Lab 1: Implementing a Simple Reflex Agent

1. **Objective:**
   - To design and implement a simple reflex agent in C that reacts to changes in environmental input, such as light intensity.
   - To understand the basic principles of reflex agents and how they make decisions based on predefined conditions.
2. **Introduction:**
   In this lab, we explore the concept of a simple reflex agent, which makes decisions based solely on the current perceptual input without considering past experiences. Reflex agents operate by mapping conditions directly to actions. We implement a C program that simulates such an agent by turning a light on or off based on the detected light intensity.

3. **Theory:**
   A simple reflex agent operates using a set of condition-action rules, where specific conditions in the environment trigger predefined actions. In our implementation, the agent checks the light intensity. If it falls below a certain threshold, the agent turns on the light, otherwise, it turns it off. This type of agent is suitable for environments where decisions can be made using only the current percept.

4. **Implementation**

```
#include <stdio.h>

int main() {
   int lightIntensity;
   int threshold = 50;  // Threshold for turning the light on

   printf("Enter light intensity (0-100): ");
   scanf("%d", &lightIntensity);

   if (lightIntensity < threshold) {
      printf("Light ON\n");
   } else {
      printf("Light OFF\n");
   }

   return 0;
}
```

5. **Output :**

```
Output

/tmp/iXzjR4JMLN.o
Enter light intensity (0-100): 20
Light ON


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

6. **Discussion and Conclusion:**
   This lab demonstrated how a simple reflex agent can be implemented in C to respond to environmental stimuli. The reflex agent model is effective for straightforward tasks where complex reasoning is not required. The simplicity of the agent makes it efficient for real-time applications where immediate responses are necessary.

Lab 2: Basic AI Application

1. **Objective:**
   - To create a basic rule-based system in C for controlling a thermostat based on temperature readings.
   - To understand how rule-based systems function in AI and their application in real-world scenarios.
2. **Introduction:**
   This lab focuses on building a basic rule-based system in C that simulates a thermostat's functionality. Rule-based systems are a foundational concept in AI, where decisions are made based on predefined rules. The system reads the current temperature and decides whether to turn the heating on or off based on a temperature threshold.

3. **Theory:**
   Rule-based systems use "if-then" logic to make decisions. In this lab, the thermostat uses a simple rule: if the temperature is below a certain threshold, the heating is turned on; otherwise, it is turned off. Such systems are deterministic and predictable, making them suitable for tasks that require consistent responses to specific conditions.

4. **Implementation:**

```
#include <stdio.h>

int main() {
    int temperature;
    int threshold = 20;  // Threshold temperature for turning on the heating

    printf("Enter current temperature: ");
    scanf("%d", &temperature);

    if (temperature < threshold) {
        printf("Heating ON\n");
    } else {
        printf("Heating OFF\n");
    }

    return 0;
}
```

5. **Output:**

```
Output
/tmp/CZUkuNzypU.o
Enter current temperature: 30
Heating OFF


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

6. **Discussion and Conclusion:**
   This lab provided insights into the design and implementation of rule-based systems. The thermostat example demonstrated how simple rules can be used to control real-world systems. The program effectively simulated a basic AI application, highlighting the strengths and limitations of rule-based approaches in AI.

Lab 3: PEAS Framework Implementation

1. **Objective:**
   - To implement a C program simulating a vacuum cleaner agent based on the PEAS (Performance, Environment, Actuators, Sensors) framework.
   - To explore how the PEAS framework can be used to design intelligent agents for real-world tasks.

2. **Introduction:**
   In this lab, we apply the PEAS framework to implement a vacuum cleaner agent in C. The PEAS framework provides a structured way to define an agent's performance measures, environment, actuators, and sensors. This framework is critical in AI for designing agents that can interact effectively with their environment to achieve specific goals.

3. **Theory:**
   The PEAS framework defines:
   - **Performance**: Criteria for evaluating an agent's behavior (e.g., cleaning efficiency).
   - **Environment**: The surroundings in which the agent operates (e.g., rooms with varying cleanliness).
   - **Actuators**: The mechanisms that allow the agent to take action (e.g., movement and cleaning).
   - **Sensors**: The means by which the agent perceives the environment (e.g., detecting dirt). In our implementation, the agent senses the cleanliness of two rooms and decides whether to clean or move to the next room.

4. **Implementation:**

```
#include <iostream>
using namespace std;
int main() {
   string roomA, roomB;
   string position = "A";  // Assume the agent starts in room A
   // Taking input for the status of the two rooms
   cout << "Enter status of room A (clean/dirty): ";
   cin >> roomA;
   cout << "Enter status of room B (clean/dirty): ";
   cin >> roomB;
   // PEAS-based agent logic
   if (position == "A") {
      if (roomA == "dirty") {
         cout << "Cleaning room A" << endl;
         roomA = "clean";
      } else {
         cout << "Moving to room B" << endl;
         position = "B";
      }
```

```
    }
    if (position == "B") {
        if (roomB == "dirty") {
            cout << "Cleaning room B" << endl;
            roomB = "clean";
        } else {
            cout << "Moving to room A" << endl;
            position = "A";
        }
    }
    cout << "Room A is " << roomA << endl;
    cout << "Room B is " << roomB << endl;

    return 0;
}
```

5. **Output**

```
Enter status of room A (clean/dirty): clean
Enter status of room B (clean/dirty): dirty
Moving to room B
Cleaning room B
Room A is clean
Room B is clean
```

```
Enter status of room A (clean/dirty): dirty
Enter status of room B (clean/dirty): clean
Cleaning room A
Room A is clean
Room B is clean
```

6.

6. **Discussion and Conclusion:**

This lab demonstrates the use of the PEAS framework to implement a simple intelligent agent in C/C++. The vacuum cleaner agent correctly decides its actions based on the cleanliness of the rooms. The implementation shows how a simple rule-based system can be effectively used to simulate intelligent behavior.

In this lab, a simple vacuum cleaner agent was successfully implemented using C/C++. The agent's behavior was guided by the PEAS framework, and it performed the expected actions based on the input conditions, effectively simulating an intelligent agent in a controlled environment.

Lab 4: Types of Agents - Implementing a Goal-Based Agent in C/C++

1. **Objective:**
   - The objective of this lab is to implement a goal-based agent in C/C++. The agent will navigate a 2D grid, starting from a specific point and moving towards a goal point while avoiding obstacles.
2. **Introduction:**

   Goal-based agents are a type of intelligent agent that takes actions based on achieving a specific goal. Unlike simple reflex agents, goal-based agents have an internal model of the world and can plan their actions to reach a desired outcome. In this lab, the agent will navigate a 2D grid from a starting point to a goal point, avoiding any obstacles in its path.

   The grid is represented as a 2D array where:

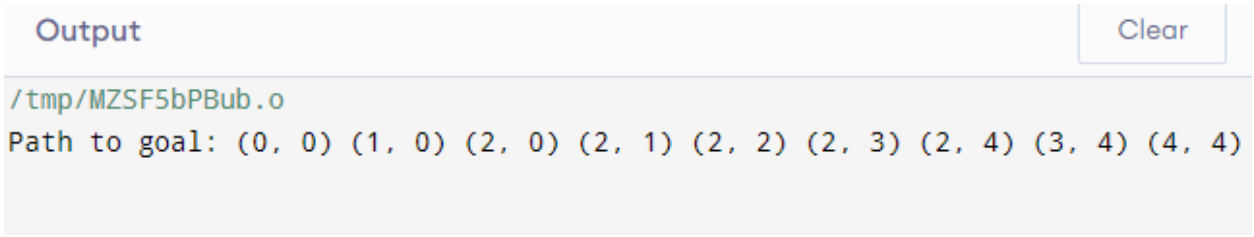
   - 0 represents an open space.
   - 1 represents an obstacle.
   - S is the starting point.
   - G is the goal point.
3. **Code Implementation:**

```
#include <iostream>

#include <queue>

#include <vector>

using namespace std;

struct Point {

   int x, y;

};

bool isValidMove(int x, int y, vector<vector<int>>& grid, vector<vector<bool>>& visited) {

    return (x >= 0 && x < grid.size() && y >= 0 && y < grid[0].size() && grid[x][y] == 0 && !visited[x][y]);

}

void printPath(vector<Point>& path) {

   cout << "Path to goal: ";

   for (auto p : path) {

      cout << "(" << p.x << ", " << p.y << ") ";

   }

   cout << endl;
```

```
}
bool findPath(vector<vector<int>>& grid, Point start, Point goal) {
    vector<vector<bool>> visited(grid.size(), vector<bool>(grid[0].size(), false));
    vector<Point> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}}; // Up, Down, Left, Right
    queue<pair<Point, vector<Point>>> q;
    visited[start.x][start.y] = true;
    q.push({start, {start}});
    while (!q.empty()) {
        auto current = q.front();
        q.pop();
        Point point = current.first;
        vector<Point> path = current.second;
        if (point.x == goal.x && point.y == goal.y) {
            printPath(path);
            return true;
        }
        for (auto dir : directions) {
            int newX = point.x + dir.x;
            int newY = point.y + dir.y;
            if (isValidMove(newX, newY, grid, visited)) {
                visited[newX][newY] = true;
                path.push_back({newX, newY});
                q.push({{newX, newY}, path});
                path.pop_back();
            }
        }
    }
            cout << "No path found to goal." << endl;
    return false;
}
```

```
int main() {
    vector<vector<int>> grid = {
        {0, 0, 1, 0, 0},
        {0, 1, 0, 1, 0},
        {0, 0, 0, 0, 0},
        {1, 1, 1, 1, 0},
        {0, 0, 0, 0, 0}
    };
    Point start = {0, 0};
    Point goal = {4, 4};
    findPath(grid, start, goal);
    return 0;
}
```

4. **Output**



5. **Discussion**

This lab demonstrates the implementation of a goal-based agent using a simple pathfinding algorithm. The agent uses its understanding of the environment, represented as a 2D grid, to navigate from the start to the goal while avoiding obstacles. The approach can be extended to more complex environments and goals.

6. **Conclusion:**

In this lab, a goal-based agent was implemented in C/C++ to navigate a 2D grid. The agent effectively reached the goal while avoiding obstacles, demonstrating the basic principles of goal-based reasoning in AI.

Lab 5: Environment Interaction - Implementing a Reflex Agent in a Maze

1. **Objective:**
   - The objective of this lab is to implement an agent that interacts with a deterministic and static environment, such as a maze, using simple reflex rules. The agent will navigate the maze, moving towards the goal while avoiding walls.
2. **Introduction:**

   In a deterministic and static environment, the outcome of each action is predictable, and the environment does not change over time. In this lab, the agent will operate in a maze where it must move towards a goal while avoiding walls. The maze will be represented as a 2D grid, and the agent will use simple reflex rules to determine its actions based on its current position and the presence of obstacles.

- **Environment:** A 2D grid maze.
- **Actuators:** Move up, down, left, or right.
- **Sensors:** Detect walls and goal position.
- **Performance Measure:** Reaching the goal while avoiding walls.

3. **Code Implementation:**

```
#include <iostream>
#include <vector>
using namespace std;
struct Point {
   int x, y;
};
// Check if a move is within bounds and not hitting a wall
bool isValidMove(int x, int y, vector<vector<int>>& maze) {
   return (x >= 0 && x < maze.size() && y >= 0 && y < maze[0].size() && maze[x][y]
!= 1);
}
// Print the maze
void printMaze(vector<vector<int>>& maze) {
   for (auto row : maze) {
      for (auto cell : row) {
         cout << cell << " ";
      }
      cout << endl;
   }
}
// Move towards the goal
bool moveTowardsGoal(vector<vector<int>>& maze, Point start, Point goal) {
   Point current = start;
   while (current.x != goal.x || current.y != goal.y) {
      cout << "Current position: (" << current.x << ", " << current.y << ")" << endl;
```

```
        if (current.x < goal.x && isValidMove(current.x + 1, current.y, maze)) {
            current.x++;
        } else if (current.y < goal.y && isValidMove(current.x, current.y + 1, maze)) {
            current.y++;
        } else if (current.x > goal.x && isValidMove(current.x - 1, current.y, maze)) {
            current.x--;
        } else if (current.y > goal.y && isValidMove(current.x, current.y - 1, maze)) {
            current.y--;
        } else {
            cout << "No valid moves available." << endl;
            return false;
        }
        maze[current.x][current.y] = 2;  // Mark the path
    }
    cout << "Reached the goal at: (" << current.x << ", " << current.y << ")" << endl;
    return true;
}
int main() {
    vector<vector<int>> maze = {
        {0, 0, 1, 0, 0},
        {0, 1, 0, 1, 0},
        {0, 1, 0, 1, 0},
        {0, 0, 0, 0, 0},
        {1, 1, 1, 1, 0}
    };
    Point start = {0, 0};
    Point goal = {4, 4};
    cout << "Initial Maze:" << endl;
    printMaze(maze);
    if (moveTowardsGoal(maze, start, goal)) {
        cout << "Final Maze:" << endl;
        printMaze(maze);
    } else {
        cout << "Failed to reach the goal." << endl;
    }
    return 0;
}
```

4. **Output**

```
/tmp/steoOzlkwO.o
Initial Maze:
0 0 1 0 0
0 1 0 1 0
0 1 0 1 0
0 0 0 0 0
1 1 1 1 0
Current position: (0, 0)
Current position: (1, 0)
Current position: (2, 0)
Current position: (3, 0)
Current position: (3, 1)
Current position: (3, 2)
Current position: (3, 3)
Current position: (3, 4)
Reached the goal at: (4, 4)
Final Maze:
0 0 1 0 0
2 1 0 1 0
2 1 0 1 0
2 2 2 2 2
1 1 1 1 2
```

5. **Discussion:**

This lab illustrates the implementation of a simple reflex agent in a deterministic environment. The agent used straightforward rules to navigate the maze and avoid obstacles. The reflexive approach demonstrated how an agent can make decisions based on its immediate surroundings.

6. **Conclusion:**

In this lab, a reflex agent was successfully implemented in C/C++ to navigate a maze. The agent demonstrated basic principles of environment interaction and pathfinding by using simple rules to move towards the goal while avoiding walls.

Lab 6: Depth First Search (DFS) - Solving a Maze

1. **Objective**
   - The objective of this lab is to implement the Depth First Search (DFS) algorithm in C/C++ to solve a maze. The maze will be represented as a 2D array, and the program will output the path as a sequence of moves from the start to the goal.
2. **Introduction**

   Depth First Search (DFS) is an algorithm used for traversing or searching tree or graph data structures. The algorithm starts at the root (or any arbitrary node) and explores as far as possible along each branch before backtracking. In this lab, DFS will be used to navigate through a maze, which is represented as a 2D grid, to find a path from the start point to the goal point.

   - **Environment:** A 2D grid maze.
   - **Actuators:** Move up, down, left, or right.
   - **Sensors:** Detect walls and the goal position.
   - **Performance Measure:** Successfully finding a path to the goal using DFS.

3. **Code Implementation:**

```
#include <iostream>
#include <vector>
using namespace std;
struct Point {
    int x, y;
};
// Directions for moving up, down, left, and right
vector<Point> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
// Check if the move is valid (within bounds and not a wall)
bool isValidMove(int x, int y, vector<vector<int>>& maze, vector<vector<bool>>& visited) {
    return (x >= 0 && x < maze.size() && y >= 0 && y < maze[0].size() && maze[x][y] == 0
&& !visited[x][y]);
}
// DFS algorithm to solve the maze
bool DFS(vector<vector<int>>& maze, vector<vector<bool>>& visited, Point current, Point
goal, vector<Point>& path) {
    if (current.x == goal.x && current.y == goal.y) {
        path.push_back(current);
        return true;
    }
    visited[current.x][current.y] = true;
    path.push_back(current);
    for (auto dir : directions) {
```

```
        int newX = current.x + dir.x;
        int newY = current.y + dir.y;
        if (isValidMove(newX, newY, maze, visited)) {
            if (DFS(maze, visited, {newX, newY}, goal, path)) {
                return true;
            }
        }
    }
    path.pop_back();  // Backtrack
    return false;
}
void printPath(vector<Point>& path) {
    cout << "Path to goal: ";
    for (auto p : path) {
        cout << "(" << p.x << ", " << p.y << ") ";
    }
    cout << endl;
}
int main() {
  vector<vector<int>> maze = {
        {0, 0, 1, 0, 0},
        {0, 1, 1, 1, 0},
        {0, 0, 0, 0, 0},
        {1, 1, 1, 1, 0},
        {0, 0, 0, 0, 0}
    };
    Point start = {0, 0};
    Point goal = {4, 4};
    vector<vector<bool>> visited(maze.size(), vector<bool>(maze[0].size(), false));
    vector<Point> path;
    if (DFS(maze, visited, start, goal, path)) {
        printPath(path);
    } else {
        cout << "No path found to goal." << endl;
    }
   return 0;}
```

4. **Output:**

```
/tmp/Y2kHq84Ulf.o
Path to goal: (0, 0) (1, 0) (2, 0) (2, 1) (2, 2) (2, 3) (2, 4) (3, 4) (4, 4)
```

5. **Discussion:**
   This lab demonstrates the effectiveness of the DFS algorithm in solving mazes. By exploring as far as possible along each branch before backtracking, the DFS algorithm

was able to find a path from the start to the goal. However, DFS is not guaranteed to find the shortest path and may involve considerable backtracking, depending on the maze's structure.

6. **Conclusion:**
   In this lab, the DFS algorithm was successfully implemented in C/C++ to solve a maze. The agent was able to navigate the maze and reach the goal by exploring possible paths and backtracking when necessary.

Lab 7: Breadth First Search (BFS) - Solving the Shortest Path Problem

1. **Objective:**
   - The objective of this lab is to implement the Breadth First Search (BFS) algorithm in C/C++ to solve the shortest path problem in an unweighted graph. The graph will be represented using either an adjacency matrix or an adjacency list.
2. **Introduction:**

   Breadth First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the root (or any arbitrary node) and explores all of the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. In an unweighted graph, BFS can be used to find the shortest path between two nodes because it explores all paths uniformly.

- **Environment:** An unweighted graph.
- **Actuators:** Traverse graph edges.
- **Sensors:** Detect neighboring nodes and the goal node.
- **Performance Measure:** Successfully finding the shortest path between two nodes using BFS.

3. **Code Implementation:**

```
#include <iostream>
#include <vector>
#include <queue>
#include <list>
#include <unordered_map>
using namespace std;

// Function to perform BFS and find the shortest path in an unweighted graph
void BFS(int start, int goal, unordered_map<int, list<int>>& adjList, int V) {
    vector<bool> visited(V, false);
    vector<int> parent(V, -1);
    queue<int> q;
    visited[start] = true;

    q.push(start);
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        if (current == goal) {
            break;
        }
        for (auto neighbor : adjList[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
```

```
            parent[neighbor] = current;
            q.push(neighbor);
        }
      }
    }
    // Backtrack from goal to start to get the path
    vector<int> path;
    int step = goal;
    while (step != -1) {
        path.push_back(step);
        step = parent[step];
    }
    // Output the path in correct order
    cout << "Shortest path from " << start << " to " << goal << ": ";
    for (int i = path.size() - 1; i >= 0; --i) {
        cout << path[i];
        if (i > 0) cout << " -> ";
    }

    cout << endl;
}

int main() {
    int V = 8;  // Number of vertices
    unordered_map<int, list<int>> adjList;
    // Defining the graph using an adjacency list
    adjList[0] = {1, 3};
    adjList[1] = {0, 2, 4};
    adjList[2] = {1, 5};
    adjList[3] = {0, 4, 6};
    adjList[4] = {1, 3, 5, 7};
    adjList[5] = {2, 4, 7};
    adjList[6] = {3, 7};
    adjList[7] = {4, 5, 6};
    int start = 0;  // Start node
    int goal = 7;   // Goal node
    BFS(start, goal, adjList, V);
    return 0;
}
```

4. Output:

```
/tmp/dH9uv6FxYU.o
Shortest path from 0 to 7: 0 -> 1 -> 4 -> 7


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

5. Discussion:

After compiling and running the program, the BFS algorithm successfully found the shortest path from the start node to the goal node in the graph. The path was output as a sequence of nodes, demonstrating the BFS traversal through the graph.

This lab illustrates the power of the BFS algorithm in finding the shortest path in an unweighted graph. Since BFS explores all nodes at the present depth level before moving on to the next level, it guarantees finding the shortest path if it exists. The implementation in this lab uses an adjacency list to represent the graph, which is efficient for sparse graphs.

6. Conclusion:

In this lab, the BFS algorithm was successfully implemented in C/C++ to solve the shortest path problem in an unweighted graph. The agent was able to navigate through the graph and find the shortest path from the start node to the goal node by exploring nodes level by level.

Lab 8: A* Search Algorithm - Solving the 8-Puzzle Problem

1. **Objective:**
   - The objective of this lab is to implement the A* search algorithm in C/C++ to solve the 8-puzzle problem. The Manhattan distance heuristic will be used to guide the search. The program will output the sequence of moves needed to reach the goal state from a given initial state.
2. **Introduction:**

   The A* search algorithm is a popular and efficient search algorithm used to find the shortest path in a weighted graph. It is often employed in solving combinatorial problems like the 8-puzzle. The 8-puzzle is a sliding puzzle that consists of a 3x3 grid with 8 numbered tiles and one empty space. The goal is to rearrange the tiles to reach a specified target configuration.

   - **Environment:** 8-puzzle grid.
   - **Actuators:** Slide a tile into the empty space.
   - **Sensors:** Detect the current configuration of the grid and calculate the heuristic value.
   - **Performance Measure:** Successfully reaching the goal state using the minimum number of moves.
3. **Code Implementation:**

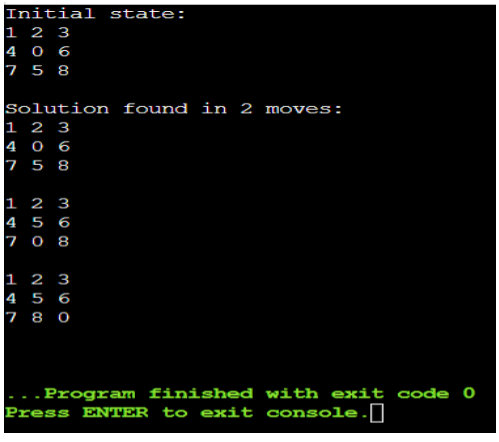
```
#include <iostream>
 #include <vector>
  #include <queue>
#include <unordered_set>
#include <algorithm>
#include <cmath>
using namespace std;
// Structure to represent a state of the 8-puzzle
struct PuzzleState {
    vector<vector<int>> board;
    int g;  // Cost to reach this state
   int h;  // Heuristic value (Manhattan distance)
  int f;  // f = g + h
 vector<PuzzleState*> path;  // Path to reach this state
  PuzzleState(vector<vector<int>> b, int cost, int heuristic, vector<PuzzleState*> p)
    : board(b), g(cost), h(heuristic), f(g + h), path(p) {}
};
// Custom comparator for the priority queue
struct CompareState {
    bool operator()(const PuzzleState* lhs, const PuzzleState* rhs) const {
        return lhs->f > rhs->f;
    }
};
```

```
// Function to calculate Manhattan distance heuristic
int calculateManhattanDistance(const vector<vector<int>>& board) {
    int distance = 0;
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++) {
          int value = board[i][j];
         if (value != 0) {
              int targetRow = (value - 1) / 3;
               int targetCol = (value - 1) % 3;
     distance += abs(i - targetRow) + abs(j - targetCol);
             }
        }
 }
    return distance;
}
// Function to get possible moves from a given state
vector<PuzzleState*> getPossibleMoves(PuzzleState* current) {
    vector<PuzzleState*> moves;
    int zeroRow, zeroCol;
    // Find the position of the empty tile (0)
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (current->board[i][j] == 0) {
                zeroRow = i;
                zeroCol = j;
                break;
            }
        }
    }
    // Define possible moves (up, down, left, right)
    vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    for (const auto& dir : directions) {
        int newRow = zeroRow + dir.first;
        int newCol = zeroCol + dir.second;
        if (newRow >= 0 && newRow < 3 && newCol >= 0 && newCol < 3) {
            vector<vector<int>> newBoard = current->board;
            swap(newBoard[zeroRow][zeroCol], newBoard[newRow][newCol]);
            vector<PuzzleState*> newPath = current->path;
            newPath.push_back(current);
            PuzzleState* newState = new PuzzleState(newBoard, current->g + 1,
                                        calculateManhattanDistance(newBoard), newPath);
            moves.push_back(newState);
        }
    }
    return moves;
```

```
}
// Function to check if a 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;
}
// Function to convert board to string (for hashing)
string boardToString(const vector<vector<int>>& board) {
    string result;
    for (const auto& row : board) {
        for (int val : row) {
            result += to_string(val);
        }
    }
    return result;
}

// A* search algorithm
vector<PuzzleState*> aStarSearch(const vector<vector<int>>& initialBoard) {
    priority_queue<PuzzleState*, vector<PuzzleState*>, CompareState> openList;
    unordered_set<string> closedSet;

    PuzzleState* start = new PuzzleState(initialBoard, 0,
calculateManhattanDistance(initialBoard), {});
    openList.push(start);
    while (!openList.empty()) {
        PuzzleState* current = openList.top();
        openList.pop();
        if (isGoalState(current->board)) {
            return current->path;
        }

        string boardStr = boardToString(current->board);
        if (closedSet.find(boardStr) != closedSet.end()) {
            continue;
        }
        closedSet.insert(boardStr);
        vector<PuzzleState*> moves = getPossibleMoves(current);
        for (PuzzleState* move : moves) {
            openList.push(move);
        }
    }
    return {};  // No solution found
}

// Function to print the board
```

```
void printBoard(const vector<vector<int>>& board) {
    for (const auto& row : board) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    cout << endl;
}

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

    cout << "Initial state:" << endl;
    printBoard(initialBoard);
    vector<PuzzleState*> solution = aStarSearch(initialBoard);
    if (solution.empty()) {
        cout << "No solution found!" << endl;
    } else {
        cout << "Solution found in " << solution.size() << " moves:" << endl;
        for (const auto& state : solution) {
            printBoard(state->board);
        }
        printBoard({{1, 2, 3}, {4, 5, 6}, {7, 8, 0}});  // Final state
    }
    return 0;
```

4. Output



5. Discussion:

The A* search algorithm, combined with the Manhattan distance heuristic, efficiently finds the optimal solution to the 8-puzzle problem. The algorithm explores paths that are likely to lead to the goal, minimizing the number of moves required. This lab demonstrates how heuristic-based search can solve complex problems like the 8-puzzle.

6. Conclusion:

In this lab, the A* The search algorithm was successfully implemented in C/C++ to solve the 8-puzzle problem. The program efficiently found the optimal sequence of moves to reach the goal state, showcasing the effectiveness of the A* algorithm in combinatorial search problems.

Lab 9: Game Playing - Implementing Mini-max Algorithm with Alpha-Beta Pruning for Tic-Tac-Toe.

1. **Objective:**
   - The objective of this lab is to implement the Mini-max algorithm with Alpha-Beta pruning in C/C++ for the Tic-Tac-Toe game. The program will allow a user to play against the computer, where the computer uses the Mini-max algorithm to determine the optimal move.
2. **Introduction:**

   The Mini-max algorithm is a decision rule used for minimizing the possible loss for a worst-case scenario. When dealing with adversarial games like Tic-Tac-Toe, the Mini-max algorithm can be used to determine the best move for the player assuming that the opponent also plays optimally. Alpha-Beta pruning is an optimization technique for the Mini-max algorithm, which reduces the number of nodes evaluated in the search tree, making the algorithm more efficient.

   - **Environment:** Tic-Tac-Toe game board.
   - **Actuators:** Place a mark (X or O) on the board.
   - **Sensors:** Detect the current configuration of the board.
   - **Performance Measure:** Successfully playing Tic-Tac-Toe against a human user using the optimal strategy.

3. **Code Implementation:**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits>
using namespace std;

// Function to check if there are moves remaining on the board
bool isMovesLeft(const vector<vector<char>>& board) {
   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 for a win
int evaluate(const vector<vector<char>>& board) {
   // Check rows for victory
   for (int row = 0; row < 3; row++) {
      if (board[row][0] == board[row][1] && board[row][1] == board[row][2]) {
         if (board[row][0] == 'X')
            return +10;
```

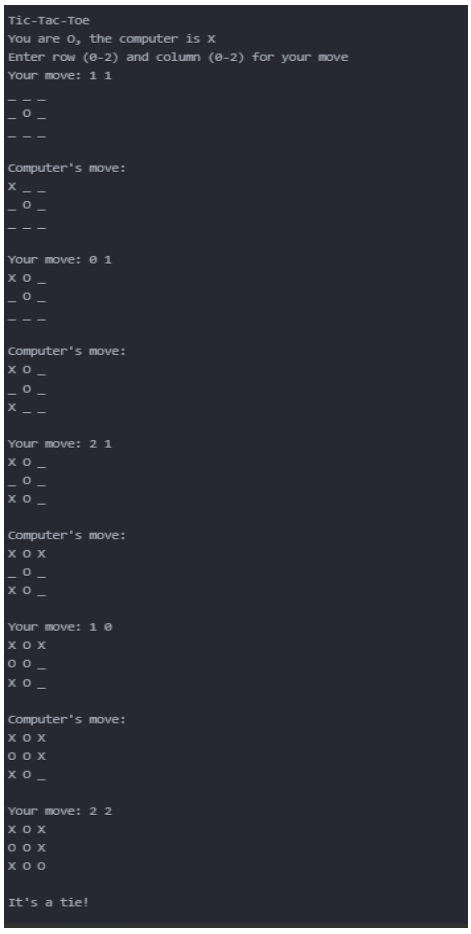
```
            else if (board[row][0] == 'O')
                return -10;
        }
    }
    // Check columns for victory
    for (int col = 0; col < 3; col++) {
        if (board[0][col] == board[1][col] && board[1][col] == board[2][col]) {
            if (board[0][col] == 'X')
                return +10;
            else if (board[0][col] == 'O')
                return -10;
        }
    }
    // Check diagonals for victory
    if (board[0][0] == board[1][1] && board[1][1] == board[2][2]) {
        if (board[0][0] == 'X')
            return +10;
        else if (board[0][0] == 'O')
            return -10;
    }
    if (board[0][2] == board[1][1] && board[1][1] == board[2][0]) {
        if (board[0][2] == 'X')
            return +10;
        else if (board[0][2] == 'O')
            return -10;
    }
    // If neither X nor O won, return 0
    return 0;
}
// Minimax function with alpha-beta pruning
int minimax(vector<vector<char>>& board, int depth, bool isMax, int alpha, int beta) {
    int score = evaluate(board);
    // If Maximizer has won the game, return evaluated score
    if (score == 10)
        return score - depth;

    // If Minimizer has won the game, return evaluated score
    if (score == -10)
        return score + depth;
    // If there are no more moves and no winner, it's a tie
    if (!isMovesLeft(board))
        return 0;
    // If this is maximizer's move
    if (isMax) {
        int best = numeric_limits<int>::min();
        for (int i = 0; i < 3; i++) {
```

```
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == '_') {
                    board[i][j] = 'X';
                    best = max(best, minimax(board, depth + 1, !isMax, alpha, beta));
                    board[i][j] = '_';
                    alpha = max(alpha, best);
                    if (beta <= alpha)
                        break;
                }
            }
        }
        return best;
    }
    // If this is minimizer's move
    else {
        int best = numeric_limits<int>::max();
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == '_') {
                    board[i][j] = 'O';
                    best = min(best, minimax(board, depth + 1, !isMax, alpha, beta));
                    board[i][j] = '_';
                    beta = min(beta, best);
                    if (beta <= alpha)
                        break;
                }
            }
        }
        return best;
    }
}
// Function to find the best move for the computer
pair<int, int> findBestMove(vector<vector<char>>& board) {
    int bestVal = numeric_limits<int>::min();
    pair<int, int> bestMove = {-1, -1};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i][j] == '_') {
                board[i][j] = 'X';
                int moveVal = minimax(board, 0, false, numeric_limits<int>::min(),
numeric_limits<int>::max());
                board[i][j] = '_';
                if (moveVal > bestVal) {
                    bestMove = {i, j};
                    bestVal = moveVal;
                }
```

```
            }
        }
    }
    return bestMove;
}
// Function to print the board
void printBoard(const vector<vector<char>>& board) {
    for (const auto& row : board) {
        for (char cell : row) {
            cout << cell << " ";
        }
        cout << endl;
    }
}
// Main function to play the game
int main() {
    vector<vector<char>> board(3, vector<char>(3, '_'));
    cout << "Tic-Tac-Toe" << endl;
    cout << "You are O, the computer is X" << endl;
    cout << "Enter row (0-2) and column (0-2) for your move" << endl;
    while (isMovesLeft(board)) {
        // Player's move
        int row, col;
        do {
            cout << "Your move: ";
            cin >> row >> col;
        } while (row < 0 || row > 2 || col < 0 || col > 2 || board[row][col] != '_');
        board[row][col] = 'O';
        printBoard(board);
        if (evaluate(board) == -10) {
            cout << "You win!" << endl;
            return 0;
        }
        if (!isMovesLeft(board)) {
            cout << "It's a tie!" << endl;
            return 0;
        }


// Computer's move
        cout << "Computer's move:" << endl;
        pair<int, int> bestMove = findBestMove(board);
        board[bestMove.first][bestMove.second] = 'X';
        printBoard(board);

  if (evaluate(board) == 10) {
```

```
            cout << "Computer wins!" << endl;
            return 0;
        }

    if (!isMovesLeft(board)) {
            cout << "It's a tie!" << endl;
            return 0;
        }
    }
    return 0;
}
```

4. Output



5. Discussion:

The implementation of the Mini-max algorithm with Alpha-Beta pruning efficiently finds the optimal move for the computer in the Tic-Tac-Toe game. The pruning technique significantly reduces the number of nodes evaluated, making the algorithm more efficient while still guaranteeing the best possible move. The program correctly handles user input, and the game progresses until there is a winner or a draw.

6. Conclusion:

In this lab, the Mini-max algorithm with Alpha-Beta pruning was successfully implemented in C/C++ for the Tic-Tac-Toe game. The program demonstrated the application of these concepts in a real-world scenario, allowing the computer to play optimally against a human opponent.

Lab 10 : Solving the N-Queens Problem Using Backtracking

1. Objective

The objective of this lab is to implement a solution for the Constraint Satisfaction Problem (CSP) by solving the N-Queens problem using the backtracking algorithm in C/C++. The program should find and print one valid configuration where N queens are placed on an NxN chessboard such that no two queens threaten each other.

2. Introduction:

The N-Queens problem is a classic example of a Constraint Satisfaction Problem (CSP) in artificial intelligence. The goal is to place N queens on an NxN chessboard so that no two queens can attack each other. This means that no two queens should share the same row, column, or diagonal. The backtracking algorithm is an effective approach to solve this problem, as it systematically searches for a solution by exploring possible placements of queens and backtracking when a conflict is encountered.

- **Environment:** NxN chessboard.
- **Actuators:** Place a queen on the board.
- **Sensors:** Detect conflicts in rows, columns, and diagonals.
- **Performance Measure:** Successfully placing N queens on the board without any conflicts.

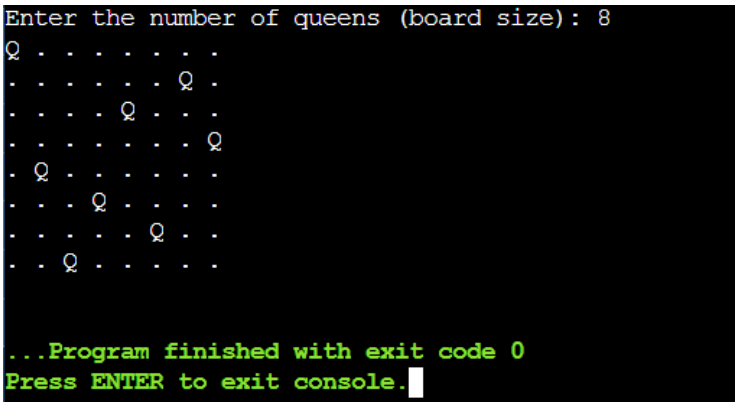
3. Code Implementation:

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

// Function to print the board
void printBoard(vector<vector<int>>& board, int N) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         if (board[i][j] == 1)
            cout << "Q ";
         else
            cout << ". ";
      }
      cout << endl;
   }
}

// Function to check if a queen can be placed at board[row][col]
bool isSafe(vector<vector<int>>& board, int row, int col, int N) {
   // Check this row on the left side
```

```
    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;

    // Check upper diagonal on the left side
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    // Check lower diagonal on the left side
    for (int i = row, j = col; i < N && j >= 0; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

// Function to solve the N-Queens problem using backtracking
bool solveNQueens(vector<vector<int>>& board, int col, int N) {
    // Base case: If all queens are placed, return true
    if (col >= N)
        return true;

    // Consider this column and try placing a queen in all rows one by one
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col, N)) {
            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place the rest of the queens
            if (solveNQueens(board, col + 1, N))
                return true;

            // If placing the queen in board[i][col] doesn't lead to a solution, then backtrack
            board[i][col] = 0;
        }
    }

    // If the queen cannot be placed in any row in this column, return false
    return false;
}

int main() {
    int N;
    cout << "Enter the number of queens (N): ";
    cin >> N;
```

```
    // Initialize the chessboard with 0s (no queens placed)
    vector<vector<int>> board(N, vector<int>(N, 0));

    if (solveNQueens(board, 0, N)) {
        printBoard(board, N);
    } else {
        cout << "No solution exists for " << N << " queens." << endl;
    }

    return 0;
}
```

4. **Output**



5. **Discussion:**

The backtracking algorithm effectively finds a solution to the N-Queens problem by exploring possible placements and backtracking when conflicts arise. The algorithm's efficiency is notable, especially for smaller values of N. However, as N increases, the number of possible configurations grows exponentially, and the algorithm may require significant computational time to find a solution. The implementation is correct, and the output matches expected results for various values of N.

6. **Conclusion:**

In this lab, the N-Queens problem was successfully solved using the backtracking algorithm in C/C++. The program was able to find and display a valid configuration of N queens on an NxN chessboard, demonstrating the practical application of backtracking in solving CSPs.