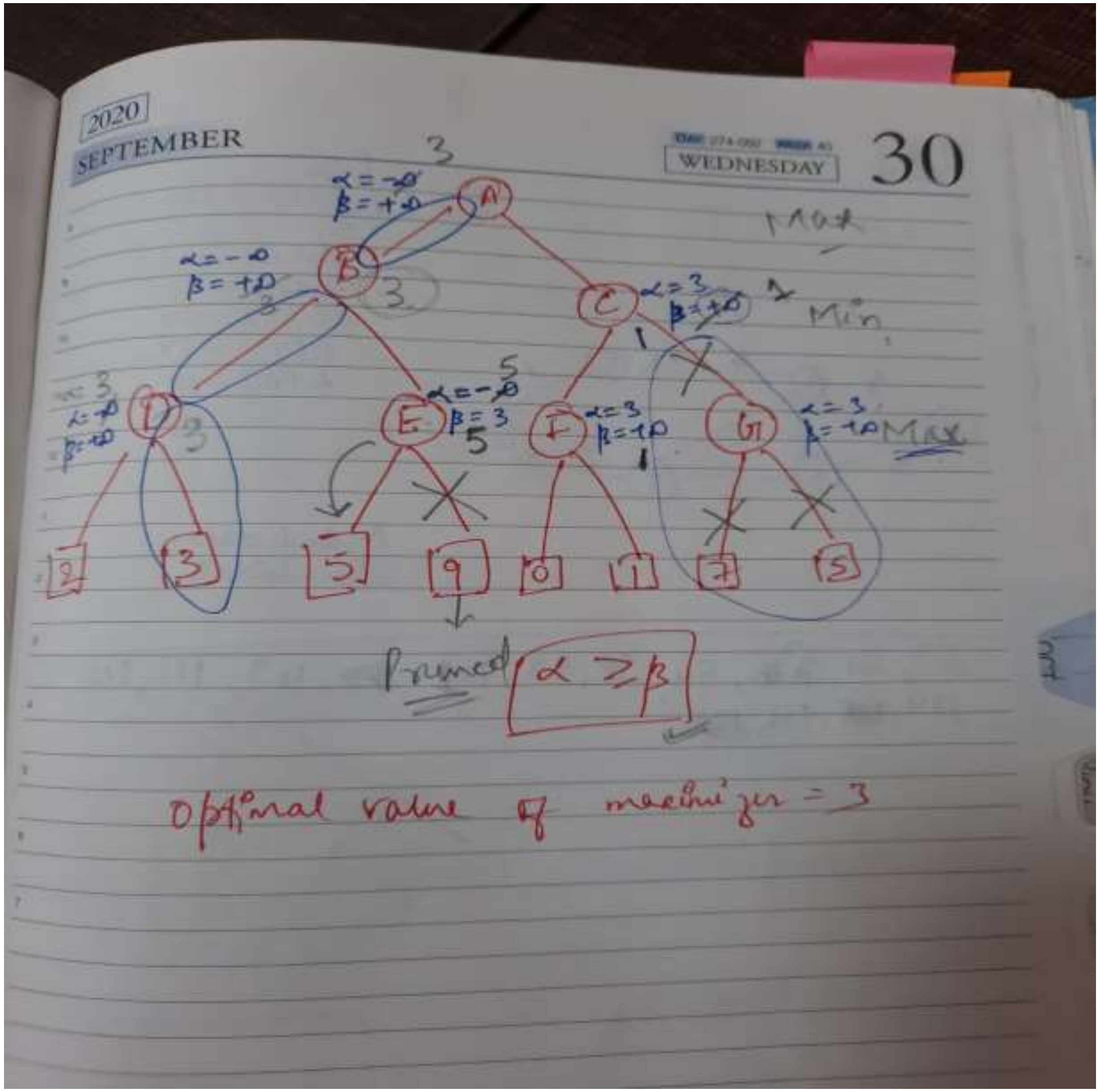
$$\alpha>=\beta$$

- Key points about alpha-beta pruning:
  - The Max player will only update the value of alpha.
  - The Min player will only update the value of beta.
  - While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
  - We will only pass the alpha, beta values to the child nodes.

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 agAI(KCS-071)n α= -∞ and β= +∞, and Node B passes the same value to its child D.



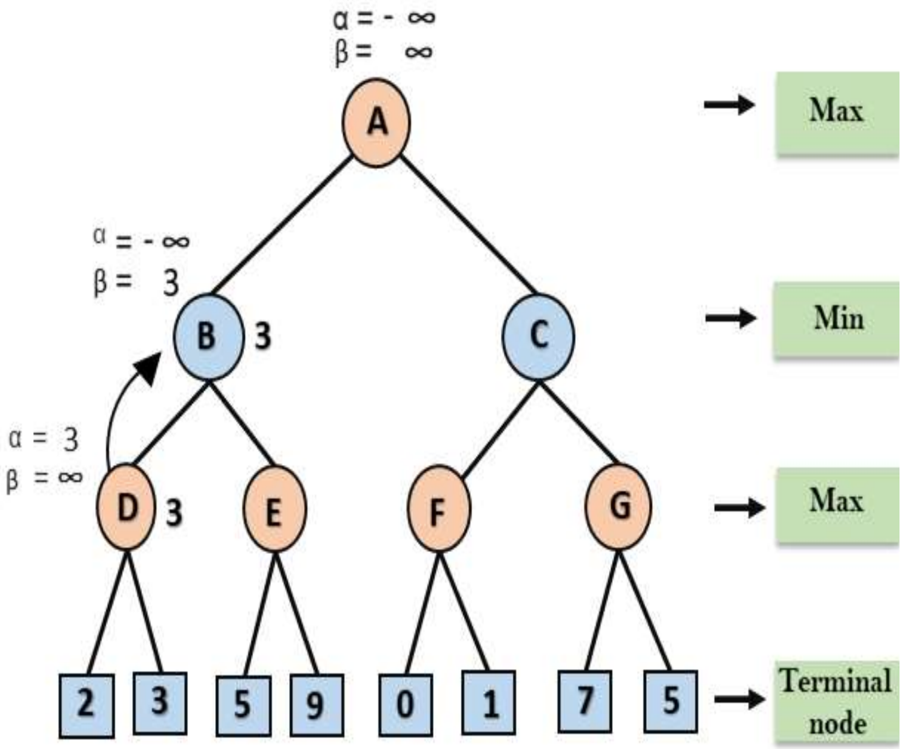


$\boxed{\alpha \geq \beta}$

Optimal value of maximizer = 3

Working of Alpha-Beta Pruning

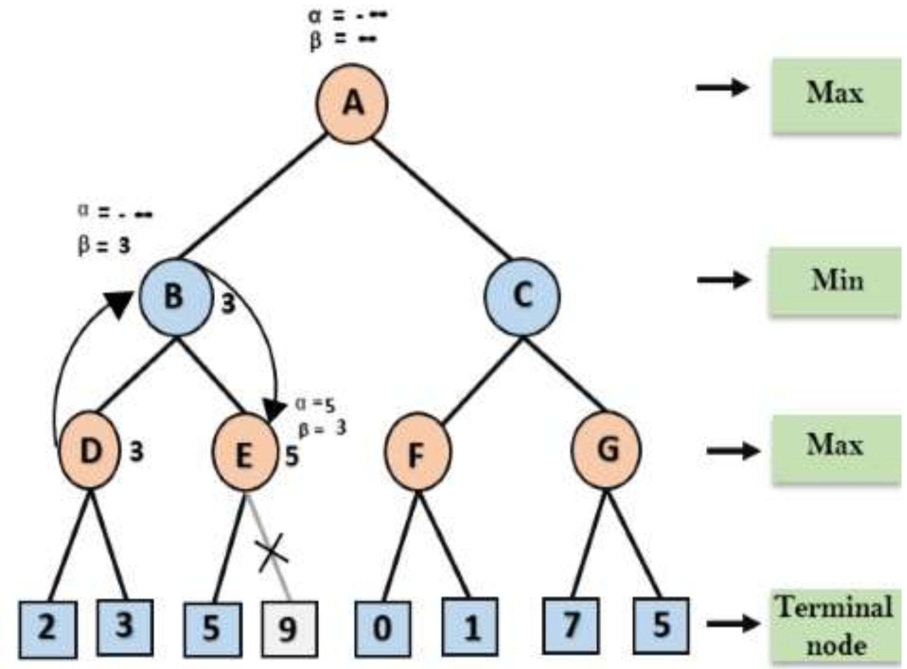
- **Step 2:** At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.
- **Step 3:** Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the avAI(KCS-071)lable 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.

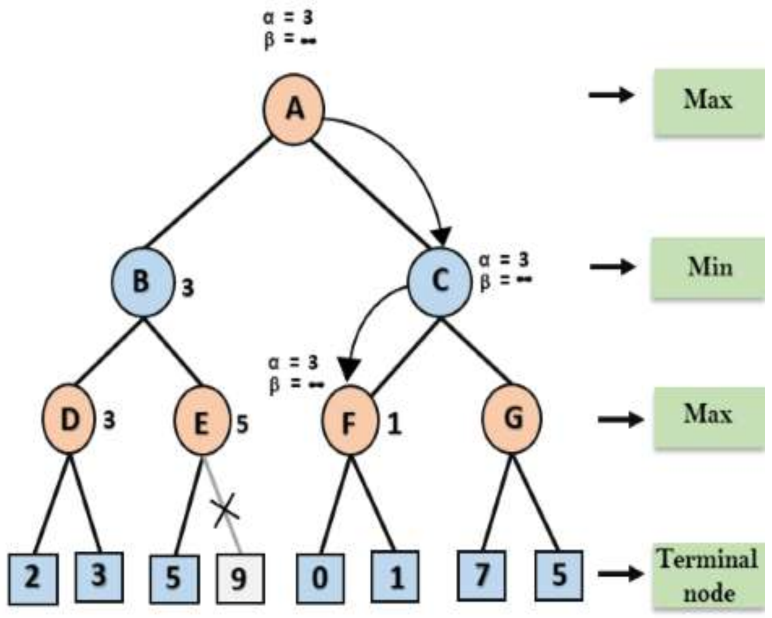
Working of Alpha-Beta Pruning

- **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.



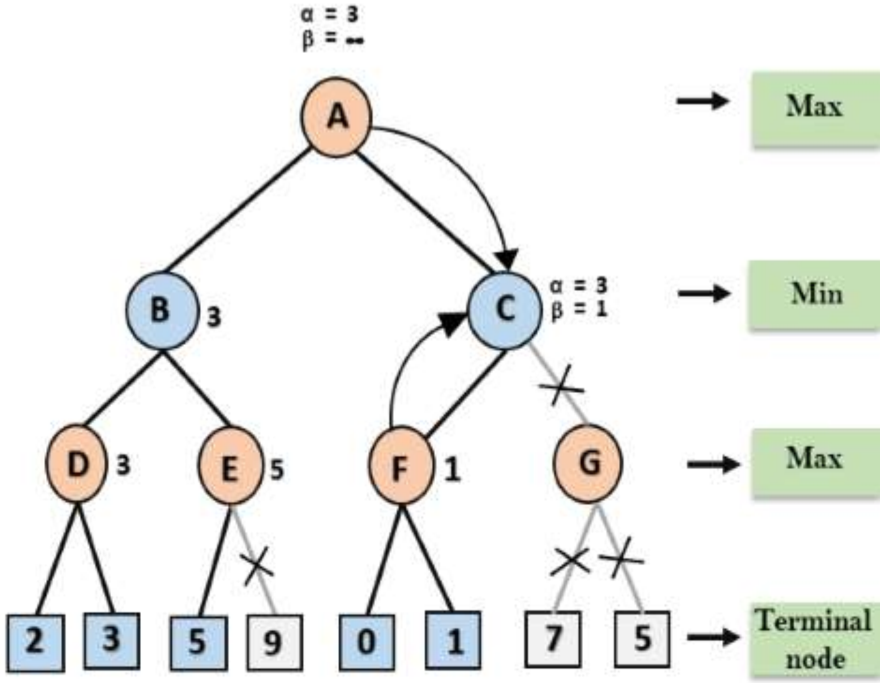
Working of Alpha-Beta Pruning

- **Step 5:** At next step, algorithm agAI(KCS-071)n backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum avAI(KCS-071)lable 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, agAI(KCS-071)n 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 remAI(KCS-071)ns 3, but the node value of F will become 1



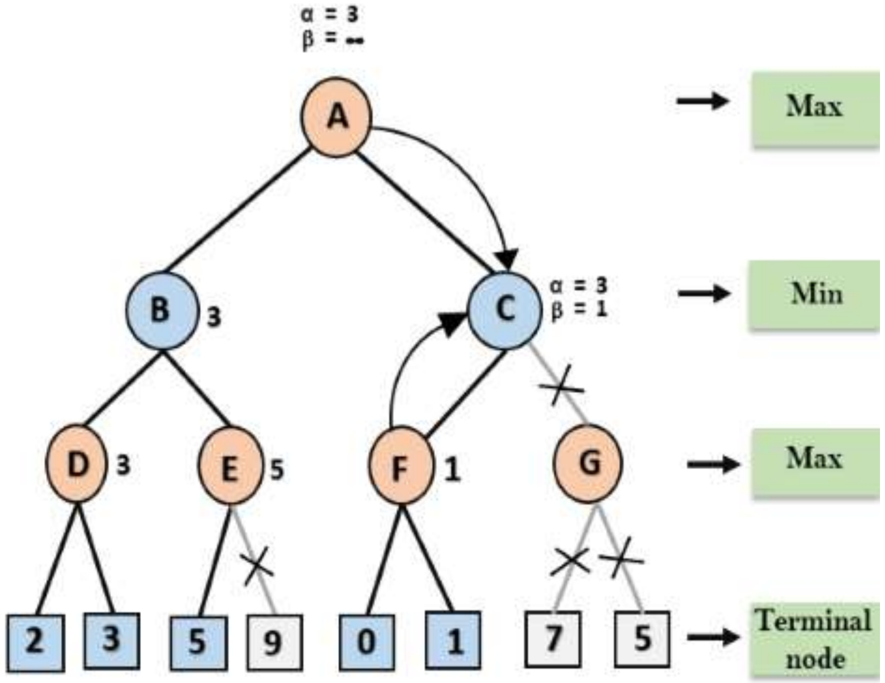
Working of Alpha-Beta Pruning

- **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 agAI(KCS-071)n 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.



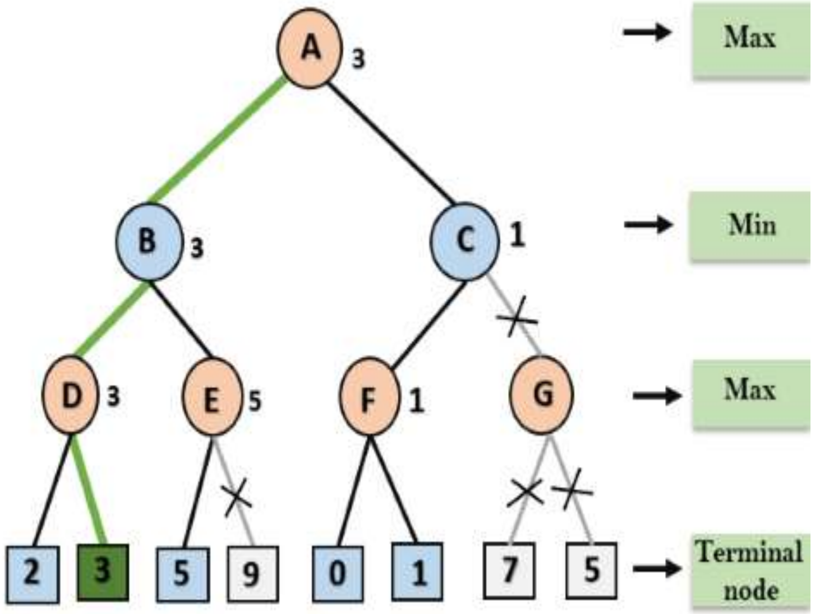
Working of Alpha-Beta Pruning

- **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 agAI(KCS-071)n 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.



Working of Alpha-Beta Pruning

- **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.


A 3
B 3
C 1
D 3
E 5
F 1
G
2
3
5
9
0
1
7
5
Max
Min
Max
Terminal node