EXPERIMENT NO. 1

Date of Performance:

Date of Submission:

Aim: One case study on AI applications published in IEEE/ACM/Springer or any prominent journal.

Theory: Automatic Quiz Generator.

Publication Year: 2021

Abstract:

In the digital age, effective knowledge retention and self-assessment play a crucial role in enhancing learning experiences. Traditional reading methods often fail to engage users, making it challenging to retain information over time. This paper presents Automatic Quiz Generator, an intelligent system that transforms text images into Multiple Choice Questions (MCQs), enriching learning through automated quiz creation. Unlike conventional summarization tools, this system extracts key information from text images, augments it with additional insights from online sources, and formulates high-quality quizzes. The core methodology integrates multiple Natural Language Processing (NLP) techniques, including Optical Character Recognition (OCR) for text extraction, keyword extraction to identify essential concepts, and web scraping to retrieve supplementary content. Leveraging deep learning models, the system ensures context-aware question generation and meaningful answer choices. Implemented using Django, NLTK, and Spacy, the model enhances adaptability across various textual formats, including handwritten notes, printed books, PDFs, and digital documents. By automating quiz creation, this system enables students, educators, and self-learners to assess their knowledge dynamically and efficiently. The proposed solution not only reduces manual effort but also fosters an interactive, data-driven approach to learning. Future enhancements will explore multilingual support, speech synthesis, and adaptive learning techniques to further personalize user experience and improve accessibility in education.

Conclusion: Thus, We have Studied case study on AI applications published in IEEE/ACM/ Springer or any prominent journal

Sign and Remark:

R1 (2 Marks)	R2 (4 Marks)	R3 (4 Marks)	Total (10 Marks)	Signature

EXPERIMENT NO. 2

Date of Performance:

Date of Submission:

Aim: Assignments on State-space formulation and PEAS representation for various AI applications

Theory:

The PEAS (Performance Measures, Environment, Actuators, and Sensors) framework is used to define and analyze intelligent agents in Artificial Intelligence. Performance measures evaluate how well the agent performs its tasks, while the environment represents the external conditions in which the agent operates. Actuators enable the agent to take actions and interact with the environment, while sensors collect data to help the agent perceive its surroundings. This framework helps in designing AI systems by clearly outlining their components and functionalities.

The PEAS (Performance Measures, Environment, Actuators, Sensors) framework for a Quiz Generator in AI would involve measuring the accuracy and relevance of generated MCQs (performance), the input text or images provided by users (environment), NLP models and web scraping tools for quiz generation (actuators), and OCR and keyword extraction for analyzing the input text (sensors).

PEAS Components:

The PEAS (Performance measure, Environment, Actuators, and Sensors) model for the Automatic Quiz Generator can be structured as follows:

PEAS Model for Automatic Quiz Generator:

1. **Performance Measure (P):**
   - Accuracy and relevance of generated MCQs.
   - Ability to extract meaningful keywords from text.
   - Effectiveness in retrieving supplementary information from the web.
   - User engagement and ease of quiz generation.
   - Reduction in manual effort for quiz creation.
   - Speed and efficiency of processing text images into quizzes.
2. **Environment (E):**
   - Input sources: Handwritten notes, printed books, PDFs, screenshots, online documents.
   - Users: Students, educators, self-learners, researchers.
   - Internet availability: Needed for web scraping additional information.
   - Device constraints: Can run on personal computers or cloud-based platforms.

3. **Actuators (A):**
   - Software modules for OCR to extract text from images.
   - Keyword extraction algorithms to identify key topics.
   - Web scraping tools to retrieve supplementary information.
   - NLP and deep learning models to generate structured MCQs.
   - Web-based Django interface to display quizzes.

4. **Sensors (S):**
   - Optical Character Recognition (OCR) engine (PyTesseract) for text extraction.
   - NLP libraries (NLTK, Spacy) for keyword extraction and text processing.
   - Web crawlers (BeautifulSoup) for online content retrieval.
   - Deep learning models for analyzing text and generating question-answer pairs.
   - User input interface to receive text images and preferences.

This PEAS model effectively describes the intelligent system's functioning and components.

Conclusion: Thus, We have Studied case study on State space formulation and PEAS representation for various AI applications.

Sign and Remark:

R1 (2 Marks)	R2 (4 Marks)	R3 (4 Marks)	Total (10 Marks)	Signature

EXPERIMENT NO. 3

Date of Performance:

Date of Submission:

Aim: To implement Breadth First Search as Uniformed Search

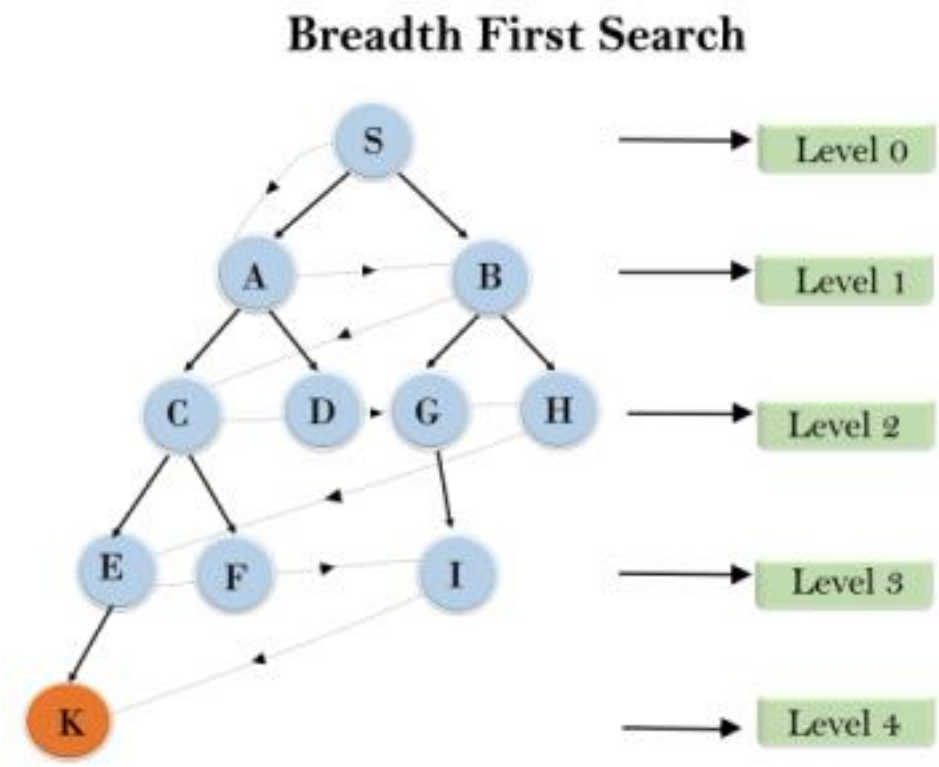
Theory:

✓ 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.



Algorithm:

The steps of the algorithm work as follow:

1. Start by putting any one of the graph's vertices at the back of the queue. 2. Now take the front item of the queue and add it to the visited list.

3. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.

4. Keep continuing steps two and three till the queue is empty.

Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

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

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

BFS Pseudocode:

The pseudocode for BFS in python goes as below:

create a queue Q

mark v as visited and put v into Q

while Q is non-empty

remove the head u of Q

mark and enqueue all (unvisited) neighbors of u

1. Uniformed Search Algorithms

a) BFS

Program:

```
from collections import defaultdict, deque
class Graph:
        def __init__(self):
                self.graph = defaultdict(list)
        def add_edge(self, u, v):
                self.graph[u].append(v)
```

```
def bfs(self, start):
        visited = set()
        queue = deque([start])
        while queue:
            current_node = queue.popleft()
            if current_node not in visited:
            print(current_node, end=" ")
            visited.add(current_node)
            for neighbor in self.graph[current_node]:
                if neighbor not in visited:
                    queue.append(neighbor)
graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)
print("BFS starting from node 2:")
graph.bfs(2)
```

Output:

BFS starting from node 2:
2 0 3 1 >

```
from collections import defaultdict, deque
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def add_edge(self, u, v):
        self.graph[u].append(v)
    def bfs(self, start):
        visited = set()
        queue = deque([start])
        while queue:
            current_node = queue.popleft()
            if current_node not in visited:
                print(current_node, end=" ")
                visited.add(current_node)
                for neighbor in self.graph[current_node]:
                    if neighbor not in visited:
                        queue.append(neighbor)
graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)
print("BFS starting from node 2:")
graph.bfs(2)
```

```
BFS starting from node 2:
2 0 3 1 >
```

b) DFS

program:

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def add_edge(self, u, v):
        self.graph[u].append(v)
    def dfs(self, start, visited=None):
        if visited is None:
            visited = set()
        if start not in visited:
            print(start, end=" ")
            visited.add(start)
            for neighbor in self.graph[start]:
                self.dfs(neighbor, visited)
# Example Usage
graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)
print("DFS starting from node 2:")
graph.dfs(2)
```

Output:

```
DFS starting from node 2:
2 0 1 3
=== Code Execution Successful ===
```

c) UCS

Program:

```
from queue import PriorityQueue
class Graph:
    def __init__(self):
        self.graph = {}
    def add_edge(self, u, v, cost):
```

```
 if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append((v, cost))
        self.graph[v].append((u, cost))  # Assuming an undirected graph
    def ucs(self, start, goal):
        visited = set()
pq = PriorityQueue()
        pq.put((0, start))  # (cost, node)
        while not pq.empty():
            cost, current = pq.get()
            if current in visited:
                continue
            print(f"Visiting Node: {current} with Cost: {cost}")
            visited.add(current)
            if current == goal:
                print(f"Reached goal {goal} with cost {cost}")
                return
            for neighbor, edge_cost in self.graph.get(current, []):
                if neighbor not in visited:
                    pq.put((cost + edge_cost, neighbor))
# Example Usage
graph = Graph()
graph.add_edge(0, 1, 1)
graph.add_edge(0, 2, 4)
graph.add_edge(1, 2, 2)
graph.add_edge(1, 3, 5)
graph.add_edge(2, 3, 1)
print("\nUCS from node 0 to 3:")
graph.ucs(0, 3)
```

Output:

```
UCS from node 0 to 3:
Visiting Node: 0 with Cost: 0
Visiting Node: 1 with Cost: 1
Visiting Node: 2 with Cost: 3
Visiting Node: 3 with Cost: 4
Reached goal 3 with cost 4

=== Code Execution Successful ===
```

2. Informed Search Algorithm

a) A* Search

Program:

```
import heapq
class Node:
    def __init__(self, name, g, h):
        self.name = name  # Node name
        self.g = g  # Cost from start node

 self.h = h  # Heuristic value
        self.f = g + h  # Evaluation function f(n) = g(n) + h(n)
        self.parent = None  # To track path

def __lt__(self, other):
        return self.f < other.f  # Priority queue comparison
def a_star_search(graph, heuristics, start, goal):
    open_list = []  # Priority queue
    closed_set = set()  # Explored nodes
    start_node = Node(start, 0, heuristics[start])
    heapq.heappush(open_list, start_node)
    while open_list:
        current_node = heapq.heappop(open_list)  # Get node with lowest f(n)
        if current_node.name == goal:  # Goal check
            path = []
            while current_node:
                path.append(current_node.name)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path
        closed_set.add(current_node.name)
        for neighbor, cost in graph[current_node.name].items():
            if neighbor in closed_set:
                continue
            g_new = current_node.g + cost
            h_new = heuristics[neighbor]
            new_node = Node(neighbor, g_new, h_new)
            new_node.parent = current_node
            heapq.heappush(open_list, new_node)  # Push new node to open list
    return None  # No path found
# Example graph representation
```

```
graph = {
   'A': {'B': 1, 'C': 4},
   'B': {'A': 1, 'D': 2, 'E': 5},
   'C': {'A': 4, 'F': 3},
   'D': {'B': 2, 'G': 1},
   'E': {'B': 5, 'H': 3},
   'F': {'C': 3, 'I': 4},
   'G': {'D': 1, 'J': 2},
   'H': {'E': 3, 'J': 6},
   'I': {'F': 4, 'J': 5},
  'J': {'G': 2, 'H': 6, 'I': 5}
}
# Example heuristic values (assumed straight-line distance to goal 'J')
heuristics = {
   'A': 7, 'B': 6, 'C': 5, 'D': 4, 'E': 5,
   'F': 3, 'G': 2, 'H': 4, 'I': 3, 'J': 0
}
# Running A* search
goal_path = a_star_search(graph, heuristics, 'A', 'J')
print("Optimal Path:", goal_path)
```

Output:

```
Optimal Path: ['A', 'B', 'D', 'G', 'J']


...Program finished with exit code 0
Press ENTER to exit console.
```

b) Greedy Best First Seach

Program:

```
def greedy_best_first_search(graph, heuristics, start, goal):
   open_list = [(heuristics[start], start)]  # Priority queue sorted by heuristic value
   came_from = {}
   closed_set = set()
   while open_list:
      open_list.sort()  # Sort by heuristic value
      _, current = open_list.pop(0)
      if current == goal:
         path = []
         while current in came_from:
            path.append(current)
```

```
current = came_from[current]
            path.append(start)
            return path[::-1]
        closed_set.add(current)
        for neighbor in graph[current]:
            if neighbor not in closed_set:
                open_list.append((heuristics[neighbor], neighbor))
                came_from[neighbor] = current
    return None
# Example graph representation
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'D': 2, 'E': 5},
    'C': {'A': 4, 'F': 3},
    'D': {'B': 2, 'G': 1},
    'E': {'B': 5, 'H': 3},
    'F': {'C': 3, 'I': 4},
    'G': {'D': 1, 'J': 2},
    'H': {'E': 3, 'J': 6},
    'I': {'F': 4, 'J': 5},
    'J': {'G': 2, 'H': 6, 'I': 5}
}
heuristics = {
    'A': 7, 'B': 6, 'C': 5, 'D': 4, 'E': 5,
    'F': 3, 'G': 2, 'H': 4, 'I': 3, 'J': 0
}
path_gbfs = greedy_best_first_search(graph, heuristics, 'A', 'J')
print("Greedy Best-First Search Path:", path_gbfs)
```

Output:

```
Greedy Best-First Search Path: ['A', 'C', 'F', 'I', 'J']


...Program finished with exit code 0
Press ENTER to exit console.
```

c. Beam Search

Program:

```
from queue import PriorityQueue
class Node:
    def __init__(self, state, cost, parent=None):
        self.state = state  # The current node state
```

```
        self.cost = cost  # The heuristic cost
        self.parent = parent  # Parent node (to reconstruct path)
    def __lt__(self, other):
        return self.cost < other.cost  # For priority queue comparison
def beam_search(start, goal, successors, heuristic, beam_width):
    queue = PriorityQueue()
    queue.put(Node(start, heuristic(start)))
    while not queue.empty():
        candidates = []
        # Get next beam_width nodes
        for _ in range(min(beam_width, queue.qsize())):
            node = queue.get()
            if node.state == goal:
                return reconstruct_path(node)
            candidates.extend(successors(node))
        # Sort candidates by heuristic value and keep the best 'beam_width' nodes
        queue = PriorityQueue()
        for candidate in sorted(candidates, key=lambda x: x.cost)[:beam_width]:
            queue.put(candidate)
    return None  # No path found
def reconstruct_path(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))
# Example Problem: Graph Representation
graph = {
    'A': [('B', 2), ('C', 3)],
    'B': [('D', 4), ('E', 1)],
    'C': [('F', 5), ('G', 2)],
    'D': [],
    'E': [('H', 3)],
    'F': [],
    'G': [('H', 1)],
    'H': []
}
def successors(node):
    """ Generate successors of a node in the graph. """
```

```
 return [Node(state, heuristic(state), node) for state, _ in graph.get(node.state, [])]
def heuristic(state):
    """ Heuristic function (assumed values for simplicity). """
    heuristic_values = {'A': 6, 'B': 4, 'C': 5, 'D': 3, 'E': 2, 'F': 4, 'G': 1, 'H': 0}
    return heuristic_values.get(state, float('inf'))
# Run Beam Search with beam width 2
start_node = 'A'
goal_node = 'H'
beam_width = 2
path = beam_search(start_node, goal_node, successors, heuristic, beam_width)
print("Path found:", path)
```

Output:

```
Path found: ['A', 'C', 'G', 'H']


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: Thus, We have Studied, implement Breadth First Search as Uniformed Search

Sign and Remark:

R1 (2 Marks)	R2 (4 Marks)	R3 (4 Marks)	Total (10 Marks)	Signature

EXPERIMENT NO. 4

Date of Performance:

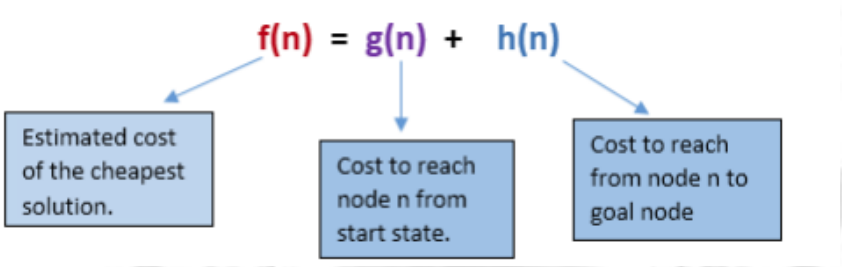
Date of Submission:

Aim: To implement A* Informed Search Algorithm.

Theory:

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

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



Algorithm 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:

1. A* search algorithm is the best algorithm than other search algorithms.
2. A* search algorithm is optimal and complete.
3. This algorithm can solve very complex problems.

Disadvantages:

1. It does not always produce the shortest path as it mostly based on heuristics and approximation.
2. A* search algorithm has some complexity issues.
3. 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.

Source Code:

```
import heapq
class Node:
    def __init__(self, name, g, h):
        self.name = name  # Node name
        self.g = g  # Cost from start node
        self.h = h  # Heuristic value
        self.f = g + h  # Evaluation function f(n) = g(n) + h(n)
        self.parent = None  # To track path
    def __lt__(self, other):
        return self.f < other.f  # Priority queue comparison
def a_star_search(graph, heuristics, start, goal):
    open_list = []  # Priority queue
    closed_set = set()  # Explored nodes
    start_node = Node(start, 0, heuristics[start])
    heapq.heappush(open_list, start_node)
    while open_list:
        current_node = heapq.heappop(open_list)  # Get node with lowest f(n)
        if current_node.name == goal:  # Goal check
            path = []
            while current_node:
                path.append(current_node.name)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path
        closed_set.add(current_node.name)
        for neighbor, cost in graph[current_node.name].items():
            if neighbor in closed_set:
                continue
            g_new = current_node.g + cost
            h_new = heuristics[neighbor]
            new_node = Node(neighbor, g_new, h_new)
            new_node.parent = current_node
```

```
            heapq.heappush(open_list, new_node)  # Push new node to open list
    return None  # No path found
# Example graph representation
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'D': 2, 'E': 5},
    'C': {'A': 4, 'F': 3},
    'D': {'B': 2, 'G': 1},
    'E': {'B': 5, 'H': 3},
    'F': {'C': 3, 'I': 4},
    'G': {'D': 1, 'J': 2},
    'H': {'E': 3, 'J': 6},
    'I': {'F': 4, 'J': 5},
   'J': {'G': 2, 'H': 6, 'I': 5}
}
# Example heuristic values (assumed straight-line distance to goal 'J')
heuristics = {
    'A': 7, 'B': 6, 'C': 5, 'D': 4, 'E': 5,
    'F': 3, 'G': 2, 'H': 4, 'I': 3, 'J': 0
}
# Running A* search
goal_path = a_star_search(graph, heuristics, 'A', 'J')
print("Optimal Path:", goal_path)
```

Output:

```
Optimal Path: ['A', 'B', 'D', 'G', 'J']


...Program finished with exit code 0
Press ENTER to exit console.
```

Sign and Remark:

R1 (2 Marks)	R2 (4 Marks)	R3 (4 Marks)	Total (10 Marks)	Signature

EXPERIMENT NO. 5

Date of Performance:

Date of Submission:

Aim: Program on Game playing algorithm.

Theory:

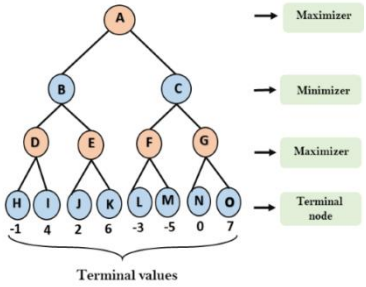
Mini-Max Algorithm in Artificial Intelligence

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Working of Min-Max Algorithm:

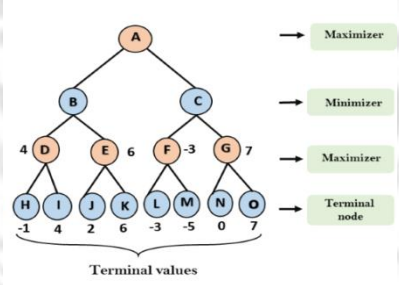
- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum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

Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



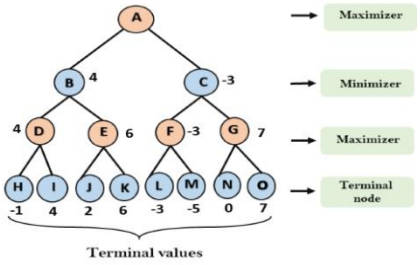
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- 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



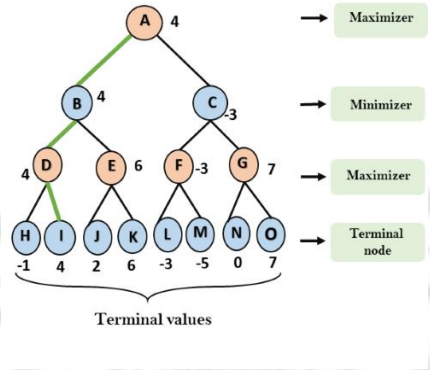
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- 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.

Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:

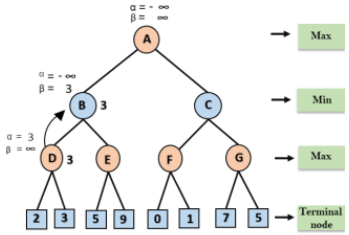
1. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
2. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

- 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.

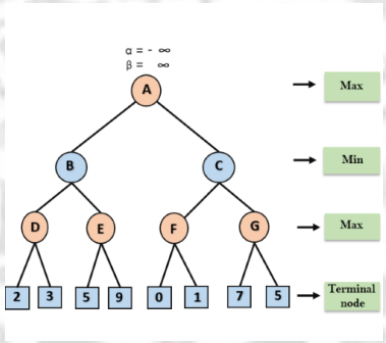
Working of Alpha-Beta Pruning:

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



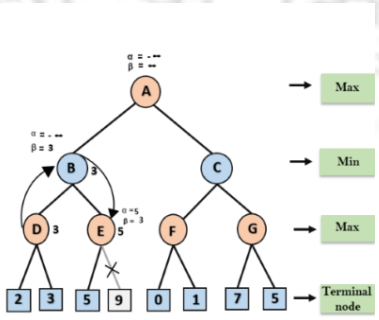
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.



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.

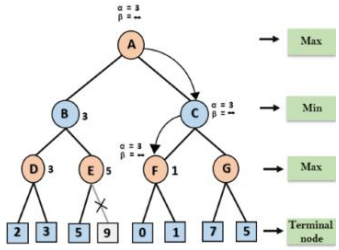
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will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



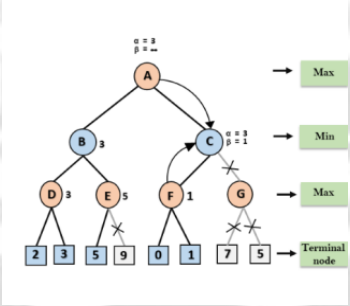
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

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

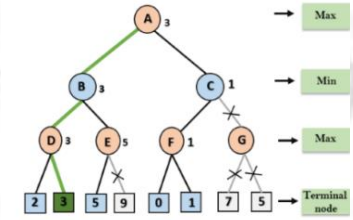
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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.



Program :

```
import math
def minimax(depth, node_index, is_max, values, alpha, beta):
    if depth == 3:  # Terminal node depth
        return values[node_index]
    if is_max:
        best = -math.inf
        for i in range(2):
```

```
val = minimax(depth + 1, node_index * 2 + i, False, values, alpha, beta)
 best = max(best, val)
          alpha = max(alpha, best)
          if beta <= alpha:
             break  # Prune remaining branches
       return best
   else:
       best = math.inf
       for i in range(2):
          val = minimax(depth + 1, node_index * 2 + i, True, values, alpha, beta)
          best = min(best, val)
          beta = min(beta, best)
          if beta <= alpha:
             break  # Prune remaining branches
       return best
# Example leaf node values
values = [-1, 4, 2, 6, -3, -5, 0, 7]
optimal_value = minimax(0, 0, True, values, -math.inf, math.inf)
print("Optimal Value:", optimal_value)
```

Output:

```
Optimal Value: 4


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: Thus, We have Studied, Program on Game playing algorithm.

Sign and Remark:

R1 (2 Marks)	R2 (4 Marks)	R3 (4 Marks)	Total (10 Marks)	Signature

सा विद्या या विमुक्तये