

**SIES College of Arts, Science and Commerce (Autonomous)**

**Sion (W), Mumbai – 400 022.**

**Department of Information Technology**

**CERTIFICATE**

**This is to certify that Mr./~~Ms.~~** **Durgam Deepesh Narendra of**   
**TY BSc [Information Technology], Semester V, Seat No. TIT2425013 has successfully completed the practical’s and submitted it online in Microsoft Teams for the subject of Artificial Intelligence as a partial fulfilment of the degree BSc (IT) during the academic year 2024-2025.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Faculty-in-charge External Examiner**

**Rajesh Yadav**

**Date: 23/09/2024**  **College Seal**

# **INDEX**

|  |  |
| --- | --- |
| **Sr No.** | **Name** |
| 1. | Write a python program to implement Breadth First Search, Depth First Search. |
| 2. | Write a python program to implement Tower of Hanoi. |
| 3. | Write a python program to implement the N queen problem where N can be anything. |
| 4. | Write a python program to implement Depth Limited Search. |
| 5. | Write a python program to implement A\* Algorithm. |
| 6. | Write a python program to implement Uniform Cost Search. |
| 7. | Write a python program to implement AO\* Algorithm. |
| 8. | Write a python program to implement Alpha Beta pruning. |
| 9. | Write a python program to implement Wumpus Problem. |
| 10. | Implementation of predicate logic using Prolog. |

**Practical 1**

a. AIM: Breadth First Search

Description: Breadth-First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. It explores all nodes at the present depth level before moving on to nodes at the next depth level, using a queue to keep track of nodes to visit. This ensures the shortest path in an unweighted graph.

1. **Initialize Structures**:
   * Create a queue and enqueue the starting node.
   * Create a set (or list) to keep track of visited nodes and mark the starting node as visited.
2. **Processing Loop**:
   * While the queue is not empty:
     + Dequeue a node from the front of the queue.
     + Process the node (e.g., print or store it).
     + For each neighbor of the dequeued node:
       - If the neighbor has not been visited:
         * Mark the neighbor as visited.
         * Enqueue the neighbor.
3. **End**:
   * Continue the process until the queue is empty.

This method ensures that nodes are visited level by level, guaranteeing that the shortest path in an unweighted graph is found if required.

CODE:

**from** collections **import** deque

**def** bfs**(**graph**,** start**):**

visited **=** **set()**

queue **=** deque**([**start**])**

visited**.**add**(**start**)**

**while** queue**:**

node **=** queue**.**popleft**()**

**print(**node**,**end**=**' '**)**

**for** neighbor **in** graph**[**node**]:**

**if** neighbor **not** **in** visited**:**

queue**.**append**(**neighbor**)**

visited**.**add**(**neighbor**)**

graph **=** **{**

'A'**:[**'B'**,**'C'**],**

'B'**:[**'D'**,**'E'**],**

'C'**:[**'F'**],**

'D'**:[],**

'E'**:[**'F'**],**

'F'**:[]**

**}**

bfs**(**graph**,**'A'**)**

**OUTPUT:**

b. AIM: Depth First Search

Description:

Depth-First Search (DFS) is an algorithm for traversing or searching tree or graph data structures. It explores as far as possible along each branch before backtracking, which means it delves deep into a node's descendants before moving to the next sibling node. DFS can be implemented using recursion or an explicit stack.

Steps for Depth-First Search

* **Initialize Structures**:
* Create a set (or list) to keep track of visited nodes.
* Use a stack (for the iterative version) or recursion (for the recursive version) to manage the nodes to be explored.
* **Start**:
* Push (or start with) the starting node onto the stack and mark it as visited.
* **Processing Loop**:
* **Iterative Version**:
  + While the stack is not empty:
    - Pop a node from the top of the stack.
    - Process the node (e.g., print or store it).
    - For each neighbor of the popped node:
      * If the neighbor has not been visited:
        + Mark it as visited.
        + Push the neighbor onto the stack.
* **Recursive Version**:
  + Process the current node (e.g., print or store it).
  + For each neighbor of the current node:
    - If the neighbor has not been visited:
      * Mark it as visited.
      * Recursively call DFS on the neighbor.
* **End**:
* Continue until all reachable nodes are visited.

DFS can be useful for tasks such as pathfinding, scheduling, and solving puzzles. It may not always find the shortest path in a weighted graph but is effective in exploring deep structures.

CODE:

**def** dfs**(**graph**,** start**,**visited**=None):**

**if** visited **is** **None:**

visited **=** **set()**

visited**.**add**(**start**)**

**print(**start**,**end**=**' '**)**

**for** neighbor **in** graph**[**start**]:**

**if** neighbor **not** **in** visited**:**

dfs**(**graph**,**neighbor**,**visited**)**

graph **=** **{**

'A'**:[**'B'**,**'C'**],**

'B'**:[**'D'**,**'E'**],**

'C'**:[**'F'**],**

'D'**:[],**

'E'**:[**'F'**],**

'F'**:[]**

**}**

dfs**(**graph**,**'A'**)**

OUTPUT: 

**Practical 2**

AIM: Write a python program to implement Tower of Hanoi.

Description:

The Tower of Hanoi is a classic puzzle involving three rods and a set of disks of different sizes. The goal is to move all disks from a source rod to a destination rod, adhering to these rules:

1. Move one disk at a time.
2. A disk can only be placed on top of a larger disk or an empty rod.

### **Steps for Solving Tower of Hanoi**

1. **Move *n−1n-1*n−1 Disks**:
   * Transfer the top *n−1n-1*n−1 disks from the source rod to an auxiliary rod, using the destination rod as a temporary holding area.
2. **Move the nth Disk**:
   * Move the largest disk from the source rod directly to the destination rod.
3. **Move *n−1n-1*n−1 Disks**:
   * Transfer the *n−1n-1*n−1 disks from the auxiliary rod to the destination rod, using the source rod as a temporary holding area.

Repeat these steps recursively until all disks are moved to the destination rod in the correct order.

CODE:

**def** moveTower**(**height**,**fromPole**,**toPole**,**withPole**):**

**if** height **>=** 1**:**

moveTower**(**height**-**1**,**fromPole**,**withPole**,**toPole**)**

moveDisk**(**fromPole**,**toPole**)**

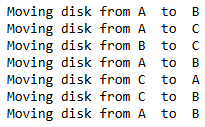
moveTower**(**height**-**1**,**withPole**,**toPole**,**fromPole**)**

**def** moveDisk**(**fp**,**tp**):**

**print(**"Moving disk from"**,**fp**,**" to "**,**tp**)**

moveTower**(**3**,**"A"**,**"B"**,**"C"**)**

OUTPUT:



**Practical 3**

**Aim:** Write a python program to implement the N queen problem where N can be anything.

**Description:**

The N-Queens problem is a classic combinatorial problem where the objective is to place *NN*N queens on an *N×NN \times N*N×N chessboard so that no two queens can attack each other. This means no two queens should share the same row, column, or diagonal.

### **Steps**

1. **Start from the First Row**:
   * Begin by placing a queen in the first row, starting from the leftmost column.
2. **Place Queens Row by Row**:
   * Move to the next row and try to place a queen in a column such that it does not conflict with the queens placed in the previous rows.
   * Check for conflicts in the same column and both diagonals.
3. **Backtrack if Necessary**:
   * If a valid position is not found in the current row, backtrack to the previous row and move the queen to the next column.
4. **Repeat Until Solved**:
   * Continue this process until all queens are placed on the board without conflicts, or until all possible configurations have been tried.
5. **Check Solutions**:
   * Once a configuration is valid, record it as a solution. If the board is filled, the solution is complete.

By using backtracking, you explore possible configurations and discard those that lead to conflicts, ultimately finding a valid arrangement of queens.

CODE:

N **=** 4

ld **=** **[**0**]** **\*** 30

rd **=** **[**0**]** **\*** 30

cl **=** **[**0**]** **\*** 30

**def** printSolution**(**board**):**

**for** i **in** **range(**N**):**

**for** j **in** **range(**N**):**

**print(**" Q " **if** board**[**i**][**j**]** **==** 1 **else** " . "**,**end**=**""**)**

**print()**

**def** solveNQUtil**(**board**,**col**):**

**if** col **>=** N**:**

**return** **True**

**for** i **in** **range(**N**):**

**if(**ld**[**i **-** col **+** N **-** 1**]** **!=** 1 **and** rd**[**i **+** col**]** **!=** 1**)** **and** cl**[**i**]** **!=** 1**:**

board**[**i**][**col**]** **=** 1

ld**[**i **-** col **+** N **-** 1**]** **=** rd**[**i **+** col**]** **=** cl**[**i**]** **=** 1

**if** solveNQUtil**(**board**,**col **+** 1**):**

**return** **True**

board**[**i**][**col**]** **=** 0 #Backtrack

ld**[**i **-** col **+** N **-** 1**]** **=** rd**[**i **+** col**]** **=** cl**[**i**]** **=** 0

**return** **False**

**def** solveNQ**():**

board **=** **[[**0 **for** \_ **in** **range(**N**)]** **for** \_ **in** **range(**N**)]**

**if** **not** solveNQUtil**(**board**,**0**):**

**print(**"Solution does not exist"**)**

**return** **False**

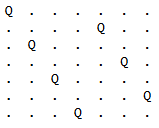
printSolution**(**board**)**

**return** **True**

**if** \_\_name\_\_ **==** "\_\_main\_\_"**:**

solveNQ**()**

OUTPUT:



**Practical 4**

**AIM:** Write a python program to implement Depth Limited Search

**DESCRIPTION:**

To explore a search space up to a specified depth limit *LL*L, avoiding the problem of infinite search spaces and potentially large memory requirements associated with unrestricted depth-first searches.

### **Steps for Depth Limited Search**

1. **Initialize the Search:**
   * **Set the Depth Limit:** Determine the maximum depth *LL*L to which the search should explore.
   * **Start Node:** Identify the initial node from which the search will begin.
2. **Push the Start Node onto a Stack:**
   * Use a stack to implement the depth-first nature of the search. Initially, the stack contains only the start node along with its current depth (0).
3. **Iterate While Stack is Not Empty:**
   * **Pop Node from Stack:** Remove the top node from the stack. This node has a certain depth.
   * **Check Depth:**
     + If the current node is at the depth limit *LL*L, skip expanding this node (or check if it is the goal if you’re searching for a specific target).
     + If the current node is not at the depth limit, proceed to the next step.
   * **Expand Node:**
     + Generate all child nodes (successors) of the current node.
     + For each child node, if it has not been visited and if its depth is less than *LL*L, push it onto the stack with its associated depth (current node depth + 1).
4. **Goal Check (if applicable):**
   * If a goal node is found during expansion, terminate the search and return the solution.
5. **Termination:**
   * If the stack becomes empty and no solution has been found, the search concludes that no solution exists within the given depth limit.

CODE:

graph**={**'A'**:[**'B'**,**'C'**],**'B'**:[**'D'**,**'E'**,**'F'**],**'C'**:[**'G'**,**'H'**],**'D'**:[**'I'**,**'J'**],**'E'**:[**'K'**,**'L'**],**'F'**:[**'M'**],**'G'**:[**'N'**,**'O'**],**'H'**:[**'P'**,**'Q'**],}**

**def** depth\_limited\_search**(**graph**,**start**,**goal**,**limit**):**

explored**=set()**

**def** recursive\_dls**(**node**,**depth**):**

**if** depth**<**0**:**

**return** **None**

**if** node**==**goal**:**

**return** **[**node**]**

explored**.**add**(**node**)**

**for** neighbor **in** graph**[**node**]:**

**if** neighbor **not** **in** explored**:**

path**=**recursive\_dls**(**neighbor**,**depth**-**1**)**

**if** path **is** **not** **None:**

**return[**node**]+**path

**return** **None**

**return** recursive\_dls**(**start**,**limit**-**1**)**

**print(**depth\_limited\_search**(**graph**,**'A'**,**'G'**,**3**))**

**print(**depth\_limited\_search**(**graph**,**'A'**,**'I'**,**4**))**

OUTPUT:



**Practical 5**

AIM: Write a python program to implement A\* Algorithm.

DESCRIPTION:

The A\* algorithm is an informed search algorithm used to find the shortest path from a start node to a goal node in a weighted graph. It combines the benefits of Dijkstra's algorithm and greedy best-first search using a cost function *f(n)f(n)*f(n), which is the sum of:

* **g(n):** The cost from the start node to node *nn*n.
* **h(n):** A heuristic estimate of the cost from node *nn*n to the goal.

The algorithm aims to efficiently find the optimal path while minimizing the search space.

### **Steps:**

1. **Initialize:**
   * **Open Set:** A priority queue or min-heap with the start node (initially with *f(start)=h(start)f(start) = h(start)*f(start)=h(start)).
   * **Closed Set:** An empty set to track nodes that have been evaluated.
   * **g(n):** The cost to reach each node, initialized to infinity except for the start node.
2. **Main Loop:**
   * **Check Open Set:** If empty, no path exists.
   * **Select Node:** Remove the node *nn*n with the lowest *f(n)f(n)*f(n) from the open set.
   * **Goal Check:** If *nn*n is the goal node, reconstruct and return the path.
   * **Expand Node:** For each neighbor *mm*m of *nn*n:
     + Calculate tentative *g(m)g(m)*g(m) as *g(n)+cost(n,m)g(n) + \text{cost}(n, m)*g(n)+cost(n,m).
     + If *mm*m is not in the closed set or the new *g(m)g(m)*g(m) is better:
       - Update *g(m)g(m)*g(m), compute *f(m)=g(m)+h(m)f(m) = g(m) + h(m)*f(m)=g(m)+h(m).
       - Add or update *mm*m in the open set.
   * **Move to Closed Set:** Add *nn*n to the closed set.
3. **Terminate:**
   * Return the path if the goal is reached.
   * If the open set is exhausted and the goal is not reached, return failure.

CODE:

graph**={**'A'**:{**'B'**:**6**,**'F'**:**3**},**

'B'**:{**'C'**:**3**,**'D'**:**2**},**

'C'**:{**'E'**:**5**,**'D'**:**1**},**

'D'**:{**'E'**:**8**},**

'E'**:{**'J'**:**5**},**

'F'**:{**'G'**:**1**,**'H'**:**7**},**

'G'**:{**'I'**:**3**},**

'H'**:{**'I'**:**2**},**

'I'**:{**'E'**:**5**,**'J'**:**3**},**

'J'**:{}}**

state**={**'A'**:**10**,**

'B'**:**8**,**

'C'**:**5**,**

'D'**:**7**,**

'E'**:**3**,**

'F'**:**6**,**

'G'**:**5**,**

'H'**:**3**,**

'I'**:**1**,**

'J'**:**0**}**

**def** A\_star**(**graph**,** start\_node**,**end\_node**,**state**):**

m**={}**

value**=**0

**if** start\_node**==**end\_node**:**

**return**

**for** node**,**v **in** graph**[**start\_node**].**items**():**

value**=**state**[**node**]+**v

m**[**node**]=**value

**if(**m **!=** **{}):**

path**[min(**m**,**key**=**m**.**get**)]=**value

A\_star**(**graph**,min(**m**,**key**=**m**.**get**),**end\_node**,**state**)**

**return** path

path**={}**

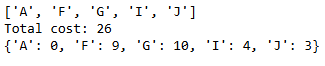
path**[**'A'**]=**0

bp**=(**A\_star**(**graph**,**'A'**,**'J'**,**state**))**

**print(list(**bp**))**

**print(**"Total cost: {}"**.format(sum(**bp**.**values**())))**

**print(**bp**)**

OUTPUT:

**Practical 6**

AIM: Write a python program to implement Uniform Cost Search.

DESCRIPTION:

Uniform Cost Search (UCS) is an algorithm used to find the least-cost path from a start node to a goal node in a weighted graph. UCS is a variant of Dijkstra's algorithm that explores nodes based on the path cost from the start node, ensuring that the path with the lowest cumulative cost is found.

### **Steps:**

1. **Initialize:**
   * **Priority Queue (Open Set):** Start with the start node in a priority queue (or min-heap) with a priority equal to the cost to reach it (initially 0).
   * **Cost Table:** Maintain a cost table to track the minimum cost to reach each node. Initialize the start node's cost to 0 and others to infinity.

**2. Main Loop:**

* + **Check Priority Queue:** If the queue is empty, no path exists.
  + **Select Node:** Remove the node *nn*n with the lowest cost from the priority queue.
  + **Goal Check:** If *nn*n is the goal node, reconstruct and return the path.
  + **Expand Node:** For each neighbor *mm*m of *nn*n:
    - Calculate the cost to reach *mm*m as *cost(n)+cost(n,m)\text{cost}(n) + \text{cost}(n, m)*cost(n)+cost(n,m).
    - If this new cost is lower than the previously recorded cost for *mm*m:
      * Update the cost for *mm*m.
      * Update the priority queue with the new cost for *mm*m.

**3. Terminate:**

* + Return the path if the goal is reached.
  + If the priority queue is exhausted and the goal is not reached, return failure.

CODE:

**import** heapq

**def** uniform\_cost\_search**(**graph**,**start**,**goal**):**

visited**=set()**

queue **=** **[(**0**,**start**,[])]**

**while** queue**:**

**(**cost**,**node**,**path**)=**heapq**.**heappop**(**queue**)**

**if** node **not** **in** visited**:**

visited**.**add**(**node**)**

path**=**path**+[**node**]**

**if** node **==** goal**:**

**return** **(**cost**,**path**)**

**for** **(**next\_node**,**c**)** **in** graph**[**node**]:**

**if** next\_node **not** **in** visited**:**

heapq**.**heappush**(**queue**,(**cost **+** c**,**next\_node**,**path**))**

**return** **float(**"inf"**)**

graph**=** **{**

'A'**:[(**'B'**,**1**),(**'C'**,**3**),(**'D'**,**7**)],**

'B'**:[(**'D'**,**5**)],**

'C'**:[(**'D'**,**12**)]**

**}**

cost**,**path **=** uniform\_cost\_search**(**graph**,**'A'**,**'D'**)**

**print(**f"Path: {path},Cost: {cost}"**)**

OUTPUT:



**Practical 7**

**Aim:** Python Program to implement AO\* Algorithm

**Description:** AO\* (which stands for "And-Or A\*") is an extension of the A\* search algorithm designed to handle problems where the search space includes both **"AND"** and **"OR"** relationships. It's used in situations where tasks or goals can be decomposed into sub-tasks that are interconnected in complex ways.

**How AO\* Works:**

1. **Graph Representation**: AO\* operates on a **AND-OR graph**, where:
   * **OR nodes** represent choices or alternatives (e.g., different actions or strategies).
   * **AND nodes** represent tasks that need all of their sub-tasks completed.
2. **Search Process**:
   * **Initial State**: Start at the root node (the initial goal or problem).
   * **Expand Nodes**: Expand nodes according to the AND-OR structure. For OR nodes, explore all possible options. For AND nodes, ensure that all required sub-tasks are resolved.
   * **Cost Calculation**: Calculate the cost of reaching the goal through different paths. The cost function in AO\* is similar to that in A\*, where it considers both the cost from the start node to the current node and the estimated cost from the current node to the goal.
3. **Heuristic Function**: Like A\*, AO\* uses a heuristic function to estimate the cost from a node to the goal. This helps in prioritizing which nodes to explore first.
4. **Solution Construction**: Once a goal is reached, AO\* constructs a solution path that combines the decisions made at OR nodes and the completion of all required tasks at AND nodes.

**Key Features of AO\*:**

* **Handling Complex Problems**: It’s particularly useful for problems where the solution requires managing both choice and dependency, such as planning and decision-making problems in AI.
* **Optimality**: AO\* aims to find an optimal solution, similar to A\*, by using heuristics to guide the search efficiently.

**Code:**

**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**]**

**print(**key**,**':'**,**Conditions**[**key**],**'>>>'**,**Cost**(**H**,**condition**,**weight**))**

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

**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**]**

Path**+=**'<--'**+**shortest\_path**(**Start**,**Updated\_cost**,**H**)**

**else:**

Path**+=**'<--('**+**key**[**Index**]+**')'

Start**=**Next**[**0**]**

Path**+=**'['**+**shortest\_path**(**Start**,**Updated\_cost**,**H**)+**'+'

Start**=**Next**[-**1**]**

Path**+=**shortest\_path**(**Start**,**Updated\_cost**,**H**)+**']'

**return** Path

H**={**'A'**:-**1**,**'B'**:**5**,**'C'**:**2**,**'D'**:**4**,**'E'**:**7**,**'F'**:**9**,**'G'**:**3**,**'H'**:**0**,**'I'**:**0**,**'J'**:**0**}**

Conditions**={**'A'**:{**'OR'**:[**'B'**],**'AND'**:[**'C'**,**'D'**]},**

'B'**:{**'OR'**:[**'E'**,**'F'**]},**

'C'**:{**'OR'**:[**'G'**],**'AND'**:[**'H'**,**'I'**]},**

'D'**:{**'OR'**:[**'J'**]}}**

weight**=**1

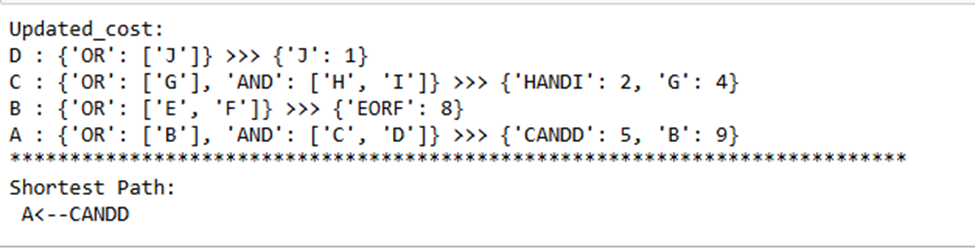
**print(**'Updated\_cost:'**)**

Updated\_cost**=**update\_cost**(**H**,**Conditions**,**weight**=**1**)**

**print(**'\*'**\***75**)**

**print(**'Shortest Path:\n'**,**shortest\_path**(**'A'**,**Updated\_cost**,**H**))**

**Output:**



**Practical 8**

**AIM:** Write a program to implement Alpha Beta pruning

**DESCRIPTION:**

The Alpha-Beta Pruning algorithm is a technique used in artificial intelligence, particularly in game theory and decision-making, to enhance the efficiency of the minimax algorithm. It reduces the number of nodes evaluated in the game tree by pruning branches that cannot influence the final decision. Here’s an explanation of how it works:

### **Minimax Algorithm Overview**

The Minimax algorithm is used in two-player games to find the optimal move for a player assuming that the opponent is also playing optimally. The algorithm explores the entire game tree, evaluating the possible outcomes of each move. However, this can be computationally expensive for complex games.

### **Alpha-Beta Pruning**

Alpha-Beta Pruning improves the efficiency of the Minimax algorithm by eliminating branches that won't affect the final decision. Here’s how it works:

#### **1. Initialization**

* **Alpha (α):** The best value that the maximizing player can guarantee so far. Initially set to negative infinity.
* **Beta (β):** The best value that the minimizing player can guarantee so far. Initially set to positive infinity.

#### **2. Search Tree Traversal**

The game tree is traversed recursively. At each node, the algorithm updates alpha and beta values and prunes branches when possible.

* **Max Node (Maximizing Player's Turn):**
  + **Evaluate Children:** For each child node, the algorithm recursively explores the child nodes and updates the alpha value.
  + **Prune:** If the value of the current child node is greater than or equal to beta, then the rest of the sibling nodes can be pruned (since the minimizing player will not choose this path).
* **Min Node (Minimizing Player's Turn):**
  + **Evaluate Children:** For each child node, the algorithm recursively explores the child nodes and updates the beta value.
  + **Prune:** If the value of the current child node is less than or equal to alpha, then the rest of the sibling nodes can be pruned (since the maximizing player will not choose this path).

#### **3. Update Values**

* For Max nodes, update the alpha value to the maximum of the current alpha and the value obtained from the child nodes.
* For Min nodes, update the beta value to the minimum of the current beta and the value obtained from the child nodes.

#### **4. Terminate Search**

The search terminates when the entire tree is evaluated or when all possible branches that can affect the outcome are pruned.

### **Example Walkthrough**

Let’s illustrate the process with a simplified example:

1. **Starting Node:** Max Node with α = -∞, β = ∞.
2. **Evaluate First Child:** Suppose this child is a Min Node.
   * **Min Node:** Evaluate its children. Update β.
   * **If β ≤ α (Alpha value from the Max Node), prune the remaining siblings of this Min Node.**
3. **Continue to the Next Sibling of the Initial Max Node:**
   * Evaluate if it needs to prune based on the updated α and β values.

By applying Alpha-Beta Pruning, the algorithm can skip over branches of the tree that don't need to be explored, thus improving computational efficiency significantly.

### **Summary**

Alpha-Beta Pruning optimizes the Minimax algorithm by reducing the number of nodes evaluated in the game tree. It does this by maintaining and updating two values (alpha and beta) that help in deciding which branches to prune, based on the potential outcomes.

**CODE:**

MAX**,**MIN**=**1000**,-**1000

**def** minimax**(**depth**,**nodeIndex**,**maximizingPlayer**,**values**,**alpha**,**beta**):**

**if** depth**==**3**:**

**return** values**[**nodeIndex**]**

**if** maximizingPlayer**:**

best**=**MIN

**for** i **in** **range(**0**,**2**):**

val**=**minimax**(**depth**+**1**,**nodeIndex**\***2**+**i**,False,**values**,**alpha**,**beta**)**

best**=max(**best**,**val**)**

alpha**=max(**alpha**,**best**)**

**if** beta**<=**alpha**:**

**break**

**return** best

**else:**

best**=**MAX

**for** i **in** **range(**0**,**2**):**

val**=**minimax**(**depth**+**1**,**nodeIndex**\***2**+**i**,True,**values**,**alpha**,**beta**)**

best**=min(**best**,**val**)**

beta**=min(**alpha**,**best**)**

**if** beta**<=**alpha**:**

**break**

**return** best

#Driver code

**if** \_\_name\_\_**==**"\_\_main\_\_"**:**

values **=** **[**3**,**5**,**6**,**9**,**1**,**2**,**0**,-**1**]**

**print(**"The optimal value is :"**,**minimax**(**0**,** 0**,** **True,** values**,**MIN**,** MAX**))**

**OUTPUT:** 

**Practical 9**

**AIM:** Wumpus problem

**Description:** The Wumpus World is a classic AI problem involving a grid where an agent must avoid hazards (pits and a Wumpus), find gold, and return to the starting point. The key components are:

* **Grid:** Typically 4x4.
* **Wumpus:** Dangerous creature.
* **Pits:** Bottomless holes.
* **Gold:** Goal to be found.
* **Agent:** Starts at the bottom-left corner.

**Algorithm Overview:**

1. **Representation:** Use logical statements to represent the environment and sensory information (e.g., stench for Wumpus, breeze for pits).
2. **Inference:** Apply logic to deduce where the Wumpus, pits, and gold are based on sensory data.
3. **Planning:** Use search algorithms (like DFS, BFS, or A\*) to plan the path to the gold and back to the start.
4. **Decision Making:** Choose actions based on current knowledge and planned path.
5. **Execution:** Move according to the plan, avoiding dangers, and adapt as new information is received.

CODE:

*wumpus***=[[**"Save"**,**"Breeze"**,**"PIT"**,**"Breeze"**],**

**[**"Smell"**,**"Save"**,**"Breeze"**,**"Save"**],**

**[**"WUMPUS"**,**"GOLD"**,**"PIT"**,**"Breeze"**],**

**[**"Smell"**,**"Save"**,**"Breeze"**,**"PIT"**]]**

row **=** 0

column **=** 0

arrow**=True**

player **=** **True**

score**=**0

**while(**player**):**

choice**=input(**"Press u to move ip\n Press d to move down \nPress l to move left\nPress r to move right\n"**)**

**if** choice**==**"u"**:**

**if** row**!=**0**:**

row**-=**1

**else:**

**print(**"move denied"**)**

**print(**"current location:"**,**wumpus**[**row**][**column**],**"\n"**)**

**elif** choice**==**"d"**:**

**if** row**!=**3**:**

row**+=**1

**else:**

**print(**"move denied"**)**

**print(**"current location:"**,**wumpus**[**row**][**column**],**"\n"**)**

**elif** choice **==** "l"**:**

**if** column **!=** 0**:**

column **-=** 1

**else:**

**print(**"move denied"**)**

**print(**"current location: "**,** wumpus**[**row**][**column**],** "\n"**)**

**elif** choice **==** "r"**:**

**if** column **!=** 3**:**

column **+=** 1

**else:**

**print(**"move denied"**)**

**print(**"current location: "**,** wumpus**[**row**][**column**],** "\n"**)**

**else:**

**print(**"Move denied"**)**

**if** wumpus**[**row**][**column**]==**"Smell" **and** arrow **!=False:**

arrow\_choice **=input(**"Do you want to throw an arrow --> \nPress y to throw \nPress n to save your arrow\n"**)**

**if** arrow\_choice **==**"y"**:**

arrow\_throw**=input(**"Press u to throw up\nPress d to throw down\nPress l to throw left \nPress r to thorw right\n"**)**

**if** arrow\_throw**==**"u"**:**

**if** wumpus**[**row**-**1**][**column**]==**"WUMPUS"**:**

**print(**"wumpus killed"**)**

score**+=**1000

**print(**"Score : "**,**score**)**

wumpus**[**row**-**1**][**column**]=**"Save"

wumpus**[**1**][**0**]=**"save"

wumpus**[**3**][**0**]=**"Save"

**else:**

**print(**"Arrow Wasted"**)**

score **-=**10

**print(**"Score : "**,**score**)**

**elif** arrow\_throw**==**"d"**:**

**if** wumpus**[**row**+**1**][**column**]==**"WUMPUS"**:**

**print(**"wumpus killed"**)**

score**+=**1000

**print(**"Score : "**,**score**)**

wumpus**[**row**+**1**][**column**]=**"Save"

wumpus**[**1**][**0**]=**"save"

wumpus**[**3**][**0**]=**"Save"

**else:**

**print(**"Arrow Wasted"**)**

score **-=**10

**print(**"Score : "**,**score**)**

**elif** arrow\_throw**==**"l"**:**

**if** wumpus**[**row**][**column**-**1**]==**"WUMPUS"**:**

**print(**"wumpus killed"**)**

score**+=**1000

**print(**"Score : "**,**score**)**

wumpus**[**row**][**column**-**1**]=**"Save"

wumpus**[**1**][**0**]=**"save"

wumpus**[**3**][**0**]=**"Save"

**else:**

**print(**"Arrow Wasted"**)**

score **-=**10

**print(**"Score : "**,**score**)**

**elif** arrow\_throw**==**"r"**:**

**if** wumpus**[**row**][**column**+**1**]==**"WUMPUS"**:**

**print(**"wumpus killed"**)**

score**+=**1000

**print(**"Score : "**,**score**)**

wumpus**[**row**][**column**+**1**]=**"Save"

wumpus**[**1**][**0**]=**"save"

wumpus**[**3**][**0**]=**"Save"

**else:**

**print(**"Arrow Wasted"**)**

score **-=**10

**print(**"Score : "**,**score**)**

arrow**=False**

**if** wumpus**[**row**][**column**]==**"WUMPUS"**:**

score**-=**1000

**print(**"\nWumpus here!!\nYou Die\nAnd your score is : "**,**score**,**"\n"**)**

**break**

**if** wumpus**[**row**][**column**]==**"GOLD"**:**

score**+=**1000

**print(**"GOLD FOUND! You Won....\nYour Score is : "**,**score**,**"\n"**)**

**break**

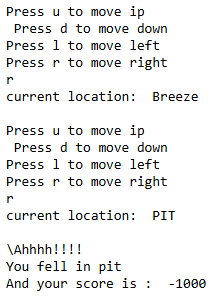
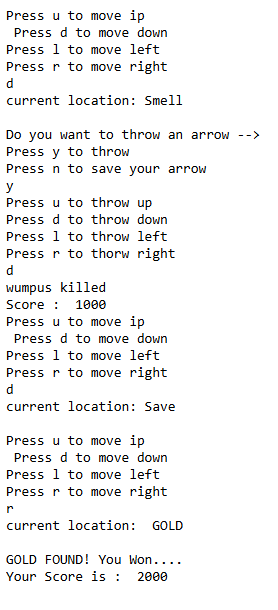
**if** wumpus**[**row**][**column**]==**"PIT"**:**

score**-=**1000

**print(**"\nAhhhh!!!!\nYou fell in pit \nAnd your score is : "**,**score**,**"\n"**)**

**break**

OUTPUT:



**Practical 10**

**Aim:** Implementation of predicate logic using Prolog.

**Description:** Prolog implements predicate logic by using a simple yet powerful structure. It relies on:

- Facts: Basic assertions about relationships or properties. For example, `parent(john, mary).` indicates that John is Mary's parent.

- Rules: Define relationships between facts, allowing Prolog to infer new information. Rules are written with a head (conclusion) and a body (conditions). For example, `grandparent(X, Y) :- parent(X, Z), parent(Z, Y).` means X is a grandparent of Y if X is a parent of Z and Z is a parent of Y.

- Queries:Requests for information based on the knowledge base. For instance, querying `?- grandparent(john, Who).` asks Prolog to find all individuals who are grandchildren of John.

In Prolog, variables are placeholders that can be unified with specific values to satisfy conditions. The language uses unification to match patterns and derive results. Through this approach, Prolog allows for dynamic problem-solving and logical reasoning based on the defined facts and rules.

1)

**Code:**

likes(Alice,tea).

likes(Alice,coffee).

likes(bob,tea).

likes(bob,coffee).

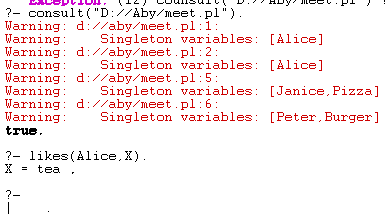
likes(Janice,Pizza).

likes(Peter,Burger).

likes(bob,X):-likes(X,pizzaa).

likes(bob,Y):-likes(peter,Y).

**Output:**



**2)**

**Code:**

female(pammi).

female(lizza).

female(patty).

female(anny).

male(jimmy).

male(bobby).

male(tomy).

male(pitter).

parent(pammi,bobby).

parent(tomy,bobby).

parent(tomy,lizza).

parent(bobby,anny).

parent(bobby,patty).

parent(patty,jimmy).

parent(bobby,pitter).

parent(pitter,jimmy).

mother(X,Y) :- parent(X,Y),female(X).

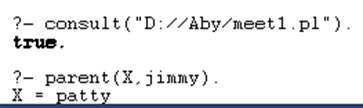
father(X,Y):- parent(X,Y),male(X).

haschild(X):- parent(X.\_).

sister(X,Y):- parent(Z,X).parent(Z,Y),female(X),X\==Y.

brother(X,Y):-parent(Z,X).parent(Z.Y),male(X).X\==Y.

**Output:**



**3)**

**Code:**

Add(X, Y, Result) :- Result is X + Y.

subtract(X, Y, Result) :- Result is X - Y.

multiply(X, Y, Result) :- Result is X \* Y.

divide(X, Y, Result) :- Result is X / Y.

modulus(X, Y, Result) :- Result is X mod Y.

**Output:**

