UNIT – II: Search Algorithms

Random search, Search with closed and open list, Depth first and Breadth first search, Heuristic search, Best first search, A* algorithm, Game Search.

Problem-solving agents:

In Artificial Intelligence, Search techniques are universal problem-solving methods. Rational agents or Problem-solving agents in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

Search Algorithm Terminologies:

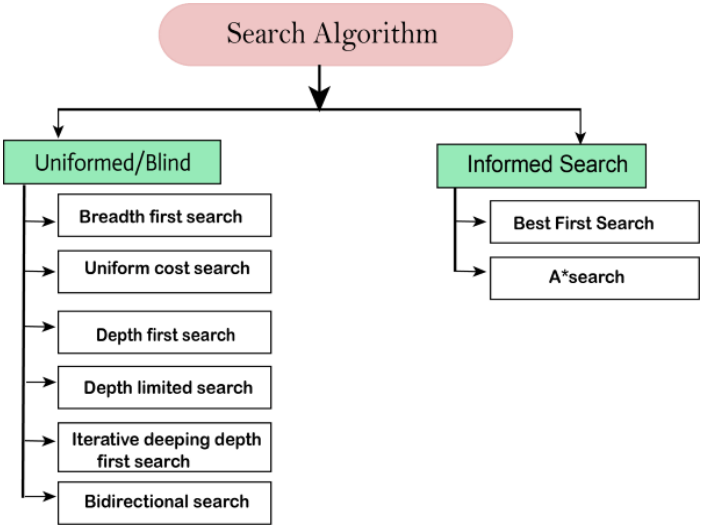
- **Search:** Searching is a step by step procedure to solve a search-problem in a given search space.
- A search problem can have three main factors:
  - **Search Space:** Search space represents a set of possible solutions, which a system may have.
  - **Start State:** It is a state from where agent 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 available 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.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

- **Completeness:** A search algorithm is said 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 said 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.

Types of Search Algorithms



Uninformed/Blind Search:

- The uniformed search is also known as Blind Search. Therefore, we can also say that examines / searches every node without any prior knowledge about the search space like initial state operators and test for the goal, has only information given by the definition of problem. So it is also called blind search algorithms. It examines each node of the tree until it achieves the goal node. It operates in a brute-force manner and hence also called brute force algorithms. Also, blind search might generate successor states distinguishing between a goal state and a non-goal state.

Informed Search

- Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy. Informed search is also called a Heuristic search.

A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.

Uninformed

- These strategies have no additional information about states beyond that provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non-goal state.

Breadth-first Search:

- **Breadth-first search** is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.
  - This algorithm searches/traverses breadth wise to search the goal in a tree or graph, so it is called breadth-first search. BFS is used where the given problem is very small and space complexity is not considered.

- In general, all the nodes are expanded at a given depth (current level) in the search tree before expanding any nodes at the next level.
- Breadth-first search is an instance of the general graph-search algorithm in which the *shallowest* unexpanded node is chosen for expansion.
- This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue- always added to the end of the OPEN list, and old nodes, which are shallower than the new nodes, get expanded first.
- Goal test is applied to each node when it is *generated* rather than when it is selected for expansion. When removing nodes from OPEN to be added to the CLOSED, check whether it is the goal node, if it is, then stops.

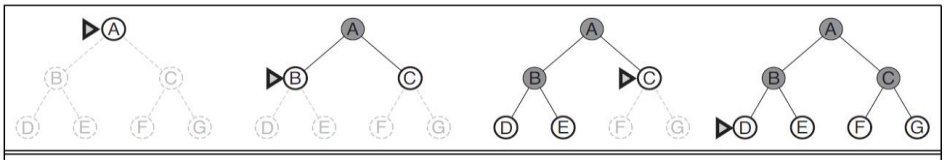
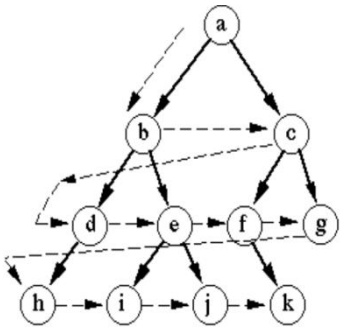


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

- The traversal is in the CLOSED list.
- But the OPEN list and CLOSED list altogether contribute to the space complexity since the total number of nodes stored in the memory is the summation of OPEN and CLOSED lists.



Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

*(For BFS, we place the children to the right side of OPEN End and chooses **shallowest node**, for DFS=> left side chooses **deepest** node)*

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 optimal solution which requires the least number of steps which provides the shortest path.=>Optimal, if step cost is 1

Disadvantages:

- if the branching factor of a node is large,
- for even small instances (e.g., chess)
- the ***space complexity*** and the ***time complexity*** are enormous
- It requires lots of memory since all nodes in current level of the tree must be saved into memory and goes to expand the next level.
- BFS needs lots of time to reach the goal state if the solution is far away from the root node.

Time Complexity:

Since the nodes beyond the goal node in the OPEN list are not processed the time complexity of BFS is associated with the number of nodes in the CLOSE list only. 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. Imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b2 at the second level. Each of *these* generates b more nodes, yielding b3 nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

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

Space Complexity:

Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$. for any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity. For breadth-first graph search in particular, every node generated remains in memory. There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier.

Completeness:

Since the state-space is explored layer-by-layer, if there is a solution, it is **definitely** found. BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution because all shallower nodes must have been generated already and failed the goal test.

Optimality:

BFS is optimal if path cost is a non-decreasing function of the depth of the node. Optimal because the goal nodes in the shallower level are found first.

Depth-first search

Depth-first search always expands the *deepest* node in the current frontier of the search tree.

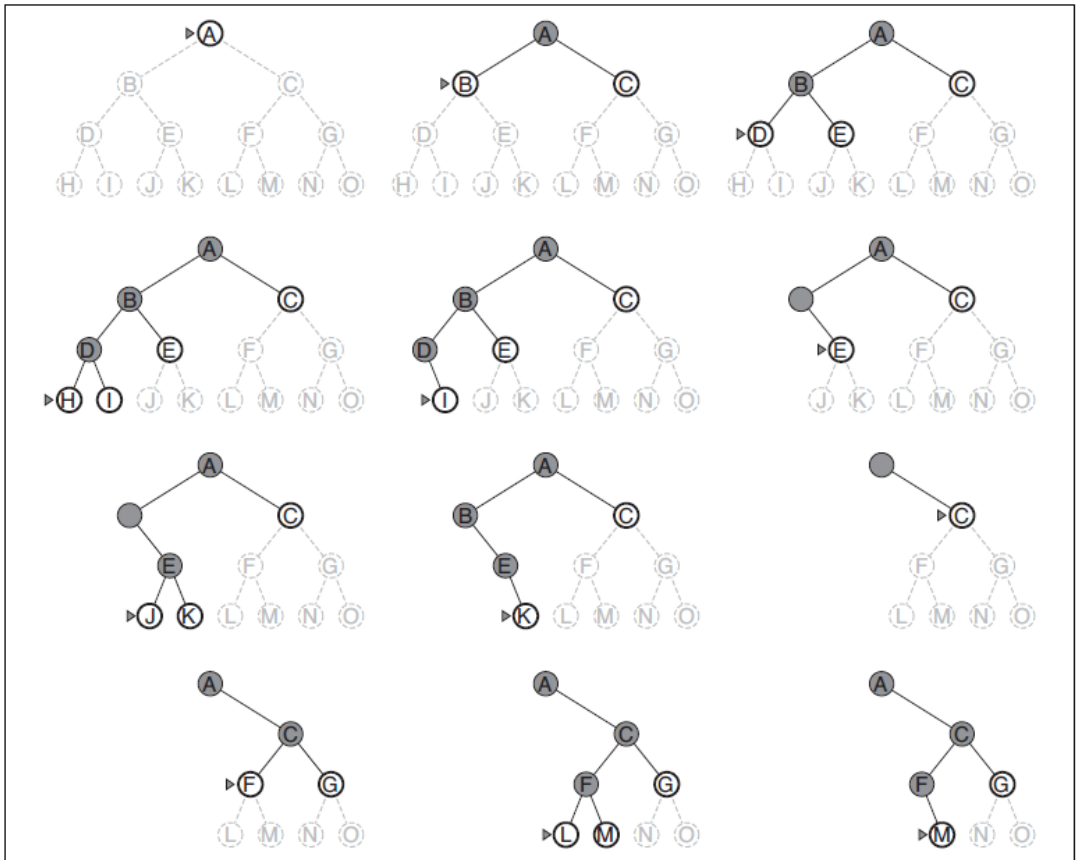


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

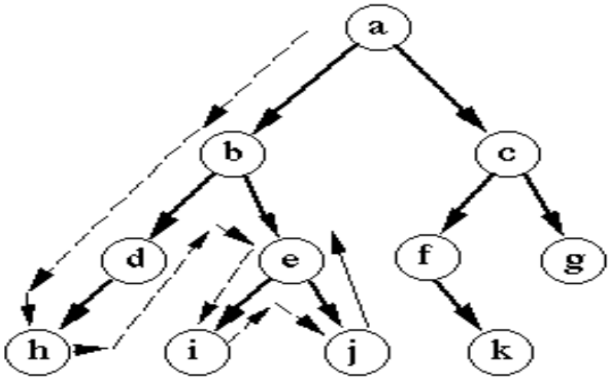
The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.

The depth-first search algorithm is an instance of the graph-search algorithm. Whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.

A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest **unexpanded node** because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

As an alternative to the GRAPH-SEARCH-style implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

- Backtracking search
  - only one successor is generated on expansion
  - rather than all successors
  - fewer memory



Depth-first search

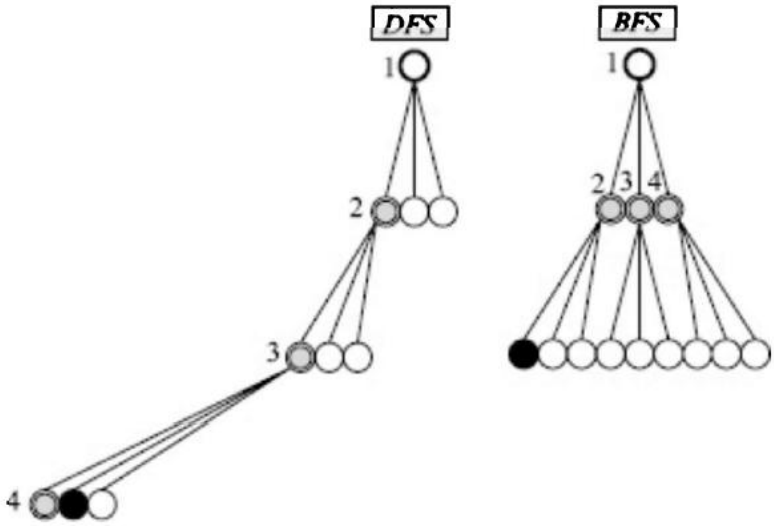


FIGURE 2.22 The search trees generated by *DFS* and *BFS* after four expansions.

Advantage:

- DFS requires very less memory as compared to BFSas 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 get trapped in an infinite loop.
- Since DFS identifies the deeper level goals before the shallower level ones, it is not optimal. DFS is not complete also. Because, if there is a loop in the one branch in the state space, even though we can find the goal node along another branch, still we cannot go to that branch due to the loop.
- Amazingly, the DFS has very low memory usage. It only stores the nodes along the branch, thus in each level minimum number of nodes are stored unlike in BFS where all the nodes get stored in each level.

Completeness:

DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity:

Bounded by the size of the state space(which may be infinite). A depth-first tree search, on the other hand, may generate all of the $O(b^m)$ nodes in the search tree, Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

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

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity:

Needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.

DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.

Optimal:

DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Depth-limited search

A depth-limited search algorithm is similar to depth-first search with a predetermined limit up to which it can traverse the nodes.

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.

The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose $l < d$, that is, the shallowest goal is beyond the depth limit. (This is likely when d is unknown.) Depth-limited search will also be non-optimal if we choose $l > d$. Its time complexity is $O(bl)$ and its space complexity is $O(bl)$.

Depth-first search can be viewed as a special case of depth-limited search with $l = \infty$.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

Advantages of DLS

- It takes lesser memory when compared to other search techniques- Memory efficient.

Disadvantages of DLS

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

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 $\ell > d$.

Iterative deepening depth-first search

Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

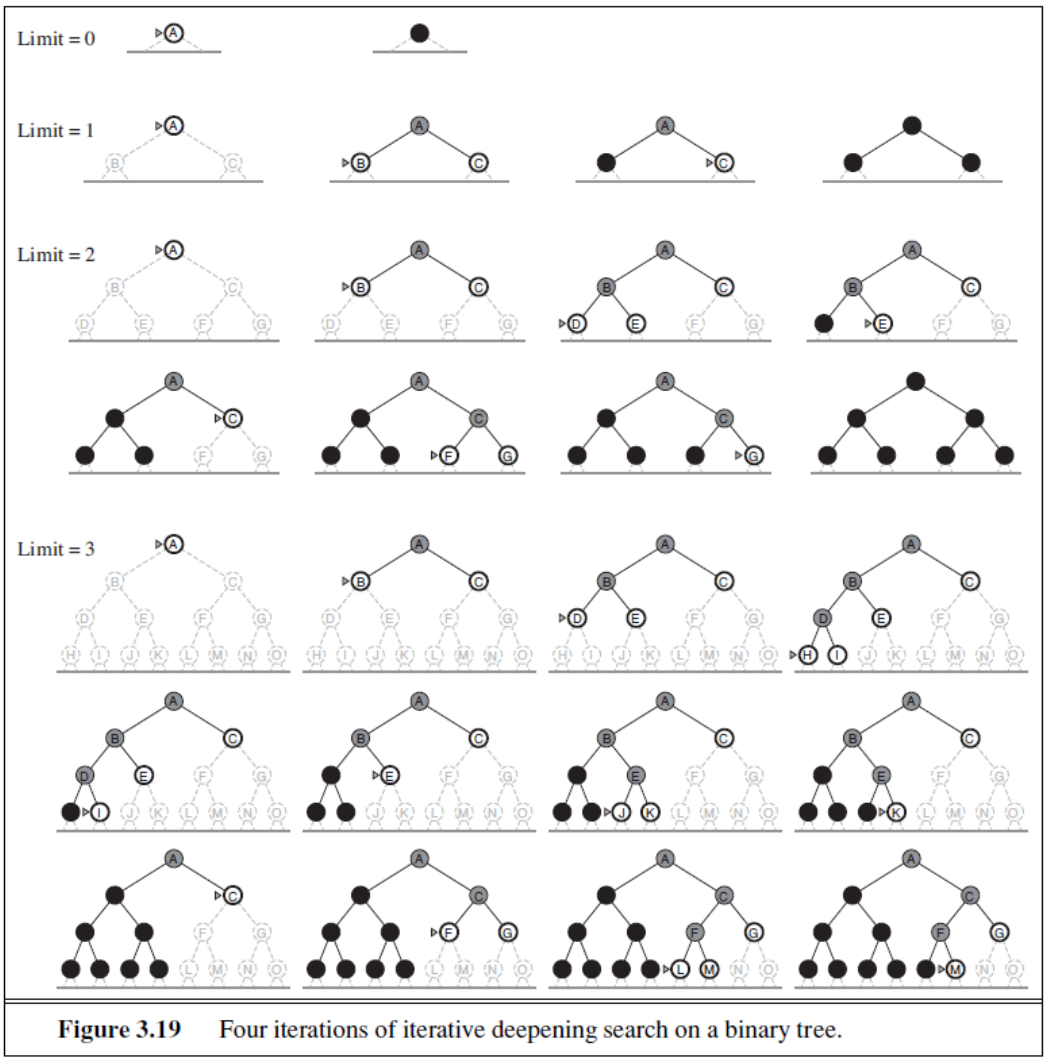


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

- It has the combined benefits of BFS and DFS search algoriths.
- Offers fast search and uses memory efficiently

Disadvantages:

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

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 $\mathbf{O(b^d)}$.

- In iterative deepening DFS, the nodes on the bottom level (depth *d*) are generated once, the nodes on the next level are generated twice and it continues up to the children of root which are generated *d* times. Therefore, the total number of nodes generated in the worst case is

$$N(IDS) = (d)b + (d-1)b^2 + \cdot\ \cdot\ \cdot + (1)b^d$$

Which gives a time complexity of $O(b^d)$—asymptotically the same as breadth-first search. There is some extra cost for generating the upper levels multiple times, but it is not large.

Space Complexity:

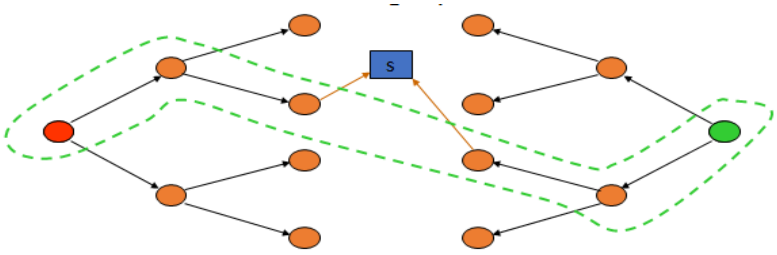
The space complexity of IDDFS will be **O(bd)**.

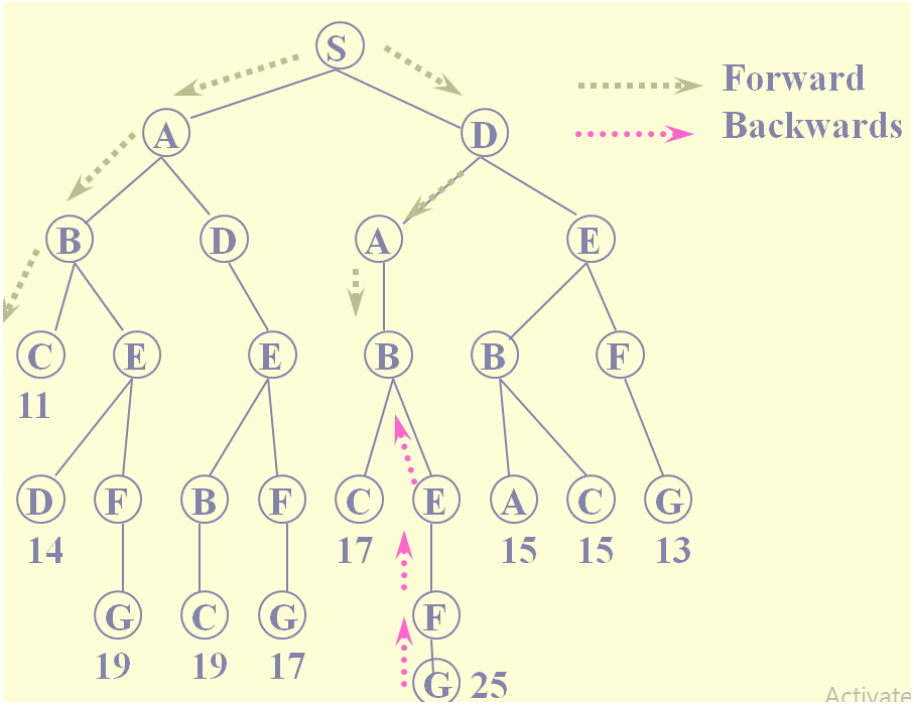
Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

Bidirectional search

2 fringe queues: FRINGE1 and FRINGE2





The bidirectional search algorithm is completely different from all other search strategies. B**idirectional 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 divides a graph** into two smaller sub-graphs. In one graph, the search is started from the initial start state and in the other graph, the search is started from the goal state. When these two nodes intersect each other, the search will be terminated.

Bidirectional search requires both start and goal start to be well defined and the branching factor to be the same in the two directions. Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.

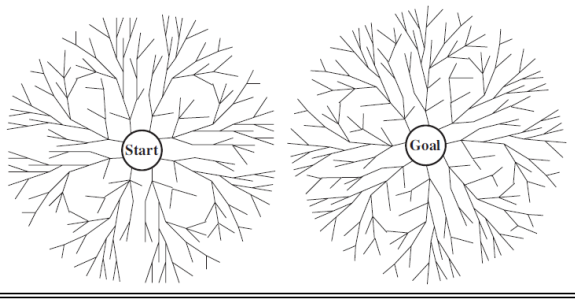


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages of bidirectional search

- This algorithm searches the graph fast.
- It requires less memory to complete its action.

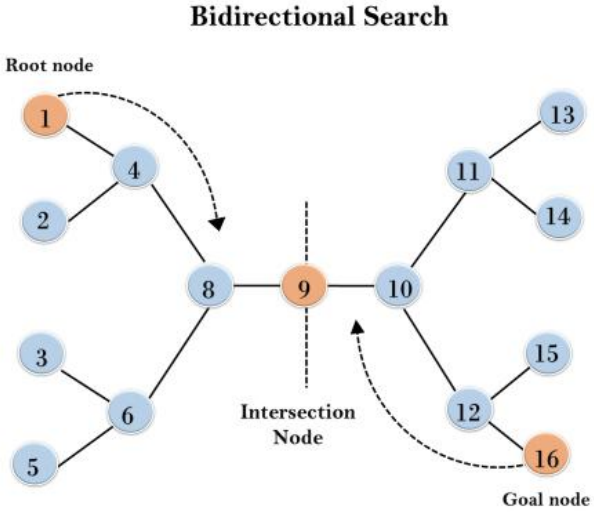
Disadvantages of bidirectional search

- 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.



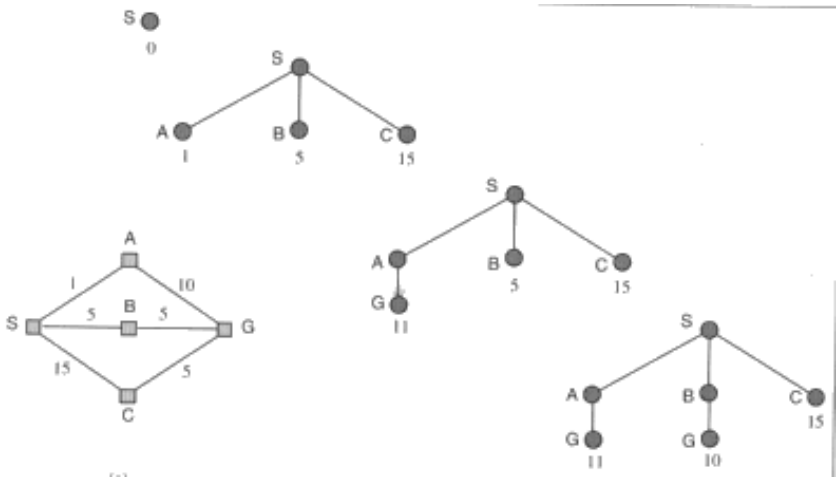
Completeness: Bidirectional Search is complete if we use BFS in both searches.

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.

Uniform cost search



- When all step costs are equal, breadth-first search is optimal because it always expands the *shallowest* unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function.
- This algorithm is mainly used when the step costs are not the same but we need the optimal solution to the goal state. Uniform cost search is considered the best search algorithm for a weighted tree or graph with costs, when a different cost is available for each edge.
- Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest *path cost* g(n) cost so far from root to node 'n', assuming costs are non-negative. This is done by storing the frontier as a priority queue ordered by g.
- The primary goal of the uniform-cost search is to find a path to the goal node by giving maximum priority to the lowest cumulative cost. Uniform cost search can be implemented using a priority queue,

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Differences from BFs:

1. ordering of the queue by path cost
2. The goal test is applied to a node when it is *selected for expansion* rather than when it is first generated. The reason is that the first goal node that is *generated* may be on a suboptimal path.
3. Test is added in case a better path is found to a node currently on the frontier.

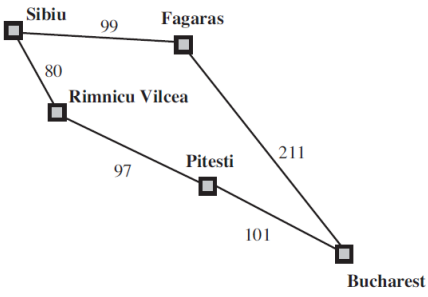


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

The problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97=177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99+211=310. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+97+101= 278. Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

For Uniform Cost Search, it is essential to continue searching even when the goal state is reached. If you stop immediately after finding a path, then you cannot guarantee that it is the shortest path, since there might have been a frontier that could have reached the goal state faster.

For example, In our Arad to Bucharest example, the first path you find (Arad - Sibiu - Fagaras - Bucharest) with the uniform cost search has the cost of 460. However, there is another path (Arad - Sibiu - Rimnicu Vilcea - Pitesti - Bucharest) that as the cost of 418.
Therefore, you need to wait until all the other frontiers have a path cost higher than the path cost of your path to the goal state. This is ensured exactly when your path to the goal state is chosen from the list of frontiers, since by definition, uniform cost search chooses the frontier with least path cost.

Path explored	**Alternative path in memory**
S->F = g(n)=>99 S->R = 80	
Chooses path with low cost S->R =80 NEW path S->R->P =80+97 = 177 Path in memory is better than currently explored, so expands that now	S->F = 99

S->F->B = 99+211 = 310 Even though goal is found checks for better paths Path in memory is better than currently explored, so expands that now	S->R->P = 177
S->R->P ->B= 177 + 101 = 278	S->F->B = 310
This current explored path is better than in memory so discards that and gives solution path as **S->R->P->B, with a path cost of 278**	Ignored

Advantages:

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

Disadvantages:

- It does not considers about the number of steps involved in searching to reach about in lowest cost I,e., only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d.

Instead, Let C* **is Cost of the optimal solution**, and **ε** is each step (action cost) to get closer to the goal node. Then the number of steps is = 1 + C*/ε. Here we have taken +1, as we start from state 0 and end to C*/ε.

Then the algorithm's worst-case time and space complexity is $\mathbf{O(b^{1 + \lfloor C^*/\varepsilon \rfloor})}$which can be much greater than b^d. This is because uniform cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + \lfloor C^*/\varepsilon \rfloor})}$.

When all step costs are the same, uniform-cost search is similar to breadth-first search, except that BFS stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost; thus uniform-cost search **does strictly more work by expanding nodes at depth d unnecessarily**

Optimal:

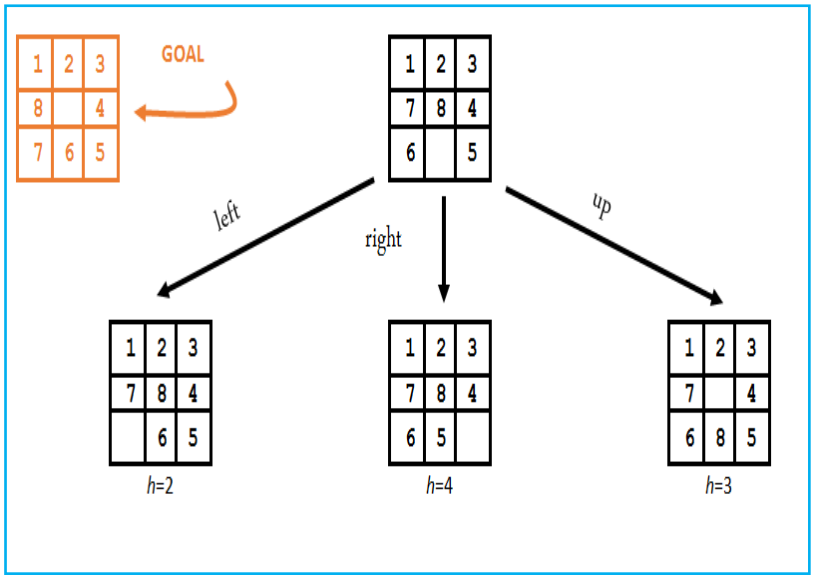
Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Informed Search Algorithms

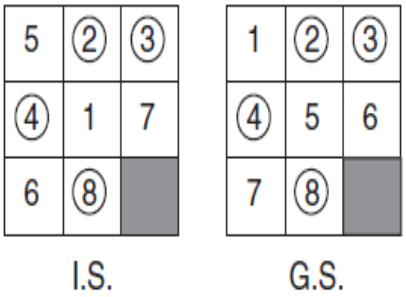
- Strategies that know whether one non-goal state is "more promising" than another are called **informed search** or **heuristic search or** directed search strategies.
- The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.
- The heuristic function is used to achieve the goal state with the lowest cost possible. This function estimates how close a state is to the goal.

- *A heuristic evaluation function, h(n), is the **estimated cost** of the cheapest path from the state at node n, to a goal state.*
- *Heuristic evaluation functions are very much dependent on the domain used. h(n) might be the estimated number of moves needed to complete a puzzle, or the estimated straight-line distance to some town in a route finder.*
- *A heuristic evaluation function which accurately represents the actual cost of getting to a goal state, tells us very clearly which nodes in the state-space to expand next, and leads us quickly to the goal state.*

Which move is best?



8 Puzzle Heuristics- Approach 1



Number of tiles in the *Correct* position.

- The **heuristic function of the game may be as follows:** Count the number of tiles that are in place with respect to the goal state. Hence,

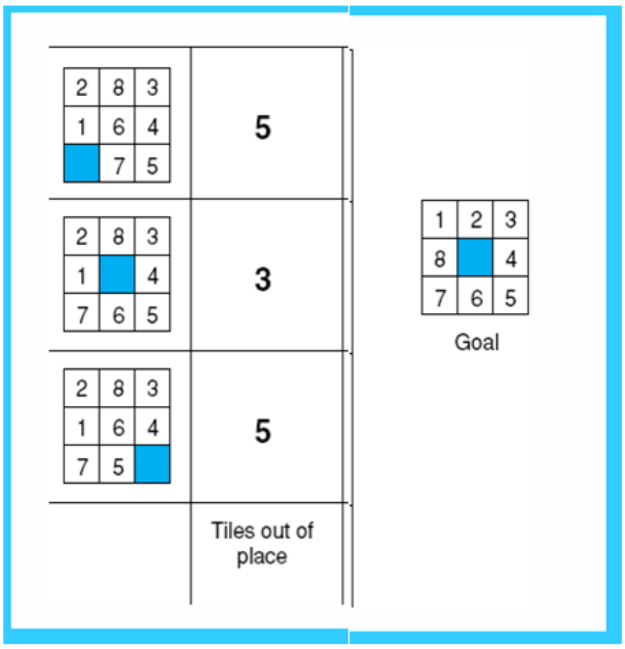
h(Initial State) = 4

- As seen in the initial state, the number of tiles that are in the correct place with respect to the goal (Tiles 2, 3, 4 and 8 are in the same place as required in the goal state) state are four. Hence, the heuristic value for the initial state is 4 (Note, as a convention we do not consider the position of blank in the nongoal state).

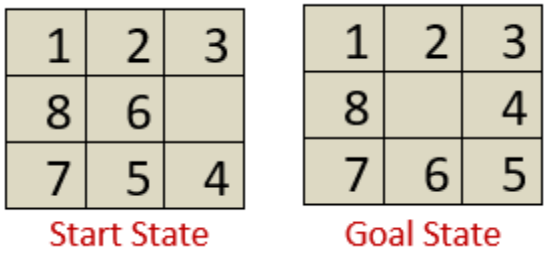
h(Goal State) = 8

- Heuristic value for the goal state will always be eight as all the tiles are in correct place in the goal state.

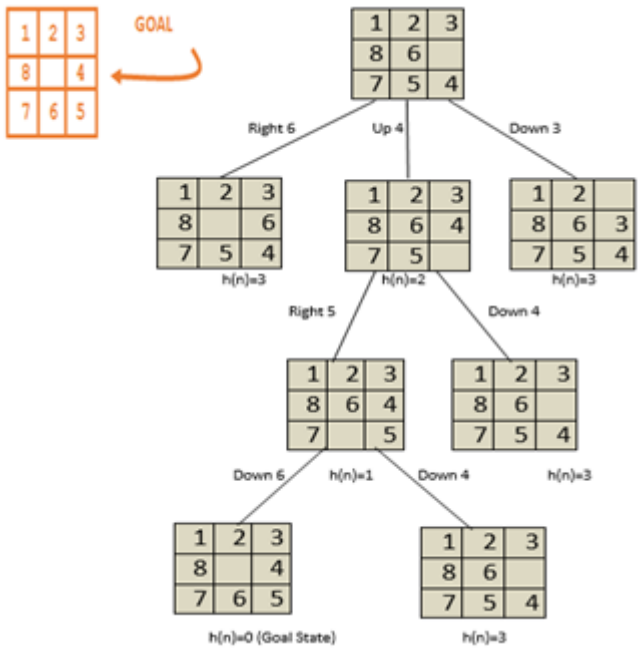
8 Puzzle Heuristics- Approach 2



- Number of tiles in the *incorrect* position/ misplaced tiles
- This can also be considered a lower bound on the number of moves from a solution!
- The "best" move is the one with the lowest number returned by the heuristic.
- Is this heuristic more than a heuristic (is it always correct?).
- Given any 2 states, does it always order them properly with respect to the minimum number of moves away from a solution?

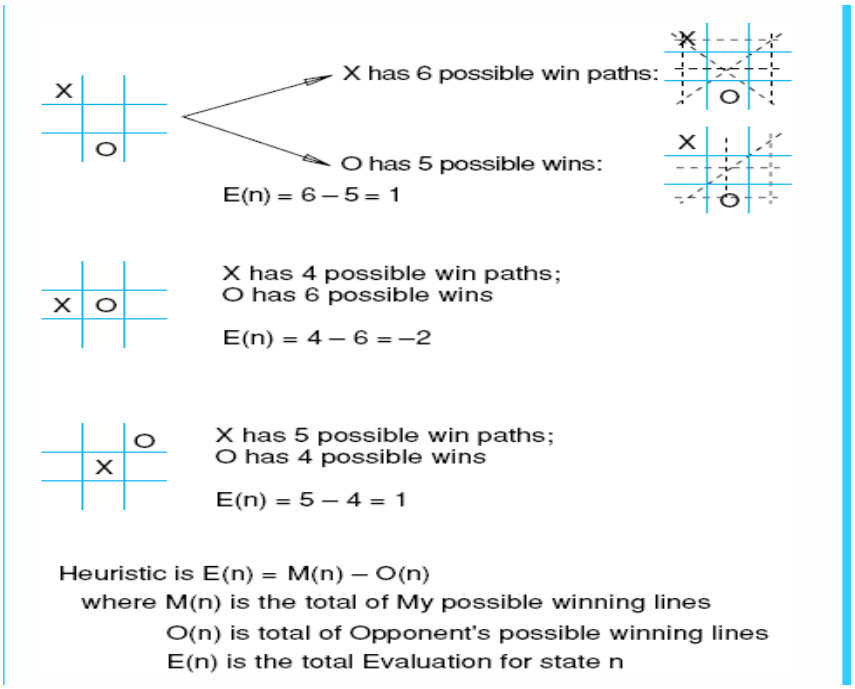


- A heuristic function for the 8-puzzle problem is defined below:
  h(n)=Number of tiles out of position.
- So, there is total of three tiles out of position i.e., 6,5 and 4. Do not count the empty tile present in the goal state). i.e. h(n)=3. Now, we require to minimize the value of **h(n) =0.**
- We can construct a state-space tree to minimize the h(n) value to 0,



It is seen from the above state space tree that the goal state is minimized from h(n)=3 to h(n)=0.

Heuristic for Tic-Tac-Toe Problem



Properties of a Heuristic search Algorithm

Use of heuristic function in a heuristic search algorithm leads to following properties of a heuristic search algorithm:

- **Admissible Condition:** An algorithm is said to be admissible, if it returns an optimal solution.
- **Completeness:** An algorithm is said to be complete, if it terminates with a solution (if the solution exists).

- **Dominance Property:** If there are two admissible heuristic algorithms **A1** and **A2** having **h1** and **h2** heuristic functions, then **A1** is said to dominate **A2** if **h1** is better than **h2** for all the values of node **n.**
- **Optimality Property:** If an algorithm is **complete, admissible**, and **dominating** other algorithms, it will be the best one and will definitely give an optimal solution.

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 pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:0/06:36

$$h(n) <= h^*(n)$$

Here h(n) is heuristic(actual) cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

In the informed search we will discuss two main algorithms which are given below:

- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

Let's discuss some of the informed search strategies.

1. Greedy best-first search algorithm

Greedy best-first search traverses the node by selecting the path which appears best at the moment. It is the combination of depth-first search and breadth-first search algorithms. The closest path is selected by using the heuristic function.

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.

Best first search algorithm:

Step 1: Place the starting node into the OPEN list.

Step 2: If the OPEN list is empty, Stop and return failure.

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:

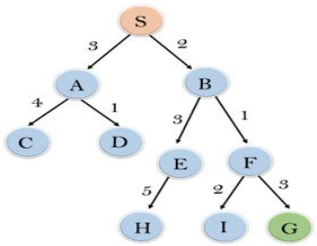
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- In the worst-case scenario, the greedy best-first search algorithm may behave like an unguided DFS.
- There are some possibilities for greedy best-first to get trapped in an infinite loop.
- The algorithm is not an optimal one.

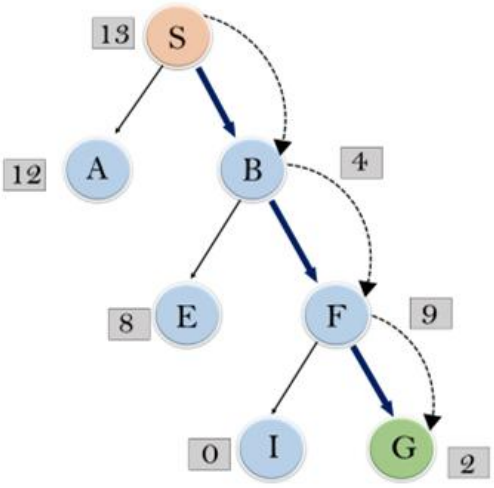
Example2:

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]

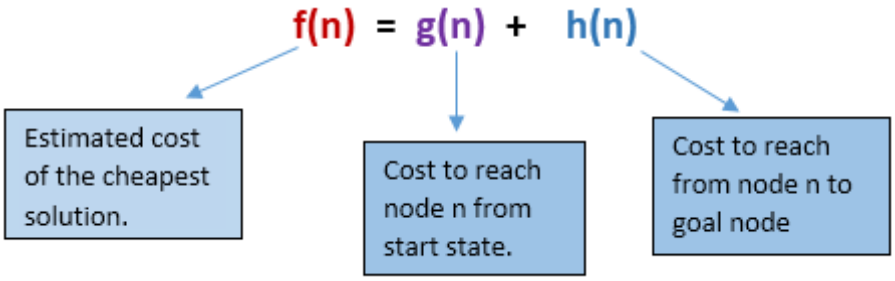
Hence the final solution path will be: **S----> B----->F----> G**

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. In greedy search, you can see we do not visit all the child nodes of a particular node. We find heuristics of each child node only the one with the lowest value is inserted to the OPEN list to be processed. Therefore greedy search is not complete.
Optimal: As well as this is not the best path (not the shortest path - we found this path in uniform cost search). Therefore greedy search is not optimal, also. As a solution, we should consider not only the heuristic value but also the path cost. Here, comes the A* algorithm.

A* Search Algorithm:

Uniform cost search=>f(n) =g(n)
Greedy BFS=>f(n) = h(n)
A* =>f(n) = g(n) + h(n)

The most widely known form of best-first search is called **A∗search**(pronounced "A-star ∗SEARCHsearch"). It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the costto get from the node to the goal:
f(n) = g(n) + h(n) .
Since g(n) gives the path cost from the start node to node n, and
h(n) is the estimated costof the cheapest path from n to the goal, we have
f(n) = estimated cost of the cheapest solution through n .
Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is thenode with the lowest value of g(n) + h(n). A∗search isboth complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH exceptthat A∗uses g + h instead of g.



At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

Algorithm of A* search:

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 failure 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:

- 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.

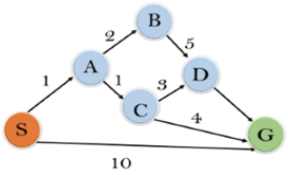
Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main 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:

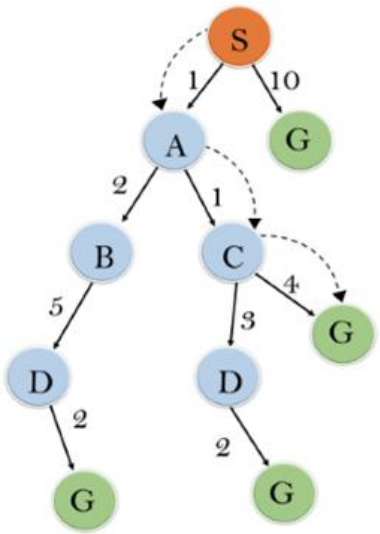
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

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.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

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)**

A* algorithm is complete since it checks all the nodes before reaching to goal node/end of the state space. It is optimal as well because considering both path cost and heuristic values, therefore the lowest path with the lowest heuristics can be found with A*. One drawback of A* is it stores all the nodes it processes in the memory.

Adversarial Search

- 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 acting in a competitive environment, 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 to win the game. 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 Game playing**.
- Here, game-playing means discussing those games where **human intelligence** and **logic factor** is used, excluding other factors such as **luck factor**. **Tic-tac-toe, chess, checkers**, etc., are such type of games where no luck factor works, only mind works.

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.'***

Techniques required getting the best optimal solution

There is always a need to choose those algorithms which provide the best optimal solution in a limited time. So, we use the following techniques which could fulfil our requirements:

- **Pruning:** A technique which allows ignoring the unwanted portions of a search tree which make no difference in its final result.
- **Heuristic Evaluation Function:** It allows approximating the cost value at each level of the search tree, before reaching the goal node.

Elements of Game Playing search

To play a game, we use a game tree to know all the possible choices and to pick the best one out. There are following elements of a game-playing:

- **INITIAL STATE (S_0):** The top node in the game-tree represents the initial state in the tree from where a game begins. and shows all the possible choice to pick out one.
- **PLAYER (s):** It defines which player is having the current turn to make a move in the state. There are two players, **MAX and MIN**. **MAX** begins the game by picking one best move and place **X** in the empty square box.
- **ACTIONS (s):** It defines the set of legal moves to be used in a state. Both the players can make moves in the empty boxes chance by chance.
- **RESULT (s, a):** It is a transition model which defines the result of a move. The moves made by **MIN** and **MAX** will decide the outcome of the game.
- **TERMINAL-TEST(s):** It defines that the game has ended and returns true.When all the empty boxes will be filled, it will be the terminating state of the game.
- **UTILITY (s,p):** It defines the final value with which the game has ended. This function is also known as **Objective function** or **Payoff function**. At the end, we will get to know who wins: **MAX** or **MIN,** and accordingly, the price will be given to them**.** The price which the winner will get i.e.
    - **(-1):** If the PLAYER MAX loses.
    - **(+1):** If the PLAYER wins.
    - **(0):** If there is a draw between the PLAYERS.

For example**, in **chess, tic-tac-toe,** we have two or three possible outcomes. Either to win, to lose, or to draw the match with values **+1,-1 or 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:

Types of algorithms in Adversarial search

In a **normal search**, we follow a sequence of actions to reach the goal or to finish the game optimally. But in an **adversarial search**, the result depends on the players which will decide the result of the game. It is also obvious that the solution for the goal state will be an optimal solution because the player will try to win the game with the shortest path and under limited time.

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.

There are following types of adversarial search:

- **Minmax Algorithm**
- **Alpha-beta Pruning.**

Minimax Strategy

In artificial intelligence, minimax is a **decision-making** strategy under **game theory,** which is used to *minimize the losing chances in a game and to maximize the winning chances.* This strategy is also known as '**Minmax,' 'MM,' or 'Saddle point.'** Basically, it is a two-player game strategy where *if one wins, the other loose the game*. This strategy simulates those games that we play in our day-to-day life. Like, if two persons are playing chess, the result will be in favour of one player and will unfavoured the other one. The person who will make his bes*t try, efforts as well as cleverness, will surely win.*

We can easily understand this strategy via **game tree**- where the *nodes represent the states of the game and edges represent the moves made by the players in the game*. **Players will be two** namely:

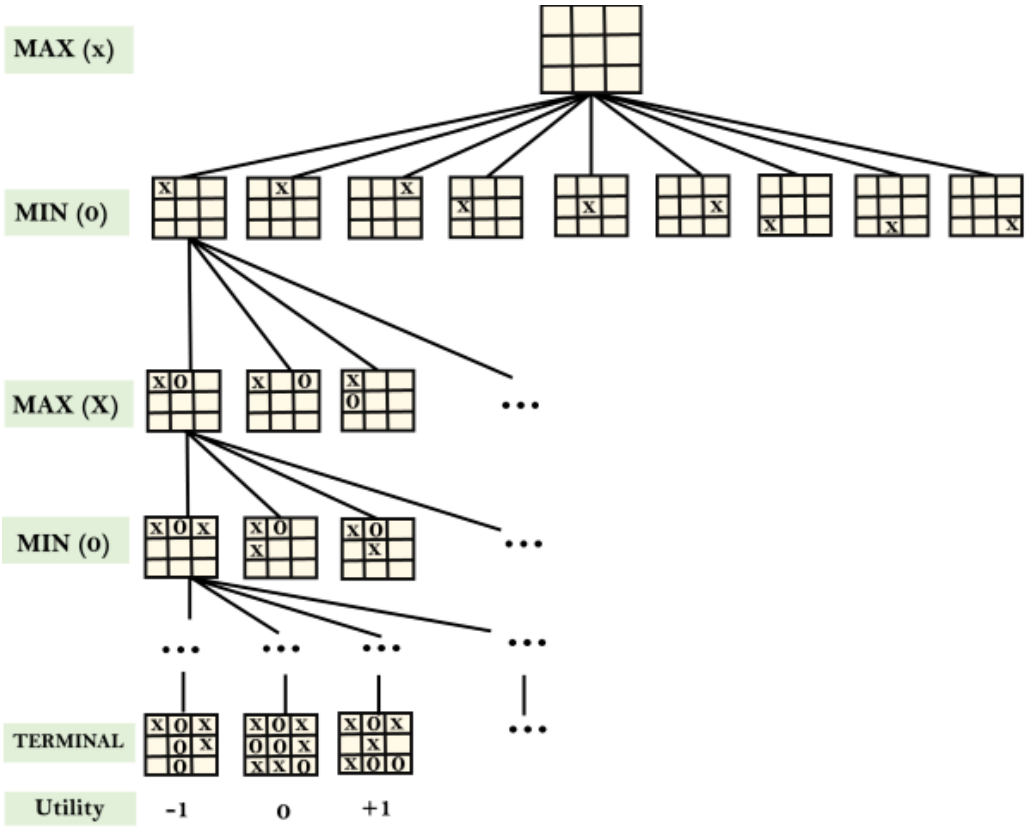
- **MIN:** Decrease the chances of **MAX** to win the game.
- **MAX:** MAXIMIZ his chances of winning the game.

They both play the game alternatively, i.e., turn by turn and following the above strategy, i.e., if one wins, the other will definitely lose it. Both players look at one another as competitors and will try to defeat one-another, giving their best.

In minimax strategy, the result of the game or the utility value is generated by a **heuristic function** by propagating from the initial node to the leaf node. It follows the **backtracking technique** and **backtracks to find the best choice.**

MAX will choose that path which will *increase its utility value* and

MIN will choose the opposite path which could help it to *minimize MAX's utility value.*



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 in the optimal way. 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:

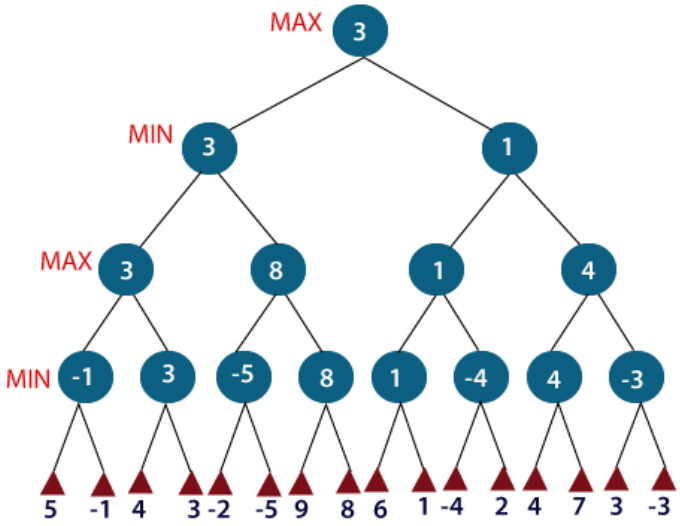
For a state S MINIMAX(s) =

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

MINIMAX Algorithm

- MINIMAX algorithm is a **backtracking algorithm** where it backtracks to pick the best move out of several choices. It provides an optimal move for the player assuming that opponent is also playing optimally.
- The minimax algorithm performs a **depth-first search algorithm** for the exploration of the complete game tree.
- Here, we have two players **MIN and MAX,** and the game is played alternatively between them, i.e., when **MAX** made a move, then the next turn is of **MIN.** It means the move made by MAX is fixed and, he cannot change it. The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

- The same concept is followed in DFS strategy, i.e., we follow the same path and cannot change in the middle. That's why in MINIMAX algorithm, instead of BFS, we follow DFS.
  - Keep on generating the game tree/ search tree till a limit **d.**
  - Compute the move using a heuristic function.
  - Propagate the values from the leaf node till the current position following the minimax strategy.
  - Make the best move from the choices.



For example, in the above figure, the two players **MAX** and **MIN** are there. **MAX** starts the game by choosing one path and propagating all the nodes of that path. Now, **MAX** will backtrack to the initial node and choose the best path where his utility value will be the maximum. After this, its **MIN** chance. **MIN** will also propagate through a path and again will backtrack, but **MIN** will choose the path which could minimize **MAX** winning chances or the utility value.

So, if the level is minimizing, the node will accept the minimum value from the successor nodes. If the level is maximizing, the node will accept the maximum value from the successor.

Note: The time complexity of MINIMAX algorithm is **O(b^d)** where b is the branching factor and d is the depth of the search tree.

Pseudo-code for MinMax Algorithm:

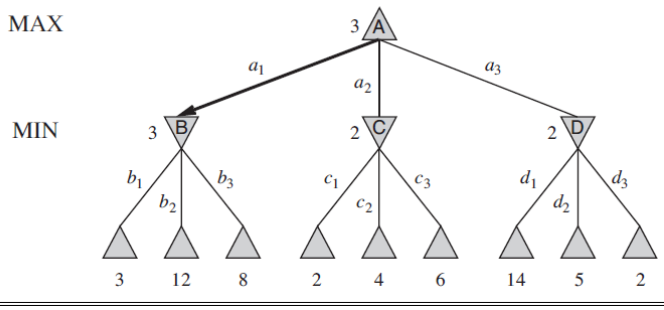


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

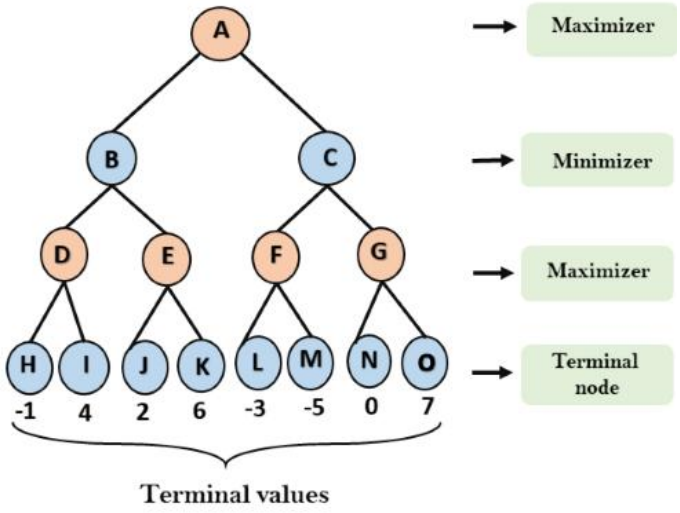
Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

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 values and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

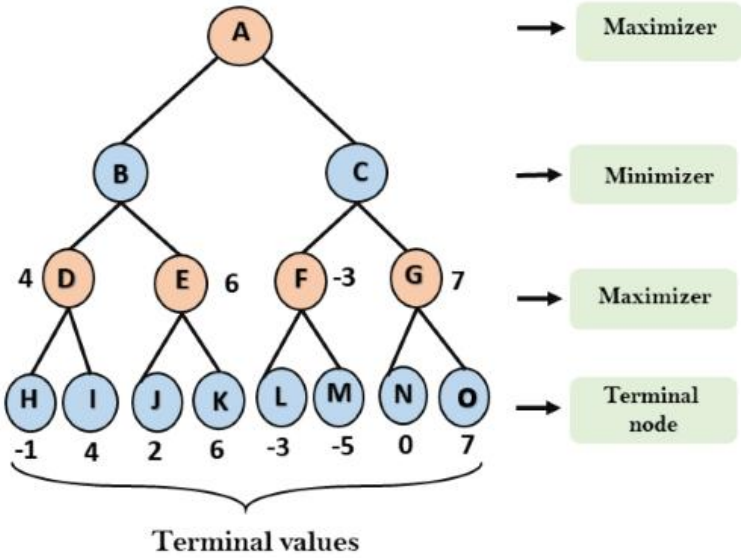
Step-1: In the first step, the algorithm generates the entire game-tree and applies the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree.

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.



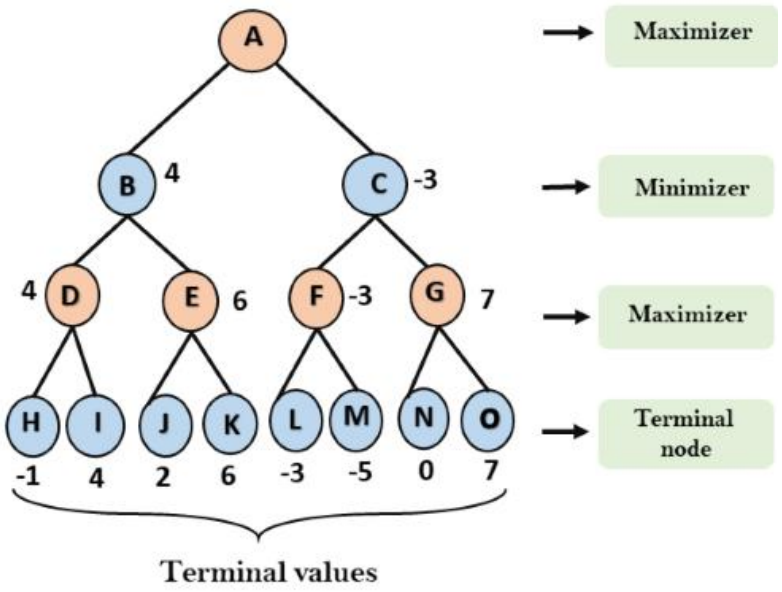
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- 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



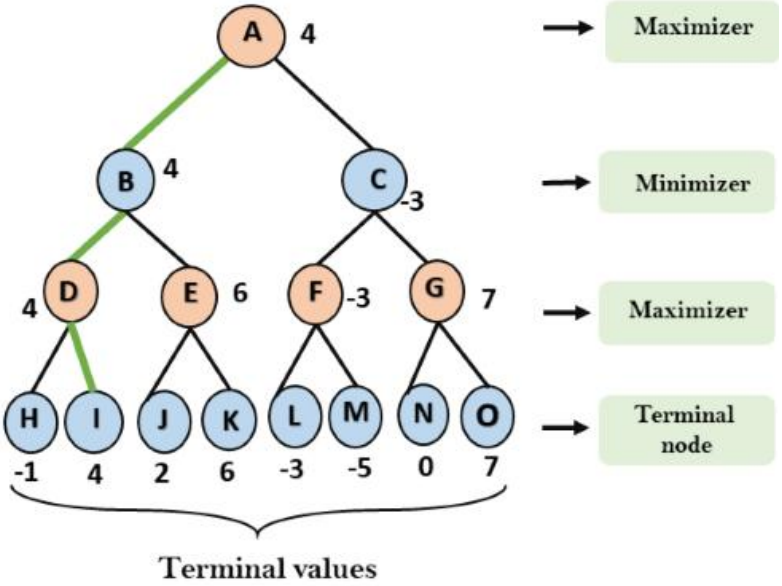
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- 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)**.

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from **alpha-beta pruning**.

Alpha-Beta Pruning

- Alpha-beta pruning is an advance version of MINIMAX algorithm. The drawback of minimax strategy is that it explores each node in the tree deeply to provide the best path among all the paths. This increases its time complexity. But as we know, the performance measure is the first consideration for any optimal algorithm. Therefore, alpha-beta pruning reduces this drawback of minimax strategy by less exploring the nodes of the search tree.
- The method used in alpha-beta pruning is that it **cutoff the search** by exploring less number of nodes. It makes the same moves as a minimax algorithm does, but it prunes the unwanted branches using the pruning technique (discussed in adversarial search). Alpha-beta pruning works on two threshold values, i.e., **α (alpha)** and **β (beta).**
- α It is the best highest value, a **MAX** player can have. It is the lower bound, which represents negative infinity value.
- β It is the best lowest value, a **MIN** player can have. It is the upper bound which represents positive infinity.

So, each MAX node has α-value, *which never decreases*, and each MIN node has β-value, which never increases.

- 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:

α>=β

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.

Pseudo-code for Alpha-beta Pruning:

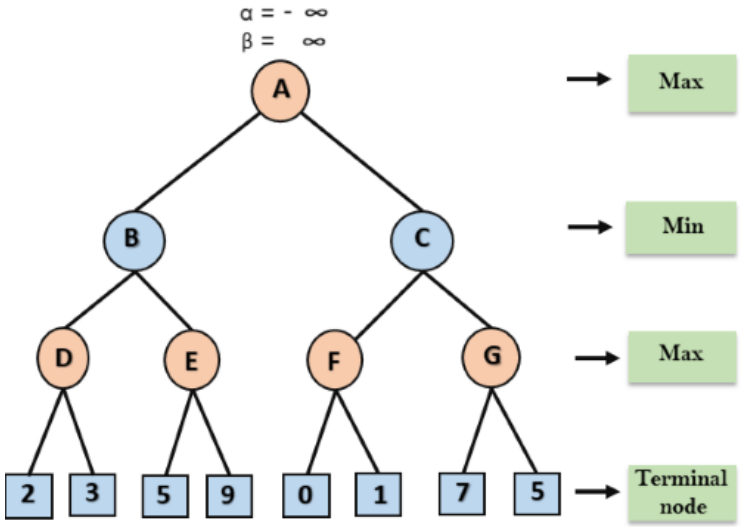
```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Working of Alpha-Beta Pruning:

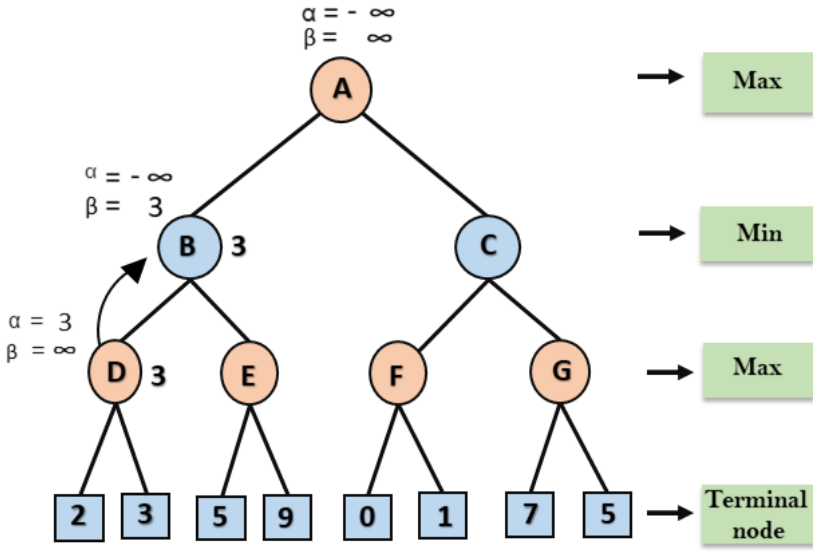
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

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



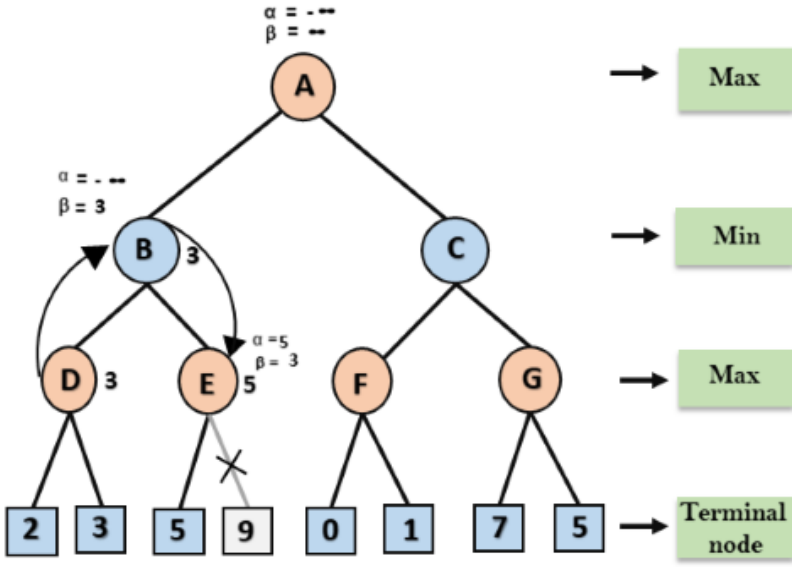
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

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



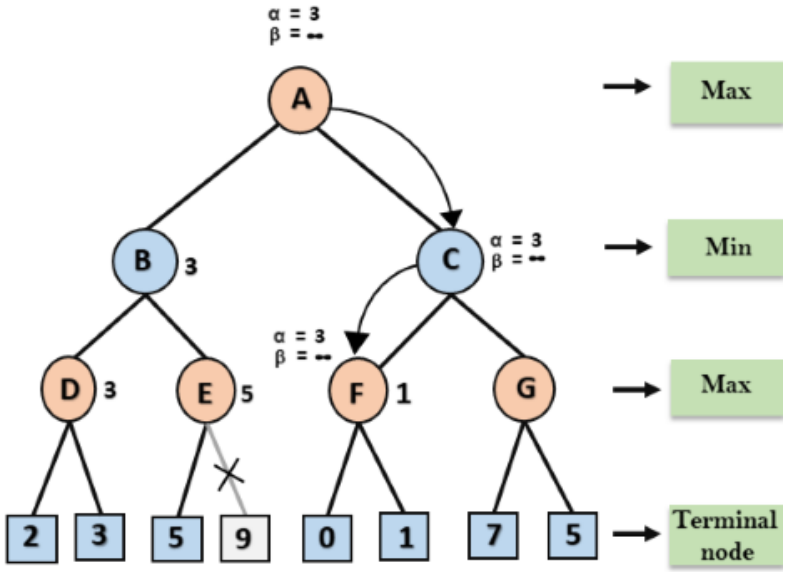
In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

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

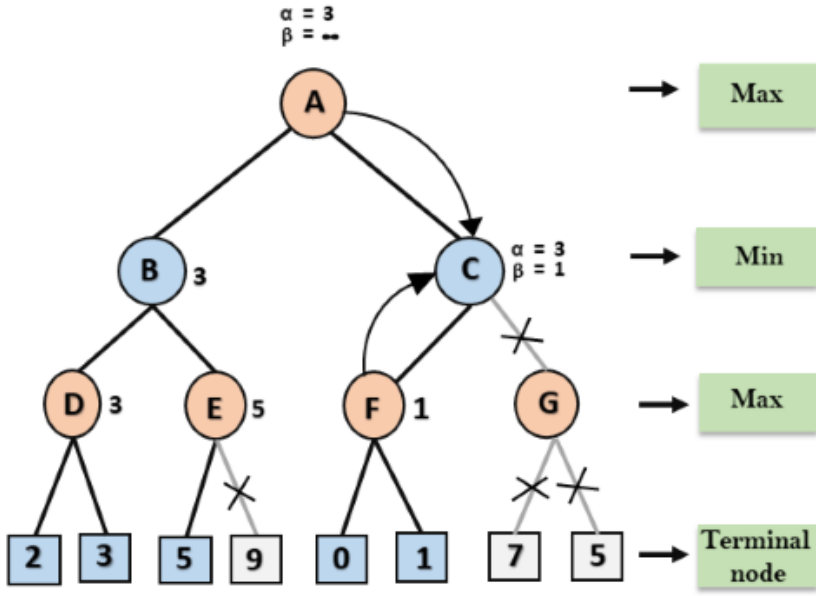


Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.
At node C, α=3 and β= +∞, and the same values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

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



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



Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.