UNIT-2

SEARCH TECHNIQUES

2.1. ISSUES IN THE DESIGN OF SEARCH PROGRAMS

When it comes to designing search programs, there are several important issues that need to be considered to ensure effective and efficient search functionality. Here are some common issues in the design of search programs:

- **Relevance:** Determining the relevance of search results is crucial to provide users with accurate and useful information. Designers need to consider various factors, such as keyword matching, context, user preferences, and relevance algorithms, to rank search results effectively.
- **Query Understanding:** Interpreting user queries correctly is essential for delivering accurate results. Search programs need to employ advanced natural language processing techniques to understand the intent behind user queries, handle synonyms, homonyms, and account for different linguistic variations.
- **Scalability:** Search programs must be designed to handle a large volume of data and user queries efficiently. As the size of the dataset grows, the search program should scale horizontally or vertically to maintain optimal performance and response times.
- **Indexing:** Effective indexing is critical for search programs to retrieve relevant results quickly. Choosing the right indexing method, such as inverted indexes or full-text indexes, and optimizing the indexing process to ensure fast and accurate retrieval of data is an important consideration.
- **User Experience:** The design of search programs should prioritize the user experience to provide intuitive and user-friendly interfaces. This includes features like auto-suggestions, query completion, filtering options, and personalized recommendations to enhance the search experience.
- **Ranking Algorithm:** Determining the order in which search results are presented requires a robust ranking algorithm. Designers need to consider factors like relevance, popularity, regency, and other user-specific parameters to develop an effective ranking algorithm that aligns with the search program's goals.
- **Query Performance:** Search programs need to deliver fast and efficient query performance to ensure a smooth user experience. This involves optimizing query processing, minimizing latency, caching frequently accessed data, and employing indexing techniques to speed up search operations.

- **Cross-Language Search:** Designing search programs that can handle multiple languages and provide accurate results across different languages is a challenge. It requires techniques like language detection, translation, and multilingual indexing to enable effective cross-language search functionality.
- **Security and Privacy**: Search programs must address security and privacy concerns. They need to handle user data securely, protect against malicious queries or attacks, and ensure compliance with data protection regulations.
- **Continuous Improvement:** Designers should consider building mechanisms for collecting user feedback and usage patterns to continuously improve the search program. Iterative refinement based on user interactions and analytics can help enhance relevance, performance, and user satisfaction over time.

Addressing these issues in the design of search programs is crucial to provide users with accurate, relevant, and efficient search results while delivering a seamless user experience.

2.2. UNIFORMED SEARCH TECHNIQUES

Uniformed search techniques, also known as blind search algorithms, are search strategies that explore the search space without any prior knowledge about the problem domain or the goal state. These techniques do not use heuristics or domain-specific information to guide the search process. Instead, they systematically traverse the search space to find a solution.

Here are some commonly used uniformed search techniques:

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

2.2.1. Breadth-first Search:

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

- **BFS algorithm:**

  A standard BFS implementation puts each vertex of the graph into one of two categories:

  1) **Visited**
  2) **Not Visited**

  The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

Step1: Start by putting any one of the graph's vertices at the back of a queue.

Step2: Take the front item of the queue and add it to the visited list.

Step3: Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

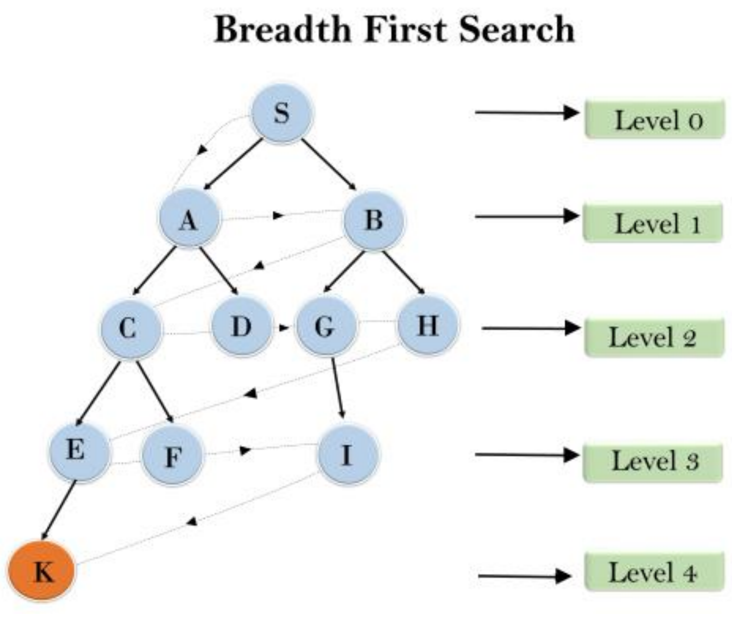
Step4: Keep repeating **steps 2 and 3** until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every 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+b2+b3+.......+ bd= O (bd)

- **Space Complexity:** Space complexity of BFS algorithm is given by the Memory size of frontier which is O(bd).

- **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.
- **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.

2.2.2. Depth-first Search:

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

- **DFS algorithm:**

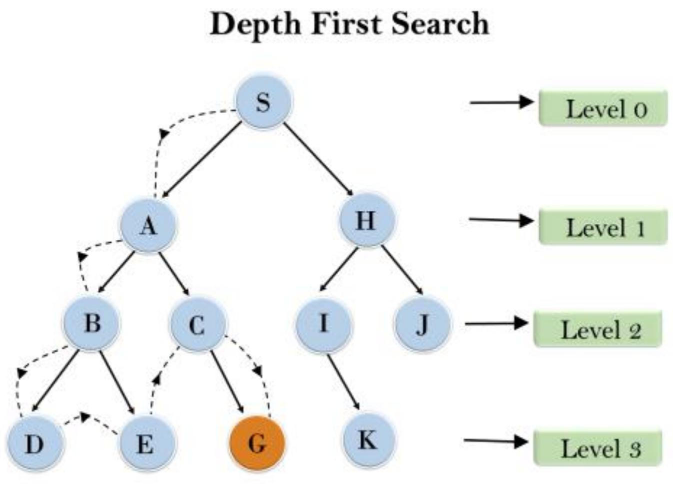
  Follow the below method to implement DFS traversal.

  Step1: Create a set or array to keep track of visited nodes.
  Step2: Choose a starting node.
  Step3: Create an empty stack and push the starting node onto the stack.
  Step4: Mark the starting node as visited.
  Step5: While the stack is not empty, do the following:
    - Pop a node from the stack.
    - Process or perform any necessary operations on the popped node.
    - Get all the adjacent neighbors of the popped node.
    - For each adjacent neighbor, if it has not been visited, do the following:
      - Mark the neighbor as visited.
      - Push the neighbor onto the stack.

  Step6: Repeat step 5 until the stack is empty.
- **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.S



- **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+ n2+ n3 +.........+ nm=O(nm)

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

2.3. HEURISTIC SEARCH TECHNIQUES

Many traditional search algorithms are used in AI applications. For complex problems, the traditional algorithms are unable to find the solutions within some practical time and space limits. Consequently, many special techniques are developed, using heuristic functions. The algorithms that use heuristic functions are called heuristic algorithms.

- Heuristic algorithms are not really intelligent; they appear to be intelligent because they achieve better performance.
- Heuristic algorithms are more efficient because they take advantage of feedback from the data to direct the search path.
- Uninformed search algorithms or Brute-force algorithms, search through the search space all possible candidates for the solution checking whether each candidate satisfies the problem's statement.
- Informed search algorithms use heuristic functions that are specific to the problem, apply them to guide the search through the search space to try to reduce the amount of time spent in searching.

A good heuristic will make an informed search dramatically outperform any uninformed search:
For example, the Traveling Salesman Problem (TSP), where the goal is to find is a good solution Instead of finding the best solution.
In such problems, the search proceeds using current information about the problem to predict which path is closer to the goal and follow it, although it does not always guarantee to find the best possible solution. Such techniques help in finding a solution within reasonable time and space (memory). Some prominent intelligent search algorithms are stated below:

1) **Hill Climbing**
2) **Generate and Test Search**
3) **A* Search**

2.3.1. Hill Climbing

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

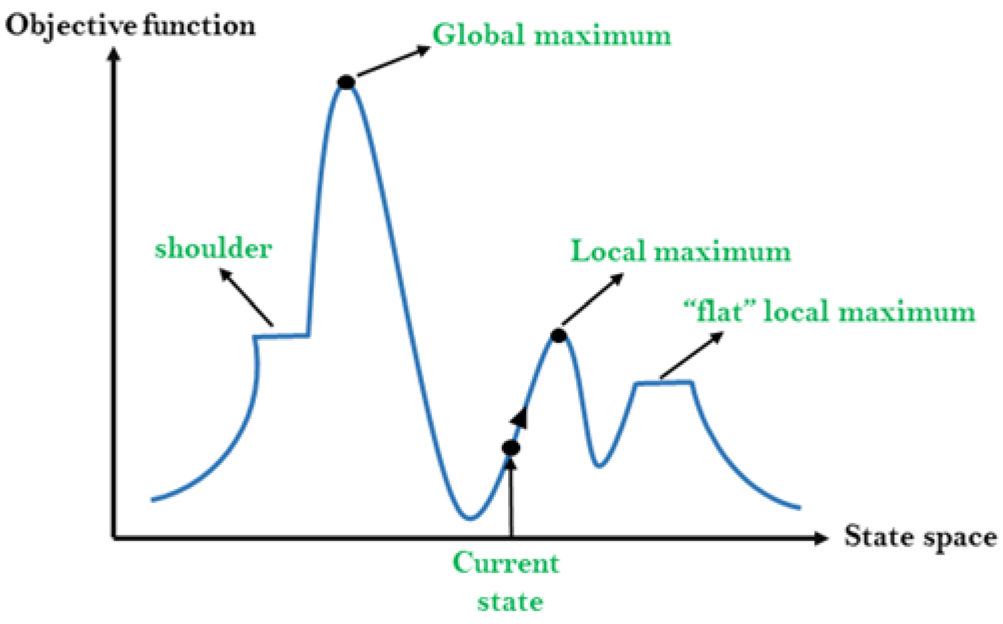
- **Features of Hill Climbing:**

  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.
- **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:
- **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:
  - If it is goal state, then return success and quit.
  - Else if it is better than the current state then assign new state as a current state.
  - Else if not better than the current state, then return to **step2**.
  - **Step 5:** Exit.
- **Steepest –Ascent hill climbing:**

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

Algorithm for Steepest-Ascent hill climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
- **Step 3:** Let SUCC be a state such that any successor of the current state will be better than it.
- **Step4:** For each operator that applies to the current state:
    - Apply the new operator and generate a new state.
    - Evaluate the new state.
    - If it is goal state, then return it and quit, else compare it to the SUCC.
    - If it is better than SUCC, then set new state as SUCC.
    - If the SUCC is better than the current state, then set current state to SUCC.
- **Step 5:** Exit.

- **Advantages of Hill Climbing:**
    - Hill Climbing can be used in continuous as well as domains.
    - These technique is very useful in job shop scheduling, automatic programming, circuit designing, and vehicle routing.
    - It is also helpful to solve pure optimization problems where the objective is to find the best state according to the objective function.

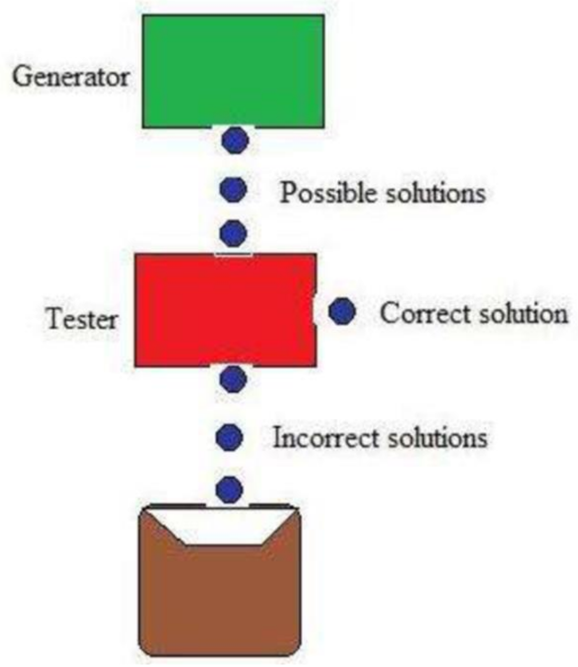
2.3.2. Generate-And-Test Search

The Generate-and-Test search algorithm is a problem-solving approach that involves generating potential solutions to a problem and then testing each solution against some criteria to determine if it satisfies the problem's requirements. If a solution meets the criteria, it is considered valid and the search process stops. If the solution does not meet the criteria, the algorithm generates and tests another solution.

Here's a general outline of the Generate-and-Test search algorithm:

1) **Generate:** Generate a potential solution to the problem.
2) **Test:** Evaluate the generated solution against the problem's criteria.
3) **Solution found:** If the generated solution meets the criteria, it is considered a valid solution, and the search process stops.
4) **Generate another solution:** If the generated solution does not meet the criteria, go back to step 1 and generate another potential solution.
5) **Termination condition:** Optionally, you can set a termination condition to stop the search after a certain number of iterations or if a specified time limit is reached.

The Generate-and-Test search algorithm is often used when the problem space is large and it's difficult to find an optimal solution using more sophisticated search methods. It is a simple and intuitive approach that can be effective in some cases, but it may not guarantee finding the best solution or be efficient for complex problems with large search spaces.



- **Example: coloured blocks**

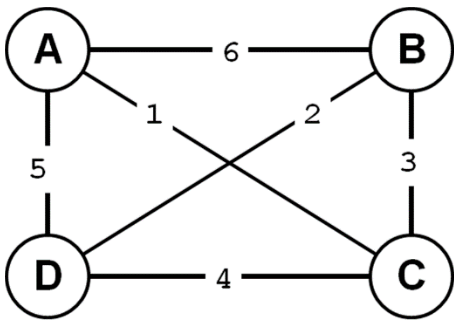
  "Arrange four 6-sided cubes in a row, with each side of each cube painted one of four colors, such that on all four sides of the row one block face of each color are showing."

  Heuristic: If there are more red faces than other colours then, when placing a block with several red faces, use few of them as possible as outside faces.

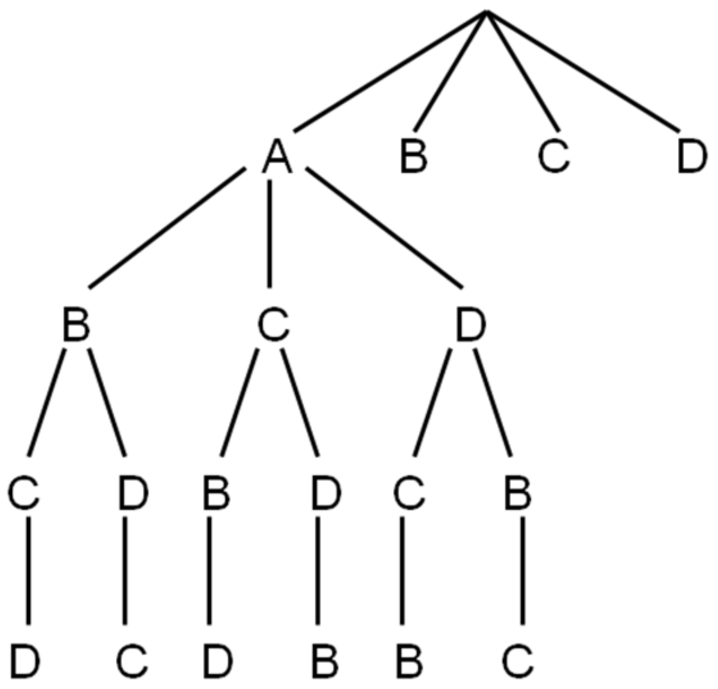
- **Example – Travelling Salesman Problem (TSP)**

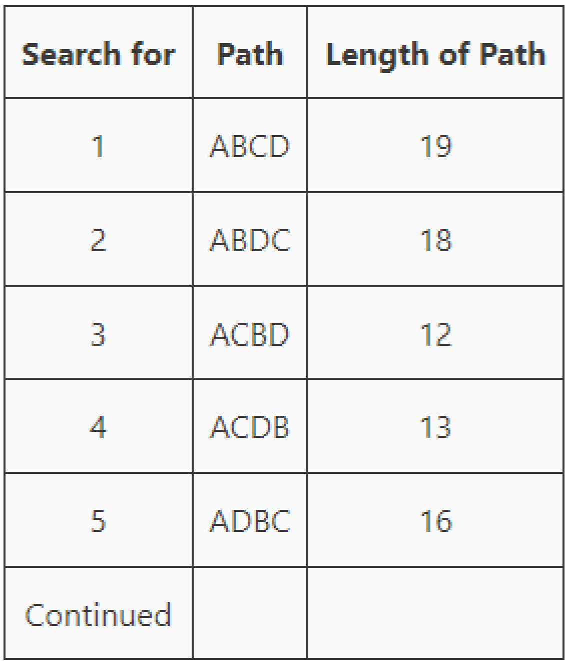
  A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.

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



Search flow with Generate and Test





Search for	Path	Length of Path
1	ABCD	19
2	ABDC	18
3	ACBD	12
4	ACDB	13
5	ADBC	16
Continued		

Finally, select the path whose length is less.

2.3.3. A* Search

The A* algorithm is a popular search algorithm used in pathfinding and graph traversal problems. It efficiently finds the optimal path between two nodes in a graph by considering both the actual cost of reaching a node from the start node and the estimated cost to the goal node. A* uses a heuristic function to estimate the cost, typically represented by the h(n) function, where n is a node in the graph.

Here's how the A* algorithm works:

1) Initialize the open set and closed set. The open set contains nodes that have been discovered but not yet explored, while the closed set contains nodes that have been visited.

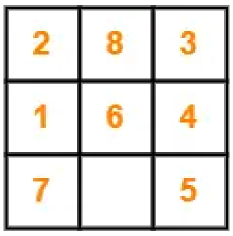
2) Add the start node to the open set and set its f-score to 0.
3) Repeat the following steps until either the goal node is reached or the open set is empty:
    1) Select the node from the open set with the lowest f-score. This node will be the current node.
    2) Move the current node from the open set to the closed set.
    3) Check if the current node is the goal node. If yes, the path has been found.
    4) Generate the neighboring nodes of the current node and calculate their g-scores (the actual cost to reach them from the start node).
    5) For each neighboring node, calculate its f-score using the formula f(n) = g(n) + h(n), where g(n) is the g-score and h(n) is the heuristic function.
    6) If a neighboring node is already in the closed set and its f-score is higher than the calculated f-score, skip it.
    7) If a neighboring node is already in the open set and its f-score is higher than the calculated f-score, update its g-score and f-score.
    8) If a neighboring node is not in the open set, add it to the open set with the calculated g-score and f-score.
4) If the open set becomes empty before reaching the goal node, there is no path available.

Once the A* algorithm terminates, the optimal path can be reconstructed by backtracking from the goal node to the start node, following the nodes with the lowest f-scores.

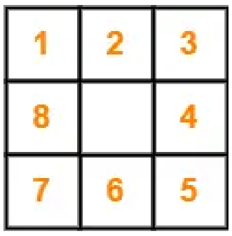
A* is considered efficient and optimal when the heuristic function h(n) satisfies certain properties, such as being admissible (never overestimating the actual cost) and consistent (satisfying the triangle inequality). The efficiency of the A* algorithm heavily depends on the choice of heuristic function and the characteristics of the problem domain.

Problem-01:

Given an initial state of a 8-puzzle problem and final state to be reached-



Initial State



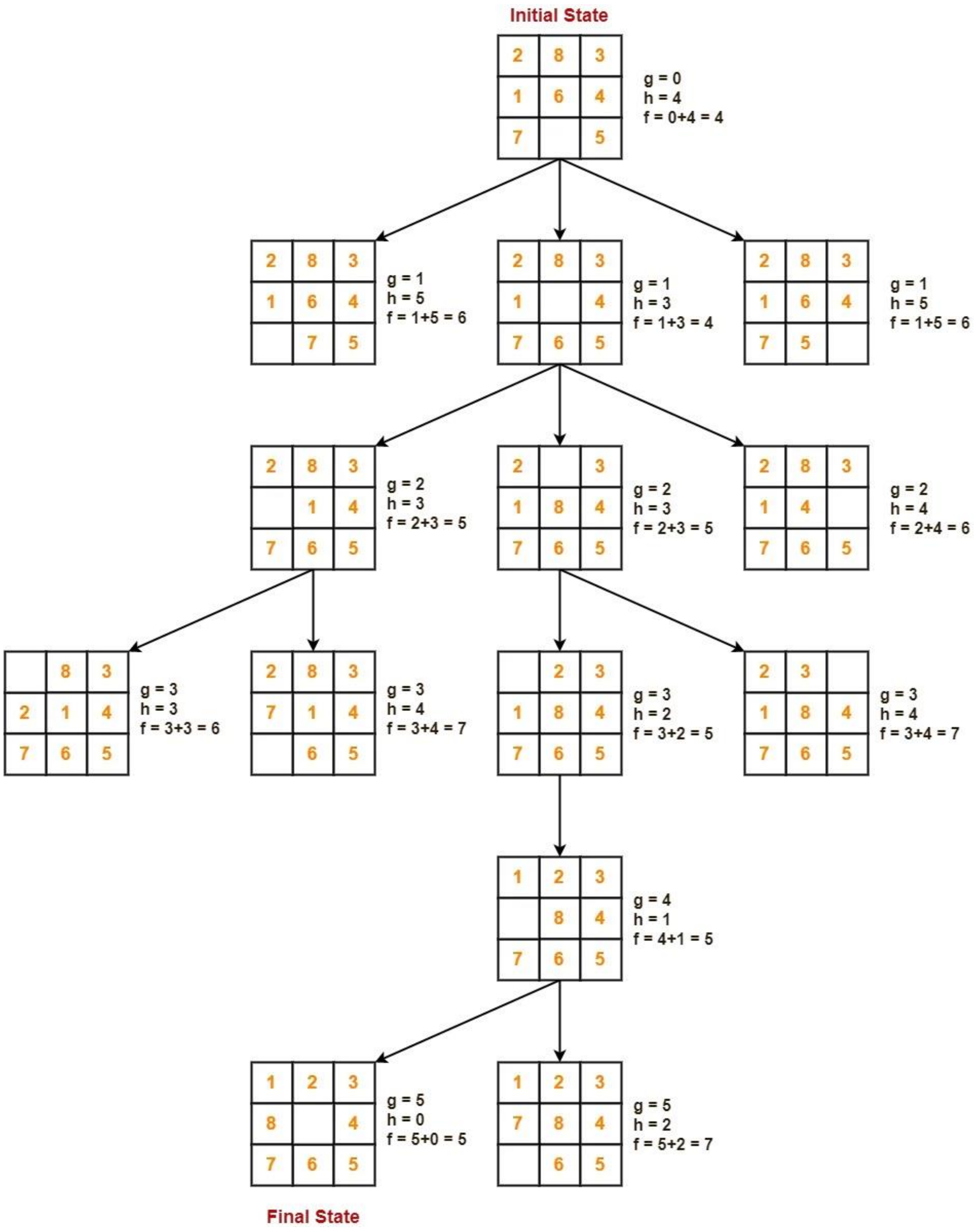
Final State

Find the most cost-effective path to reach the final state from initial state using A* Algorithm.

Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

Solution-

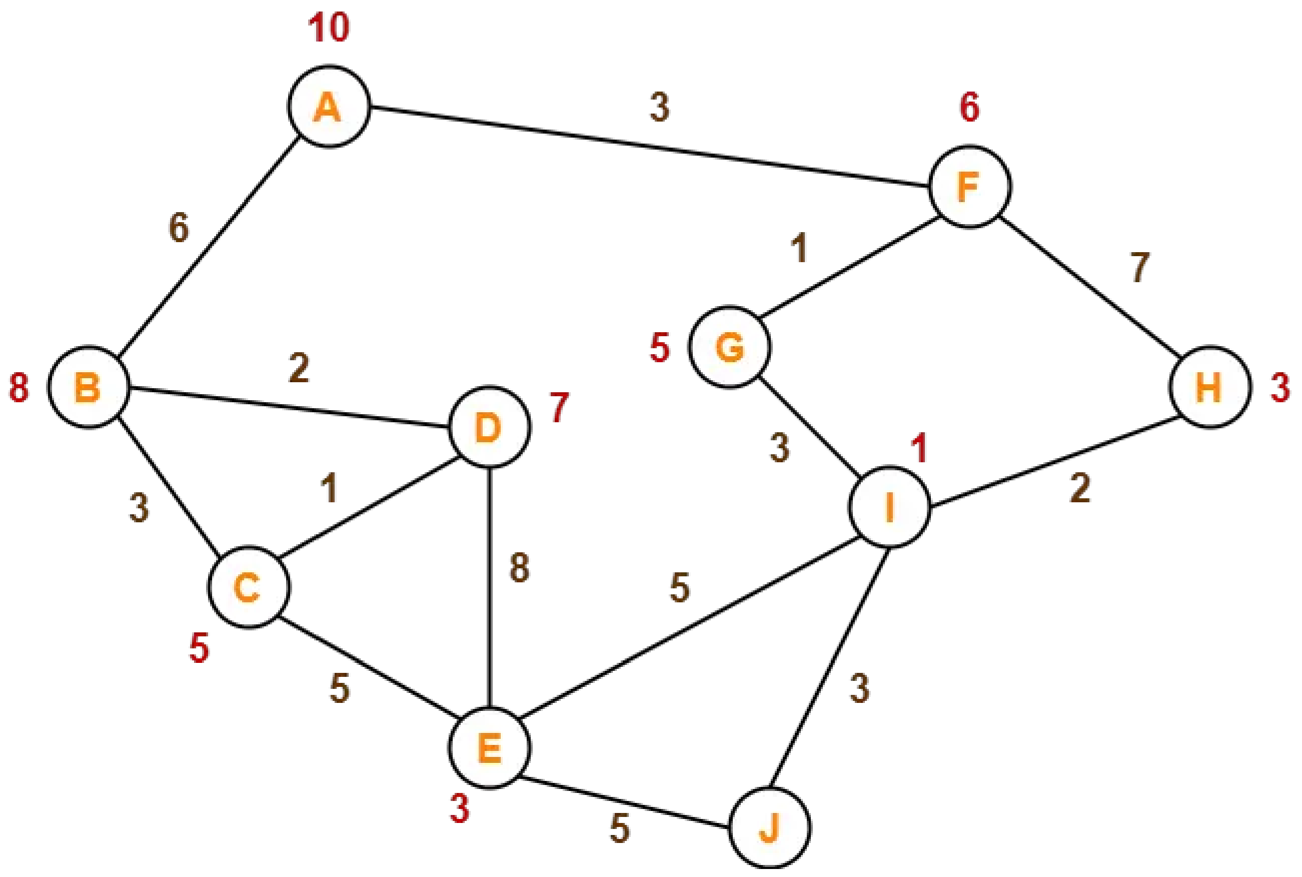
- A* Algorithm maintains a tree of paths originating at the initial state.



- It extends those paths one edge at a time.
- It continues until final state is reached.

Problem-02:

Consider the following graph-



- The numbers written on edges represent the distance between the nodes. The numbers written on nodes represent the heuristic value. Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.

- **Solution-**

  - **Step-01:**
    - We start with node A.
    - Node B and Node F can be reached from node A.

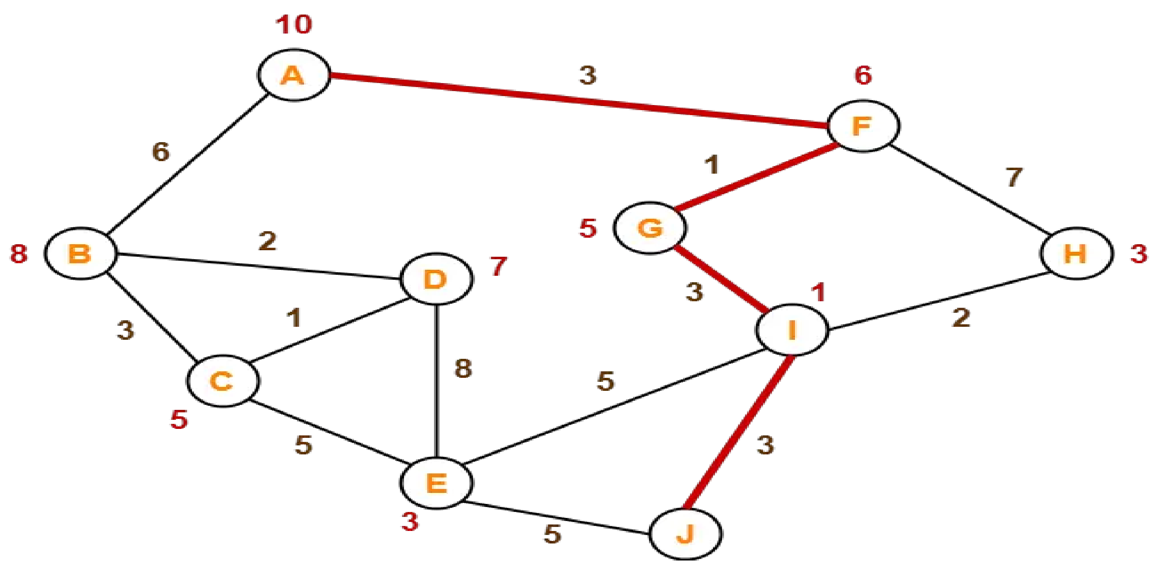
      A* Algorithm calculates f(B) and f(F).

      - f(B) = 6 + 8 = 14
      - f(F) = 3 + 6 = 9

    - Since f(F) < f(B), so it decides to go to node F.

Path- A → F

- **Step-02:**
  - Node G and Node H can be reached from node F.
  - A* Algorithm calculates f(G) and f(H).
    - f(G) = (3+1) + 5 = 9
    - f(H) = (3+7) + 3 = 13
  - Since f(G) < f(H), so it decides to go to node G.
  - Path- A → F → G
  - **Step-03:**
    - Node I can be reached from node G.
    - A* Algorithm calculates f(I).
    - f(I) = (3+1+3) + 1 = 8
    - It decides to go to node I.
    - Path- A → F → G → I
  - **Step-04:**
    - Node E, Node H and Node J can be reached from node I.
    - A* Algorithm calculates f(E), f(H) and f(J).
    - f(E) = (3+1+3+5) + 3 = 15
    - f(H) = (3+1+3+2) + 3 = 12
    - f(J) = (3+1+3+3) + 0 = 10
    - Since f(J) is least, so it decides to go to node J.
    - Path- A → F → G → I → J
    - This is the required shortest path from node A to node J.



2.4. Adversarial search techniques

- AI Adversarial search: Adversarial search is a game-playing technique where the agents are surrounded by a competitive environment. A conflicting goal is given to the agents (multiagent). These agents compete with one another and try to defeat one another in order to win the game. Such conflicting goals give rise to the adversarial search. 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.'

My Interest
"I Win"
Your Interest
"You Loose"

We are opponents- I win, you loose.

- Techniques required to get 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 fulfill 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 to approximate the cost value at each level of the search tree, before reaching the goal node.

2.4.1. 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.

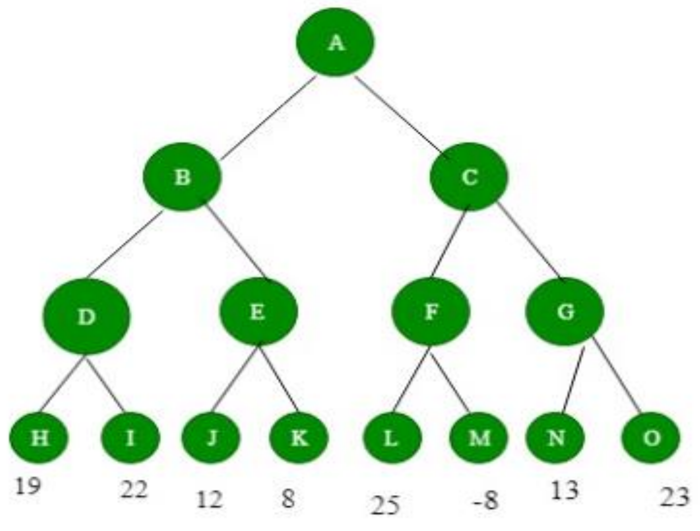


Figure 1: Before backing-up of values

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.

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.

- **Advantages of Game Playing in Artificial Intelligence:**
  - **Advancement of AI:** Game playing has been a driving force behind the development of artificial intelligence and has led to the creation of new algorithms and techniques that can be applied to other areas of AI.
  - **Education and training:** Game playing can be used to teach AI techniques and algorithms to students and professionals, as well as to provide training for military and emergency response personnel.
  - **Research:** Game playing is an active area of research in AI and provides an opportunity to study and develop new techniques for decision-making and problem-solving.
  - **Real-world applications:** The techniques and algorithms developed for game playing can be applied to real-world applications, such as robotics, autonomous systems, and decision support systems.
- **Disadvantages of Game Playing in Artificial Intelligence:**
  - **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.
  - **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.

2.4.2. MINIMAX algorithm

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 favor of one player and will unfavor the other one. The person who will make his best 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: Increases 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.

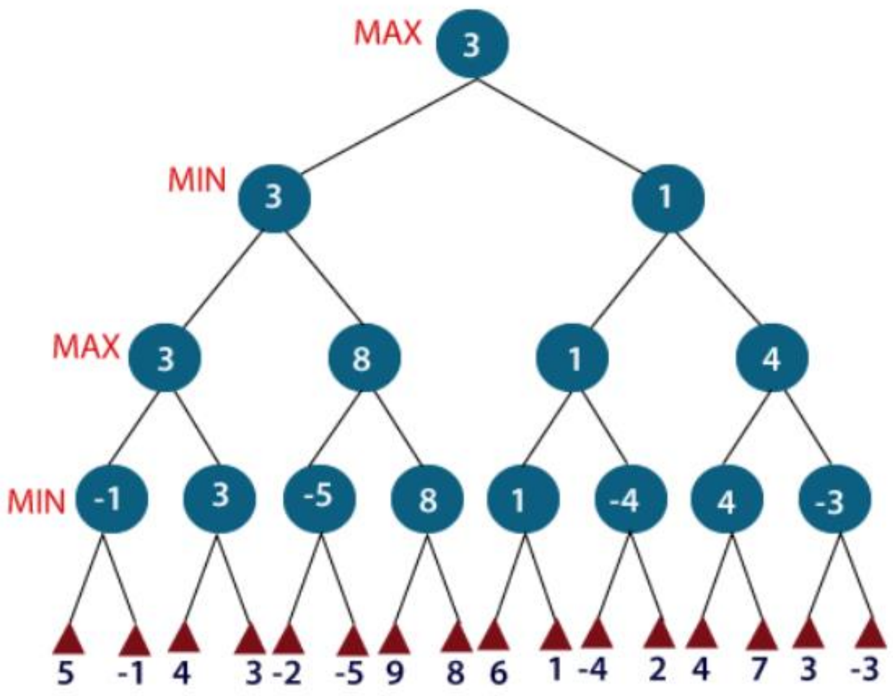
In MINIMAX strategy, the result of the game or the utility value is generated by a heuristic function by propagating from the initial node to the root node. It follows the backtracking technique and backtracks to find the best choice. MAX will choose that path which will increase its utility value and MIN will choose the opposite path which could help it to minimize MAX’s utility value.

MINIMAX Algorithm

MINIMAX algorithm is a backtracking algorithm where it backtracks to pick the best move out of several choices. MINIMAX strategy follows the DFS (Depth-first search) concept. 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 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.