**Name: Gourav Kumar Shaw**

**Enrollment No. : 2020CSB010**

**Subject : Artificial Intelligence Laboratory.**

**Semester: 8th**

# Assignment – 4

**Qn1.** Develop an advanced program to explore and generate solutions for the placement of eight queens on an 8x8 chessboard. The objective is to devise a strategy ensuring that no queen can attack another, presenting a challenge that demands the implementation of a sophisticated algorithm for systematic exploration of feasible queen configurations, introducing heightened complexity in managing the constraints associated with preventing mutual attacks among the queens.

**Code:**

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

using namespace std;

class EightQueens {
private:
    const int board_size = 8;
    int solution_count = 0;

    bool isSafe(vector<vector<int>>& board, int row, int col) {
        // Check if there is a queen in the same column
        for (int i = 0; i < row; ++i) {
            if (board[i][col] == 1)
                return false;
```

```
        }

        // Check upper left diagonal
        for (int i = row, j = col; i >= 0 && j >= 0; --i, --j) {
            if (board[i][j] == 1)
                return false;
        }

        // Check upper right diagonal
        for (int i = row, j = col; i >= 0 && j < board_size; --i, ++j) {
            if (board[i][j] == 1)
                return false;
        }

        return true;
    }

    void solveQueens(vector<vector<int>>& board, int row) {
        if (row == board_size) {
            printSolution(board);
            solution_count++;
            return;
        }

        for (int col = 0; col < board_size; ++col) {
            if (isSafe(board, row, col)) {
                board[row][col] = 1;
                solveQueens(board, row + 1);
                board[row][col] = 0; // Backtrack
            }
        }
    }

    void printSolution(vector<vector<int>>& board) {
        cout << "Solution " << solution_count + 1 << ":\n";
        for (const auto& row : board) {
            for (int cell : row) {
                cout << cell << " ";
            }
            cout << endl;
        }
        cout << endl;
    }

public:
    void findSolutions() {
        vector<vector<int>> board(board_size, vector<int>(board_size, 0));
        solveQueens(board, 0);
```

```
    }
};

int main() {
    EightQueens eq;
    eq.findSolutions();
    return 0;
}
```

**Qn2.** In the context of a room with a door, a monkey finds itself hungry and positioned at the room's entrance. In the center of the room, a banana dangles tantalizingly from the ceiling. However, the monkey's current stature prevents it from reaching the banana directly from the floor. Adjacent to a window, there exists a box that the monkey can potentially employ to its advantage. The monkey is equipped with a repertoire of actions: walking on the floor, climbing the box, pushing the box (if already at it), and grasping the banana if standing on the box directly beneath it.

a. Investigate the feasibility of the monkey successfully obtaining the banana, factoring in its initial position, physical constraints, and the spectrum of actions available within the room.

b. Devise a meticulous and intricate strategy outlining the specific sequence of actions the monkey should execute to secure the banana. This plan should intricately incorporate floor navigation, box utilization, and precise movements to align the monkey underneath the hanging banana. The complexity of this problem-solving task is heightened due to the intricacies involved in orchestrating a successful sequence of actions.

## Code:

```
#include <iostream>
#include <cmath>

using namespace std;

struct Point {
    int x, y;
};

class MonkeyBananaProblem {
private:
    int room_width, room_length, ceiling_height, box_height, monkey_height;
    Point gate, window;

public:
    void getUserInputs() {
        cout << "Enter room width: ";
        cin >> room_width;
        cout << "Enter room length: ";
        cin >> room_length;
        cout << "Enter ceiling height: ";
        cin >> ceiling_height;
        cout << "Enter box height: ";
        cin >> box_height;
        cout << "Enter monkey height: ";
        cin >> monkey_height;

        cout << "Enter coordinates of the gate (x y): ";
        cin >> gate.x >> gate.y;

        cout << "Enter coordinates of the window (x y) (summing bottom left
corner as 0,0): ";
        cin >> window.x >> window.y;
    }

    bool areCoordinatesValid() {
        return gate.x >= 0 && gate.x <= room_width && gate.y >= 0 && gate.y <=
room_length &&
               window.x >= 0 && window.x <= room_width && window.y >= 0 &&
window.y <= room_length &&
               ceiling_height > 0 && box_height >= 0 && monkey_height >= 0;
    }

    bool canReachBanana() {
       return box_height+monkey_height >= ceiling_height;
    }
```

```
    void printStepsToBanana() {
        if (!areCoordinatesValid()) {
            cout << "Invalid coordinates entered. Please enter valid
coordinates.\n";
            return;
        }

        if (canReachBanana()) {
            cout << "Steps to reach the banana:\n";
            if (monkey_height >= ceiling_height) {
                cout << "1. Walk towards the center of the room.\n";
                cout << "2. Jump and grab the banana directly.\n";
            } else {
                cout << "1. Walk towards the window.\n";
                cout << "2. Push the box towards the banana.\n";
                cout << "3. Climb onto the box.\n";
                cout << "4. Reach out and grasp the banana.\n";
            }
        } else {
            cout << "The monkey cannot reach the banana.\n";
        }
    }
};

int main() {
    MonkeyBananaProblem problem;
    problem.getUserInputs();
    problem.printStepsToBanana();
    return 0;
}
```

**Qn3.** Develop a program to generate a step-by-step solution for the problem of transporting three missionaries and three cannibals across a river using a boat with a maximum capacity of two individuals. The challenge is subject to the constraint that on both banks, the number of missionaries must not be outnumbered by cannibals. Violating this constraint would result in cannibals consuming the missionaries. Furthermore, the boat cannot traverse the river

alone without any passengers. This programming task necessitates the creation of an algorithm capable of systematically determining the sequence of movements that adhere to the specified constraints, showcasing a heightened level of difficulty due to the intricacies involved in managing the compositions of missionaries and cannibals during each crossing.

## Code:

```
#include <bits/stdc++.h>

using namespace std;

struct state {
    int ml, cl, pos, mr, cr;
    state *parent;
    int op;
    bool operator==(const state & rhs) const{
        return ((ml == rhs.ml) && (cl == rhs.cl) && (mr == rhs.mr) && (cr ==
rhs.cr) && (pos  == rhs.pos));
    }
};

state * nextState(state *Z, const int j) {
  state * S = new state();
  (*S) = (*Z);
  (*S).op  = j;
  switch (j)
  {
    case 0: {  (*S).pos  -= 1;
               (*S).ml += 0;
               (*S).cl += 1;
               (*S).mr -= 0;
               (*S).cr -= 1;}
            break;
    case 1: {  (*S).pos  -= 1;
               (*S).ml += 0;
               (*S).cl += 2;
               (*S).mr -= 0;
               (*S).cr -= 2;}
            break;
    case 2: {  (*S).pos  -= 1;
               (*S).ml += 1;
```

```
            (*S).cl += 0;
            (*S).mr -= 1;
            (*S).cr -= 0;}
         break;
   case 3: {  (*S).pos  -= 1;
            (*S).ml += 2;
            (*S).cl += 0;
            (*S).mr -= 2;
            (*S).cr -= 0;}
         break;
   case 4: {  (*S).pos  -= 1;
            (*S).ml += 1;
            (*S).cl += 1;
            (*S).mr -= 1;
            (*S).cr -= 1;}
         break;
   case 5: {  (*S).pos  += 1;
            (*S).mr += 0;
            (*S).cr += 1;
            (*S).ml -= 0;
            (*S).cl -= 1;}
         break;
   case 6: {  (*S).pos  += 1;
            (*S).mr += 0;
            (*S).cr += 2;
            (*S).ml -= 0;
            (*S).cl -= 2;}
         break;
   case 7: {  (*S).pos  += 1;
            (*S).mr += 1;
            (*S).cr += 0;
            (*S).ml -= 1;
            (*S).cl -= 0;}
         break;
   case 8: {  (*S).pos  += 1;
            (*S).mr += 2;
            (*S).cr += 0;
            (*S).ml -= 2;
            (*S).cl -= 0;}
         break;
   case 9: {  (*S).pos  += 1;
            (*S).mr += 1;
            (*S).cr += 1;
            (*S).ml -= 1;
            (*S).cl -= 1;}
         break;
}
return S;
```

```
}

bool validState(state *S){
  if(( (*S).cl < 0) || ( (*S).cl > 3)) return false;
  if(( (*S).cr < 0) || ( (*S).cr > 3)) return false;
  if(( (*S).ml < 0) || ( (*S).ml > 3)) return false;
  if(( (*S).mr < 0) || ( (*S).mr > 3)) return false;
  if(( (*S).pos != 0) && ( (*S).pos != 1))  return false;
  if( (( (*S).cl >  (*S).ml) && ( (*S).ml > 0)) || (( (*S).cr >  (*S).mr) && (
(*S).mr > 0)) ) return false;
  return true;
}

bool notFound(state *s, vector<state *> &vis){
    for(int i=0; i<vis.size(); i++){
        if((*s) == (*(vis[i]))) return false;
    }
    return true;
}

void addChildren(queue<state *> &q, state *s, vector<state *> &vis){
    state *temp;
    for(int i=0; i<10; i++){
        temp = nextState(s, i);
        if(validState(temp) && notFound(temp, vis)) {
            (*temp).parent = s;
            q.push(temp);
        } else delete temp;
    }
}

void printOP(int n) {
  switch (n)
  {
    case 0:  cout << "C(0,1,0)" << endl; break;
    case 1:  cout << "C(0,2,0)" << endl; break;
    case 2:  cout << "C(1,0,0)" << endl; break;
    case 3:  cout << "C(2,0,0)" << endl; break;
    case 4:  cout << "C(1,1,0)" << endl; break;
    case 5:  cout << "C(0,1,1)" << endl; break;
    case 6:  cout << "C(0,2,1)" << endl; break;
    case 7:  cout << "C(1,0,1)" << endl; break;
    case 8:  cout << "C(2,0,1)" << endl; break;
    case 9:  cout << "C(1,1,1)" << endl; break;
  }
}

int main(){
```

```
    bool fl=false;
    state START = {3,3,0,0,0,NULL,-1};
    state GOAL = {0,0,1,3,3,NULL};

    queue<state *> q;
    vector<state *> vis;

    q.push(&START);

    state* temp;

    while(!q.empty()){
        temp = q.front();
        q.pop();
        if((*temp) == GOAL){
            fl=true;
            break;
        } else{
            addChildren(q, temp, vis);
            vis.push_back(temp);
        }
    }

    if(fl){
        cout << "Path found!\n";
        cout << "C(missionary, cannibal, position of boat)\n";

        stack<int> ops;
        for(state* i=temp; i!=NULL; i = (*i).parent){
            ops.push((*i).op);
        }

        while(!ops.empty()){
            printOP(ops.top());
            ops.pop();
        }
    } else{
        cout << "Path not found!\n";
    }

    return 0;
}
```

## Output:

```
Sem_AI_Assignment\Assignment_4> cd "c:\Users\Gourav Kumar Shaw\Desktop\8th_Sem_AI_Assign
ment\Assignment_4\" ; if ($?) { g++ Qn3.cpp -o Qn3 } ; if ($?) { .\Qn3 }
Path found!
C(missionary, cannibal, position of boat)
C(0,2,1)
C(0,1,0)
C(0,2,1)
C(0,1,0)
C(2,0,1)
C(1,1,0)
C(2,0,1)
C(0,1,0)
C(0,2,1)
C(0,1,0)
C(0,2,1)
```

**Qn4.** Create a program to systematically illustrate the steps for a farmer to safely transport a tiger, a goat, and a cabbage across a river using a small boat. The boat can only carry one belonging at a time, and the farmer faces the challenge of ensuring that, at no point during the crossings, the tiger is left alone with the goat or the goat is left alone with the cabbage. Otherwise, the tiger would eat the goat, or the goat would eat the cabbage. The program must generate a sequence of movements that adhere to these constraints, demonstrating a heightened level of difficulty due to the complex interactions between the farmer, the tiger, the goat, and the cabbage during each crossing.

**Code:**

```cpp
#include <bits/stdc++.h>

using namespace std;

struct state {
    int tl, tr, cl, cr, gl, gr, pos;
    state *parent;
    int op;
    bool operator==(const state & rhs) const{
```

```
        return ((tl == rhs.tl) && (tr == rhs.tr) && (cl == rhs.cl) && (cr ==
rhs.cr) && (pos  == rhs.pos) && (gl == rhs.gl) && (gr == rhs.gr));
    }
};

state * nextState(state *Z, const int j) {
  state * S = new state();
  (*S) = (*Z);
  (*S).op  = j;
  switch (j)
  {
    case 0: {  (*S).pos  -= 1;
               (*S).tl += 1;
               (*S).tr -= 1;}
            break;
    case 1: {  (*S).pos  -= 1;
               (*S).cl += 1;
               (*S).cr -= 1;}
            break;
    case 2: {  (*S).pos  -= 1;
               (*S).gl += 1;
               (*S).gr -= 1;}
            break;
    case 3: {  (*S).pos  -= 1;}
            break;
    case 4: {  (*S).pos  += 1;
               (*S).tr += 1;
               (*S).tl -= 1;}
            break;
    case 5: {  (*S).pos  += 1;
               (*S).cr += 1;
               (*S).cl -= 1;}
            break;
    case 6: {  (*S).pos  += 1;
               (*S).gr += 1;
               (*S).gl -= 1;}
            break;
    case 7: {  (*S).pos  += 1;}
            break;
  }
  return S;
}

bool validState(state *S){
  if(( (*S).cl < 0) || ( (*S).cl > 1)) return false;
  if(( (*S).cr < 0) || ( (*S).cr > 1)) return false;
  if(( (*S).gl < 0) || ( (*S).gl > 1)) return false;
  if(( (*S).gr < 0) || ( (*S).gr > 1)) return false;
```

```
  if(( (*S).tl < 0) || ( (*S).tl > 1)) return false;
  if(( (*S).tr < 0) || ( (*S).tr > 1)) return false;
  if(( (*S).pos != 0) && ( (*S).pos != 1))  return false;
  if(( (*S).tl == 1 && (*S).gl == 1 && (*S).pos == 1)) return false;
  if(( (*S).cl == 1 && (*S).gl == 1 && (*S).pos == 1)) return false;
  if(( (*S).tr == 1 && (*S).gr == 1 && (*S).pos == 0)) return false;
  if(( (*S).cr == 1 && (*S).gr == 1 && (*S).pos == 0)) return false;

  return true;
}

bool notFound(state *s, vector<state *> &vis){
    for(int i=0; i<vis.size(); i++){
        if((*s) == (*(vis[i]))) return false;
    }
    return true;
}

void addChildren(queue<state *> &q, state *s, vector<state *> &vis){
    // cout << "S\n";
    // cout << (*s).tl << " " << (*s).gl << " " << (*s).cl << " " << (*s).pos
<< "\n";
    state *temp;
    for(int i=0; i<8; i++){
        temp = nextState(s, i);
        // cout << (*temp).tl << " " << (*temp).gl << " " << (*temp).cl << " "
<< (*temp).pos << "\n";
        if(validState(temp) && notFound(temp, vis)) {
            (*temp).parent = s;
            q.push(temp);
        } else delete temp;
    }
}

void printOP(int n) {
  switch (n)
  {
    // tiger, goat, cabbage, boat
    case 0:  cout << "C(1,0,0,0)" << endl; break;
    case 1:  cout << "C(0,0,1,0)" << endl; break;
    case 2:  cout << "C(0,1,0,0)" << endl; break;
    case 3:  cout << "C(0,0,0,0)" << endl; break;
    case 4:  cout << "C(1,0,0,1)" << endl; break;
    case 5:  cout << "C(0,0,1,1)" << endl; break;
    case 6:  cout << "C(0,1,0,1)" << endl; break;
    case 7:  cout << "C(0,0,0,1)" << endl; break;
  }
}
```

```
int main(){

    bool fl=false;
    state START = {1,0,1,0,1,0,0,NULL,-1};
    state GOAL = {0,1,0,1,0,1,1,NULL};

    queue<state *> q;
    vector<state *> vis;

    q.push(&START);

    state* temp;

    while(!q.empty()){
        temp = q.front();
        q.pop();
        if((*temp) == GOAL){
            fl=true;
            break;
        } else{
            addChildren(q, temp, vis);
            vis.push_back(temp);
        }
    }

    if(fl){
        cout << "Path found!\n";
        cout << "C(tiger, goat, cabbage, position of boat)\n";

        stack<int> ops;
        for(state* i=temp; i!=NULL; i = (*i).parent){
            ops.push((*i).op);
        }

        while(!ops.empty()){
            printOP(ops.top());
            ops.pop();
        }
    } else{
        cout << "Path not found!\n";
    }

    return 0;
}
```

## Output:

```
PS C:\Users\Gourav Kumar Shaw\Desktop\8th_Sem_AI_Assignment\Assignment_4> cd "c:\Users\G
ourav Kumar Shaw\Desktop\8th_Sem_AI_Assignment\Assignment_4\" ; if ($?) { g++ Qn4.cpp -o
 Qn4 } ; if ($?) { .\Qn4 }
Path found!
C(tiger, goat, cabbage, position of boat)
C(0,1,0,1)
C(0,0,0,0)
C(1,0,0,1)
C(0,1,0,0)
C(0,0,1,1)
C(0,0,0,0)
C(0,1,0,1)
```

**Qn5.** Develop a comprehensive program that effectively addresses a puzzle problem. The puzzle involves a 3x3 grid with eight numbered tiles and an empty space, initially arranged randomly. The task is to create a program that can systematically rearrange the tiles from the initial configuration to a predefined goal state, usually in ascending order from left to right, adhering to the constraints of permissible moves.

## Code:

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

using namespace std;

// Define the goal state
vector<vector<int>> goalState = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 0}
};

// Define structure for puzzle node
struct PuzzleNode {
    vector<vector<int>> state;
    int heuristic;
```

```
    int moves;
    PuzzleNode* parent;

    // Constructor
    PuzzleNode(vector<vector<int>> s, int h, int m, PuzzleNode* p) : state(s),
heuristic(h), moves(m), parent(p) {}

    // Compare function for priority queue
    bool operator>(const PuzzleNode& other) const {
        return heuristic + moves > other.heuristic + other.moves;
    }

    // Compare function for set
    bool operator<(const PuzzleNode& other) const {
        return state < other.state;
    }
};

// Function to calculate the heuristic (number of incorrect pieces)
int calculateHeuristic(const vector<vector<int>>& state) {
    int heuristic = 0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (state[i][j] != goalState[i][j] && state[i][j] != 0) {
                heuristic++;
            }
        }
    }
    return heuristic;
}

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

// Function to generate possible moves
vector<vector<vector<int>>> generateMoves(const vector<vector<int>>& state) {
    vector<vector<vector<int>>> moves;
    int zeroX, zeroY;

    // Find the position of the empty tile (0)
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (state[i][j] == 0) {
                zeroX = i;
                zeroY = j;
                break;
```

```
            }
        }
    }

    // Generate moves by swapping empty tile with neighboring tiles
    if (zeroX > 0) {
        vector<vector<int>> newState = state;
        swap(newState[zeroX][zeroY], newState[zeroX - 1][zeroY]);
        moves.push_back(newState);
    }
    if (zeroX < 2) {
        vector<vector<int>> newState = state;
        swap(newState[zeroX][zeroY], newState[zeroX + 1][zeroY]);
        moves.push_back(newState);
    }
    if (zeroY > 0) {
        vector<vector<int>> newState = state;
        swap(newState[zeroX][zeroY], newState[zeroX][zeroY - 1]);
        moves.push_back(newState);
    }
    if (zeroY < 2) {
        vector<vector<int>> newState = state;
        swap(newState[zeroX][zeroY], newState[zeroX][zeroY + 1]);
        moves.push_back(newState);
    }

    return moves;
}

// Function to print the state of the puzzle
void printState(const vector<vector<int>>& state) {
    for (const auto& row : state) {
        for (int value : row) {
            cout << value << " ";
        }
        cout << endl;
    }
}

// Function to trace back the solution path
void printSolutionPath(PuzzleNode* node) {
    vector<PuzzleNode*> path;
    while (node != nullptr) {
        path.push_back(node);
        node = node->parent;
    }
    reverse(path.begin(), path.end());
```

```
    for (PuzzleNode* n : path) {
        printState(n->state);
        cout << endl;
    }
}

// A* algorithm implementation
void solvePuzzle(const vector<vector<int>>& initialState) {
    priority_queue<PuzzleNode*, vector<PuzzleNode*>, greater<PuzzleNode*>>
frontier;
    set<vector<vector<int>>> explored;

    PuzzleNode* initialNode = new PuzzleNode(initialState,
calculateHeuristic(initialState), 0, nullptr);
    frontier.push(initialNode);

    while (!frontier.empty()) {
        PuzzleNode *currentNode = frontier.top();
        frontier.pop();

        if (isGoalState(currentNode->state)) {
            cout << "Solution found in " << currentNode->moves << " moves." <<
endl;
            printSolutionPath(currentNode);
            return;
        }

        explored.insert(currentNode->state);

        // Generate possible moves
        vector<vector<vector<int>>> possibleMoves = generateMoves(currentNode-
>state);
        for (const auto& move : possibleMoves) {
            if (explored.find(move) == explored.end()) {
                PuzzleNode* newNode = new PuzzleNode(move,
calculateHeuristic(move), currentNode->moves + 1, currentNode);
                frontier.push(newNode);
            }
        }
    }

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

int main() {
    vector<vector<int>> initialState = {
        {1, 2, 3},
        {4, 0, 6},
```

```
        {7, 5, 8}
    };

    cout << "Enter initial state in the form of a 3x3 matrix.\n";

    for(int i=0; i<3; i++){
        for(int j=0; j<3; j++){
            cin >> initialState[i][j];
        }
    }

    cout << "Enter goal state in the form of a 3x3 matrix.\n";

    for(int i=0; i<3; i++){
        for(int j=0; j<3; j++){
            cin >> goalState[i][j];
        }
    }

    cout << "Initial state:" << endl;
    printState(initialState);
    cout << endl;

    solvePuzzle(initialState);

    return 0;
}
```