

## Experiment 1

### Analysis of Breadth First Search (BFS) Program

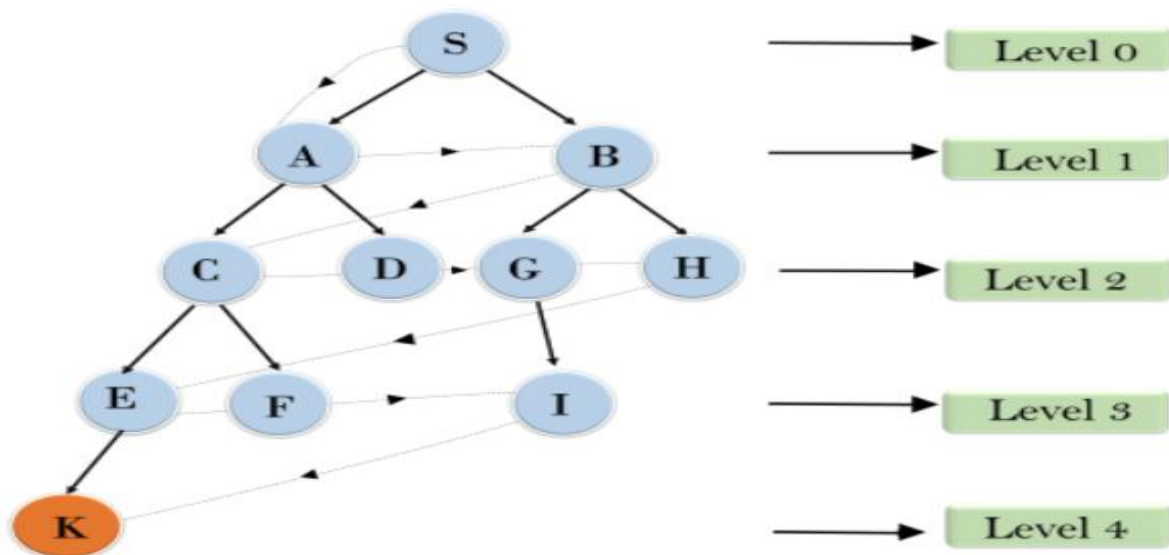
#### Aim:

The Breadth-First Search (BFS) algorithm traverses a graph level by level starting from a given node, exploring all neighboring nodes at the present depth before moving on to nodes at the next depth level.

#### Description:

##### I. Overview:

- Breadth-First Search (BFS) is an algorithm used to traverse or search through graph data structures.
- It begins at a specified starting node and explores all of its neighboring nodes at the current depth level before moving on to nodes at the next depth level.
- BFS uses a queue data structure to keep track of the nodes that need to be explored.



##### II. Traversal Mechanism:

- BFS traverses the graph level by level.
- It starts from a given node and explores all its neighbours before moving on to the neighbours' neighbours.

##### III. Queue Data Structure:

- BFS uses a queue to keep track of nodes to be explored.
- Nodes are enqueued when discovered and dequeued when processed.

##### IV. Visited Set:

- A visited set is used to keep track of nodes that have already been explored.
- This prevents reprocessing of nodes and avoids infinite loops.

## **V. Initialization:**

- The starting node is enqueued and marked as visited.

## **VI. Traversal Process:**

- Dequeue a node from the queue.
- Process the node (e.g., record its value or perform some operation).
- Enqueue all unvisited neighbours of the node.

## **VI. Termination:**

- The algorithm terminates when the queue is empty, indicating that all reachable nodes have been explored.

## **VII. Shortest Path:**

- BFS is effective for finding the shortest path in unweighted graphs since it explores all nodes at the current depth level before moving to the next level.

## **VIII. Applications:**

### **1. Shortest Path in Unweighted Graphs:**

- Finds the shortest path from the starting node to any other node.

### **2. Connected Components:**

- Identifies all nodes connected to a given starting node.

### **3. Level Order Traversal:**

- Used in trees to traverse nodes level by level.

## **IX. Complexity:**

### **1. Time Complexity:**

- $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges.

### **2. Space Complexity:**

- $O(V)$ , due to the storage of the queue and visited set.

## Code of Analysis of Breadth First Search (BFS) Program (in Python):

```
from collections import deque

def bfs(graph, start):
    visited = []
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.append(node)
            queue.extend(graph[node])

    return visited

graph = {
    'A': ['B', 'C', 'D'],
    'B': ['E'],
    'C': ['F'],
    'D': ['H'],
    'E': [],
    'F': [],
    'H': ['I', 'J'],
    'I': [],
    'J': []
}

print("BFS(Breadth First Search is):", bfs(graph, 'A'))
```

### Output:

BFS(Breadth First Search is): ['A', 'B', 'C', 'D', 'E', 'F', 'H', 'I', 'J']

### Analysis:

#### I. Initialization:

- The BFS function takes a graph (represented as an adjacency list) and a starting node as inputs.
- visited is an empty list that will store the nodes in the order they are visited.
- queue is a deque initialized with the starting node.

#### II. Traversal Mechanism:

- The BFS traversal begins by processing nodes in the queue.
- A while loop runs as long as the queue is not empty.

#### III. Node Processing:

- The node at the front of the queue is removed using pop left().
- If this node has not been visited, it is added to the visited list.

#### IV. Exploring Neighbours:

- All neighbours of the current node (obtained from the graph) are added to the end of the queue.
- This ensures that nodes are processed in the order they are discovered, maintaining the level-order traversal characteristic of BFS.

## **V. Termination:**

- The algorithm terminates when the queue is empty, meaning all reachable nodes have been visited.

## **VI. Output:**

- The function returns the visited list, which contains the nodes in the order they were visited during the BFS traversal.
- The BFS traversal starting from node 'A' proceeds as follows:
  1. Start at 'A': queue = ['A'], visited = []
  2. Visit 'A', enqueue neighbour's 'B', 'C', 'D': queue = ['B', 'C', 'D'], visited = ['A']
  3. Visit 'B', enqueue neighbour 'E': queue = ['C', 'D', 'E'], visited = ['A', 'B']
  4. Visit 'C', enqueue neighbour 'F': queue = ['D', 'E', 'F'], visited = ['A', 'B', 'C']
  5. Visit 'D', enqueue neighbour's 'H': queue = ['E', 'F', 'H'], visited = ['A', 'B', 'C', 'D']
  6. Visit 'E': queue = ['F', 'H'], visited = ['A', 'B', 'C', 'D', 'E']
  7. Visit 'F': queue = ['H'], visited = ['A', 'B', 'C', 'D', 'E', 'F']
  8. Visit 'H', enqueue neighbours 'I', 'J': queue = ['I', 'J'], visited = ['A', 'B', 'C', 'D', 'E', 'F', 'H']
  9. Visit 'I': queue = ['J'], visited = ['A', 'B', 'C', 'D', 'E', 'F', 'H', 'I']
  10. Visit 'J': queue = [], visited = ['A', 'B', 'C', 'D', 'E', 'F', 'H', 'I', 'J']

## **Conclusion:**

The provided BFS implementation effectively traverses a graph using a queue to explore nodes level by level. It ensures each node is visited once by maintaining a 'visited' list, preventing redundant processing and cycles. The algorithm is efficient with a time complexity of  $O(V + E)$  and space complexity of  $O(V)$ , making it suitable for various applications such as finding the shortest path in unweighted graphs. The example graph traversal demonstrates BFS's capability to explore all reachable nodes from the starting point in an orderly manner.

## Experiment 2

### Analysis of Depth First Search (DFS) Program

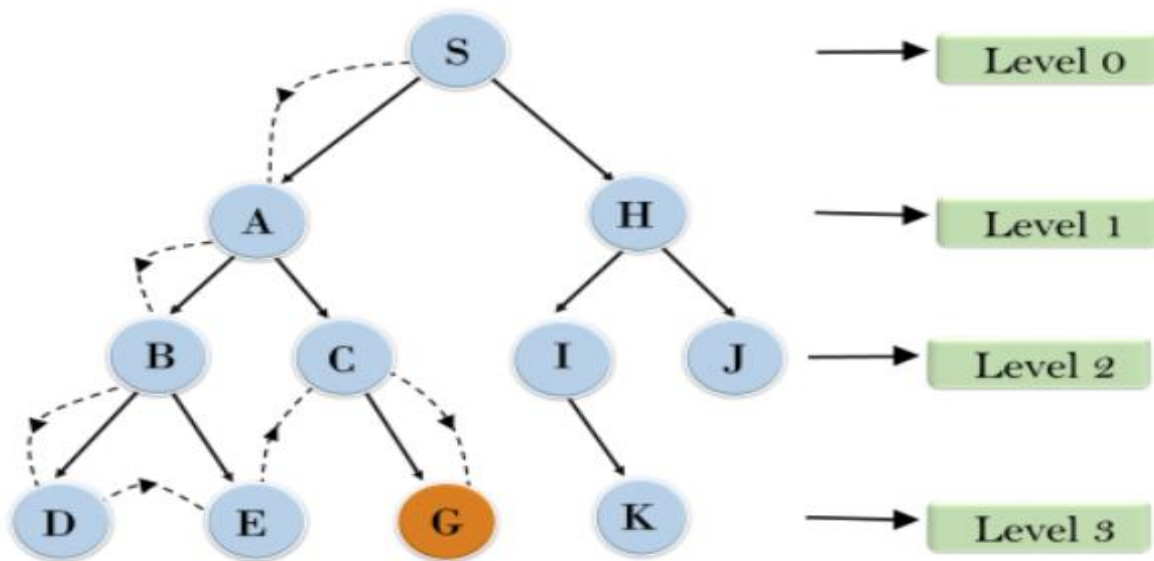
#### Aim:

The aim of analyzing a Depth-First Search (DFS) program is to understand how it explores a graph by diving deep into each branch before backtracking, utilizing a stack (either explicit or via recursion) to track the nodes.

#### Description:

##### I. Overview:

- Depth-First Search (DFS) is an algorithm used to traverse or search through graph data structures.
- It starts at a specified starting node and explores as far down each branch as possible before backtracking to explore other branches.
- DFS can be implemented using a stack (for an iterative approach) or recursion, leveraging the call stack to keep track of the nodes.



##### II. Traversal Mechanism:

###### 1. Deep Exploration:

- DFS explores the graph by going as deep as possible along each branch before backtracking.

###### 2. Branch-by-Branch:

- Each branch is fully explored before moving on to the next, ensuring thorough coverage of the graph.

### **III. Graph Representation:**

- The graph is typically represented as an adjacency list, where each node points to a list of its neighbours.
- This representation is efficient for storing sparse graphs and allows quick access to each node's neighbours.

### **IV. Stack Usage:**

- Uses an explicit stack to manage nodes, ensuring non-recursive depth-first traversal.
- Relies on the call stack to manage nodes, leveraging recursion for depth-first traversal.

### **V. Visited Set:**

- The visited set keeps track of nodes that have been processed to avoid redundant work.
- Helps prevent infinite loops by ensuring each node is visited only once.

### **VI. Node Processing:**

- Nodes are marked as visited upon first encounter.
- Each visited node can be processed, such as recording its value or performing specific operations.

### **VII. Exploring Neighbours:**

- For each unvisited neighbour of the current node, DFS is called recursively or the neighbour is pushed onto the stack.
- Ensures that all neighbours are explored before backtracking, maintaining the depth-first nature.

### **VIII. Backtracking:**

- When a node has no unvisited neighbours, DFS backtracks to the previous node.
- The algorithm continues exploration from nodes that have unexplored neighbours.

### **IX. Termination:**

- DFS terminates when all reachable nodes have been visited and the stack is empty (for iterative DFS) or all recursive calls are completed.
- Ensures that all nodes connected to the starting node are processed.

### **X. Complexity:**

- $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges. Each vertex and edge is processed once.
- $O(V)$ , due to the storage requirements for the visited set and the stack or call stack.

### Code of Analysis of Depth First Search (DFS) Program (in Python):

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = []
        visited.append(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

    return visited

graph = {
    'A': ['B', 'C', 'D'],
    'B': ['E'],
    'C': ['F'],
    'D': ['H'],
    'E': [],
    'F': [],
    'H': ['I', 'J'],
    'I': [],
    'J': []
}

print("DFS(Depth First Search is):", dfs(graph, 'A'))
```

### Output:

DFS(Depth First Search is): ['A', 'B', 'E', 'C', 'F', 'D', 'H', 'I', 'J']

### Analysis:

#### I. Function Definition:

- The dfs function is defined to perform a depth-first search on a graph.
- It takes three parameters: graph, start, and an optional visited list.

#### II. Initialization:

- The visited list is initialized as an empty list if not provided, ensuring each call has a list to track visited nodes.

#### III. Node Processing:

- The starting node (start) is appended to the visited list upon first encounter, marking it as processed.

#### **IV. Recursive Exploration:**

- The function iterates over each neighbour of the current node (start).
- If a neighbour has not been visited, the dfs function is called recursively with the neighbour as the new starting node.
- This deep exploration continues until all reachable nodes are visited.

#### **V. Backtracking:**

- When a node has no unvisited neighbours, the function backtracks to the previous node by returning from the recursive call.
- This process ensures all branches are fully explored before moving to another branch.

#### **VI. Termination:**

- The function returns the visited list after all nodes have been explored, providing the order of traversal.

#### **VII. Graph Representation:**

- The graph is represented as an adjacency list, where each node points to a list of its neighbours.
- This representation allows efficient access to the neighbours of each node.

#### **VIII. Example Graph Traversal:**

- Given the example graph, the DFS traversal starting from node 'A' will visit nodes in the order: A, B, E, C, F, D, H, I, J.
- The DFS traversal starting from node 'A' proceeds as follows:

##### **1. Start at 'A':**

- Initial state: visited = []
- Visit 'A': visited = ['A']

##### **2. Visit 'B' (neighbour of 'A'):**

- Current state: visited = ['A']
- Visit 'B': visited = ['A', 'B']

##### **3. Visit 'E' (neighbour of 'B'):**

- Current state: visited = ['A', 'B']
- Visit 'E': visited = ['A', 'B', 'E']

##### **4. Backtrack to 'B':**



- All neighbours of 'E' are visited, so return to 'B'.

#### **5. Backtrack to 'A':**

- All neighbours of 'B' are visited, continue with the next neighbour of 'A'.

#### **6. Visit 'C' (next neighbour of 'A'):**

- Current state: visited = ['A', 'B', 'E']
- Visit 'C': visited = ['A', 'B', 'E', 'C']

#### **7. Visit 'F' (neighbour of 'C'):**

- Current state: visited = ['A', 'B', 'E', 'C']
- Visit 'F': visited = ['A', 'B', 'E', 'C', 'F']

#### **8. Backtrack to 'C':**

- All neighbours of 'F' are visited, so return to 'C'.

#### **9. Backtrack to 'A':**

- All neighbours of 'C' are visited, continue with the next neighbour of 'A'.

#### **10. Visit 'D' (next neighbour of 'A'):**

- Current state: visited = ['A', 'B', 'E', 'C', 'F']
- Visit 'D': visited = ['A', 'B', 'E', 'C', 'F', 'D']

#### **11. Visit 'H' (neighbour of 'D'):**

- Current state: visited = ['A', 'B', 'E', 'C', 'F', 'D']
- Visit 'H': visited = ['A', 'B', 'E', 'C', 'F', 'D', 'H']

#### **12. Visit 'I' (neighbour of 'H'):**

- Current state: visited = ['A', 'B', 'E', 'C', 'F', 'D', 'H']
- Visit 'I': visited = ['A', 'B', 'E', 'C', 'F', 'D', 'H', 'I']

#### **13. Backtrack to 'H':**

- All neighbours of 'I' are visited, so return to 'H'.

#### **14. Visit 'J' (next neighbour of 'H'):**

- Current state: visited = ['A', 'B', 'E', 'C', 'F', 'D', 'H', 'I']
- Visit 'J': visited = ['A', 'B', 'E', 'C', 'F', 'D', 'H', 'I', 'J']

### **IX. Complexity:**

#### **1. Time Complexity:**

- $O(V + E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges. Each vertex and edge are processed once.

## **2. Space Complexity:**

- $O(V)$ , due to the storage requirements for the visited list and the call stack for recursive calls.

## **Conclusion:**

The DFS traversal starting from node 'A' systematically explores each node's depth before backtracking, visiting nodes in the order: A, B, E, C, F, D, H, I, J. This depth-first approach ensures that all reachable nodes are thoroughly explored from the starting point. By leveraging recursion, the algorithm maintains a stack of nodes to visit, marking nodes as visited to avoid redundant processing. The DFS traversal is efficient with a time complexity of  $O(V + E)$  and a space complexity of  $O(V)$ , making it suitable for tasks requiring comprehensive exploration of graph structures.

## Experiment 3

### Analysis of 8 Puzzle Problem

#### Code of Analysis of 8 Puzzle Problem without using Heuristics (in Python):

```
import numpy as np
from collections import deque
class PuzzleState:
    def __init__(self, matrix, moves=0, previous=None):
        self.matrix = np.array(matrix)
        self.moves = moves
        self.previous = previous
        self.blank_position = tuple(np.argwhere(self.matrix == 0)[0])
    def is_goal(self):
        goal = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
        return np.array_equal(self.matrix, goal)
    def generate_successors(self):
        successors = []
        x, y = self.blank_position
        possible_moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        for dx, dy in possible_moves:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_matrix = self.matrix.copy()
                new_matrix[x, y], new_matrix[new_x, new_y] = new_matrix[new_x,
new_y], new_matrix[x, y]
                successors.append(PuzzleState(new_matrix, self.moves + 1, self))
        return successors
```

```

def solve_puzzle(initial_state):
    start_state = PuzzleState(initial_state)
    if start_state.is_goal():
        print(start_state.matrix)
        return
    queue = deque([start_state])
    visited = set()
    visited.add(start_state.matrix.tobytes())
    while queue:
        current_state = queue.popleft()

        for successor in current_state.generate_successors():
            if successor.matrix.tobytes() not in visited:
                if successor.is_goal():
                    print(successor.matrix)
                    return
                queue.append(successor)
                visited.add(successor.matrix.tobytes())

if __name__ == "__main__":
    initial_state = [
        [1, 2, 3],
        [4, 8, 0],
        [7, 6, 5]
    ]
    solve_puzzle(initial_state)

```

### Output:

```

[[1 2 3]
 [4 5 6]
 [7 8 0]]

```

## Code of Analysis of 8 Puzzle Problem using Heuristics (in Python):

```
import numpy as np
from collections import deque
class PuzzleState:
    def __init__(self, matrix, moves=0, previous=None):
        self.matrix = np.array(matrix)
        self.moves = moves
        self.previous = previous
        self.blank_position = tuple(np.argwhere(self.matrix == 0)[0])
        self.heuristic = self.calculate_heuristic()
    def calculate_heuristic(self):
        goal = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
        return np.sum(self.matrix != goal) - 1 # Subtracting 1 to exclude the blank tile
    def is_goal(self):
        goal = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
        return np.array_equal(self.matrix, goal)
    def generate_successors(self):
        successors = []
        x, y = self.blank_position
        possible_moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in possible_moves:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_matrix = self.matrix.copy()
                new_matrix[x, y], new_matrix[new_x, new_y] = new_matrix[new_x,
new_y], new_matrix[x, y]
                successors.append(PuzzleState(new_matrix, self.moves + 1, self))
        successors.sort(key=lambda state: state.heuristic)
        return successors
```

```

def solve_puzzle(initial_state):
    start_state = PuzzleState(initial_state)
    if start_state.is_goal():
        print_solution_path(start_state)
        return
    current_state = start_state
    visited = set()
    visited.add(current_state.matrix.tobytes())
    while not current_state.is_goal():
        successors = current_state.generate_successors()
        for successor in successors:
            if successor.matrix.tobytes() not in visited:
                current_state = successor
                visited.add(successor.matrix.tobytes())
                break
        if current_state.is_goal():
            print_solution_path(current_state)
            return
def print_solution_path(state):
    print(state.matrix)
if __name__ == "__main__":
    initial_state = [
        [1, 2, 3],
        [0, 4, 6],
        [7, 5, 8]
    ]
    solve_puzzle(initial_state)

```

### Output:

```

[[1 2 3]
 [4 5 6]
 [7 8 0]]

```

## Experiment 4

### Analysis of Best First Search Problem

#### Code of Analysis of Best First Search Problem (in Python):

```
import heapq
class Graph:
    def __init__(self):
        self.graph = {}
        self.heuristics = {}
    def add_edge(self, node, neighbor, cost):
        if node not in self.graph:
            self.graph[node] = []
        self.graph[node].append((cost, neighbor))
    def set_heuristic(self, node, h_value):
        self.heuristics[node] = h_value
    def best_first_search(self, start, goal):
        open_list = []
        heapq.heappush(open_list, (self.heuristics[start], start))
        came_from = {start: None}
        visited = set()
        while open_list:
            _, current_node = heapq.heappop(open_list)
            if current_node == goal:
                print(f'Goal {goal} found!')
                return self.reconstruct_path(came_from, start, goal)
            if current_node in visited:
                continue
            visited.add(current_node)
```

```

        for cost, neighbor in self.graph.get(current_node, []):
            if neighbor not in visited:
                heapq.heappush(open_list, (self.heuristics[neighbor], neighbor))
                came_from[neighbor] = current_node
                print(f"Visited: {current_node}, Exploring: {neighbor}")
        print("Goal not found.")
        return None

def reconstruct_path(self, came_from, start, goal):
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = came_from[current]
    path.reverse()
    return path

g = Graph()
g.add_edge('S', 'A', 3)
g.add_edge('S', 'B', 2)
g.add_edge('A', 'C', 4)
g.add_edge('A', 'D', 1)
g.add_edge('B', 'E', 3)
g.add_edge('B', 'F', 1)
g.add_edge('E', 'H', 5)
g.add_edge('F', 'T', 2)
g.add_edge('F', 'G', 3)
g.set_heuristic('S', 13)
g.set_heuristic('A', 12)
g.set_heuristic('B', 4)
g.set_heuristic('C', 7)

```



```
g.set_heuristic('D', 3)
g.set_heuristic('E', 8)
g.set_heuristic('F', 2)
g.set_heuristic('G', 0)
g.set_heuristic('H', 4)
g.set_heuristic('I', 9)
path = g.best_first_search('S', 'G')
print("Path:", path)
```

### **Output:**

```
Visited: S, Exploring: A
Visited: S, Exploring: B
Visited: B, Exploring: E
Visited: B, Exploring: F
Visited: F, Exploring: I
Visited: F, Exploring: G
Goal G found!
Path: ['S', 'B', 'F', 'G']
```

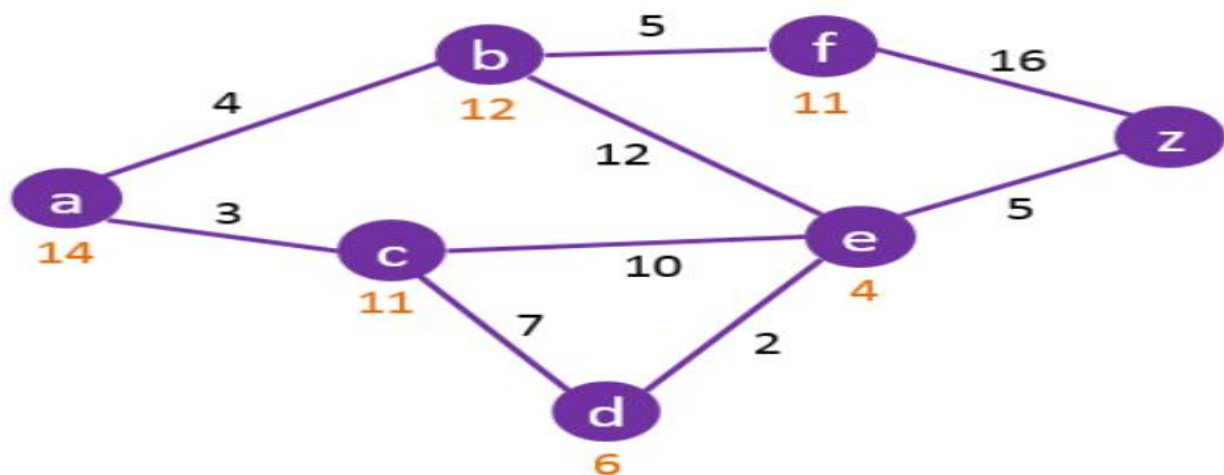
## Experiment 5

### Analysis of A\* Algorithm

#### Aim:

The aim of analyzing the A\* algorithm is to assess its efficiency in finding the shortest path by evaluating its balance between exploration and exploitation, and its performance in terms of time and space complexity.

#### Description:



The analysis of the A\* algorithm involves evaluating several key aspects to understand its performance and effectiveness:

- **Efficiency:** Assess how well the algorithm finds the shortest path in terms of computational resources, including time complexity.
- **Heuristic Function:** Evaluate the impact of different heuristic functions on the algorithm's performance, accuracy, and efficiency.
- **Optimality:** Determine if the algorithm guarantees finding the optimal path, given the heuristic is admissible and consistent.
- **Completeness:** Analyse whether the algorithm always finds a solution if one exists, and under what conditions.
- **Space Complexity:** Examine the amount of memory required by the algorithm to store nodes and paths during execution.
- **Scalability:** Investigate how the algorithm performs with increasing problem size and complexity.
- **Comparative Analysis:** Compare the A\* algorithm's performance with other pathfinding algorithms in various scenarios to understand its relative strengths and weaknesses.

## Code of Analysis of Analysis of A\* Algorithm (in Python):

```
import heapq
class Node:
    def __init__(self, name, parent=None):
        self.name = name
        self.parent = parent
        self.g = 0
        self.h = 0
        self.f = 0 |
    def __eq__(self, other):
        return self.name == other.name
    def __lt__(self, other):
        return self.f < other.f

    def __hash__(self):
        return hash(self.name)
def a_star_all_paths(graph, start, goal, heuristic):
    start_node = Node(start)
    goal_node = Node(goal)
    open_list = []
    paths = []
    heapq.heappush(open_list, start_node)
    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.name == goal:
            path = []
            total_cost = current_node.g
            while current_node:
                path.append(current_node.name)
                current_node = current_node.parent
            paths.append((path[::-1], total_cost))
            continue
        neighbors = graph.get(current_node.name, {})
        for neighbor_name, cost in neighbors.items():
            neighbor_node = Node(neighbor_name, current_node)
            neighbor_node.g = current_node.g + cost
            neighbor_node.h = heuristic.get(neighbor_name, 0)
            neighbor_node.f = neighbor_node.g + neighbor_node.h
            heapq.heappush(open_list, neighbor_node)
    return paths
def find_best_path(paths):
    if not paths:
        return None, None
    best_path = min(paths, key=lambda x: x[1])
    return best_path
graph = {
    'S': {'A': 1, 'G': 10},
    'A': {'B': 2, 'C': 1},
    'B': {'D': 5},
    'C': {'D': 3, 'G': 4},
    'D': {'G': 2},
    'G': {}
}
heuristic = {
    'S': 5,
    'A': 3,
    'B': 4,
    'C': 2,
    'D': 6,
    'G': 0
}
start = 'S'
goal = 'G'
all_paths = a_star_all_paths(graph, start, goal, heuristic)
print("All paths and their respective costs are:")
for path, cost in all_paths:
    print(f"Path: {' -> '.join(path)}, Cost: {cost}")
if all_paths:
    best_path, best_cost = find_best_path(all_paths)
    print("\nBest Path:")
    print(f"Path: {' -> '.join(best_path)}, Cost: {best_cost}")
else:
    print("\nNo path found from start to goal.")
```

## Output:

All paths and their respective costs are:

Path: S -> A -> C -> G, Cost: 6

Path: S -> G, Cost: 10

Path: S -> A -> C -> D -> G, Cost: 7

Path: S -> A -> B -> D -> G, Cost: 10

Best Path:

Path: S -> A -> C -> G, Cost: 6

## Analysis:

### ➤ Functionality:

The code finds all possible paths from the start node to the goal node using the A\* algorithm. It computes paths and their costs, then identifies the optimal path based on the lowest cost.

### ➤ Algorithm:

#### 1. Node Class:

Defines nodes with attributes for cost (g, h, f), and supports comparison and hashing.

#### 2. A Pathfinding\*:

Uses a priority queue (min-heap) to explore nodes, tracking paths and costs. Paths are stored and returned once the goal is reached.

#### 3. Heuristic:

A heuristic function estimates the cost from a node to the goal, helping to prioritize nodes.

### ➤ Performance:

#### 1. Time Complexity:

Depends on the size of the graph and the priority queue operations. Generally, A\* performs efficiently with a good heuristic.

#### 2. Space Complexity:

Stores all nodes in the priority queue and the paths list, which can grow with the graph size.

### ➤ Output:

Displays all found paths with their costs and highlights the optimal path.

## **Conclusion:**

The A\* algorithm implementation effectively finds and evaluates all potential paths from the start node to the goal node. By using a heuristic function to guide the search, it optimizes the pathfinding process, balancing exploration and exploitation. The algorithm provides a comprehensive solution by storing and displaying all possible paths and their respective costs. It identifies and highlights the optimal path with the lowest total cost, demonstrating its efficiency and suitability for complex pathfinding problems. Overall, this implementation showcases A\*'s capability to deliver precise and cost-effective pathfinding results.

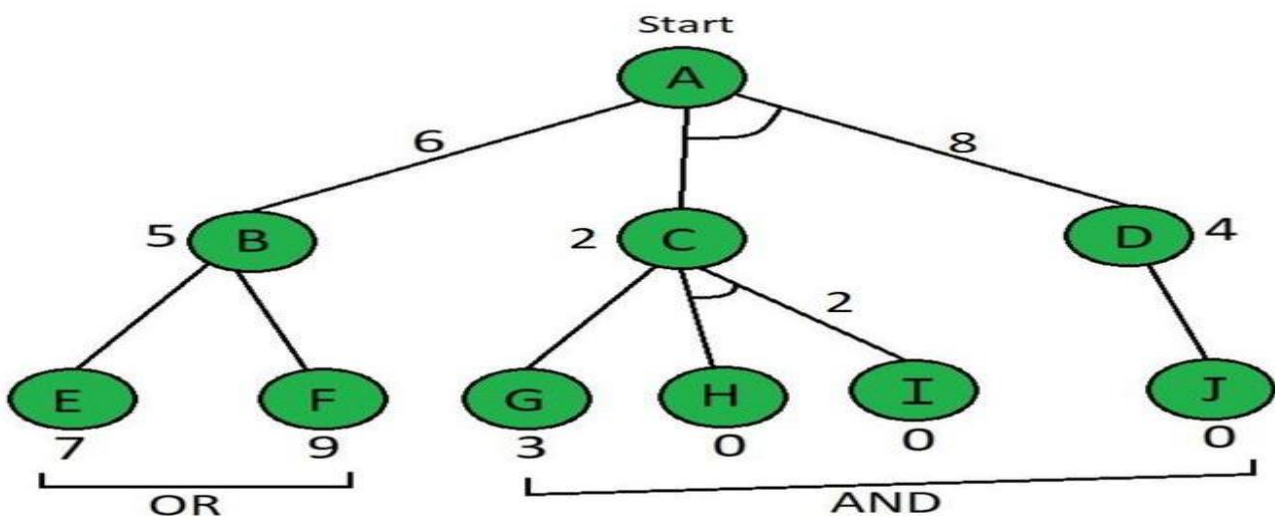
## Experiment 6

### Analysis of AO\* Algorithm

#### Aim:

The aim of the AO\* algorithm is to find the optimal path in a search space that includes both "AND" and "OR" nodes. It minimizes total cost while effectively handling complex structures, evaluates the costs of reaching each node based on heuristics, and incrementally updates paths. This makes it suitable for dynamic decision-making scenarios, such as planning and resource allocation.

#### Description:



#### 1. Node Types:

- **OR Nodes:** Represent choices where at least one child must be selected to progress. These nodes are used to model decisions with multiple alternatives.
- **AND Nodes:** Represent conditions where all child nodes must be satisfied. These nodes are used to model requirements that must be fulfilled simultaneously.

#### 2. Goal:

The primary goal of the AO\* algorithm is to identify the least-cost path from a starting node to a goal node by effectively exploring both AND and OR relationships.

#### 3. Cost Evaluation:

- Each node has an associated heuristic value, which estimates the cost to reach the goal from that node.
- The algorithm computes the total cost for each node based on the costs of its child nodes and their respective relationships (AND or OR).

## 4. Search Strategy:

- AO\* uses a best-first search strategy, exploring nodes in order of increasing estimated cost.
- It incrementally updates the cost of nodes as it explores the graph, allowing previously computed paths to be reused and adjusted as necessary.

## 5. Pathfinding:

- The algorithm maintains the best path found so far for each node and updates it when a cheaper path is discovered.
- Once a node is solved (i.e., its best path has been determined), it does not need to be revisited.

## 6. Applications:

- AO\* is particularly useful in domains like AI planning, resource allocation, and situations where decisions involve both alternatives and constraints.

## Code of Analysis of Analysis of AO\* Algorithm (in Python)

```
def Cost(H, condition, weight=1):
    cost = {}
    if 'AND' in condition:
        AND_nodes = condition['AND']
        Path_A = ' AND '.join(AND_nodes)
        PathA = sum(H[node] + weight for node in AND_nodes)
        cost[Path_A] = PathA
    if 'OR' in condition:
        OR_nodes = condition['OR']
        Path_B = ' OR '.join(OR_nodes)
        PathB = min(H[node] + weight for node in OR_nodes)
        cost[Path_B] = PathB
    return cost

def update_cost(H, Conditions, weight=1):
    Main_nodes = list(Conditions.keys())
    Main_nodes.reverse()
    least_cost = {}
    for key in Main_nodes:
        condition = Conditions[key]
        c = Cost(H, condition, weight)
        H[key] = min(c.values())
        least_cost[key] = Cost(H, condition, weight)
    return least_cost

def shortest_path(Start, Updated_cost, H):
    Path = Start
    total_cost = H[Start] if Start not in Updated_cost else min(Updated_cost[Start].values())
    if Start in Updated_cost.keys():
        Min_cost = min(Updated_cost[Start].values())
        key = list(Updated_cost[Start].keys())
        values = list(Updated_cost[Start].values())
        Index = values.index(Min_cost)
        Next = key[Index].split()
        if len(Next) == 1:
            Start = Next[0]
            sub_path, _ = shortest_path(Start, Updated_cost, H)
            Path += '-' + sub_path
        else:
            Path += '-' + key[Index] + ')'
            Start = Next[0]
            sub_path_1, _ = shortest_path(Start, Updated_cost, H)
            Start = Next[-1]
            sub_path_2, _ = shortest_path(Start, Updated_cost, H)
            Path += '[' + sub_path_1 + ' + ' + sub_path_2 + ']'
    return Path, total_cost

H = {'A': -1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}
Conditions = {
    'A': {'OR': ['D'], 'AND': ['B', 'C']},
    'B': {'OR': ['G', 'H']},
    'D': {'AND': ['E', 'F']}
}
weight = 1
Updated_cost = update_cost(H, Conditions, weight)
path, least_cost = shortest_path('A', Updated_cost, H)
print('Shortest Path:\n', path)
print(f'Least Cost at A: {least_cost}')
```

## Output:

```
Shortest Path:  
A-D-(E AND F) [E + F]  
Least Cost at A: 11
```

## Analysis:

### I. Components

1. **Heuristic Dictionary (H)**: Stores the costs associated with each node.
2. **Conditions Dictionary**: Defines relationships (AND/OR) between nodes.
3. **Weight**: A constant used to adjust cost calculations.

### II. Functions

#### ➤ **Cost(H, condition, weight)**:

1. Calculates costs for nodes based on their conditions:
  - i. For **AND** nodes, sums the costs of all children.
  - ii. For **OR** nodes, finds the minimum cost among children.
2. Returns a dictionary of computed costs.

#### ➤ **update\_cost(H, Conditions, weight)**:

1. Updates the heuristic values in H for each node based on its conditions.
2. Utilizes the Cost function to determine minimum costs and returns them.

#### ➤ **shortest\_path(Start, Updated\_cost, H)**:

1. Constructs the path starting from the specified node.
2. Recursively calculates the total cost and builds the path by evaluating children nodes.
3. Handles both AND and OR conditions for path construction.
4. Returns the constructed path and total cost.

### III. Execution Flow

1. Initializes heuristic values and conditions.
2. Calls update\_cost to compute costs.
3. Uses shortest\_path to find and display the shortest path and least cost.



## **Conclusion:**

The AO\* algorithm efficiently solves problems with both AND and OR conditions by finding the least-cost path. It dynamically updates heuristic values and prunes unnecessary paths, focusing on the most promising routes. This makes it effective for complex decision-making tasks, ensuring faster and optimal solutions compared to traditional algorithms.

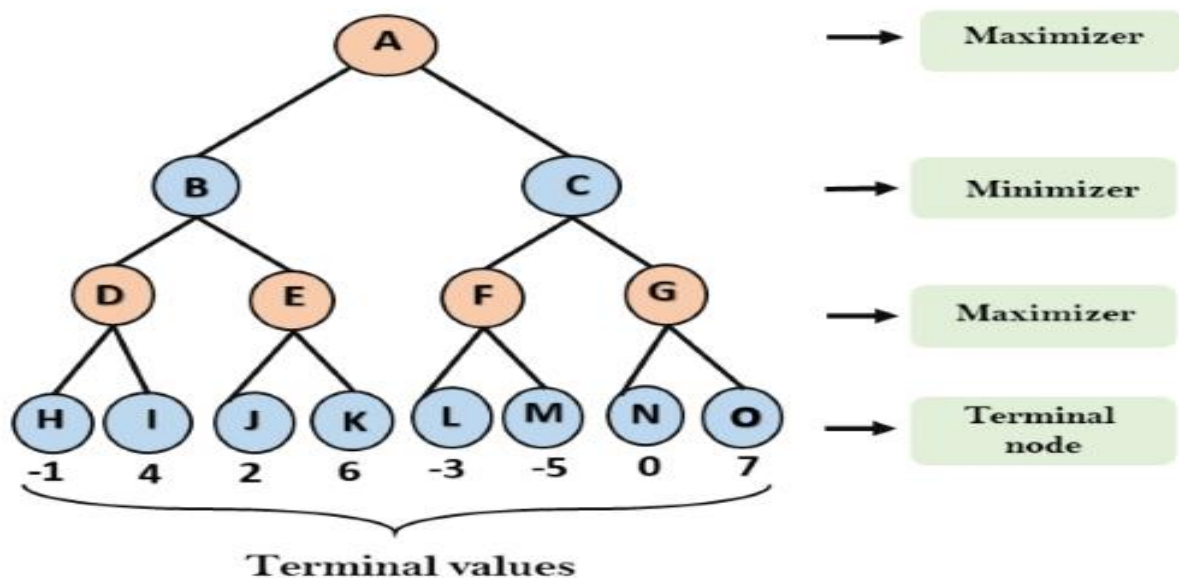
## Experiment 7

### Analysis of Min-Max Algorithm

#### Aim:

The aim of analyzing the Min-Max algorithm in Artificial Intelligence is to evaluate decision-making strategies in competitive environments. It helps identify the optimal move for a player by minimizing the possible loss for a worst-case scenario, assuming the opponent plays optimally. This algorithm is primarily used in two-player games like chess to simulate intelligent behavior.

#### Description:



## I. Introduction

The Min-Max algorithm is a decision-making strategy used in artificial intelligence, especially in two-player games like chess and tic-tac-toe. The algorithm helps simulate the optimal decision for a player by considering the possible moves of both players.

## II. Objective of Min-Max Algorithm

- To determine the best possible move for a player assuming the opponent is also playing optimally.
- Minimize the possible loss in the worst-case scenario (hence the name Min-Max).

## III. Working of Min-Max Algorithm

## **1. Game Tree Representation**

- The game is represented as a tree where each node represents a game state.
- The branches represent the possible moves by both players.

## **2. Maximizing Player (Max)**

- This player tries to maximize their chances of winning.
- The algorithm assumes that the maximizing player will choose the move that leads to the highest score.

## **3. Minimizing Player (Min)**

- This player attempts to minimize the maximizing player's score.
- The algorithm assumes that the opponent will also play optimally to minimize the chances of the maximizing player winning.

## **4. Evaluation Function**

- At the leaf nodes of the game tree (where the game ends), an evaluation function assigns scores to represent the result of the game.
- For instance, in a game like tic-tac-toe, a win for Max might score +1, a loss -1, and a draw 0.

## **IV. Steps of the Algorithm**

### **1. Generate the Game Tree**

- Construct the game tree by simulating all possible moves by both players.

### **2. Evaluate Terminal Nodes**

- Apply the evaluation function to terminal states (end of the game) to assign a score to each outcome.

### **3. Backpropagate Scores**

- From the terminal nodes, backpropagate the scores to the parent nodes. Max nodes choose the maximum score, and Min nodes choose the minimum score.

### **4. Choose the Optimal Move**

- At the root of the tree (initial game state), the Max player selects the move that leads to the highest score, assuming the Min player tries to minimize it.

## **V. Advantages of Min-Max Algorithm**

- Provides a clear framework for decision-making in competitive games.
- Ensures an optimal strategy if both players play optimally.

## VI. Limitations

- **Time Complexity:** The algorithm can be slow due to the exponential growth of the game tree.
- **Inability to Handle Uncertainty:** It assumes both players play perfectly, which may not be the case in real-world scenarios.

## Alpha-Beta Pruning (Optimization)

- An optimization technique that reduces the number of nodes evaluated in the game tree.
- Its “prunes” or cuts off branches of the game tree that won’t affect the final decision, making the Min-Max algorithm more efficient.

## Code of Analysis of Analysis of Min Max Algorithm (in Python)

```
import math
tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [3, 5],
    'E': [6, 9],
    'F': [1, 2],
    'G': [0, -1]
}
leaf_values = {
    'D': [3, 5],
    'E': [6, 9],
    'F': [1, 2],
    'G': [0, -1]
}
def is_terminal(state):
    return state in leaf_values
def utility(state):
    return max(leaf_values[state])
def actions(state):
    return tree[state] if state in tree else []
def result(state, action):
    return action
def minmax(state, depth, maximizing_player):
    if is_terminal(state) or depth == 0:
        return utility(state)
    if maximizing_player:
        max_eval = -math.inf
        for action in actions(state):
            eval = minmax(result(state, action), depth - 1, False)
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = math.inf
        for action in actions(state):
            eval = minmax(result(state, action), depth - 1, True)
            min_eval = min(min_eval, eval)
        return min_eval
optimal_value = minmax('A', 3, True)
```

## Output:

The final value at the root node is: 5

## Analysis:

### 1. Introduction

This code demonstrates the Min-Max algorithm, used for decision-making in two-player games. It evaluates the best move by exploring a tree of game states and simulates both players playing optimally.

### 2. Key Functions

- **is\_terminal(state)**: Determines if a state is a terminal (leaf) node.
- **utility(state)**: Retrieves the maximum value of a terminal node.
- **actions(state)**: Returns the list of possible moves (child nodes) from a given state.
- **minmax(state, depth, maximizing\_player)**: Recursively explores the game tree, maximizing for one player and minimizing for the opponent.

### 3. Working

- **Maximizing Player**: Tries to select the move with the highest value.
- **Minimizing Player**: Simulates the opponent, selecting the lowest possible value.
- **Base Case**: The recursion ends when a terminal node is reached or the depth limit is 0.

### 4. Execution

- Starting from root 'A', the algorithm explores all possible moves.
- After evaluating the children's nodes of 'A', the final optimal value returned to the root is 5.

## Conclusion:

The Min-Max algorithm effectively models decision-making in two-player, zero-sum games by simulating all possible moves and selecting the optimal strategy. In this code, the algorithm recursively evaluates the game tree, alternating between maximizing the player's score and minimizing the opponent's score. By using terminal nodes and evaluating their utility, the algorithm ensures that the best possible outcome is achieved under the assumption that both players play optimally.

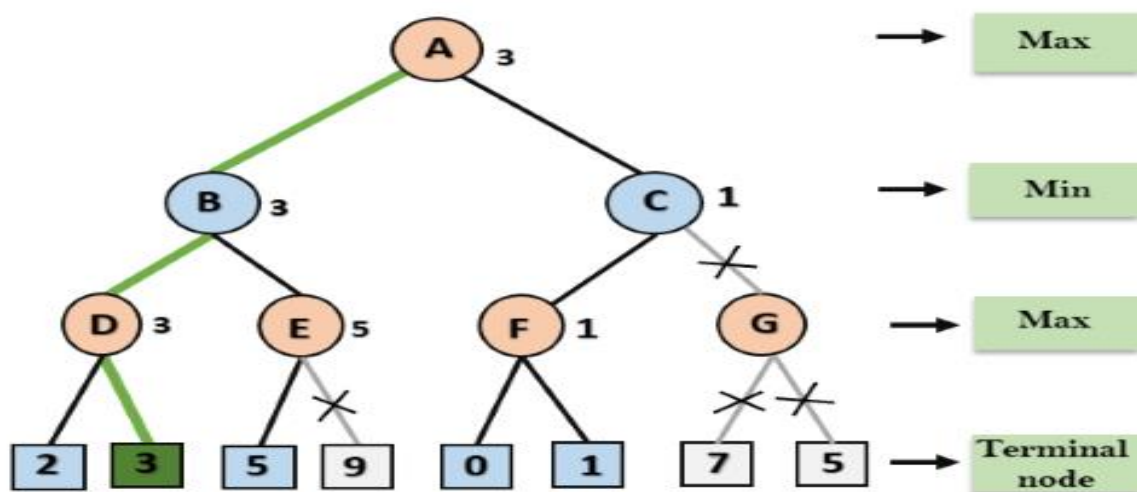
## Experiment 8

### Analysis of Min-Max Algorithm Using Alpha Beta Pruning

#### Aim:

The aim of analyzing the **Min-Max algorithm with Alpha-Beta Pruning** is to optimize decision-making in competitive environments by identifying the best possible move while reducing unnecessary calculations. This approach enhances the efficiency of evaluating game states, assuming both players play optimally, and is commonly used in two-player games like chess.

#### Description:



#### I. Introduction

The Min-Max algorithm with Alpha-Beta Pruning is a decision-making strategy used in artificial intelligence, especially for two-player games like chess and tic-tac-toe. It helps simulate the optimal decision by reducing the number of moves evaluated, making the process more efficient without compromising the outcome.

#### II. Objective of Min-Max Algorithm with Alpha-Beta Pruning

- To determine the best possible move for a player while assuming the opponent plays optimally.
- To optimize the efficiency of the Min-Max algorithm by eliminating unnecessary calculations, reducing the time complexity of decision-making in large game trees.

#### III. Working of Min-Max Algorithm with Alpha-Beta Pruning

##### 1. Game Tree Representation

- The game is represented as a tree where each node is a game state, and branches are possible moves.

## 2. Maximizing Player (Max)

- This player tries to maximize their score, selecting moves that lead to the highest possible gain.

## 3. Minimizing Player (Min)

- This player tries to minimize the score of the maximizing player, aiming to minimize losses.

## 4. Alpha-Beta Pruning

- Alpha represents the best score that the maximizing player can guarantee.
- Beta represents the best score that the minimizing player can guarantee.
- When evaluating nodes, if a certain branch cannot produce a better outcome than already considered options (determined by alpha and beta), that branch is "pruned" or skipped, saving computational effort.

## 5. Evaluation Function

- An evaluation function assigns scores at the leaf nodes of the game tree (end of the game), representing the outcome of that game state.

# IV. Steps of the Algorithm

## 1. Generate the Game Tree

- Construct the game tree by simulating all possible moves for both players.

## 2. Apply Alpha-Beta Pruning

- Set initial alpha to  $-\infty$  and beta to  $+\infty$ .
- Traverse the game tree while maintaining and updating alpha and beta values.
- If a node's value is worse than the current alpha or beta, that branch is pruned (skipped).

## 3. Evaluate Terminal Nodes

- The algorithm applies the evaluation function to the terminal states, scoring them based on the outcome (e.g., +1 for Max's win, -1 for Min's win, 0 for a draw).

## 4. Backpropagate Scores

- From terminal nodes, scores are backpropagated to parent nodes. Max nodes choose the maximum score, while Min nodes select the minimum score.

## 5. Choose the Optimal Move

- At the root node, the maximizing player selects the move that leads to the highest score, considering the opponent's optimal strategy and after pruning irrelevant branches.

## V. Advantages of Alpha-Beta Pruning

- **Reduces Computation:** Significantly cuts down the number of nodes that need to be evaluated, making the algorithm faster and more efficient.
- **Retains Optimal Decision:** Pruning does not affect the final decision, ensuring the outcome remains optimal while speeding up the search.

## VI. Limitations

- **Complexity with Large Trees:** Although pruning improves efficiency, large and complex game trees can still pose a challenge, especially if the branching factor is high.
- **Perfect Play Assumption:** It assumes both players play perfectly, which may not always be realistic in human behaviour.

## Code of Analysis of Analysis of Min Max Algorithm (in Python)

```
import math
tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [3, 5],
    'E': [6, 9],
    'F': [1, 2],
    'G': [0, -1]
}
leaf_values = {
    'D': [3, 5],
    'E': [6, 9],
    'F': [1, 2],
    'G': [0, -1]
}
def is_terminal(state):
    return state in leaf_values
def utility(state):
    return max(leaf_values[state])
def actions(state):
    return tree[state] if state in tree else []
def result(state, action):
    return action
def alphabeta(state, depth, alpha, beta, maximizing_player):
    if is_terminal(state) or depth == 0:
        return utility(state)
    if maximizing_player:
        max_eval = -math.inf
        for action in actions(state):
            eval = alphabeta(result(state, action), depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = math.inf
        for action in actions(state):
            eval = alphabeta(result(state, action), depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval
optimal_value = alphabeta('A', 3, -math.inf, math.inf, True)
print(f"The optimal value at the root node is: {optimal_value}")
```



## Output:

The optimal value at the root node is: 5

## Analysis:

### 1. Introduction

The code implements the Alpha-Beta Pruning algorithm, an optimization of Min-Max for two-player games. It reduces the number of nodes evaluated by pruning irrelevant branches, improving efficiency.

### 2. Data Structures

- **Tree Representation:** The game is modelled as a tree, where nodes represent game states and leaves represent outcomes.
- **Leaf Values:** Terminal nodes hold utility values that indicate the result of the game.

### 3. Key Functions

- **is\_terminal(state):** Checks if a node is terminal (end of the game).
- **utility(state):** Returns the utility value for terminal states.
- **actions(state):** Provides possible moves (child nodes) from a state.

### 4. Alpha-Beta Pruning Algorithm

- **Maximizing Player:** Tries to maximize the game's outcome.
- **Minimizing Player:** Tries to minimize the maximizing player's score.
- **Alpha and Beta:** Alpha tracks the best option for Max, and Beta tracks the best for Min. Pruning occurs when further evaluation won't change the outcome.

## Conclusion:

The Alpha-Beta Pruning algorithm efficiently enhances the Min-Max algorithm by reducing the number of nodes evaluated in a game tree, without affecting the optimal decision. By pruning branches that don't influence the final outcome, it optimizes decision-making in competitive games, making it more computationally efficient while maintaining the same result as the standard Min-Max algorithm. This makes it particularly useful for games like chess, where deep decision trees can grow exponentially.

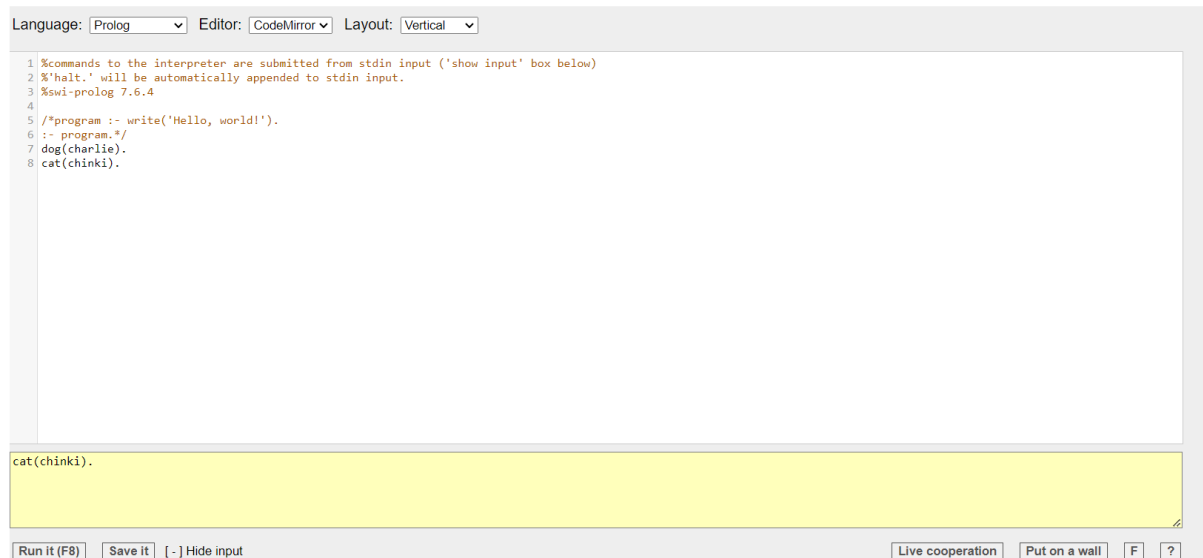
## Experiment 9

### Analysis of Prolog Programming Language

#### What is Prolog?

Prolog is a declarative programming language, widely used in artificial intelligence research, as opposed to the imperative programming paradigm such as Java or C.

#### 1. compile prolog online



```
1 %commands to the interpreter are submitted from stdin input ('show input' box below)
2 %'halt.' will be automatically appended to stdin input.
3 %swi-prolog 7.6.4
4
5 /*program :- write('Hello, world!').
6 :- program.*
7 dog(charlie).
8 cat(chinki).
```

cat(chinki).

Run it (F8) Save it [ - ] Hide input Live cooperation Put on a wall F ?

#### Rules:

##### 1. Full Stop (.):

- In Prolog, every fact or rule must end with a **full stop (.)**. This indicates that the statement is complete, and Prolog can interpret it.

##### 2. Fact 1: dog(charlie).

- This is a fact in Prolog, which states that "charlie" is a dog. This means whenever Prolog is asked if "charlie" is a dog, it will return true.

##### 3. Fact 2: cat(chinki).

- Similarly, this is another fact that states "chinki" is a cat. So, if you ask Prolog if "chinki" is a cat, it will return true.

Run it (F8) Save it [ - ] Hide input

Absolute running time: 0.3 sec, cpu time: 0.15 sec, memory peak: 7 Mb, absolute service time: 0,45 sec

true.

This is a question asked to the Prolog interpreter: "Is chinki a cat?" Based on the fact given earlier, Prolog will answer true, as we already declared that chinki is indeed a cat.

## Output:

The output "true." confirms that Prolog has successfully matched the query to the fact cat(chinki)..

## How Prolog Works:

- In Prolog, **facts** like dog(charlie). or cat(chinki). are simple declarations of truth. They don't require further reasoning.
- **Queries** allow you to ask Prolog questions based on these facts. In this case, cat(chinki) is a query, and since this matches the fact provided, the answer is true.

## 2. compile prolog online

```
1 %commands to the interpreter are submitted from stdin input ('show input' box below)
2 %'halt.' will be automatically appended to stdin input.
3 %swi-prolog 7.6.4
4
5 /*program :- write('Hello, world!').
6 :- program.*/
7
8 dog(mani).
9 cat(chinki).
10
11 is_animal(X):-dog(X);cat(X).
```

is\_animal(chinki).

## Rules:

### 1. Full Stop (.):

- In Prolog, every fact or rule must end with a **full stop (.)**. This indicates that the statement is complete, and Prolog can interpret it.

### 2. Fact 1: dog(mani).

- This fact states that "**mani is a dog.**" Whenever Prolog is asked if mani is a dog, it will return true.

### 3. Fact 2: cat(chinki).

- This fact states that "**chinki is a cat.**" Whenever Prolog is asked if chinki is a cat, it will return true.

#### 4. Rule 1: `is_animal(X) :- dog(X); cat(X).`

- This is a **rule** that defines the condition under which something is an animal. It states that **X is an animal if X is a dog or X is a cat**. The ; operator means "or." The :- symbol means "if".
- This rule checks if the argument (X) is either a dog or a cat, and if so, concludes that it is an animal.

Run it (F8)

Save it

[ - ] Hide input

Absolute running time: 0.3 sec, cpu time: 0.15 sec, memory peak: 7 Mb, absolute service time: 0,42 sec

true.

#### Output:

The output "true." confirms that Prolog has successfully matched the query to the fact `cat(chinki)`.

---

#### How Prolog Works:

- In Prolog, **facts** like `dog(mani).` or `cat(chinki).` are simple declarations of truth. These are direct statements that Prolog accepts without further reasoning.
- **Queries** allow you to ask Prolog questions based on these facts. In the given case, the query `cat(chinki)` checks if "chinki" is a cat. Since this matches a previously defined fact, Prolog returns the answer "true.", indicating the match.