UNIT-2

Searching

Searching for solutions

For finding the solution one can make use of explicit search tree that is generated by the initial state and the successor function that together define the state space. In general, we may have search graph rather than a search tree as the same state can be reached from multiple paths.

"A set of all possible states for a given problem is known as the state space of the problem".

Consider a example of making a coffee:

If one wants to make a cup of coffee, then state space representation of this problem can be done as follows:

1) Analyze the problem, i.e., verify whether the necessary ingredients like instant coffee powder, milk powder, sugar, kettle, stove etc., are available or not.

2) If ingredients are available then steps to solve the problem are -

i) Boil half cup of water in the kettle.

ii) Take some of the boiled water in a cup add necessary amount of instant coffee powder to make decoction.

iii) Add milk powder to the remaining boiling water to make milk.

iv) Mix decoction and milk.

v) Add sufficient quantity of sugar to your taste and the coffee is ready.

3) Now, by representing all the steps done sequentially we can make state space representation of this problem as shown in Fig.

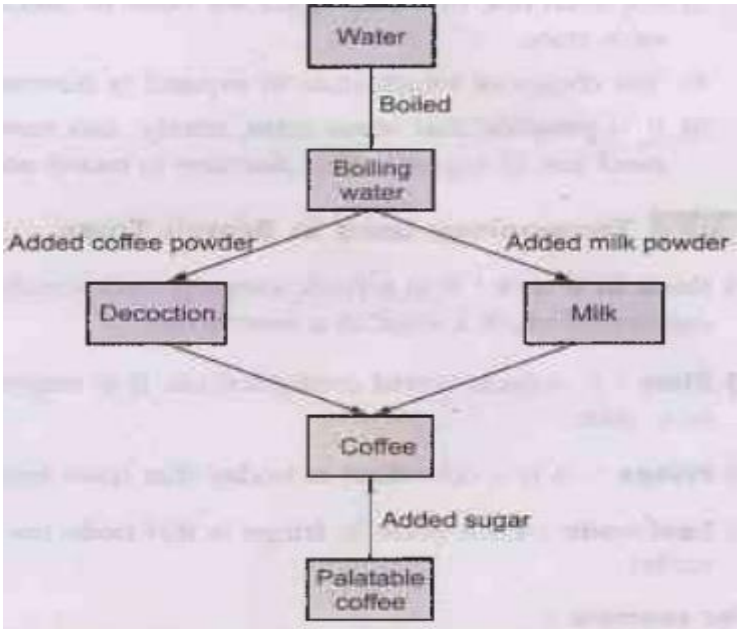


Fig: State space representation of coffee making

4) Westarted with the ingredients (initial state), followed by a sequence of steps (called states) and at last had a cup of coffee (goal state).

We added only needed amount of coffee powder, milk powder and sugar (operators). Thus, every AI problem has to do the process of searching for the solution steps as they are not explicit in nature. This searching is needed for solution steps are not known beforehand and have to be found out. Basically to do a search process, the following are needed -

i) The initial state description of the problem.

ii) A set of legal operators that changes the state.

iii) The final or the goal state.

Given these, searching can be defined, as a sequence of steps that transforms the initial state to the goal state. In other words, we can say that a problem can be defined as a state space search.

Construction of State Space

1) The root of search tree is a search node corresponding to initial state. In this state only we can check if goal is reached.

2) If goal is not reached we need to consider another state. Such a can be done by to expanding from the current state by applying successor function which generates new state. From this we may get multiple states.

3) For each one of these, again we need to check goal test or else repeat expansion of each state.

4) The choice of which state to expand is determined by the search strategy.

5) It is possible that some state, surely, can never lead to goal state. Such a state we need not to expand. This decision is based on various conditions of the problem.

Terminology used in Search Trees

1) Node in a tree: It is a book keeping data structure to represent the structure configuration of a state in a search tree.

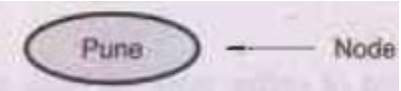
2) State: It reflects world configuration. It is mapping of state and action to another new state.

3) Fringe: It is a collection of nodes that have been generated but not yet expanded.

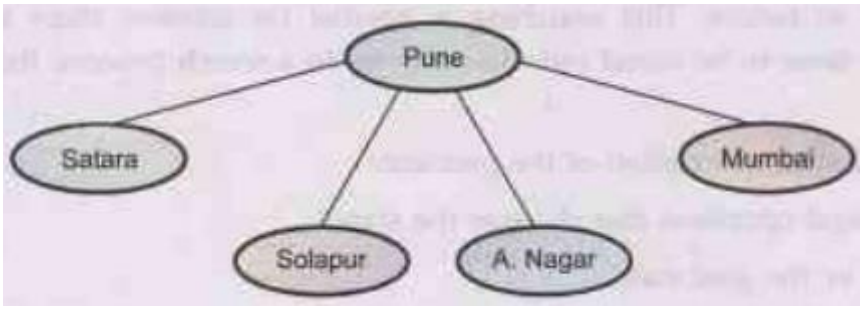
4) Leaf node: Each node in fringe is leaf node (as it does not have further successor node).

For example:

a) The initial state



b) After expanding Pune



c) After expanding Satara

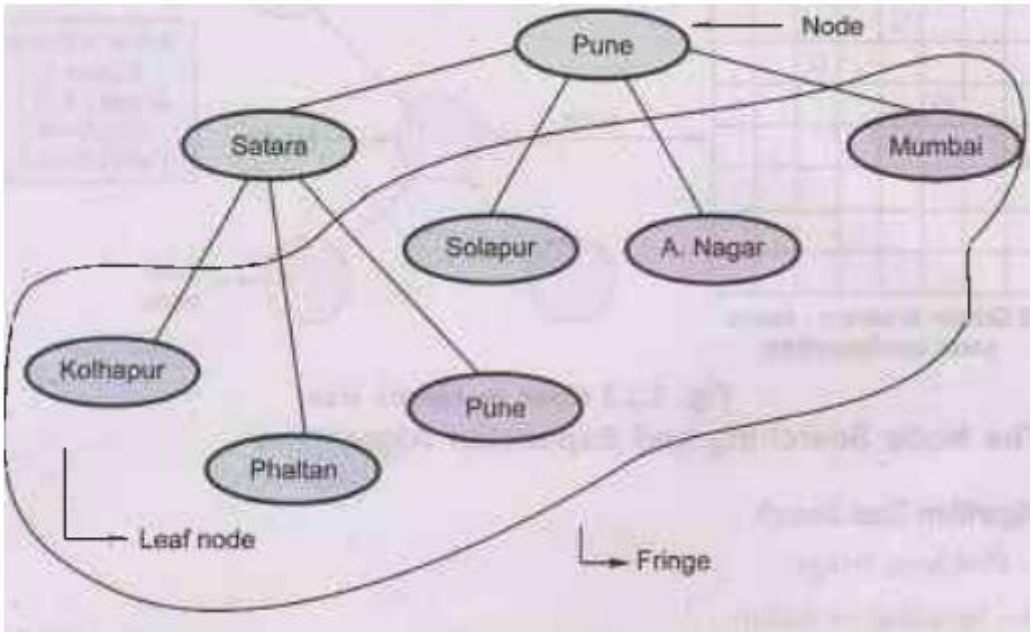


Fig: Partial search trees for finding root from Pune to Mumbai

5) Search strategy: It is a function that selects the next node to be expanded from current fringe. Strategy looks for best node for further expansion.

For finding best node each node needs to be examined. If fringe has many nodes then it would be computationally expensive.

The collection of un-expanded nodes (fringe) is implemented as queue, provided with, the sets of operations to work with queue. These operations can be CREATE Queue, INSERT in Queue, DELETE from Queue and all necessary operations which can be used for general tree-search algorithm.

Uninformed search strategies

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search. These algorithms can only generate the successors and differentiate between the goal state and non-goal state.

Following are the various types of uninformed search algorithms:

1. Breadth-first Search
2. Depth-first Search

3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search
6. Bidirectional Search

Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor 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:

- 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:

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:

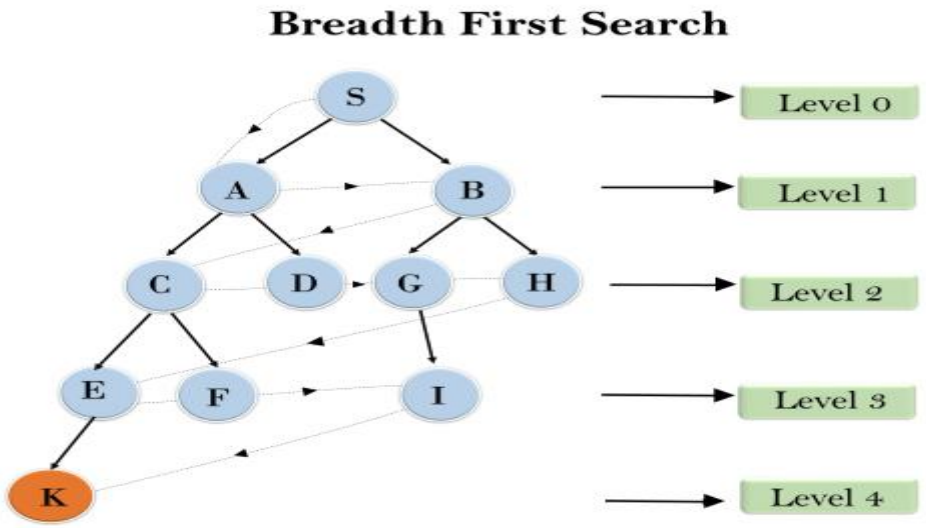
S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K

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.

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

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

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

Advantage:

- 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:

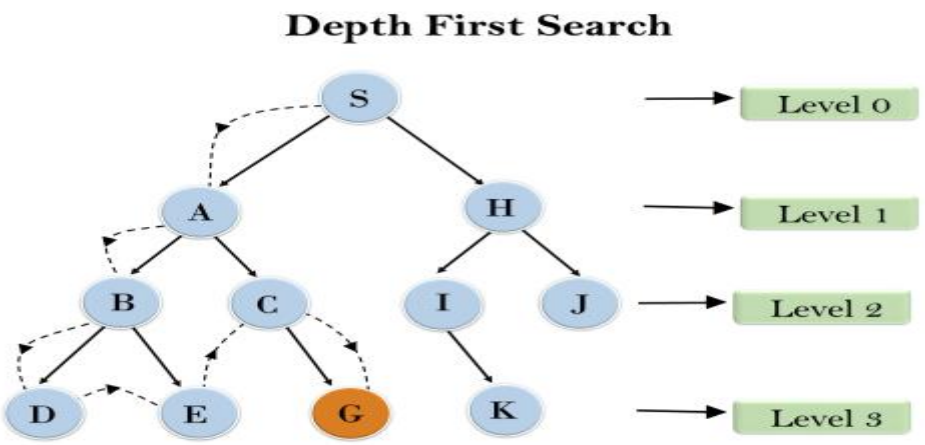
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

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

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



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.

Search with partial information (Heuristic search) Hill climbing

Hill climbing is a simple optimization algorithm used in Artificial Intelligence (AI) to find the best possible solution for a given problem. It belongs to the family of local search algorithms and is often used in optimization problems where the goal is to find the best solution from a set of possible solutions.

- In Hill Climbing, the algorithm starts with an initial solution and then iteratively makes small changes to it in order to improve the solution. These changes are based on a heuristic function that evaluates the quality of the solution. The algorithm continues to make these small changes until it reaches a local maximum, meaning that no further improvement can be made with the current set of moves.
- There are several variations of Hill Climbing, including steepest ascent Hill Climbing, first-choice Hill Climbing, and simulated annealing. In steepest ascent Hill Climbing, the algorithm evaluates all the possible moves from the current solution and selects the one that leads to the best improvement. In first-choice Hill Climbing, the algorithm randomly selects a move and accepts it if it leads to an improvement, regardless of whether it is the best move. Simulated

annealing is a probabilistic variation of Hill Climbing that allows the algorithm to occasionally accept worse moves in order to avoid getting stuck in local maxima.

Hill Climbing can be useful in a variety of optimization problems, such as scheduling, route planning, and resource allocation. However, it has some limitations, such as the tendency to get stuck in local maxima and the lack of diversity in the search space. Therefore, it is often combined with other optimization techniques, such as genetic algorithms or simulated annealing, to overcome these limitations and improve the search results.

Advantages of Hill Climbing algorithm:

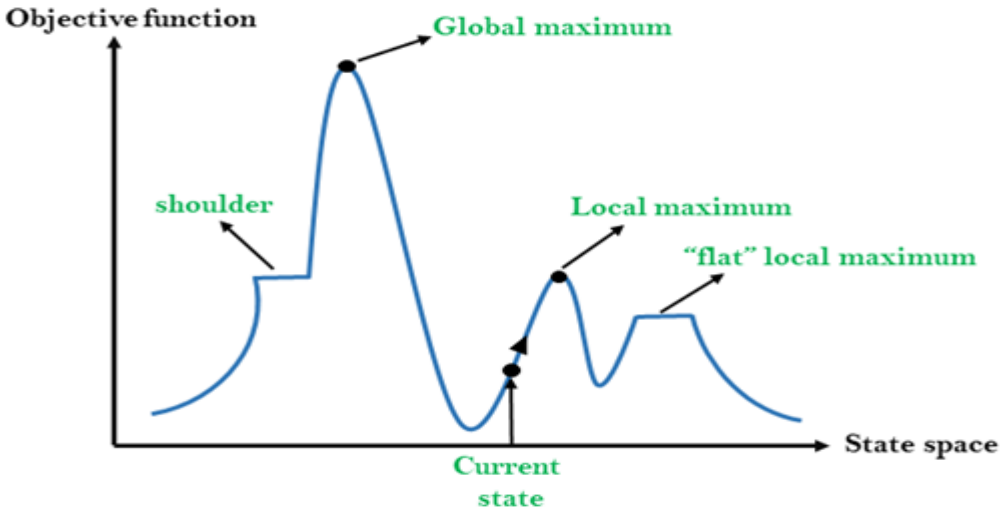
1. Hill Climbing is a simple and intuitive algorithm that is easy to understand and implement.
2. It can be used in a wide variety of optimization problems, including those with a large search space and complex constraints.
3. Hill Climbing is often very efficient in finding local optima, making it a good choice for problems where a good solution is needed quickly.
4. The algorithm can be easily modified and extended to include additional heuristics or constraints.

Features of Hill Climbing:

Following are some main features of Hill Climbing Algorithm:

- **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.

Ridge: It is a region that is higher than its neighbors but itself has a slope. It is a special kind of local maximum.

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:
- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:

Simple Hill Climbing:

Simple hill climbing 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

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:

a. If it is goal state, then return success and quit.

b. Else if it is better than the current state then assign new state as a current state.

c. Else if not better than the current state, then return to step2.

- **Step 5:** Exit.

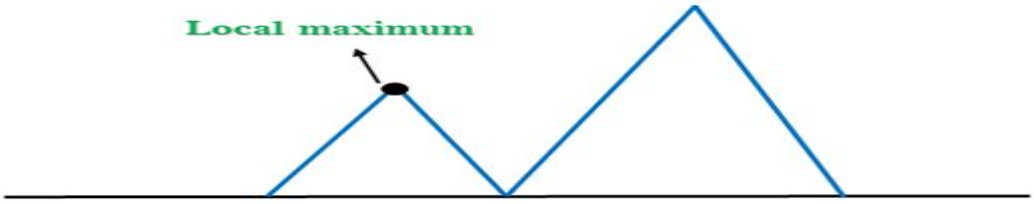
Stochastic hill climbing:

Stochastic hill climbing 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.

Problems in Hill Climbing Algorithm:

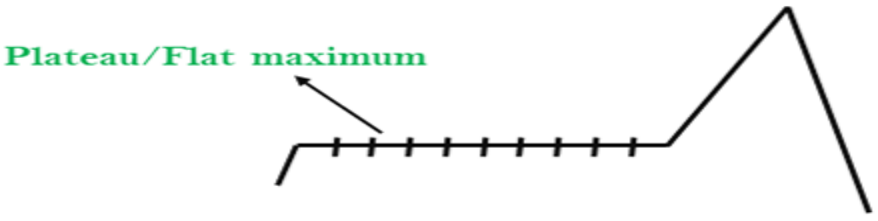
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

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.



2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains 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.



3. 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.

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.

A* Algorithm

A* (pronounced "A-star") is a powerful graph traversal and pathfinding algorithm widely used in artificial intelligence and computer science. It is mainly used to find the shortest path between two nodes in a graph, given the estimated cost of getting from the current node to the destination node. The main advantage of the algorithm is its ability to provide an optimal path by exploring the graph in a more informed way compared to traditional search algorithms such as Dijkstra's algorithm.

Algorithm A* combines the advantages of two other search algorithms: Dijkstra's algorithm and Greedy Best-First Search. Like Dijkstra's algorithm, A* ensures that the path found is as short as possible but does so more efficiently by directing its search through a heuristic similar to Greedy Best-First Search. A heuristic function, denoted h(n), estimates the cost of getting from any given node n to the destination node.

The main idea of A* is to evaluate each node based on two parameters:

1. **g(n):** the actual cost to get from the initial node to node n. It represents the sum of the costs of node n outgoing edges.
2. **h(n):** Heuristic cost (also known as "estimation cost") from node n to destination node n. This problem-specific heuristic function must be acceptable, meaning it never overestimates the actual cost of achieving the goal. The evaluation function of node n is defined as f(n) = g(n) h(n).

Algorithm A* selects the nodes to be explored based on the lowest value of f(n), preferring the nodes with the lowest estimated total cost to reach the goal. The A* algorithm works:

1. Create an open list of foundbut not explored nodes.

2. Create a closed list to hold already explored nodes.
3. Add a startingnode to the open list with an initial value of g
4. Repeat the following steps until the open list is empty or you reachthe target node:

a. Find the node with the smallest f-value (i.e., the node with the minor g(n) h(n)) in the open list.

b. Move the selected node from the open list to the closed list.

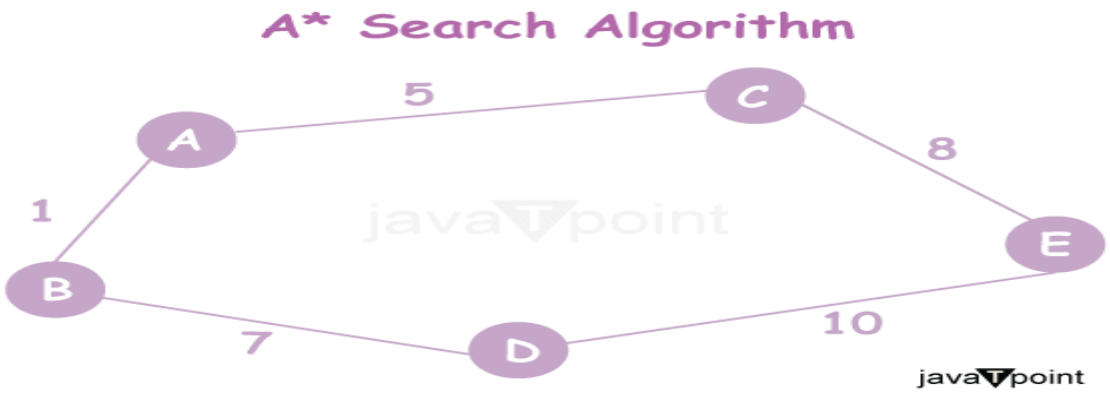
c. Createall valid descendantsof the selected node.

d. For each successor, calculateits g-value as the sum of the current node's g value and the cost of movingfrom the current node to the successor node. Update the g-value of the tracker when a better path is found.

e. If the followeris not in the open list, add it with the calculated g-value and calculate its h-value. If it is already in the open list, update its g value if the new path is better.

f. Repeat the cycle. Algorithm A* terminates when the target node is reached or when the open list empties, indicating no paths from the start node to the target node. The A* search algorithm is widely used in various fields such as robotics, video games, network routing, and design problems because it is efficient and can find optimal paths in graphs or networks.

However, choosing a suitable and acceptable heuristic function is essential so that the algorithm performs correctly and provides an optimal solution.



The A* (pronounced "letter A") search algorithm is a popular and widely used graph traversal algorithm in artificial intelligence and computer science. It is used to find the shortest path from a start node to a destination node in a weighted graph. A* is an informed search algorithm that uses heuristics to guide the search efficiently. The search algorithm A* works as follows:

The algorithm starts with a priority queue to store the nodes to be explored. It also instantiates two data structures g(n): The cost of the shortest path so far from the starting node to node n and h(n), the estimated cost (heuristic) from node n to the destination node. It is often a reasonable heuristic, meaning it never overestimates the actual cost of achieving a goal. Put the initial node in the priority queue and set its g(n) to 0. If the priority queue is not empty, Remove the node with the lowest f(n) from the priority queue. f(n) = g(n) h(n). If the deleted node is the destination node, the algorithm ends, and the path is found. Otherwise, expand the node and create its neighbors. For each neighbor node, calculate its initial g(n) value, which is the sum of the g value of the current node and the cost of moving from the current node to a neighboring node. If the neighbor node is not in priority order or the

original g(n) value is less than its current g value, update its g value and set its parent node to the current node. Calculate the f(n) value from the neighbor node and add it to the priority queue.

If the cycle ends without finding the destination node, the graph has no path from start to finish. The key to the efficiency of A* is its use of a heuristic function h(n) that provides an estimate of the remaining cost of reaching the goal of any node. By combining the actual cost g (n) with the heuristic cost h (n), the algorithm effectively explores promising paths, prioritizing nodes likely to lead to the shortest path. It is important to note that the efficiency of the A* algorithm is highly dependent on the choice of the heuristic function. Acceptable heuristics ensure that the algorithm always finds the shortest path, but more informed and accurate heuristics can lead to faster convergence and reduced search space.

Advantages of A* Search Algorithm in Artificial Intelligence

The A* search algorithm offers several advantages in artificial intelligence and problem-solving scenarios:

1. **Optimal solution:** A* ensures finding the optimal (shortest) path from the start node to the destination node in the weighted graph given an acceptable heuristic function. This optimality is a decisive advantage in many applications where finding the shortest path is essential.
2. **Completeness:** If a solution exists, A* will find it, provided the graph does not have an infinite cost This completeness property ensures that A* can take advantage of a solution if it exists.
3. **Efficiency:** A* is efficient ifan efficient and acceptable heuristic function is used. Heuristics guide the search to a goal by focusing on promising paths and avoiding unnecessary exploration, making A* more efficient than non-aware search algorithms such as breadth-first search or depth-first search.
4. **Versatility:** A* is widely applicable to variousproblem areas, including wayfinding, route planning, robotics, game development, and more. A* can be used to find optimal solutions efficiently as long as a meaningful heuristic can be defined.
5. **Optimized search:** A* maintains a priority order to select the nodes with the minor f(n) value (g(n) and h(n)) for expansion. This allows it to explore promising paths first, which reduces the search space and leads to faster convergence.
6. **Memory efficiency:** Unlike some other search algorithms, such as breadth-first search, A* stores only a limited number of nodes in the priority queue, which makes it memory efficient, especially for large graphs.
7. **Tunable Heuristics:** A*'s performancecan be fine-tuned by selecting different heuristic functions. More educated heuristics can lead to faster convergence and less expanded nodes.
8. **Extensively researched:** A* is a well-established algorithm with decades of research and practical applications. Many optimizations and variations have been developed, making it a reliable and well-understood troubleshooting tool.

9. **Web search:** A* can be used for web-based path search, where the algorithm constantly updates the path according to changes in the environment or the appearance of new It enables real-time decision-making in dynamic scenarios.

AO* Algorithm

The **AO* algorithm**, short for **"Anytime Optimistic"** algorithm, is a search algorithm used in artificial intelligence and computer science to find the optimal path or solution in a graph or state space. It is particularly useful in applications like robotics, pathfinding, and planning, where finding the best possible solution is essential.

The AO* algorithm belongs to the family of **informed search algorithms**, meaning it utilizes heuristics or estimated cost functions to guide the search process. It efficiently balances the trade-off between **computation time** and the **quality of the solution**. Unlike some other algorithms that focus solely on finding the optimal solution, AO* provides a series of progressively improving solutions, making it adaptable to various scenarios.

Key Features of AO* Algorithm:

1. **Anytime Nature:** AO* provides solutions incrementally, allowing users to interrupt the search at any time and retrieve the best solution found so far. This feature is crucial in real-time systems where immediate responses are necessary.
2. **Optimistic Search:** AO* maintains an "optimistic" cost estimate for each state, which serves as a lower bound on the true cost. This estimate helps prioritize promising paths during the search.
3. **Adaptive Behaviour:** AO* can adapt its search strategy based on the available computational resources and user requirements. It can allocate more time for an exhaustive search if needed or return a solution quickly when computational resources are limited.
4. **Heuristic Function:** Like other informed search algorithms, AO* relies on a heuristic function that estimates the cost from the current state to the goal state. This function guides the search by prioritizing states that appear more promising.

Working of AO* Algorithm

The **AO* algorithm** is designed to efficiently search through a state space or graph to find the optimal solution while providing the flexibility to return intermediate solutions at any time. Its operation can be broken down into several key steps:

1. Initialization

The algorithm begins with the initialization of critical components:

- **Start State:** It starts from the initial state, which represents the current state of the problem or the starting point in a graph.
- **Cost Estimates:** For each state, AO* maintains an "optimistic" cost estimate, denoted as g*(s), which serves as a lower bound on the true cost from the start state to that state. Initially, these cost estimates are set to infinity for all states except the start state, which has a cost estimate of zero.
- **Priority Queue:** AO* uses a priority queue (often implemented as a binary heap) to keep track of states that need to be expanded. States are prioritized in the queue based on their g*(s) values, with states having lower cost estimates being higher in priority.

2. Iterative Expansion

The core of AO* is an iterative process that repeatedly selects and expands the most promising state from the priority queue. This process continues until certain termination conditions are met. Here's how it works:

- **Selecting a State:** The algorithm selects the state with the lowest g*(s) value from the priority queue. This state represents the most promising path discovered so far.
- **Expanding a State:** Once a state is selected, AO* generates its successor states, which are the states reachable from the current state by taking valid actions or moving along edges in the graph. These successor states are generated and evaluated.
- **Updating Cost Estimates:** For each successor state, the algorithm updates its g*(s) value. The updated value depends on the cost of reaching that successor state from the current state and the g*(s) value of the current state.
- **Adding to Priority Queue:** The newly generated states, along with their updated g*(s) values, are added to the priority queue.

3. Termination

The search process continues until certain termination conditions are met. These conditions can include:

- **A predefined time limit:** The algorithm stops after a specified amount of time has elapsed.
- **A user request:** The user can request the algorithm to stop and return the best solution found so far.
- **The discovery of an optimal solution:** If AO* finds a solution that satisfies the problem constraints, it can terminate.

4. Solution Retrieval

One of the unique features of AO* is its ability to return solutions incrementally. At any point during the search, the user can decide to stop the algorithm and retrieve the best solution found so far. This flexibility is particularly valuable in real-time systems where immediate responses are required, and waiting for the optimal solution may not be feasible.

5. Adaptation

Another essential aspect of AO* is its adaptability. It can adjust its search strategy based on available computational resources and user requirements. If more time or computational power is available, AO* can perform a more exhaustive search to improve the solution quality. Conversely, if resources are limited, it can return a solution quickly without completing the entire search.

Example: Robotic Navigation

In the field of robotics, AO* can be used for path planning and navigation in dynamic environments. Let's consider a scenario where a robot needs to navigate through an environment with moving obstacles.

1. Initialization:

- **Start State:** The robot's current position is the initial state.

- **Cost Estimates:** Initially, cost estimates for states are set based on the distance from the current position to possible goal positions.
- **Priority Queue:** The priority queue is initialized with the robot's current position.

2. Iterative Expansion:

- The algorithm selects the state with the lowest g*(s) value from the priority queue, which is the robot's current position initially.
- It generates successor states by simulating possible movements of the robot.
- The cost estimates for these successor states are updated based on the estimated time or energy required to reach those states.
- The successor states and their updated cost estimates are added to the priority queue.

3. Termination:

- The search process continues until a termination condition is met. This could be when the robot reaches the goal, when a user intervention occurs, or when a timeout occurs.

4. Solution Retrieval:

- AO* allows the robot to start moving toward the goal based on the best path found so far, even if it's not the optimal path. This is particularly useful in scenarios where the environment is changing, and the robot needs to adapt its path in real time.

Advantages of AO* Algorithm

1. **Adaptability:** AO* can adapt to changing requirements and computational resources, making it suitable for real-time systems.
2. **Incremental Solutions:** It provides incremental solutions, allowing users to make progress while the search continues.
3. **Optimistic Estimates:** The use of optimistic cost estimates can guide the search efficiently.
4. **Heuristic Guidance:** Like A*, it benefits from heuristic guidance, improving search efficiency.

Difference between A* Algorithm and AO* Algorithm

Aspect	A* Algorithm	AO* Algorithm
Anytime Nature	No	Yes (Provides incremental solutions)
Optimistic Cost Estimates	Single cost estimate based on heuristics	Optimistic cost estimates for each state
Adaptability	Fixed-depth search	Adapts to available time and resources
Prioritization	Always seeks the single optimal path	Balances quality vs. time efficiently
Solution Completeness	Finds the single optimal solution	Provides a series of progressively improving solutions
Termination	Typically continues until optimal	Can be terminated at any time, returning the

Aspect	A* Algorithm	AO* Algorithm
Condition	solution found	best solution found so far
Use Cases	Used when finding the absolute best solution is top priority	Preferred in real-time systems and applications requiring adaptability
Real-time Applications	Less suitable for real-time systems	Widely used in robotics, video games, network routing, autonomous vehicles, etc.
Quality vs. Time Trade-off	Prioritizes solution quality over time efficiency	Balances between solution quality and computation time efficiently
Search Strategy	Fixed-depth search with no adaptability	Adapts search strategy based on available computational resources

Problem reduction

In the field of artificial intelligence, problem reduction is a crucial concept that plays a fundamental role in problem-solving. It involves breaking down a complex problem into smaller, more manageable subproblems to find a solution. This approach enables artificial intelligence systems to handle complex tasks more efficiently and effectively. Problem reduction is a crucial technique in artificial intelligence that enables systems to apply specific problem-solving methods to each subproblem, making it easier to find a solution for the overall problem. This technique allows AI systems to handle a wide range of problems, from simple to highly complex ones, by dividing them into manageable parts. This approach enables efficient use of computational resources and reduces the complexity of solving complex problems. Problem reduction also promotes modularity and reusability in AI systems. Once a subproblem is solved, the solution can be reused for similar problems, saving time and computational resources.

Furthermore, problem reduction enables AI systems to handle uncertainty and incomplete information by focusing on specific aspects of a problem. In conclusion, problem reduction is a critical concept in artificial intelligence that enables the effective and efficient solving of complex problems. It allows AI systems to divide problems into manageable subproblems, apply problem-solving techniques to each subproblem, and promote modularity and reusability. Overall, problem reduction plays a vital role in advancing the capabilities of artificial intelligence systems.

Problem Reduction Techniques in AI

In artificial intelligence, problem reduction is a powerful technique used to break down complex problems into smaller, more manageable subproblems, illustrating the concept of problem solving in artificial intelligence. This approach allows AI systems to solve problems by breaking them down into simpler, solvable components. The basic idea behind problem reduction is to identify a large problem and then repeatedly break it down into smaller, more solvable problems until a base case is reached. Each subproblem is solved independently, and the solutions are combined to form a solution to the original problem. There are several problem reduction techniques commonly used in AI, including 1. Subgoal decomposition: This technique involves decomposing a problem into subgoals, where each subgoal represents a simpler task or problem to be solved. Subgoal decomposition allows the AI system to focus on solving one subgoal at a time, gradually working toward the solution of the main problem. 2. Means-ends analysis: Means-ends analysis is a problem-solving approach that involves identifying the difference between the current state and the goal state, and then finding ways to reduce that difference step by step.

The AI system uses a set of operators or actions to move from the current state to the goal state, iteratively narrowing the problem until a solution is found. 3. Divide and Conquer: This technique involves dividing a problem into smaller, independent subproblems, solving each subproblem

separately, and then combining the solutions to obtain the final solution. Divide and conquer is particularly useful for large problems because it allows for parallelization and efficient computation. 4. Macro Operators: Macro operators are predefined procedures or sequences of actions that simplify the problem-solving process. These higher-level operators allow the AI system to treat a complex task as a single step, reducing the complexity of the problem and improving efficiency. Overall, problem reduction techniques play a critical role in artificial intelligence, enabling AI systems to tackle complex problems by breaking them down into smaller, more manageable components. These techniques provide a systematic and efficient approach to problem solving, enhancing the capabilities of AI systems in various domains.

Advantages of Problem Reduction Approaches

Problem reduction is a fundamental concept in artificial intelligence that offers several advantages for solving complex problems. In this approach, a complex problem is broken down into smaller, more manageable subproblems, making it easier to find a solution. One of the main benefits of problem reduction approaches is their ability to simplify the problem-solving process. By breaking down a problem into smaller pieces, each component becomes easier to understand and analyze. This allows AI systems to focus their efforts on solving specific subproblems, increasing the efficiency and effectiveness of the overall solution. Another advantage of problem reduction approaches is their flexibility. This approach allows AI systems to adapt and adjust their problem-solving strategies based on the specific requirements of each subproblem. It enables the system to switch between different problem-solving methods and algorithms, optimizing the solution for each particular subproblem. In addition, problem reduction approaches promote modularity and reusability. By decomposing a problem into smaller subproblems, it becomes possible to reuse solutions from previous subproblems in different contexts. This not only saves time and effort, but also facilitates knowledge transfer and sharing within the AI system.

Furthermore, problem reduction approaches facilitate collaboration and cooperation between different AI systems or agents. By decomposing a complex problem into smaller subproblems, it becomes easier to assign specific tasks to different agents or systems, allowing them to work together toward a common goal. In summary, problem reduction approaches offer several advantages for artificial intelligence systems. They simplify the problem-solving process, increase flexibility and adaptability, promote modularity and reusability, and facilitate collaboration and cooperation. By exploiting these advantages, AI systems can solve complex problems more effectively and efficiently.

Game Playing:

Game Playing is an important domain of artificial intelligence. Games don't require much knowledge; the only knowledge we need to provide is the rules, legal moves and the conditions of winning or losing the game. Both players try to win the game. So, both of them try to make the best move possible at each turn. Searching techniques like BFS(Breadth First Search) are not accurate for this as the branching factor is very high, so searching will take a lot of time. So, we need another search procedures that improve –

- **Generate procedure** so that only good moves are generated.
- **Test procedure** so that the best move can be explored first.

Game playing is a popular application of artificial intelligence that involves the development of computer programs to play games, such as chess, checkers, or Go. The goal of game playing in artificial intelligence is to develop algorithms that can learn how to play games and make decisions that will lead to winning outcomes.

1. One of the earliest examples of successful game playing AI is the chess program Deep Blue, developed by IBM, which defeated the world champion Garry Kasparov in 1997. Since then, AI has been applied to a wide range of games, including two-player games, multiplayer games, and video games.

There are two main approaches to game playing in AI, rule-based systems and machine learning-based systems.

1. **Rule-based systems** use a set of fixed rules to play the game.
2. **Machine learning-based systems** use algorithms to learn from experience and make decisions based on that experience.

In recent years, machine learning-based systems have become increasingly popular, as they are able to learn from experience and improve over time, making them well-suited for complex games such as Go. For example, AlphaGo, developed by DeepMind, was the first machine learning-based system to defeat a world champion in the game of Go.

Game playing in AI is an active area of research and has many practical applications, including game development, education, and military training. By simulating game playing scenarios, AI algorithms can be used to develop more effective decision-making systems for real-world applications.

The most common search technique in game playing is Minimax search procedure. It is depth-first depth-limited search procedure. It is used for games like chess and tic-tac-toe.

Minimax algorithm uses two functions:
MOVEGEN : It generates all the possible moves that can be generated from the current position.
STATICEVALUATION : It returns a value depending upon the goodness from the viewpoint of two-player

This algorithm is a two player game, so we call the first player as PLAYER1 and second player as PLAYER2. The value of each node is backed-up from its children. For PLAYER1 the backed-up value is the maximum value of its children and for PLAYER2 the backed-up value is the minimum value of its children. It provides most promising move to PLAYER1, assuming that the PLAYER2 has make the best move. It is a recursive algorithm, as same procedure occurs at each level.

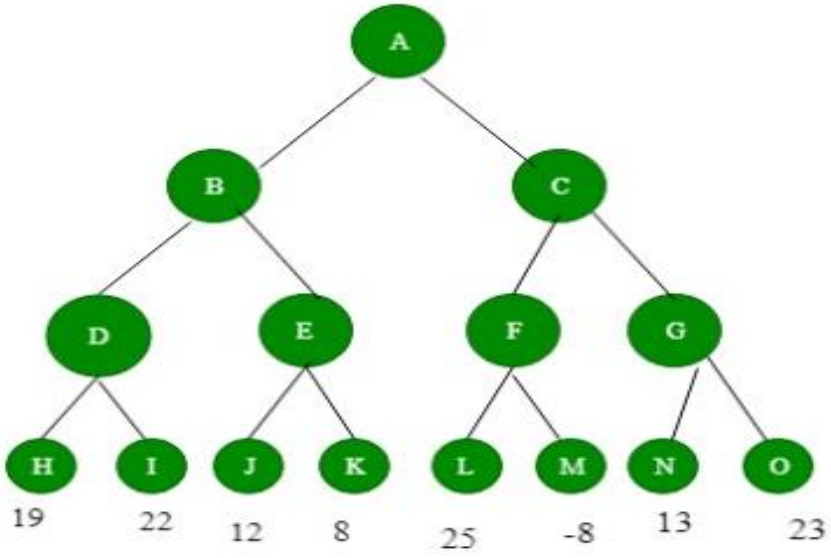


Figure 1: Before backing-up of values

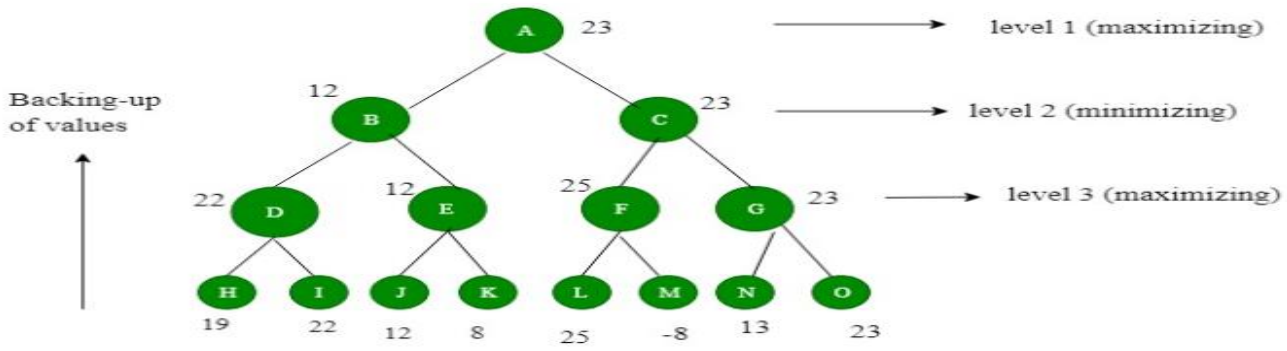


Figure 2: After backing-up of values We assume that PLAYER1 will start the game.

4 levels are generated. The value to nodes H, I, J, K, L, M, N, O is provided by STATICEVALUATION function. Level 3 is maximizing level, so all nodes of level 3 will take maximum values of their children. Level 2 is minimizing level, so all its nodes will take minimum values of their children. This process continues. The value of A is 23. That means A should choose C move to win.

Adversial 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 against us.

- In previous topics, we have studied the search strategies which are only associated with a single agent that aims to find the solution which often expressed in the form of a sequence of actions.
- 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 against 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 modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

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 have 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.

Formalization of the problem:

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

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called 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.

Example: Tic-Tac-Toe game tree:

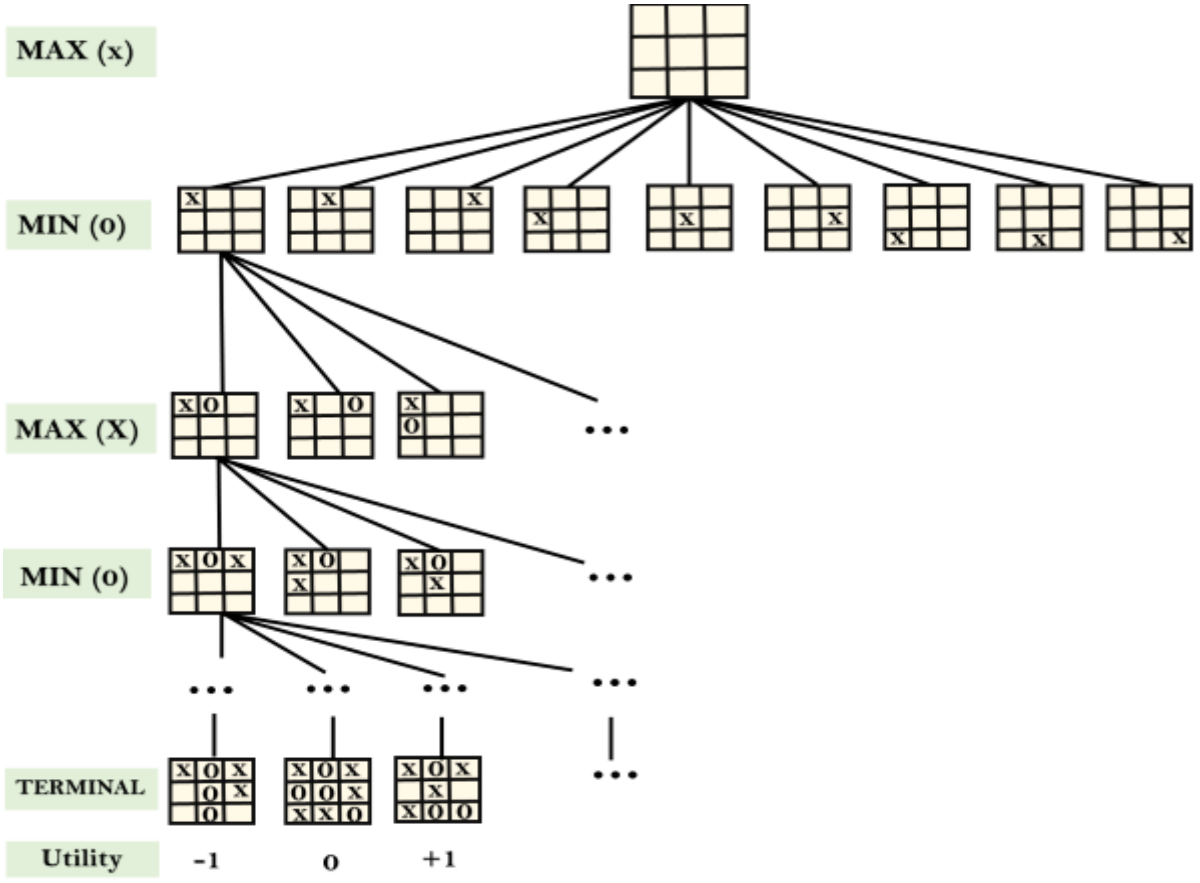
The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

- 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 against 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 against 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.



Hence adversarial Search for the minimax procedure works as follows:

- It aims 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 prefer to move to a state of minimum value then:

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

Games:

AI in gaming refers to responsive and adaptive video game experiences. These AI-powered interactive experiences are usually generated via non-player characters, or NPCs, that act intelligently or creatively, as if controlled by a human game-player. AI is the engine that determines an NPC's behavior in the game world.

While AI in some form has long appeared in video games, it is considered a booming new frontier in how games are both developed and played. AI games increasingly shift the control of the game experience toward the player, whose behavior helps produce the game experience.

AI procedural generation, also known as procedural storytelling, in game design refers to game data being produced algorithmically rather than every element being built specifically by a developer.

Why Does AI in Gaming Matter?

AI in gaming is all about enhancing a player's experience. It is especially important as developers deliver gaming experiences to different devices. No longer is gaming simply a choice between console or desktop computer. Rather, players expect immersive game experiences on a vast array of mobile and wearable devices, from smartphones to VR headsets, and more. AI enables developers to deliver console-like experiences across device types.

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.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. 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 the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm 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.

Pseudo-code for MinMax Algorithm:

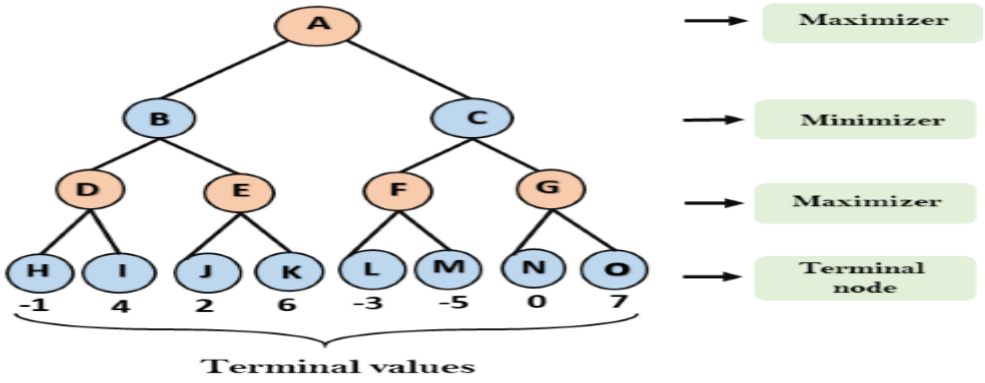
```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

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
```

Initial call:
Minimax(node, 3, true)

Working of Min-Max Algorithm:

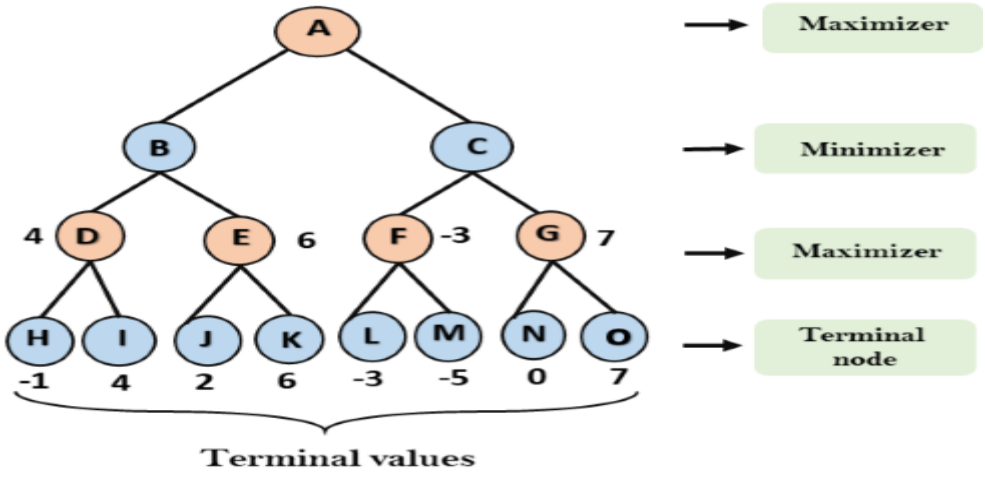
- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum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

Step-1: In the first step, the algorithm generates the entire game-tree and applies 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.



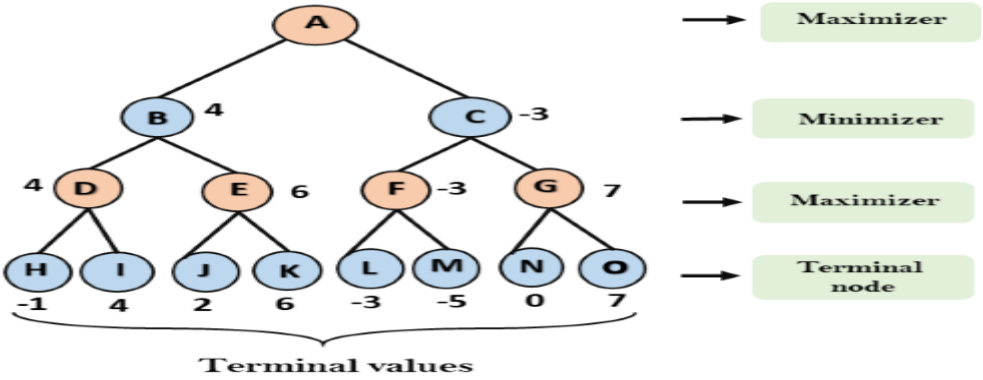
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- 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



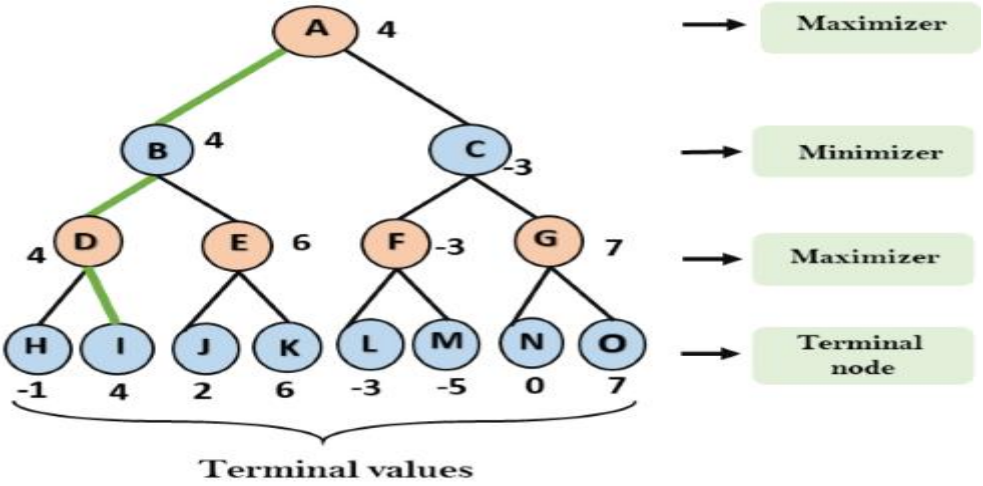
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

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



Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. 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



That was the complete workflow of the minimax two player game.

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)**.

Optimal decisions in multiplayer games:

Optimal decision-making in games is a fundamental challenge in the field of Artificial Intelligence (AI). Various techniques and approaches have been developed to enable AI agents to make optimal decisions in games. Here are some key concepts and methods:

Minimax Algorithm:

- Minimax is a decision-making algorithm commonly used in two-player, zero-sum games (e.g., chess, tic-tac-toe).
- It involves creating a game tree that represents all possible moves and counter-moves for both players.
- The algorithm seeks to minimize the maximum possible loss (hence the name "minimax") by selecting the best move at each turn.

Alpha-Beta Pruning:

- Alpha-beta pruning is an optimization technique used in conjunction with the minimax algorithm.
- It reduces the number of nodes evaluated in the game tree by eliminating branches that are guaranteed to be suboptimal.
- This technique can significantly speed up the search process and make it feasible to search deeper into the game tree.

Monte Carlo Tree Search (MCTS):

- MCTS is a popular technique for making decisions in games with large branching factors and complex decision spaces.
- It combines random simulations (rollouts) with tree exploration to focus on promising branches of the game tree.
- MCTS has been highly successful in games like Go and has led to breakthroughs in AI game playing.

Reinforcement Learning (RL):

- RL involves training agents to make decisions by interacting with an environment and receiving rewards based on their actions.
- Agents learn to optimize their decisions over time to maximize cumulative rewards.
- Deep Reinforcement Learning (DRL) uses neural networks to approximate optimal decision policies.

Q-Learning:

- Q-learning is a model-free reinforcement learning technique where an agent learns to make decisions by updating a Q-value function.
- The Q-values represent the expected cumulative rewards for taking certain actions in specific states.
- Q-learning is well-suited for discrete action spaces and has been used in games like checkers and backgammon.

Policy Gradient Methods:

- Policy gradient methods aim to directly learn a policy (a mapping from states to actions) that maximizes expected rewards.
- These methods are suitable for both discrete and continuous action spaces.
- They have been applied to games like poker and continuous control tasks.

Evolutionary Algorithms:

- Evolutionary algorithms involve generating and evolving a population of candidate solutions over multiple generations.
- They have been used for game playing by evolving strategies and decision policies.
- Evolutionary algorithms are adaptable to a wide range of game scenarios.

Heuristic Search and Pattern Databases:

- For games with large state spaces, heuristic search methods and pattern databases can be used to estimate the optimal value of game states.
- These techniques provide approximate solutions that guide decision-making.

Neural Network-based Approaches:

- Deep learning methods, such as neural networks, have been applied to various aspects of game playing, including decision-making.
- Neural networks can learn to approximate decision functions, predict outcomes, and model opponent behavior.

Domain-Specific Knowledge and Expert Systems:

- In some games, domain-specific knowledge and expert systems can be used to guide decision-making.
- These systems encode human expertise and strategies to help the AI agent make informed decisions.

Ultimately, the choice of approach depends on the nature of the game, the complexity of the decision space, the available computational resources, and the specific goals of the AI agent. Many modern AI game-playing systems combine multiple techniques to achieve optimal decisions in complex game environments.

Problem in Game playing:

1. **Limited scope:** The techniques and algorithms developed for game playing may not be well-suited for other types of applications and may need to be adapted or modified for different domains.
2. **Computational cost:** Game playing can be computationally expensive, especially for complex games such as chess or Go, and may require powerful computers to achieve real-time performance.

Alpha-Beta Pruning:

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.

- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence 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**. It is also called as **Alpha-Beta Algorithm**.
- 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:
    a. **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 -∞.
    b. **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 +∞.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning is:

$$\alpha >= \beta$$

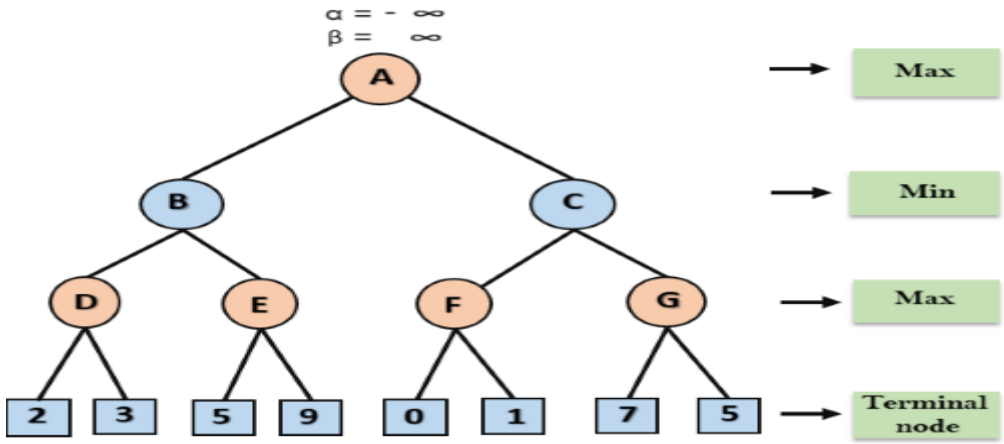
Key points about alpha-beta pruning:

- 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:

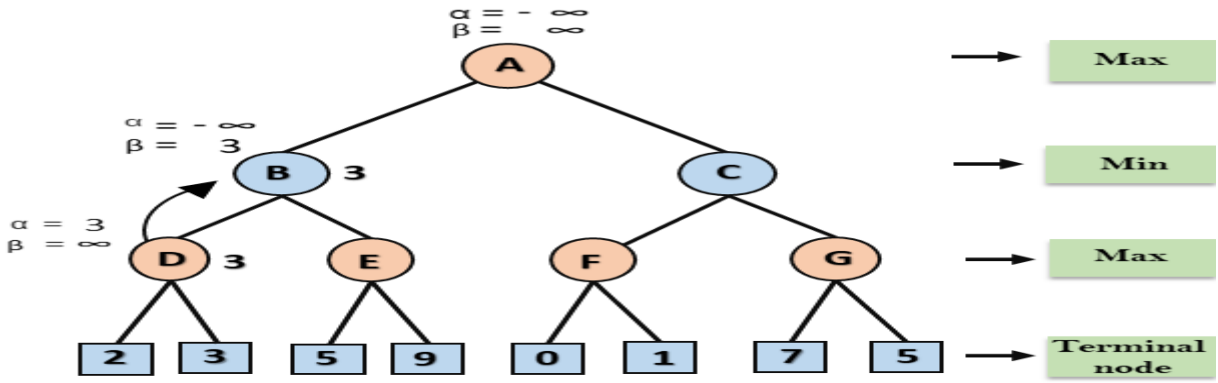
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

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



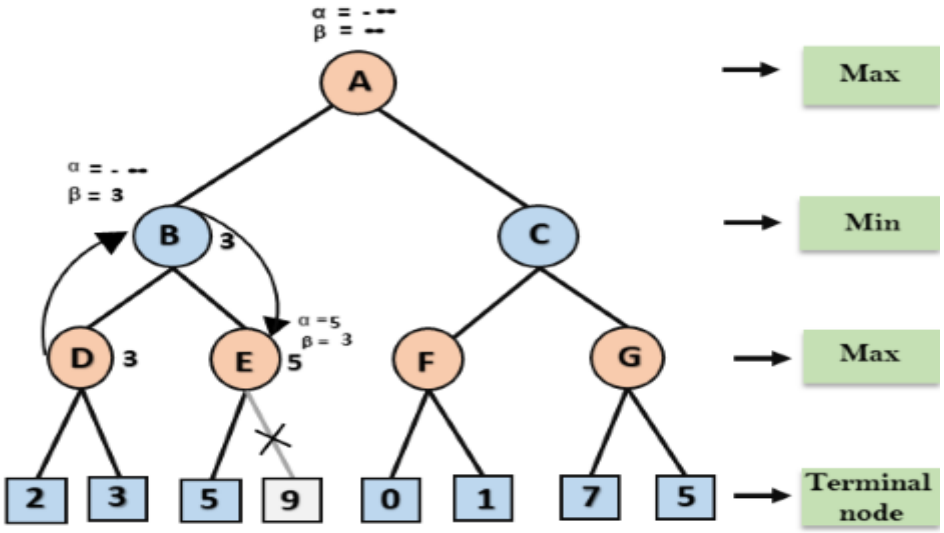
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

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.

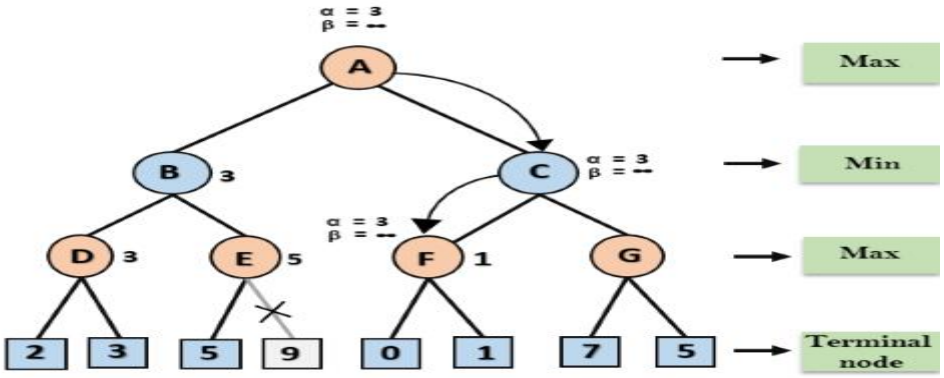
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will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



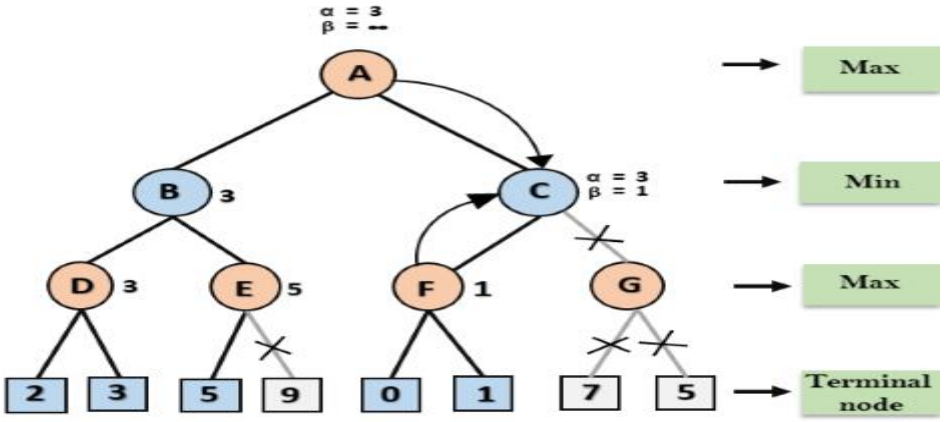
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

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

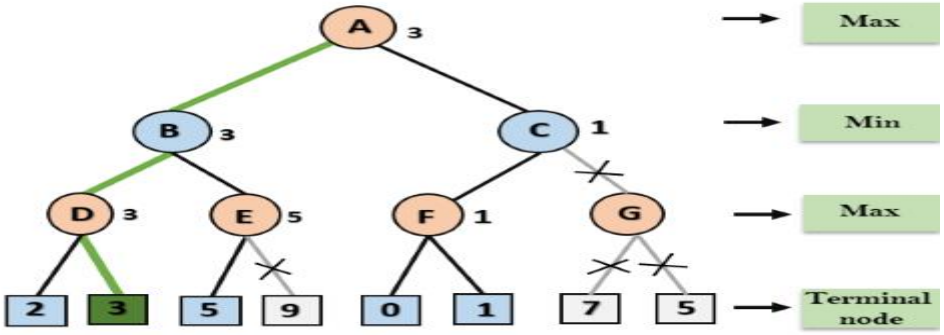
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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.



Evaluation functions:

In 1950, Shannon proposed that programs should cut off the search earlier to apply a heuristic evaluation function to states in the search.

- This technique effectively turns non-terminal nodes into terminal leaves.
- The idea is to replace the utility function by a heuristic evaluation function (EVAL), which estimates the position's utility, and replace the terminal test by a cutoff test that decides when to apply EVAL.
- An evaluation function returns an estimate of the expected utility of the game from a give position, just as heuristic functions return an estimate of the distance to the goal.

The performance of a game-playing program depends strongly on the quality of its evaluation function – but how to design?

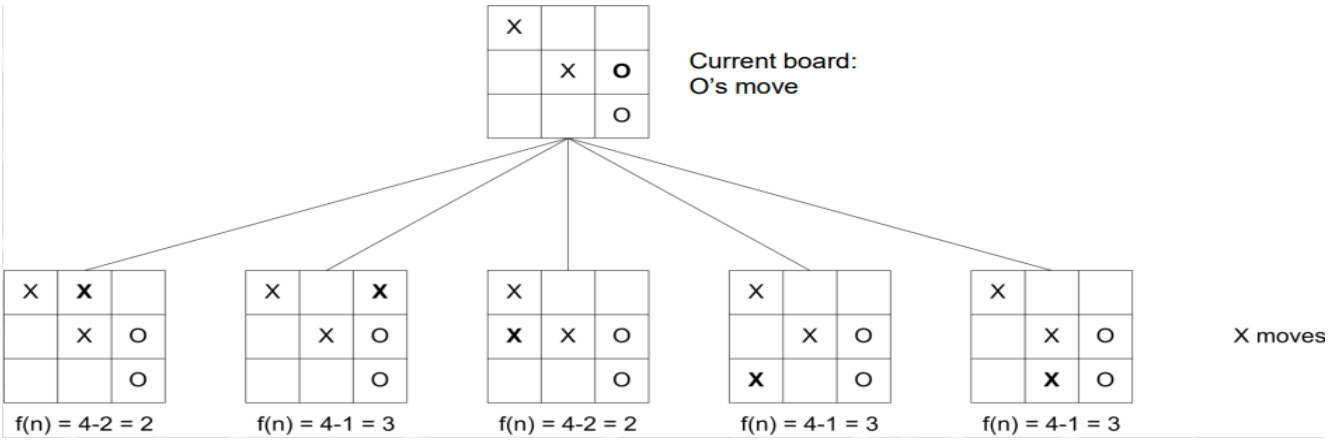
(1) The evaluation function should order the terminal states in the same way as the true utility function: states that are wins must evaluate better than draws, etc.

(2) The computation must not take too long! (3) For nonterminal states, the evaluation function should be strongly correlated with the actual "chances of winning."

Note that if the search must be cut off at nonterminal states, then the algorithm will necessarily be uncertain about the final outcomes of those states.

- Usually, an evaluation function calculates various features of a state (e.g. number of pawns, etc.).
- The features define various categories or equivalence classes of states: the states in each category have the same values for all features.
- For example: suppose experience suggests that 72% of states encountered in a two-pawn vs. one-pawn category lead to a win (utility: +1); 20% to a loss (0) and 8% to a draw (1/2).
- Expected value: (0.72 x 1)+(0.20 x 0) + (0.08 x ½) = 0.76.

Most evaluation functions compute separate numerical contributions from each feature and then combine them to find the total value.
– Typical evaluation function is a linear sum of features – Eval(s) = w1 f 1 (s) + w2 f 2 (s) + … + wn fn (s)
• w1 = 9
• f 1 (s) = number of white queens) – number of black queens • etc.



Evaluation function f(n) measures "goodness" of board configuration n. Assumed to be better estimate as search is deepened (i.e., at lower levels of game tree).
Evaluation function here: "Number of possible wins (rows, columns, diagonals) not blocked by opponent, minus number of possible wins for opponent not blocked by current player."