



L. D. College of Engineering

Opp. Gujarat University, Navrangpura, Ahmedabad - 380015

LAB MANUAL

Branch: Computer Engineering

**Artificial Intelligence
(3170716)**

Semester: VII

Faculty Details:

- 1. Prof. Dr. V. B. Vaghela**
- 2. Prof. H. D. Rajput**

CERTIFICATE

This is to certify that Mr./Ms. _____,

Enrollment Number _____ has satisfactorily completed the practical work
in “Artificial Intelligence” subject at L D College of Engineering, Ahmedabad-380015.

Date of Submission: _____

Sign of Faculty: _____

Head of Department: _____

L. D. College of Engineering, Ahmedabad
Department of Computer Engineering
Practical List

Subject Name: Artificial Intelligence (3170716)

Term: 2024-2025

Sr. No.	Title	Date	Page No.	CO	Marks (10)	Sign
PART-A AI Programs						
1	Write a program to implement Tic-Tac-Toe game problem.			CO3		
2	Write a program to implement BFS (for 8 puzzle problem or Water Jug problem or any AI search problem).			CO1		
3	Write a program to implement DFS (for 8 puzzle problem or Water Jug problem or any AI search problem).			CO1		
4	Write a program to implement Single Player Game (Using any Heuristic Function).			CO3		
5	Write a program to Implement A* Algorithm.			CO4		
6	Write a program to implement mini-max algorithm for any game development.			CO4		
PART – B Prolog Programs						
1	Write a PROLOG program to represent Facts. Make some queries based on the facts.			CO2		
2	Write a PROLOG program to represent Rules. Make some queries based on the facts and rules			CO2		
3	Write a PROLOG program to define the relations using the given predicates.			CO5		
4	Write a PROLOG program to perform addition, subtraction, multiplication and division of two numbers using arithmetic operators.			CO5		
5	Write a PROLOG program to display the numbers from 1 to 10 by simulating the loop using recursion.			CO5		
6	Write a PROLOG program to count number of elements in a list.			CO5		
7	Write a PROLOG program to perform various operations on a list.			CO5		
8	Write a PROLOG program to reverse the list.			CO5		
9	Write a PROLOG program to demonstrate the use of CUT and FAIL predicate.			CO5		
10	Write a PROLOG program to solve the Tower of Hanoi problem.			CO5		
11	Write a PROLOG program to solve the N-Queens problem.			CO5		
12	Write a PROLOG program to solve the Travelling Salesman problem			CO5		

L. D. College of Engineering, Ahmedabad
Department of Computer Engineering

Practical Rubrics

Subject Name: Artificial Intelligence					Subject Code: (3170716)	
Term: 2024-2025						
Rubrics ID	Criteria	Marks	Excellent (3)	Good (2)	Satisfactory (1)	Need Improvement (0)
RB1	Regularity	02	--	High (>70%)	Moderate (40-70%)	Poor (0-40%)
RB2	Problem Analysis and Development of Solution	03	Appropriate & Full Identification of the Problem & Complete Solution for the Problem	Limited Identification of the Problem / Incomplete Solution for the Problem	Very Less Identification of the Problem / Very Less Solution for the Problem	Not able to analyze the problem and develop the solution
RB3	Concept Clarity and Understanding	03	Concept is very clear with proper understanding	Concept is clear at moderate level.	Just overview of the concept is known.	Concept is not clear.
RB4	Documentation	02	--	Documentation completed neatly.	Not up to standard.	Proper format not followed, incomplete.

SIGN OF FACULTY

L. D. College of Engineering, Ahmedabad
Department of Computer Engineering
LABORATORY PRACTICALS ASSESSMENT

Subject Name: Artificial Intelligence (3170716)

Term: ODD 2024-25

Enroll. No.:

Name:

Pract. No.	RB1 (2)	RB2 (3)	RB3 (3)	RB4 (2)	Total (10)	Date	Faculty Sign
PART – A AI Programs							
1							
2							
3							
4							
5							
6							
PART – B Prolog Programs							
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							

Practical – 1

Aim: Write a program to implement Tic-Tac-Toe game problem.

- **Objective:** The objective is to implement a two-player Tic-Tac-Toe game, where players take turns marking a 3x3 grid, and the game checks for a winner or a draw after each move.

- **Theory:**

Rules of the Game:

- The game is to be played between two people (in this program between HUMAN and COMPUTER).
- One of the player chooses 'O' and the other 'X' to mark their respective cells.
- The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character ('O' or 'X').
- If no one wins, then the game is said to be draw.

O	X	O
O	X	X
X	O	X

Implementation In our program the moves taken by the computer and the human are chosen randomly. We use rand() function for this. What more can be done in the program? The program is not played optimally by both sides because the moves are chosen randomly. The program can be easily modified so that both players play optimally (which will fall under the category of Artificial Intelligence). Also the program can be modified such that the user himself gives the input (using scanf() or cin). The above changes are left as an exercise to the readers. **Winning Strategy – An Interesting Fact** If both the players play optimally then it is destined that you will never lose (“although the match can still be drawn”). It doesn’t matter whether you play first or second. In another ways – “Two expert players will always draw”.

- **Tools / Material Needed:**

- **Hardware:** Computer/Laptop
- **Software:** VS code or any IDE, Python

- **Implementation:**

```
#include <iostream>
using namespace std;

// Function to draw the game board
void drawBoard(char board[3][3])
{
    cout << " " << board[0][0] << " | " << board[0][1] << " | " << board[0][2] << endl;
    cout << "---+---+---" << endl;
    cout << " " << board[1][0] << " | " << board[1][1] << " | " << board[1][2] << endl;
    cout << "---+---+---" << endl;
    cout << " " << board[2][0] << " | " << board[2][1] << " | " << board[2][2] << endl;
}

// Function to check for a win
bool checkWin(char board[3][3], char player)
{
    // Check rows and columns
    for (int i = 0; i < 3; i++)
    {
        if (board[i][0] == player && board[i][1] == player && board[i][2] == player)
        {
            return true;
        }
        if (board[0][i] == player && board[1][i] == player && board[2][i] == player)
        {
            return true;
        }
    }
    // Check diagonals
    if ((board[0][0] == player && board[1][1] == player && board[2][2] == player) ||
        (board[0][2] == player && board[1][1] == player && board[2][0] == player))
    {
        return true;
    }
    return false;
}

// Function to check for a draw
bool checkDraw(char board[3][3])
{
    for (int i = 0; i < 3; i++)
    {
```

```
for (int j = 0; j < 3; j++)
{
    if (board[i][j] == ' ')
    {
        return false;
    }
}
}
return true;
}

int main()
{
    char board[3][3] = {{ ' ', ' ', ' ' }, { ' ', ' ', ' ' }, { ' ', ' ', ' ' }};
    char player = 'X';
    int row, col;

    while (true)
    {
        drawBoard(board);
        cout << "Player " << player << ", enter row (1-3) and column (1-3): ";
        cin >> row >> col;

        // Validate input
        if (row < 1 || row > 3 || col < 1 || col > 3)
        {
            cout << "Invalid input. Please try again." << endl;
            continue;
        }

        // Check if space is already occupied
        if (board[row - 1][col - 1] != ' ')
        {
            cout << "Space already occupied. Please try again." << endl;
            continue;
        }

        // Update board
        board[row - 1][col - 1] = player;

        // Check for win or draw
        if (checkWin(board, player))
        {
            drawBoard(board);
            cout << "Player " << player << " wins!" << endl;
            break;
        }
    }
}
```



```

    }
    else if (checkDraw(board))
    {
        drawBoard(board);
        cout << "It's a draw!" << endl;
        break;
    }

    // Switch player
    player = (player == 'X') ? 'O' : 'X';
}

return 0;
}

```

- **Output:**

```

PS D:\AI Practical> cd "d:\AI Practical\" ; if ($?) { g++ pra1.cpp -o pra1 } ; if ($?) { .\pra1 }
  |  |
  |  |
  |  |
  |  |
Player X, enter row (1-3) and column (1-3): 1 1
X |  |
  |  |
  |  |
  |  |
Player O, enter row (1-3) and column (1-3): 1 3
X |  | O
  |  |
  |  |
  |  |
Player X, enter row (1-3) and column (1-3): 2 2
X |  | O
  |  |
  | X |
  |  |
  |  |
Player O, enter row (1-3) and column (1-3): 2 3
X |  | O
  |  |
  | X | O
  |  |
  |  |
Player X, enter row (1-3) and column (1-3): 3 3
X |  | O
  | X | O
  |  |
  |  | X
Player X wins!

```

Signature of Faculty:

Grade:

Practical – 2

Aim: Write a program to implement BFS (for 8 puzzle problem or Water Jug problem or any AI search problem).

- **Objective:** To implement the Breadth-First Search (BFS) algorithm to solve an AI search problem, such as the Water Jug problem, where the goal is to measure a specific amount of water using two jugs of different capacities.

- **Theory:**

The Water Jug Problem is a classic puzzle in artificial intelligence (AI) that involves using two jugs with different capacities to measure a specific amount of water. It is a popular problem to teach problem-solving techniques in AI, particularly when introducing search algorithms. The Water Jug Problem highlights the application of AI to real-world puzzles by breaking down a complex problem into a series of states and transitions that a machine can solve using an intelligent algorithm.

Problem Description: Water Jug Problem

The Water Jug Problem typically involves two jugs with different capacities. The objective is to measure a specific quantity of water by performing operations like filling a jug, emptying a jug, or transferring water between the two jugs. The problem can be stated as follows:

- You are given two jugs, one with a capacity of **X** liters and the other with a capacity of **Y** liters.
- You need to measure exactly **Z** liters of water using these two jugs.
- The allowed operations are:
 - Fill one of the jugs.
 - Empty one of the jugs.
 - Pour water from one jug into another until one jug is either full or empty.

Example:

Input: $m = 3, n = 5, d = 4$

Output: 6

Explanation: Operations are as follow:

Initially, both jugs are empty ($\text{jug1} = 0, \text{jug2} = 0$).

Step 1: Fill the 5 liter jug $\rightarrow (0, 5)$.

Step 2: Pour from the 5 liter jug to the 3 liter jug $\rightarrow (3, 2)$.

Step 3: Empty the 3 liter jug $\rightarrow (0, 2)$.

Step 4: Pour the 2 liters from the 5-liter jug to the 3 liter jug -> (2, 0).

Step 5: Fill the 5 liter jug again -> (2, 5).

Step 6: Pour 1 liter from the 5 liter jug into the 3 liter jug -> (3, 4).

Now, the 5 liter jug contains exactly 4 liters, so we stop and return 6 steps.

Approach:

To solve this problem, we can think like it as a state exploration problem where each state represents the amount of water in both jugs at a particular point in time. From any given state, a set of possible operations can be performed to move to a new state and this continues until we either reach the desired amount d in one of the jugs or determining that it's not possible.

For this problem, the state is represented as a pair (jug1, jug2), where jug1 is the amount of water in the first jug and jug2 is the amount in the second jug. The initial state is (0, 0) because both jugs start empty. Since we're looking for the minimum number of operations, Breadth-First Search (BFS) is a good choice as it explores all possible states level by level and this make sure that the first time we find the solution, it's with the minimum number of steps.

There are six possible operations that can be applied at any given state:

1. Fill Jug 1: Fill the first jug to its maximum capacity (m liters).
2. Fill Jug 2: Fill the second jug to its maximum capacity (n liters).
3. Empty Jug 1: Completely empty the first jug.
4. Empty Jug 2: Completely empty the second jug.
5. Pour Jug 1 into Jug 2: Pour as much water as possible from the first jug into the second jug until the second jug is full or the first jug is empty.
6. Pour Jug 2 into Jug 1: Pour as much water as possible from the second jug into the first jug until the first jug is full or the second jug is empty.

- **Tools / Material Needed:**

- **Hardware:** Computer/Laptop
- **Software:** VS code or any IDE, Python

- **Implementation:**

```
#include <iostream>
#include <queue>
#include <set>
#include <vector>
#include <map>
#include <algorithm>
```

```
using namespace std;

// Function to check if the current state is the target state
bool is_target_state(pair<int, int> state, int target)
{
    return state.first == target || state.second == target;
}

// Function to generate all possible next states from the current state
vector<pair<int, int>> get_next_states(int jug1, int jug2, pair<int, int> curr_state)
{
    vector<pair<int, int>> next_states;
    int x = curr_state.first, y = curr_state.second;

    // Fill jug1
    next_states.push_back({jug1, y});
    // Fill jug2
    next_states.push_back({x, jug2});
    // Empty jug1
    next_states.push_back({0, y});
    // Empty jug2
    next_states.push_back({x, 0});

    // Pour from jug1 to jug2
    int pour_to_jug2 = min(x, jug2 - y);
    next_states.push_back({x - pour_to_jug2, y + pour_to_jug2});

    // Pour from jug2 to jug1
    int pour_to_jug1 = min(y, jug1 - x);
    next_states.push_back({x + pour_to_jug1, y - pour_to_jug1});

    return next_states;
}

// BFS to solve the Water Jug problem
vector<pair<int, int>> bfs_water_jug(int jug1, int jug2, int target)
{
    // Initial state (0, 0)
    pair<int, int> initial_state = {0, 0};

    // Queue for BFS
    queue<pair<int, int>> queue;
    queue.push(initial_state);

    // Set to track visited states
```

```
set<pair<int, int>> visited;
visited.insert(initial_state);

// Map to store the path
map<pair<int, int>, pair<int, int>> parent_map;
parent_map[initial_state] = {-1, -1}; // Special value to indicate the root node

while (!queue.empty())
{
    pair<int, int> current_state = queue.front();
    queue.pop();

    // Check if the current state is the target
    if (is_target_state(current_state, target))
    {
        // Backtrack to find the path
        vector<pair<int, int>> path;
        while (current_state != make_pair(-1, -1))
        {
            path.push_back(current_state);
            current_state = parent_map[current_state];
        }
        reverse(path.begin(), path.end());
        return path;
    }

    // Generate next states
    vector<pair<int, int>> next_states = get_next_states(jug1, jug2, current_state);
    for (auto next_state : next_states)
    {
        if (visited.find(next_state) == visited.end())
        {
            visited.insert(next_state);
            queue.push(next_state);
            parent_map[next_state] = current_state;
        }
    }
}

return {}; // No solution found
}

int main()
{
    // Jug capacities
    int jug1_capacity = 4;
```

```

int jug2_capacity = 3;
// Target amount of water
int target_amount = 2;

// Perform BFS to find solution
vector<pair<int, int>> solution_path = bfs_water_jug(jug1_capacity, jug2_capacity,
target_amount);

if (!solution_path.empty())
{
    cout << "Solution path to achieve the target:" << endl;
    for (size_t step = 0; step < solution_path.size(); ++step)
    {
        cout << "Step " << step << ": Jug1 = " << solution_path[step].first
            << ", Jug2 = " << solution_path[step].second << endl;
    }
}
else
{
    cout << "No solution found." << endl;
}

return 0;
}

```

• Output:

```

PS D:\AI Practical> cd "d:\AI Practical"
PS D:\AI Practical> cd "d:\AI Practical\" ; if ($?) { g++ p2.c++ -o p2 } ; if ($?) { .\p2 }
Solution path to achieve the target:
Step 0: Jug1 = 0, Jug2 = 0
Step 1: Jug1 = 0, Jug2 = 3
Step 2: Jug1 = 3, Jug2 = 0
Step 3: Jug1 = 3, Jug2 = 3
Step 4: Jug1 = 4, Jug2 = 2
PS D:\AI Practical>

```

Signature of Faculty:

Grade:

Practical – 3

Aim: Write a program to implement DFS (for 8 puzzle problem or Water Jug problem or any AI search problem).

- **Objective:** To implement the Breadth-First Search (BFS) algorithm to solve an AI search problem, such as the Water Jug problem, where the goal is to measure a specific amount of water using two jugs of different capacities.

- **Theory:**

The Water Jug Problem is a classic puzzle in artificial intelligence (AI) that involves using two jugs with different capacities to measure a specific amount of water. It is a popular problem to teach problem-solving techniques in AI, particularly when introducing search algorithms. The Water Jug Problem highlights the application of AI to real-world puzzles by breaking down a complex problem into a series of states and transitions that a machine can solve using an intelligent algorithm.

Problem Description: Water Jug Problem

The Water Jug Problem typically involves two jugs with different capacities. The objective is to measure a specific quantity of water by performing operations like filling a jug, emptying a jug, or transferring water between the two jugs. The problem can be stated as follows:

- You are given two jugs, one with a capacity of **X** liters and the other with a capacity of **Y** liters.
- You need to measure exactly **Z** liters of water using these two jugs.
- The allowed operations are:
 - Fill one of the jugs.
 - Empty one of the jugs.
 - Pour water from one jug into another until one jug is either full or empty.

Example:

Input: $m = 3, n = 5, d = 4$

Output: 6

Explanation: Operations are as follow:

Initially, both jugs are empty ($\text{jug1} = 0, \text{jug2} = 0$).

Step 1: Fill the 5 liter jug $\rightarrow (0, 5)$.

Step 2: Pour from the 5 liter jug to the 3 liter jug $\rightarrow (3, 2)$.

Step 3: Empty the 3 liter jug $\rightarrow (0, 2)$.

Step 4: Pour the 2 liters from the 5-liter jug to the 3 liter jug $\rightarrow (2, 0)$.

Step 5: Fill the 5 liter jug again $\rightarrow (2, 5)$.

Step 6: Pour 1 liter from the 5 liter jug into the 3 liter jug $\rightarrow (3, 4)$.

Now, the 5 liter jug contains exactly 4 liters, so we stop and return 6 steps.

Approach:

Solving the Water Jug Problem Using State Space Representation and Depth-First Search (DFS):

For instance, given two jugs with capacities of 3 liters and 5 liters, and a goal of measuring 4 liters, the search for a solution begins from the initial state and moves through various possible states by filling, emptying, and pouring the water between the two jugs.

In this solution, we use a Depth-First Search (DFS) algorithm to solve the Water Jug Problem, where the jugs have capacities of 3 liters and 5 liters, and the goal is to measure 4 liters of water. In DFS, the algorithm explores deeper paths first before backtracking if the solution is not found.

Defining the State Space:

We represent each state as a pair (x, y) where:

1. x is the amount of water in the 3-liter jug.
2. y is the amount of water in the 5-liter jug.

The initial state is $(0, 0)$ because both jugs start empty, and the goal is to reach any state where either jug contains exactly 4 liters of water.

Operations in State Space:

The following operations define the possible transitions from one state to another:

1. Fill the 3-liter jug: Move to $(3, y)$.
2. Fill the 5-liter jug: Move to $(x, 5)$.
3. Empty the 3-liter jug: Move to $(0, y)$.
4. Empty the 5-liter jug: Move to $(x, 0)$.
5. Pour water from the 3-liter jug into the 5-liter jug: Move to $(\max(0, x - (5 - y)), \min(5, x + y))$.
6. Pour water from the 5-liter jug into the 3-liter jug: Move to $(\min(3, x + y), \max(0, y - (3 - x)))$.

• **Tools / Material Needed:**

- **Hardware:** Computer/Laptop
- **Software:** VS code or any IDE, Python

- **Implementation:**

```
#include <iostream>
#include <set>
#include <vector>

using namespace std;

// Function to check if the current state is the target state
bool is_target_state(pair<int, int> state, int target)
{
    return state.first == target || state.second == target;
}

// Function to generate all possible next states from the current state
vector<pair<int, int>> get_next_states(int jug1, int jug2, pair<int, int> curr_state)
{
    vector<pair<int, int>> next_states;
    int x = curr_state.first, y = curr_state.second;

    // Fill jug1
    next_states.push_back({jug1, y});
    // Fill jug2
    next_states.push_back({x, jug2});
    // Empty jug1
    next_states.push_back({0, y});
    // Empty jug2
    next_states.push_back({x, 0});

    // Pour from jug1 to jug2
    int pour_to_jug2 = min(x, jug2 - y);
    next_states.push_back({x - pour_to_jug2, y + pour_to_jug2});

    // Pour from jug2 to jug1
    int pour_to_jug1 = min(y, jug1 - x);
    next_states.push_back({x + pour_to_jug1, y - pour_to_jug1});

    return next_states;
}

// DFS to solve the Water Jug problem
bool dfs_water_jug(int jug1, int jug2, int target, pair<int, int> current_state,
set<pair<int, int>> &visited, vector<pair<int, int>> &path)
{

```

```
// Base case: check if the current state is the target
if (is_target_state(current_state, target))
{
    path.push_back(current_state);
    return true;
}

// Mark the current state as visited
visited.insert(current_state);

// Add the current state to the solution path
path.push_back(current_state);

// Generate all possible next states
vector<pair<int, int>> next_states = get_next_states(jug1, jug2, current_state);
for (auto next_state : next_states)
{
    if (visited.find(next_state) == visited.end())
    {
        // Recursively perform DFS on the next state
        if (dfs_water_jug(jug1, jug2, target, next_state, visited, path))
        {
            return true;
        }
    }
}

// Backtrack if no solution is found in this path
path.pop_back();
return false;
}

int main()
{
    // Jug capacities
    int jug1_capacity = 3; // 3-liter jug
    int jug2_capacity = 5; // 5-liter jug

    // Target amount of water
    int target_amount = 4; // Goal is to measure exactly 4 liters

    // Initial state (0, 0)
    pair<int, int> initial_state = {0, 0};

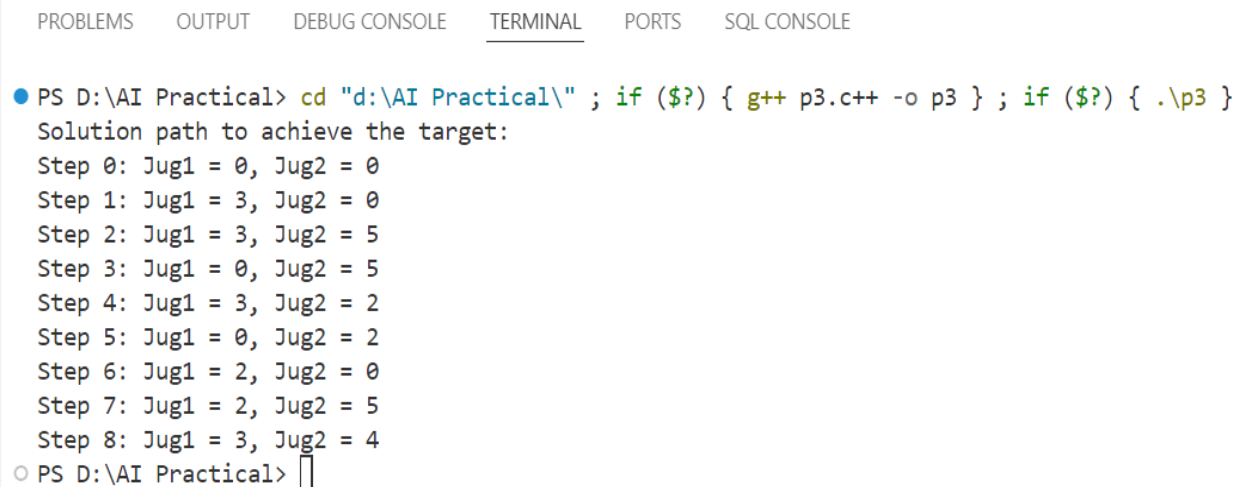
    // Set to track visited states
    set<pair<int, int>> visited;
```

```
// Vector to store the solution path
vector<pair<int, int>> solution_path;

// Perform DFS to find the solution
if (dfs_water_jug(jug1_capacity, jug2_capacity, target_amount, initial_state,
visited, solution_path))
{
    cout << "Solution path to achieve the target:" << endl;
    for (size_t step = 0; step < solution_path.size(); ++step)
    {
        cout << "Step " << step << ": Jug1 = " << solution_path[step].first
            << ", Jug2 = " << solution_path[step].second << endl;
    }
}
else
{
    cout << "No solution found." << endl;
}

return 0;
}
```

- **Output:**



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL CONSOLE

● PS D:\AI Practical> cd "d:\AI Practical\" ; if ($?) { g++ p3.cpp -o p3 } ; if ($?) { .\p3 }
Solution path to achieve the target:
Step 0: Jug1 = 0, Jug2 = 0
Step 1: Jug1 = 3, Jug2 = 0
Step 2: Jug1 = 3, Jug2 = 5
Step 3: Jug1 = 0, Jug2 = 5
Step 4: Jug1 = 3, Jug2 = 2
Step 5: Jug1 = 0, Jug2 = 2
Step 6: Jug1 = 2, Jug2 = 0
Step 7: Jug1 = 2, Jug2 = 5
Step 8: Jug1 = 3, Jug2 = 4
○ PS D:\AI Practical> 
```

Signature of Faculty:

Grade:

Practical – 4

Aim: Write a program to implement Single Player Game (Using any Heuristic Function).

- **Objective:** To solve the 8-puzzle using A* with a heuristic function (Manhattan distance) that estimates how close the current state is to the goal.

- **Theory:**

Single Player 8-Puzzle Game Using A* Algorithm (Manhattan Distance Heuristic)

The **8-puzzle** problem is a classic puzzle problem where the goal is to arrange a set of numbered tiles on a 3x3 grid, leaving one empty space (denoted as 0). The objective is to move the tiles into a specified goal configuration by sliding tiles into the empty space. The puzzle can be solved using various search algorithms, and one of the most effective methods is the *A algorithm**, particularly when combined with a heuristic function like the **Manhattan distance**.

Explanation:

The **A*** algorithm is a popular pathfinding and search algorithm used to solve problems like puzzles and navigation tasks. In the context of the 8-puzzle, A* is used to explore possible moves from the current state to the goal state by evaluating the cost of each move and an estimate of how close the current state is to the goal state.

The A* algorithm uses two key metrics:

1. **$g(n)$** : The cost of reaching the current state from the starting state.
2. **$h(n)$** : A heuristic function estimating the cost of reaching the goal from the current state. In the 8-puzzle, **Manhattan distance** is commonly used as the heuristic function. This measures the total distance each tile needs to move from its current position to its goal position.

The total cost **$f(n)$** is the sum of the two:

- **$f(n) = g(n) + h(n)$**

At each step, the algorithm selects the state with the lowest **$f(n)$** , making it a very efficient approach for solving the puzzle.

Heuristic Function (Manhattan Distance):

The **Manhattan distance** heuristic is used to estimate the cost of moving each tile to its correct position. It is calculated by summing the absolute differences between the current and goal positions for each tile. This heuristic is admissible, meaning it never overestimates the cost, ensuring that A* will find an optimal solution if one exists.

Algorithm:

The A* algorithm for solving the 8-puzzle can be described as follows:

1. Initialization:

- Initialize two lists: an **open list** (for states to explore) and a **closed list** (for explored states).
- Add the starting state to the open list and set its **g** value to 0 and its **h** value to the Manhattan distance between the current state and the goal.

2. Exploration:

- While the open list is not empty, pick the state with the lowest **f(n)** from the open list. Let's call this state **q**.
- If **q** is the goal state, terminate the search, and trace back the path from the goal to the start to get the solution.
- Otherwise, generate all possible valid moves (neighbor states) from **q** by sliding the empty space (0) in one of the four directions: up, down, left, or right.

3. Evaluation:

- For each neighboring state, calculate its **g(n)** (cost to reach this state) and **h(n)** (heuristic value using Manhattan distance).
- If the state has already been explored and its current **f(n)** is lower, discard it.
- Otherwise, add the state to the open list for further exploration.

4. Termination:

- The process continues until the goal state is reached or the open list is empty (no solution found).

- **Tools / Material Needed:**

- **Hardware:** Computer/Laptop
- **Software:** VS code or any IDE, Python

- **Implementation:**

```

? #include <iostream>
? #include <vector>
? #include <queue>
? #include <set>
? #include <algorithm>
? #include <cmath>
? #include <iterator>
? #include <tuple>
?
? using namespace std;
?
? // Manhattan Distance Heuristic Function
? int manhattan_distance(const vector<vector<int>> &state, const vector<int> &goal)
? {

```

```

int total_distance = 0;
for (int i = 0; i < 3; ++i)
{
    for (int j = 0; j < 3; ++j)
    {
        if (state[i][j] != 0)
        {
            int value = state[i][j];
            auto goal_pos = find(goal.begin(), goal.end(), value);
            int goal_idx = distance(goal.begin(), goal_pos);
            int goal_x = goal_idx / 3;
            int goal_y = goal_idx % 3;
            total_distance += abs(goal_x - i) + abs(goal_y - j);
        }
    }
}
return total_distance;
}

// Function to find the index of '0' (empty space)
pair<int, int> find_zero(const vector<vector<int>> &state)
{
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            if (state[i][j] == 0)
            {
                return {i, j};
            }
        }
    }
    return {-1, -1}; // Should never reach here
}

// Class to represent each node (state) in the search space
class Node
{
public:
    vector<vector<int>> state;
    Node *parent;
    string move;
    int g; // Cost from start to this node
    int h; // Heuristic value
    int f; // Total cost (f = g + h)

```

```

Node(const vector<vector<int>> &state, Node *parent = nullptr, const string
&move = "", int g = 0, int h = 0)
    : state(state), parent(parent), move(move), g(g), h(h), f(g + h) {}

// Comparison function for priority queue
bool operator<(const Node &other) const
{
    return f > other.f; // Greater f means lower priority in the min-heap
}

};

// Function to generate the possible moves from the current state
vector<pair<vector<vector<int>>, string>> generate_neighbors(const
vector<vector<int>> &state)
{
    vector<pair<vector<vector<int>>, string>> neighbors;
    int x, y;
    tie(x, y) = find_zero(state);

    vector<pair<string, pair<int, int>>> moves = {
        {"up", {x - 1, y}},
        {"down", {x + 1, y}},
        {"left", {x, y - 1}},
        {"right", {x, y + 1}}};

    for (const auto &move : moves)
    {
        int new_x = move.second.first;
        int new_y = move.second.second;
        if (0 <= new_x && new_x < 3 && 0 <= new_y && new_y < 3)
        { // Check boundaries
            vector<vector<int>> new_state = state;
            swap(new_state[x][y], new_state[new_x][new_y]); // Swap empty space
            with adjacent tile
            neighbors.push_back({new_state, move.first});
        }
    }
    return neighbors;
}

// A* Algorithm to solve the 8-puzzle
pair<vector<vector<vector<int>>>, vector<string>> astar_8_puzzle(const
vector<vector<int>> &start, const vector<vector<int>> &goal)
{
    vector<int> flat_goal;
    for (const auto &row : goal)

```

```

{
    flat_goal.insert(flat_goal.end(), row.begin(), row.end());
}

Node *start_node = new Node(start, nullptr, "", 0, manhattan_distance(start,
flat_goal));
priority_queue<Node> open_list;
open_list.push(*start_node);
set<vector<vector<int>>> closed_list;

while (!open_list.empty())
{
    Node current_node = open_list.top();
    open_list.pop();

    if (current_node.state == goal)
    {
        vector<vector<vector<int>>> path;
        vector<string> moves;
        Node *node_ptr = &current_node;
        while (node_ptr != nullptr)
        {
            path.push_back(node_ptr->state);
            moves.push_back(node_ptr->move);
            node_ptr = node_ptr->parent;
        }
        reverse(path.begin(), path.end());
        reverse(moves.begin(), moves.end());
        return {path, moves};
    }

    closed_list.insert(current_node.state);

    for (const auto &neighbor : generate_neighbors(current_node.state))
    {
        const auto &neighbor_state = neighbor.first;
        const auto &move = neighbor.second;

        if (closed_list.find(neighbor_state) != closed_list.end())
        {
            continue;
        }

        int g = current_node.g + 1;
        int h = manhattan_distance(neighbor_state, flat_goal);
    }
}

```



```

Node *neighbor_node = new Node(neighbor_state, new Node(current_node),
move, g, h);
    open_list.push(*neighbor_node);
}
}
return {{}, {}}; // No solution found
}

int main()
{
    // Initial state (random configuration)
    vector<vector<int>> start_state = {
        {1, 2, 3},
        {4, 0, 6},
        {7, 5, 8}};

    // Goal state (target configuration)
    vector<vector<int>> goal_state = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 0}};

    // Run the A* algorithm
    auto result = astar_8_puzzle(start_state, goal_state);
    auto path = result.first;
    auto moves = result.second;

    if (!path.empty())
    {
        cout << "Solution found!" << endl;
        cout << "Moves to reach the goal:" << endl;
        for (const auto &move : moves)
        {
            if (!move.empty())
            {
                cout << move << endl;
            }
        }
        cout << "\nPath to the solution:" << endl;
        for (const auto &step : path)
        {
            for (const auto &row : step)
            {
                for (int num : row)
                {
                    cout << num << " ";

```

```

    }
    cout << endl;
}
cout << endl; // Newline after each step
}
}
else
{
    cout << "No solution found." << endl;
}

return 0;
}

```

• Output:

```

PS D:\AI Practical> cd "d:\AI Practical\" ; if ($?) { g++ p4.c++ -o p4 } ; if ($?) { .\p4 }
Solution found!
Moves to reach the goal:
down
right

Path to the solution:
1 2 3
4 0 6
7 5 8

1 2 3
4 5 6
7 0 8

1 2 3
4 5 6
7 8 0

PS D:\AI Practical>

```

Signature of Faculty:

Grade:

Practical – 5

Aim: Write a program to Implement A* Algorithm.

- **Objective:** To implement the A* algorithm, a widely used search algorithm in AI, for finding the shortest path in a graph. A* combines the advantages of Breadth-First Search (BFS) and Dijkstra's Algorithm by using both path cost and heuristic information to guide its search.

- **Theory:**

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Explanation:

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.

What A* Search Algorithm does is that at each step it picks the node according to a value-‘f’ which is a parameter equal to the sum of two other parameters – ‘g’ and ‘h’. At each step it picks the node/cell having the lowest ‘f’, and process that node/cell.

We define ‘g’ and ‘h’ as simply as possible below

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this ‘h’ which are discussed in the later sections.

Algorithm:

We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

// A* Search Algorithm

1. Initialize the open list

2. Initialize the closed list

 put the starting node on the open

 list (you can leave its f at zero)

3. while the open list is not empty
 - a) find the node with the least f on the open list, call it "q"
 - b) pop q off the open list
 - c) generate q's 8 successors and set their parents to q
 - d) for each successor
 - i) if successor is the goal, stop search
 - ii) else, compute both g and h for successor
successor.g = q.g + distance between successor and q
successor.h = distance from goal to successor (This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)

successor.f = successor.g + successor.h
 - iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
 - iv) if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor
otherwise, add the node to the open list
 - end (for loop)
 - e) push q on the closed list
 - end (while loop)

- **Tools / Material Needed:**

- **Hardware:** Computer/Laptop
- **Software:** VS code or any IDE, Python

- **Implementation:**

```

#include <iostream>
#include <vector>
#include <queue>
#include <set>
#include <map>
#include <cmath>
#include <memory>
#include<algorithm>

using namespace std;

// Class to represent a position in the grid
class Node
{
public:
    pair<int, int> position;
    shared_ptr<Node> parent;
    int g; // Cost from start to this node
    int h; // Heuristic: estimated cost to goal
    int f; // Total cost (f = g + h)

    Node(pair<int, int> pos, shared_ptr<Node> parent = nullptr)
        : position(pos), parent(parent), g(0), h(0), f(0) {}

    // Comparison operator for the priority queue (min-heap)
    bool operator<(const Node &other) const
    {
        return this->f > other.f; // In priority queue, higher `f` value
means lower priority
    }
};

// Heuristic function: Manhattan distance
int heuristic(const pair<int, int> &current, const pair<int, int> &goal)
{

```

```
    return abs(current.first - goal.first) + abs(current.second -  
goal.second);  
}
```

```
// A* algorithm function
```

```
vector<pair<int, int>> astar_algorithm(const vector<vector<int>> &grid,  
pair<int, int> start, pair<int, int> goal)  
{
```

```
    // Initialize the start and goal nodes
```

```
    shared_ptr<Node> start_node = make_shared<Node>(start);
```

```
    shared_ptr<Node> goal_node = make_shared<Node>(goal);
```

```
    // Open list as a priority queue (min-heap)
```

```
    priority_queue<Node> open_list;
```

```
    open_list.push(*start_node);
```

```
    // Closed list to track visited nodes
```

```
    set<pair<int, int>> closed_list;
```

```
    // Map to store the cost from the start node to each visited node
```

```
    map<pair<int, int>, int> g_score;
```

```
    g_score[start] = 0;
```

```
    while (!open_list.empty())
```

```
    {
```

```
        // Get the node with the lowest f value
```

```
        Node current_node = open_list.top();
```

```
        open_list.pop();
```

```
        // If the goal is reached, reconstruct the path
```

```
        if (current_node.position == goal_node->position)
```

```
        {
```

```
            vector<pair<int, int>> path;
```

```
            shared_ptr<Node> current = make_shared<Node>(current_node);
```

```
            while (current != nullptr)
```

```
            {
```

```
                path.push_back(current->position);
```

```
                current = current->parent;
```

```

    }
    reverse(path.begin(), path.end());
    return path; // Return reversed path from start to goal
}

// Add the current node to the closed list
closed_list.insert(current_node.position);

// Possible moves: right, left, down, up
vector<pair<int, int>> neighbors = {
    {0, 1}, // Right
    {0, -1}, // Left
    {1, 0}, // Down
    {-1, 0} // Up
};

for (const auto &move : neighbors)
{
    pair<int, int> neighbor_pos = {current_node.position.first +
move.first, current_node.position.second + move.second};

    // Check if the neighbor is out of bounds or a wall
    (inaccessible)
    if (neighbor_pos.first >= 0 && neighbor_pos.first < grid.size()
&&
        neighbor_pos.second >= 0 && neighbor_pos.second <
grid[0].size() &&
        grid[neighbor_pos.first][neighbor_pos.second] == 0)
    {

        // Skip if the neighbor is already in the closed list
        if (closed_list.find(neighbor_pos) != closed_list.end())
            continue;

        // Create a new node for this neighbor
        shared_ptr<Node> neighbor_node =
make_shared<Node>(neighbor_pos, make_shared<Node>(current_node));

```

```

        // Calculate g, h, and f values
        int tentative_g = g_score[current_node.position] + 1; //
Assuming uniform cost for each move

        neighbor_node->g = tentative_g;
        neighbor_node->h = heuristic(neighbor_node->position,
goal_node->position);
        neighbor_node->f = neighbor_node->g + neighbor_node->h;

        // If the neighbor has not been visited or a better path is
found, add to the open list
        if (g_score.find(neighbor_pos) == g_score.end() ||
tentative_g < g_score[neighbor_pos])
        {
            g_score[neighbor_pos] = tentative_g;
            open_list.push(*neighbor_node);
        }
    }
}

return {}; // No path found
}

int main()
{
    // Define a grid where 0 represents free space and 1 represents
obstacles
    vector<vector<int>> grid = {
        {0, 1, 0, 0, 0},
        {0, 1, 0, 1, 0},
        {0, 0, 0, 1, 0},
        {1, 1, 0, 1, 0},
        {0, 0, 0, 0, 0}};

    // Start and goal positions
    pair<int, int> start = {0, 0}; // Top-left corner
    pair<int, int> goal = {4, 4}; // Bottom-right corner

```



```

// Run the A* algorithm
vector<pair<int, int>> path = astar_algorithm(grid, start, goal);

if (!path.empty())
{
    cout << "Path found:" << endl;
    for (const auto &step : path)
    {
        cout << "(" << step.first << ", " << step.second << ")" <<
endl;
    }
}
else
{
    cout << "No path found." << endl;
}

return 0;
}

```

• Output:

```

● PS D:\AI Practical> cd "d:\AI Practical"
● PS D:\AI Practical> cd "d:\AI Practical\" ; if ($?) { g++ p5.c++ -o p5 } ; if ($?) { .\p5 }
Path found:
(0, 0)
(1, 0)
(2, 0)
(2, 1)
(2, 2)
(3, 2)
(4, 2)
(4, 3)
(4, 4)
○ PS D:\AI Practical> 

```

Signature of Faculty:

Grade:

Practical – 6

Aim: Write a program to implement mini-max algorithm for any game development.

- **Objective:** To implement the Mini-Max algorithm to make an optimal move for the computer in a game of Tic-Tac-Toe. The computer plays as X and the human player plays as O.

- **Theory:**

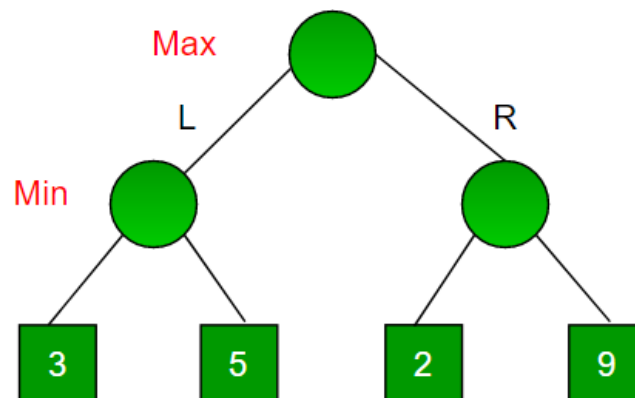
Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

Example:

Consider a game which has 4 final states and paths to reach final state are from root to 4 leaves of a perfect binary tree as shown below. Assume you are the maximizing player and you get the first chance to move.



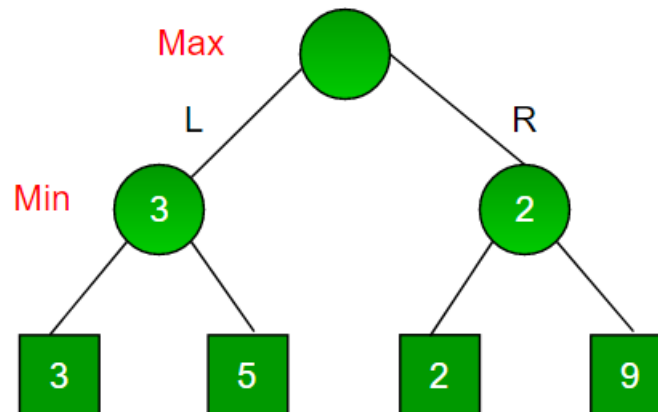
Since this is a backtracking based algorithm, it tries all possible moves, then backtracks and makes a decision.

Maximizer goes LEFT: It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the minimizer it will definitely choose the least among both, that is 3

Maximizer goes RIGHT: It is now the minimizers turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values.

Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.

Now the game tree looks like below :



The above tree shows two possible scores when maximizer makes left and right moves.

• Tools / Material Needed:

- **Hardware:** Computer/Laptop
- **Software:** VS code or any IDE, Python

• Implementation:

```

#include <iostream>
#include <limits.h>

using namespace std;

#define PLAYER_X 1
#define PLAYER_O -1
#define EMPTY 0

// Function to print the Tic-Tac-Toe board
void printBoard(int board[3][3])
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {

```

```
        if (board[i][j] == PLAYER_X)
            cout << "X ";
        else if (board[i][j] == PLAYER_O)
            cout << "O ";
        else
            cout << "- ";
    }
    cout << endl;
}
}

// Function to check if the game is over (win or tie)
int checkWin(int board[3][3])
{
    // Checking Rows
    for (int i = 0; i < 3; i++)
    {
        if (board[i][0] == board[i][1] && board[i][1] == board[i][2] && board[i][0] !=
EMPTY)
        {
            return board[i][0];
        }
    }

    // Checking Columns
    for (int i = 0; i < 3; i++)
    {
        if (board[0][i] == board[1][i] && board[1][i] == board[2][i] && board[0][i] !=
EMPTY)
        {
            return board[0][i];
        }
    }

    // Checking Diagonals
    if (board[0][0] == board[1][1] && board[1][1] == board[2][2] && board[0][0] !=
EMPTY)
    {
        return board[0][0];
    }
    if (board[0][2] == board[1][1] && board[1][1] == board[2][0] && board[0][2] !=
EMPTY)
    {
        return board[0][2];
    }
}
```

```
    return 0; // No win
}

// Function to check if there are moves left
bool isMovesLeft(int board[3][3])
{
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (board[i][j] == EMPTY)
                return true;
    return false;
}

// Mini-Max function
int minimax(int board[3][3], int depth, bool isMaximizingPlayer)
{
    int score = checkWin(board);

    // If X has won
    if (score == PLAYER_X)
        return 10 - depth;

    // If O has won
    if (score == PLAYER_O)
        return depth - 10;

    // If no moves are left and no one has won, it's a tie
    if (!isMovesLeft(board))
        return 0;

    // Maximizing player's move (X)
    if (isMaximizingPlayer)
    {
        int best = INT_MIN;

        // Traverse all cells
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                // Check if cell is empty
                if (board[i][j] == EMPTY)
                {
                    // Make the move
                    board[i][j] = PLAYER_X;
```

```

        // Call minimax recursively and choose the maximum value
        best = max(best, minimax(board, depth + 1, false));

        // Undo the move
        board[i][j] = EMPTY;
    }
}
return best;
}

// Minimizing player's move (O)
else
{
    int best = INT_MAX;

    // Traverse all cells
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            // Check if cell is empty
            if (board[i][j] == EMPTY)
            {
                // Make the move
                board[i][j] = PLAYER_0;

                // Call minimax recursively and choose the minimum value
                best = min(best, minimax(board, depth + 1, true));

                // Undo the move
                board[i][j] = EMPTY;
            }
        }
    }
    return best;
}
}

// Function to find the best move for X
pair<int, int> findBestMove(int board[3][3])
{
    int bestVal = INT_MIN;
    pair<int, int> bestMove = {-1, -1};

    // Traverse all cells, evaluate minimax function for all empty cells

```

```

for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
    {
        // Check if cell is empty
        if (board[i][j] == EMPTY)
        {
            // Make the move
            board[i][j] = PLAYER_X;

            // Compute the value of the move
            int moveVal = minimax(board, 0, false);

            // Undo the move
            board[i][j] = EMPTY;

            // If the value of the current move is more than the best value, update
bestMove

            if (moveVal > bestVal)
            {
                bestMove.first = i;
                bestMove.second = j;
                bestVal = moveVal;
            }
        }
    }
}

return bestMove;
}

int main()
{
    int board[3][3] = {
        {PLAYER_X, PLAYER_O, PLAYER_X},
        {PLAYER_O, PLAYER_O, PLAYER_X},
        {EMPTY, EMPTY, EMPTY}};

    cout << "Current board state:\n";
    printBoard(board);

    pair<int, int> bestMove = findBestMove(board);

    cout << "\nThe best move for X is: Row = " << bestMove.first
        << ", Column = " << bestMove.second << endl;
}

```

```
    return 0;  
}
```

- **Output:**

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SQL CONSOLE  
  
● PS D:\AI Practical> cd "d:\AI Practical"  
● PS D:\AI Practical> cd "d:\AI Practical\" ; if ($?) { g++ p6.cpp -o p6 } ; if ($?) { .\p6 }  
Current board state:  
X O X  
O O X  
- - -  
  
The best move for X is: Row = 2, Column = 2  
○ PS D:\AI Practical>
```

Signature of Faculty:

Grade:

