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. **Dodiya Dhrupalkumar Jesingbhai**, Enrollment Number **210280107059** 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 of this practical is to implement a simple two-player **Tic-Tac-Toe game** using C++. The program should allow players to take turns marking the board, check for winning conditions, handle invalid inputs, and declare the game as either a win, loss, or draw.

Theory:

Tic-Tac-Toe is a popular two-player game that involves a grid of 3x3 squares. Each player takes turns marking a space in the grid, with one player using 'X' and the other 'O'. The goal of the game is for a player to align three of their marks either horizontally, vertically, or diagonally. If all spaces are filled without any player achieving this, the game ends in a draw.

In this practical, the game is implemented using the C++ programming language. The primary concepts involved in the implementation are:

1. **Array Representation**:
   The game board is represented using a 2D array of size 3x3. Initially, each cell is filled with a number representing the position (1-9), allowing players to choose a spot.
2. **Player Input and Validation**:
   Players input their desired position, which is validated to ensure that the spot is available and within the valid range (1-9). If a position is already taken, the player is prompted to select a different position.
3. **Win Conditions**:
   The game checks for winning conditions after every turn by examining all rows, columns, and diagonals. If a player successfully marks three consecutive cells in any direction, they are declared the winner.

4. **Draw Condition**:
   The game also checks for a draw condition, which occurs when all positions are filled, and no player has won.

5. **Game Loop**:
   A loop runs continuously until a win or draw is detected. The game alternates between players after each valid move, ensuring that both players get turns.

This project helps in understanding essential programming concepts like:

- **2D Arrays**: For board representation.
- **Conditional Statements**: To check for win or draw conditions.
- **Loops**: For managing player turns.
- **Functions**: To modularize the code for various tasks like displaying the board, checking for a win, and handling player input.

The implementation of Tic-Tac-Toe is an excellent exercise in basic game logic, array manipulation, and input validation in C++.

Code:

```
#include <iostream>

using namespace std;


// Global 3x3 board array initialized with numbers 1-9

char board[3][3] = { {'1', '2', '3'}, {'4', '5', '6'}, {'7', '8', '9'} };


// Function to display the board

void displayBoard() {

   cout << "-------------\n";

   for (int i = 0; i < 3; i++) {

      cout << "| ";

      for (int j = 0; j < 3; j++) {

         cout << board[i][j] << " | ";
```

```
        }
        cout << "\n-------------\n";
    }
}

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

// Function to check if the game is a draw
bool checkDraw() {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i][j] != 'X' && board[i][j] != 'O') {
                return false;
            }
```

```
        }
    }
    return true;
}

// Function to update the board based on player's move
bool makeMove(int player, int position) {
    char mark = (player == 1) ? 'X' : 'O';

    // Map the position number to board row and column
    int row = (position - 1) / 3;
    int col = (position - 1) % 3;

    // Check if the position is available
    if (board[row][col] != 'X' && board[row][col] != 'O') {
        board[row][col] = mark;
        return true;
    } else {
        cout << "Position already taken! Choose another one.\n";
        return false;
    }
}

// Main function to run the game
int main() {
    int player = 1; // Player 1 starts
    int position;
    bool gameOver = false;

    cout << "Welcome to Tic-Tac-Toe!\n";
```

```
while (!gameOver) {
    displayBoard();
    cout << "Player " << player << "'s turn (Enter position 1-9): ";
    cin >> position;

    if (position < 1 || position > 9) {
        cout << "Invalid position! Please enter a number between 1 and 9.\n";
        continue;
    }

    // Make the move and check if it's valid
    if (!makeMove(player, position)) {
        continue;
    }

    // Check if the player has won
    if (checkWin()) {
        displayBoard();
        cout << "Player " << player << " wins the game!\n";
        gameOver = true;
    }
    // Check if the game is a draw
    else if (checkDraw()) {
        displayBoard();
        cout << "The game is a draw!\n";
        gameOver = true;
    }

    // Switch player turn
    player = (player == 1) ? 2 : 1;
}
```

```
    return 0;
}
```

OUTPUT:

```
PS C:\Users\Dhrupalsinh\Desktop\submition\AI code partA> cd "c:\Users\Dhrupalsinh\Desktop\submition\AI co
de partA\" ; if ($?) { g++ 1.cpp -o 1 } ; if ($?) { .\1 }
Welcome to Tic-Tac-Toe!
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
Player 1's turn (Enter position 1-9):
```

```
-------------
| X | O | 3 |
-------------
| 4 | X | 6 |
-------------
| 7 | O | X |
-------------
Player 1 wins the game!
PS C:\Users\Dhrupalsinh\Desktop\submition\AI code partA>
```

Conclusion:

In this practical, we implemented a simple Tic-Tac-Toe game using C++. The project helped in understanding key programming concepts like 2D arrays, conditional statements, and loops. We successfully managed player input, checked for win/draw conditions, and developed a functional game loop. Overall, this exercise enhanced our skills in building interactive console-based games.

Practical:2

AIM: Write a program to implement BFS for Water Jug problem

Objective:

To implement the **Breadth-First Search (BFS)** algorithm to solve the **Water Jug Problem**, where we measure a specific amount of water using two jugs with given capacities through a series of operations like filling, emptying, and pouring water.

Theory:

The **Water Jug Problem** is a classic problem in **Artificial Intelligence** that involves finding a solution to measure an exact quantity of water using two jugs with different capacities. By using operations such as filling, emptying, and transferring water between the jugs, we can reach the desired amount. The BFS algorithm is ideal for exploring all possible states (jug configurations) systematically, ensuring the shortest solution path. BFS uses a queue to track the states and expands each state to explore possible moves until the goal is reached.

Code:

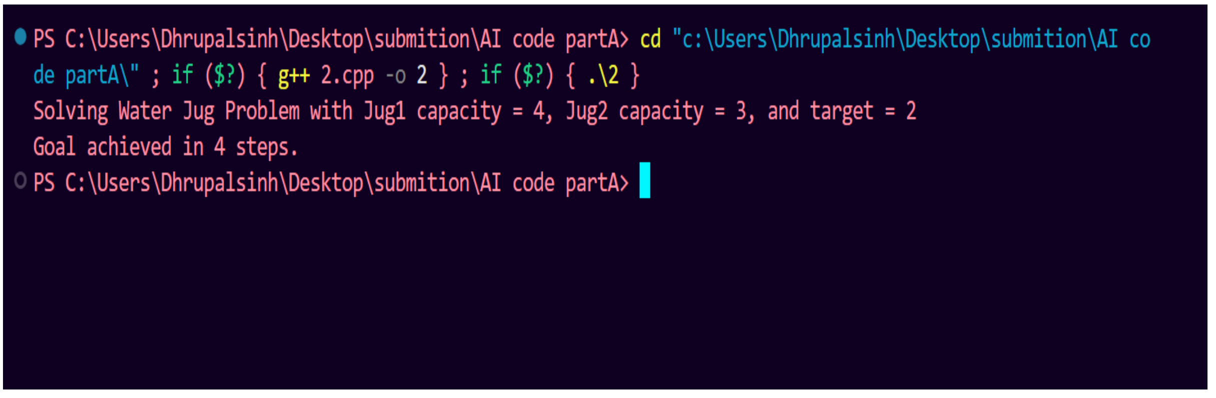
```
#include <iostream>
#include <queue>
#include <set>

using namespace std;

// State to represent the current amount of water in Jug1 and Jug2
struct State {
    int jug1, jug2;
    int step;  // to keep track of the number of steps
```

```
    State(int a, int b, int s) : jug1(a), jug2(b), step(s) {}
};

// Function to check if the goal state is achieved
bool isGoalState(int jug1, int jug2, int target) {
    return (jug1 == target || jug2 == target);
}

// Function to perform BFS to solve the Water Jug problem
void bfs(int jug1Capacity, int jug2Capacity, int target) {
    set<pair<int, int>> visited;  // To keep track of visited states
    queue<State> q;  // Queue for BFS

    // Start with both jugs empty
    q.push(State(0, 0, 0));
    visited.insert({0, 0});

    while (!q.empty()) {
        State current = q.front();
        q.pop();

        // If the target is reached in either jug, we are done
        if (isGoalState(current.jug1, current.jug2, target)) {
            cout << "Goal achieved in " << current.step << " steps." << endl;
            return;
        }

        // Possible operations (fill, empty, pour)
        vector<State> nextStates;
```

```
        // Fill Jug1 completely
        nextStates.push_back(State(jug1Capacity, current.jug2, current.step + 1));
        // Fill Jug2 completely
        nextStates.push_back(State(current.jug1, jug2Capacity, current.step + 1));
        // Empty Jug1
        nextStates.push_back(State(0, current.jug2, current.step + 1));
        // Empty Jug2
        nextStates.push_back(State(current.jug1, 0, current.step + 1));

        // Pour water from Jug1 to Jug2
        int pourToJug2 = min(current.jug1, jug2Capacity - current.jug2);
        nextStates.push_back(State(current.jug1 - pourToJug2, current.jug2 + pourToJug2, current.step +
1));

        // Pour water from Jug2 to Jug1
        int pourToJug1 = min(current.jug2, jug1Capacity - current.jug1);
        nextStates.push_back(State(current.jug1 + pourToJug1, current.jug2 - pourToJug1, current.step +
1));

        // Enqueue all valid next states
        for (State nextState : nextStates) {
            if (visited.find({nextState.jug1, nextState.jug2}) == visited.end()) {
                visited.insert({nextState.jug1, nextState.jug2});
                q.push(nextState);
            }
        }
    }

    cout << "No solution found." << endl;
}
```

```
int main() {
    int jug1Capacity = 4;  // Capacity of Jug1
    int jug2Capacity = 3;  // Capacity of Jug2
    int target = 2;  // Target amount of water to measure

    cout << "Solving Water Jug Problem with Jug1 capacity = " << jug1Capacity
         << ", Jug2 capacity = " << jug2Capacity << ", and target = " << target << endl;

    bfs(jug1Capacity, jug2Capacity, target);
    return 0;
}
```

OUTPUT:



```
PS C:\Users\Dhrupalsinh\Desktop\submition\AI code partA> cd "c:\Users\Dhrupalsinh\Desktop\submition\AI co
de partA\" ; if ($?) { g++ 2.cpp -o 2 } ; if ($?) { .\2 }
Solving Water Jug Problem with Jug1 capacity = 4, Jug2 capacity = 3, and target = 2
Goal achieved in 4 steps.
PS C:\Users\Dhrupalsinh\Desktop\submition\AI code partA>
```

Conclusion:

In this implementation of the Water Jug problem using Breadth-First Search (BFS), we successfully demonstrated how BFS can be utilized to explore all possible states in an unweighted search space. The algorithm efficiently finds the shortest path to reach the goal state, ensuring that all potential jug combinations are considered. By representing each state as a node and applying the BFS strategy, we can guarantee that the solution, if it exists, will be found with minimal operations. This approach can be extended to other problems involving similar state-space exploration and optimization scenarios.

Practical:3

AIM: Write a program to implement DFS for Water Jug problem.

Objective:

To implement the **Depth-First Search (DFS)** algorithm to solve the **Water Jug Problem**, where we aim to measure a specific amount of water using two jugs through a series of operations such as filling, emptying, and pouring.

Theory:

The **Water Jug Problem** is an Artificial Intelligence problem where two jugs with different capacities are used to measure a target quantity of water. The **DFS** algorithm explores each possible sequence of operations, diving deep into one path before backtracking to explore other paths. By maintaining a stack and keeping track of visited states, DFS finds a solution by exhaustively searching through all possibilities.

Code:

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

using namespace std;

// State to represent the current amount of water in Jug1 and Jug2
struct State {
    int jug1, jug2;
```

```
    State(int a, int b) : jug1(a), jug2(b) {}
};

// Function to check if the goal state is achieved
bool isGoalState(int jug1, int jug2, int target) {
    return (jug1 == target || jug2 == target);
}

// Function to print the current state
void printState(int jug1, int jug2) {
    cout << "Jug1: " << jug1 << " | Jug2: " << jug2 << endl;
}

// Function to perform DFS to solve the Water Jug problem
void dfs(int jug1Capacity, int jug2Capacity, int target) {
    set<pair<int, int>> visited;  // To keep track of visited states
    stack<State> s;  // Stack for DFS

    // Start with both jugs empty
    s.push(State(0, 0));
    visited.insert({0, 0});

    while (!s.empty()) {
        State current = s.top();
        s.pop();

        // Display current state
        printState(current.jug1, current.jug2);

        // If the target is reached in either jug, we are done
        if (isGoalState(current.jug1, current.jug2, target)) {
```

```
        cout << "Goal achieved: " << current.jug1 << " in Jug1, "
             << current.jug2 << " in Jug2." << endl;
        return;
    }

    // Possible operations (fill, empty, pour)
    vector<State> nextStates;

    // Fill Jug1 completely
    nextStates.push_back(State(jug1Capacity, current.jug2));
    // Fill Jug2 completely
    nextStates.push_back(State(current.jug1, jug2Capacity));
    // Empty Jug1
    nextStates.push_back(State(0, current.jug2));
    // Empty Jug2
    nextStates.push_back(State(current.jug1, 0));

    // Pour water from Jug1 to Jug2
    int pourToJug2 = min(current.jug1, jug2Capacity - current.jug2);
    nextStates.push_back(State(current.jug1 - pourToJug2, current.jug2 + pourToJug2));

    // Pour water from Jug2 to Jug1
    int pourToJug1 = min(current.jug2, jug1Capacity - current.jug1);
    nextStates.push_back(State(current.jug1 + pourToJug1, current.jug2 - pourToJug1));

    // Push all valid next states onto the stack
    for (State nextState : nextStates) {
        if (visited.find({nextState.jug1, nextState.jug2}) == visited.end()) {
            visited.insert({nextState.jug1, nextState.jug2});
            s.push(nextState);
        }
```

```
        }
    }
    cout << "No solution found." << endl;
}


int main() {
    int jug1Capacity = 4;  // Capacity of Jug1
    int jug2Capacity = 3;  // Capacity of Jug2
    int target = 2;  // Target amount of water to measure

    cout << "Solving Water Jug Problem with Jug1 capacity = " << jug1Capacity
         << ", Jug2 capacity = " << jug2Capacity << ", and target = " << target << endl;

    dfs(jug1Capacity, jug2Capacity, target);
    return 0;
}
```

OUTPUT:

```
PS C:\Users\Dhrupalsinh\Desktop\submition\AI code partA> cd "c:\Users\Dhrupalsinh\Desktop\submition\AI co
de partA\" ; if ($?) { g++ 3.cpp -o 3 } ; if ($?) { .\3 }
Solving Water Jug Problem with Jug1 capacity = 4, Jug2 capacity = 3, and target = 2
Jug1: 0 | Jug2: 0
Jug1: 0 | Jug2: 3
Jug1: 3 | Jug2: 0
Jug1: 3 | Jug2: 3
Jug1: 4 | Jug2: 2
Goal achieved: 4 in Jug1, 2 in Jug2.
PS C:\Users\Dhrupalsinh\Desktop\submition\AI code partA>
```

Conclusion:

The Water Jug Problem was successfully solved using the **Depth-First Search (DFS)** algorithm. This approach explored various configurations of water levels in the two jugs by simulating different operations until the goal was achieved. The practical enhanced our understanding of how DFS works in AI search problems, exploring deep paths before backtracking to find solutions.

Practical:4

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

Objective:

To implement a **single-player game** (8-puzzle problem) using the *A search algorithm** with the **Manhattan Distance heuristic** to efficiently find the solution to the puzzle.

Theory:

The 8-puzzle is a classic single-player problem in Artificial Intelligence. The puzzle can be solved using *A search**, a heuristic-based algorithm. By applying the **Manhattan Distance** as a heuristic function, the algorithm explores states in a way that minimizes the total number of moves required to reach the goal.

Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <cmath>

using namespace std;

// Size of the puzzle
#define N 3

// Structure to represent a state of the puzzle
struct PuzzleState {
```

```
    vector<vector<int>> board;  // Puzzle board
    int x, y;                   // Blank tile position
    int g, h;                   // g = steps taken, h = heuristic value
    PuzzleState* parent;        // Pointer to parent state for tracing the solution

    // Constructor
    PuzzleState(vector<vector<int>> b, int x, int y, int g, int h, PuzzleState* p)
        : board(b), x(x), y(y), g(g), h(h), parent(p) {}
};

// Function to check if the current state is the goal state
bool isGoalState(const vector<vector<int>>& board) {
    vector<vector<int>> goal = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};
    return board == goal;
}

// Manhattan Distance Heuristic Function
int calculateManhattanDistance(const vector<vector<int>>& board) {
    int distance = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int value = board[i][j];
            if (value != 0) {
                int targetX = (value - 1) / N;
                int targetY = (value - 1) % N;
                distance += abs(i - targetX) + abs(j - targetY);
            }
        }
    }
    return distance;
}
```

```
// Function to print the puzzle board
void printBoard(const vector<vector<int>>& board) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         if (board[i][j] == 0)
            cout << "  ";
         else
            cout << board[i][j] << " ";
      }
      cout << endl;
   }
   cout << endl;
}

// Custom comparator for priority queue
struct Compare {
   bool operator()(const PuzzleState* a, const PuzzleState* b) {
      return (a->g + a->h) > (b->g + b->h);  // Minimize f = g + h
   }
};

// Function to solve the 8-puzzle problem using A* search with Manhattan Distance heuristic
void solvePuzzle(vector<vector<int>> start) {
   // Possible moves (up, down, left, right)
   int dx[] = {-1, 1, 0, 0};
   int dy[] = {0, 0, -1, 1};

   // Initial state of the puzzle
   int startX, startY;
   for (int i = 0; i < N; i++) {
```

```
        for (int j = 0; j < N; j++) {
            if (start[i][j] == 0) {
                startX = i;
                startY = j;
            }
        }
    }

    // Priority queue for A* search
    priority_queue<PuzzleState*, vector<PuzzleState*>, Compare> pq;

    // Set of visited states
    map<vector<vector<int>>, bool> visited;

    // Push the initial state into the priority queue
    PuzzleState* initialState = new PuzzleState(start, startX, startY, 0,
calculateManhattanDistance(start), nullptr);
    pq.push(initialState);
    visited[start] = true;

    while (!pq.empty()) {
        PuzzleState* current = pq.top();
        pq.pop();
        // If the goal state is reached, print the solution path
        if (isGoalState(current->board)) {
            cout << "Solution found!" << endl;
            vector<PuzzleState*> solutionPath;
            PuzzleState* temp = current;
            // Trace the path from goal to initial state
            while (temp != nullptr) {
                solutionPath.push_back(temp);
```

```
            temp = temp->parent;
        }

        // Print the solution path
        for (int i = solutionPath.size() - 1; i >= 0; i--) {
            printBoard(solutionPath[i]->board);
        }
        return;
    }
    // Try all possible moves
    for (int i = 0; i < 4; i++) {
        int newX = current->x + dx[i];
        int newY = current->y + dy[i];

        if (newX >= 0 && newX < N && newY >= 0 && newY < N) {
            vector<vector<int>> newBoard = current->board;
            swap(newBoard[current->x][current->y], newBoard[newX][newY]);

            if (!visited[newBoard]) {
                int h = calculateManhattanDistance(newBoard);
                PuzzleState* newState = new PuzzleState(newBoard, newX, newY, current->g + 1, h,
current);
                pq.push(newState);
                visited[newBoard] = true;
            }}} }
  cout << "No solution found." << endl;
}

int main() {
  // Initial configuration of the puzzle (0 represents the blank space)
  vector<vector<int>> start = {
```

```
        {1, 2, 3},
        {4, 0, 5},
        {7, 8, 6}
    };
    cout << "Starting Puzzle:" << endl;
    printBoard(start);
    // Solve the puzzle using A* search with Manhattan Distance heuristic
    solvePuzzle(start);
    return 0;
}
```

OUTPUT:

```
Starting Puzzle:
1 2 3
4   5
7 8 6

Solution found!
1 2 3
4   5
7 8 6

1 2 3
4   5
7 8 6

1 2 3
4 5
4   5
7 8 6

1 2 3
1 2 3
4 5
7 8 6

1 2 3
4 5 6
7 8
```

Conclusion:

The 8-puzzle problem was successfully solved using the _A search algorithm_* with the **Manhattan Distance heuristic**. This approach efficiently explored the puzzle states and found the optimal solution path, demonstrating the effectiveness of heuristic functions in AI search problems

Practical:5

AIM: Write a program to implement A* Algorithm.

Objective:

The objective of this program is to implement the A* algorithm to solve the 8-puzzle problem, efficiently finding the shortest path from the initial state to the goal state using the Manhattan distance heuristic.

Theory:

8-Puzzle Problem Overview:

- The 8-puzzle consists of a 3x3 grid, with eight numbered tiles (1–8) and one blank space.
- The goal is to move the tiles to reach a specified goal configuration by sliding them into the blank space.

Heuristic:

We'll use the Manhattan distance heuristic, which calculates the sum of the distances of the tiles from their goal positions.

Code:

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

using namespace std;
```

```
struct Node {
    vector<vector<int>> state;
    int g, h, f; // g(n) = cost from start, h(n) = heuristic, f(n) = g(n) + h(n)
    int x, y; // Position of the blank tile
    Node* parent;

    Node(vector<vector<int>> state, int g, int h, int x, int y, Node* parent = nullptr)
        : state(state), g(g), h(h), f(g + h), x(x), y(y), parent(parent) {}
};

// Function to calculate Manhattan distance heuristic
int calculateHeuristic(const vector<vector<int>>& state, const vector<vector<int>>& goal) {
    int h = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (state[i][j] != 0) {
                for (int x = 0; x < 3; x++) {
                    for (int y = 0; y < 3; y++) {
                        if (state[i][j] == goal[x][y]) {
                            h += abs(i - x) + abs(j - y); // Manhattan distance
                        }}}}}}
    return h;
}

// Function to check if the current state is the goal state
bool isGoalState(const vector<vector<int>>& state, const vector<vector<int>>& goal) {
    return state == goal;
}

// Function to print the current state
void printState(const vector<vector<int>>& state) {
```

```
    for (const auto& row : state) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    cout << endl;
}

// Comparison operator for priority queue (min-heap based on f-value)
struct compare {
    bool operator()(Node* a, Node* b) {
        return a->f > b->f;
    }
};
// Function to get the neighbors (possible moves) of the current state
vector<Node*> getNeighbors(Node* node, const vector<vector<int>>& goal) {
    vector<Node*> neighbors;
    vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}}; // Up, Down, Left, Right

    for (auto dir : directions) {
        int newX = node->x + dir.first;
        int newY = node->y + dir.second;

        if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {
            vector<vector<int>> newState = node->state;
            swap(newState[node->x][node->y], newState[newX][newY]);

            int g = node->g + 1;
            int h = calculateHeuristic(newState, goal);
            neighbors.push_back(new Node(newState, g, h, newX, newY, node));
```

```
        }
    }
    return neighbors;
}

// A* Algorithm function
void aStarSearch(const vector<vector<int>>& start, const vector<vector<int>>& goal) {
    priority_queue<Node*, vector<Node*>, compare> openList; // Priority queue for A*
    set<vector<vector<int>>> closedList; // To avoid revisiting states

    int x, y;
    // Find the position of the blank tile (0)
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (start[i][j] == 0) {
                x = i;
                y = j;
                break;
            }}}

    Node* root = new Node(start, 0, calculateHeuristic(start, goal), x, y);
    openList.push(root);

    while (!openList.empty()) {
        Node* current = openList.top();
        openList.pop();
        // If goal state is reached
        if (isGoalState(current->state, goal)) {
            cout << "Goal state reached!" << endl;
            cout << "Steps to reach the goal:" << endl;
            while (current != nullptr) {
```

```
            printState(current->state);
            current = current->parent;
        }
        return;
    }

    closedList.insert(current->state);

    // Explore neighbors
    vector<Node*> neighbors = getNeighbors(current, goal);
    for (Node* neighbor : neighbors) {
        if (closedList.find(neighbor->state) == closedList.end()) {
            openList.push(neighbor);
        }} }

  cout << "No solution found!" << endl;
}

int main() {
  vector<vector<int>> start = {
    {1, 2, 3},
    {0, 4, 6},
    {7, 5, 8}
  };
  vector<vector<int>> goal = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 0}
  };

  cout << "Initial State:" << endl;
```

```
    printState(start);
    cout << "Goal State:" << endl;
    printState(goal);
    cout << "Performing A* Search..." << endl;
    aStarSearch(start, goal);
    return 0;
}
```

OUTPUT:

```
Initial State:
1 2 3
0 4 6
7 5 8

Goal State:
1 2 3
4 5 6
7 8 0

Performing A* Search...
Goal state reached!
Steps to reach the goal:
1 2 3
4 5 6
7 8 0

1 2 3
4 5 6
7 0 8

1 2 3
4 0 6
7 5 8

1 2 3
0 4 6
```

Conclusion:

The **A*** algorithm was successfully implemented to solve the **8-puzzle problem** using the Manhattan distance as a heuristic function. The algorithm efficiently finds the optimal solution by balancing the actual cost and the estimated cost to reach the goal.

Practical:6

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

Objective:

The objective of this program is to implement the **Minimax algorithm** to develop an AI player for a simple **Tic-Tac-Toe** game that always plays optimally.

Theory:

The Minimax Algorithm is a popular decision-making algorithm in game theory and artificial intelligence, often used in turn-based games like Tic-Tac-Toe, Chess, or Checkers. It allows a player to minimize the maximum possible loss by assuming the opponent is playing optimally

Code:

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

using namespace std;

char board[3][3] = {
    {'_', '_', '_'},
    {'_', '_', '_'},
    {'_', '_', '_'}
};

// Function to print the board
void printBoard() {
    for (int i = 0; i < 3; i++) {
```

```
        for (int j = 0; j < 3; j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

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

// Function to evaluate the board
int evaluate(char b[3][3]) {
    // Checking rows for victory
    for (int row = 0; row < 3; row++) {
        if (b[row][0] == b[row][1] && b[row][1] == b[row][2]) {
            if (b[row][0] == 'X')
                return +10;
            else if (b[row][0] == 'O')
                return -10;
        }
    }

    // Checking columns for victory
```

```
    for (int col = 0; col < 3; col++) {
        if (b[0][col] == b[1][col] && b[1][col] == b[2][col]) {
            if (b[0][col] == 'X')
                return +10;
            else if (b[0][col] == 'O')
                return -10;
        }
    }

    // Checking diagonals for victory
    if (b[0][0] == b[1][1] && b[1][1] == b[2][2]) {
        if (b[0][0] == 'X')
            return +10;
        else if (b[0][0] == 'O')
            return -10;
    }

    if (b[0][2] == b[1][1] && b[1][1] == b[2][0]) {
        if (b[0][2] == 'X')
            return +10;
        else if (b[0][2] == 'O')
            return -10;
    }

    return 0;
}

// Minimax function to calculate the best move
int minimax(char board[3][3], int depth, bool isMax) {
    int score = evaluate(board);
```

```
  // If Maximizer (X) has won
  if (score == 10)
    return score;

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

  // If no moves left and no winner, it's a draw
  if (isMovesLeft(board) == false)
    return 0;

  // If it's Maximizer's move (X)
  if (isMax) {
    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] == '_') {
          // Make the move
          board[i][j] = 'X';

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

          // Undo the move
          board[i][j] = '_';
        }
      }
```

```
        }
        return best;
    }

    // If it's Minimizer'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] == '_') {
                    // Make the move
                    board[i][j] = 'O';

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

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

// Function to find the best move for X
pair<int, int> findBestMove(char 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] == '_') {
                // Make the move
                board[i][j] = 'X';

                // Compute evaluation function for this move
                int moveVal = minimax(board, 0, false);

                // Undo the move
                board[i][j] = '_';

                // If the value of the current move is more than the best value, update best
                if (moveVal > bestVal) {
                    bestMove = {i, j};
                    bestVal = moveVal;
                }
            }
        }
    }
    return bestMove;
}

int main() {
    cout << "Tic-Tac-Toe Game\n";
    printBoard();
```

```
    int turn = 0; // 0 for human (O), 1 for AI (X)
    while (isMovesLeft(board)) {
        int row, col;
        if (turn == 0) {
            cout << "Enter your move (row and column): ";
            cin >> row >> col;
            if (board[row][col] == '_') {
                board[row][col] = 'O';
                turn = 1;
            } else {
                cout << "Invalid move! Try again.\n";
                continue;
            }
        } else {
            cout << "AI is making its move...\n";
            pair<int, int> bestMove = findBestMove(board);
            board[bestMove.first][bestMove.second] = 'X';
            turn = 0;
        }

        printBoard();
        int score = evaluate(board);
        if (score == 10) {
            cout << "AI (X) wins!\n";
            break;
        } else if (score == -10) {
            cout << "You (O) win!\n";
            break;
        } else if (!isMovesLeft(board)) {
            cout << "It's a draw!\n";
            break;
```

```
        }
    }

    return 0;
}
```

OUTPUT:

```
PS C:\Users\Dhrupalsinh\Desktop\submition\AI code partA> cd "c:\Users\Dhrupalsinh\Desktop\submition\AI co
de partA\" ; if ($?) { g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFil
e }
Tic-Tac-Toe Game
_ _ _
_ _ _
_ _ _
Enter your move (row and column): 1 3
_ _ _
_ _ _
O _ _
AI is making its move...
_ _ _
_ X _
O _ _
```

Conclusion:

The **Minimax Algorithm** was successfully implemented to create an AI player that plays optimally in the **Tic-Tac-Toe** game. The AI considers all possible moves and makes decisions that maximize its chances of winning, ensuring that the player faces a challenging