AI LAB EXAM

Non AI Game:

#include <stdio.h>

char board[3][3];

char currentPlayer = 'X';

void initializeBoard() {

    for (int i = 0; i < 3; i++)

        for (int j = 0; j < 3; j++)

            board[i][j] = ' ';

}

void printBoard() {

    for (int i = 0; i < 3; i++) {

        printf(" %c | %c | %c \n", board[i][0], board[i][1], board[i][2]);

        if (i < 2) printf("---|---|---\n");

    }

}

int checkWin() {

    for (int i = 0; i < 3; i++) {

        if ((board[i][0] == board[i][1] && board[i][1] == board[i][2] && board[i][0] != ' ') ||

            (board[0][i] == board[1][i] && board[1][i] == board[2][i] && board[0][i] != ' '))

            return 1;

    }

    return ((board[0][0] == board[1][1] && board[1][1] == board[2][2] && board[0][0] != ' ') ||

            (board[0][2] == board[1][1] && board[1][1] == board[2][0] && board[0][2] != ' '));

}

int isMovesLeft() {

    for (int i = 0; i < 3; i++)

        for (int j = 0; j < 3; j++)

            if (board[i][j] == ' ')

                return 1;

    return 0;

}

void computerMove() {

    for (int i = 0; i < 3; i++) {

        for (int j = 0; j < 3; j++) {

            if (board[i][j] == ' ') {

                board[i][j] = 'O';

                if (checkWin()) return;

                board[i][j] = ' ';

            }

        }

    }

    for (int i = 0; i < 3; i++) {

        for (int j = 0; j < 3; j++) {

            if (board[i][j] == ' ') {

                board[i][j] = 'X';

                if (checkWin()) {

                    board[i][j] = 'O';

                    return;

                }

                board[i][j] = ' ';

            }

        }

    }

    for (int i = 0; i < 3; i++)

        for (int j = 0; j < 3; j++)

            if (board[i][j] == ' ') {

                board[i][j] = 'O';

                return;

            }

}

void makeMove() {

    int row, col;

    if (currentPlayer == 'X') {

        printf("Enter row and column (0, 1, or 2) for player %c: ", currentPlayer);

        scanf("%d %d", &row, &col);

        if (row >= 0 && row < 3 && col >= 0 && col < 3 && board[row][col] == ' ')

            board[row][col] = currentPlayer;

        else {

            printf("Invalid move. Try again.\n");

            makeMove();

        }

        currentPlayer = 'O';

    } else {

        computerMove();

        currentPlayer = 'X';

    }

}

int main() {

    initializeBoard();

    for (int moves = 0; moves < 9 && !checkWin(); moves++) {

        printBoard();

        makeMove();

    }

    printBoard();

    if (checkWin())

        printf("Player %c wins!\n", (currentPlayer == 'X') ? 'O' : 'X');

    else

        printf("It's a draw!\n");

    return 0;

}

AI GAME:

#include <stdio.h>

#include <limits.h>

char board[3][3];

char player = 'X', computer = 'O';

void initializeBoard() {

    for (int i = 0; i < 3; i++)

        for (int j = 0; j < 3; j++)

            board[i][j] = ' ';

}

void printBoard() {

    for (int i = 0; i < 3; i++) {

        for (int j = 0; j < 3; j++) {

            printf(" %c ", board[i][j]);

            if (j < 2) printf("|");

        }

        printf("\n");

        if (i < 2) printf("---|---|---\n");

    }

}

int checkWin(char mark) {

    for (int i = 0; i < 3; i++)

        if ((board[i][0] == mark && board[i][1] == mark && board[i][2] == mark) ||

            (board[0][i] == mark && board[1][i] == mark && board[2][i] == mark))

            return 1;

    return (board[0][0] == mark && board[1][1] == mark && board[2][2] == mark) ||

           (board[0][2] == mark && board[1][1] == mark && board[2][0] == mark);

}

int minimax(int depth, int isMax) {

    if (checkWin(computer)) return 10 - depth;

    if (checkWin(player)) return depth - 10;

    if (depth == 9) return 0;

    int best = isMax ? INT\_MIN : INT\_MAX, score;

    for (int i = 0; i < 3; i++)

        for (int j = 0; j < 3; j++)

            if (board[i][j] == ' ') {

                board[i][j] = isMax ? computer : player;

                score = minimax(depth + 1, !isMax);

                board[i][j] = ' ';

                best = isMax ? (score > best ? score : best) : (score < best ? score : best);

            }

    return best;

}

void findBestMove() {

    int bestVal = INT\_MIN, row = -1, col = -1, score;

    for (int i = 0; i < 3; i++)

        for (int j = 0; j < 3; j++)

            if (board[i][j] == ' ') {

                board[i][j] = computer;

                score = minimax(0, 0);

                board[i][j] = ' ';

                if (score > bestVal) {

                    bestVal = score;

                    row = i;

                    col = j;

                }

            }

    board[row][col] = computer;

}

void playerMove() {

    int row, col;

    printf("Enter row and column (0, 1, or 2) for player %c: ", player);

    scanf("%d %d", &row, &col);

    if (row >= 0 && row < 3 && col >= 0 && col < 3 && board[row][col] == ' ')

        board[row][col] = player;

    else {

        printf("Invalid move. Try again.\n");

        playerMove();

    }

}

int main() {

    initializeBoard();

    for (int moves = 0; moves < 9 && !checkWin(player) && !checkWin(computer); moves++) {

        printBoard();

        if (moves % 2 == 0)

            playerMove();

        else

            findBestMove();

    }

    printBoard();

    if (checkWin(player))

        printf("Player %c wins!\n", player);

    else if (checkWin(computer))

        printf("Player %c wins!\n", computer);

    else

        printf("It's a draw!\n");

    return 0;

}

BFS IMPLEMENTATION:

**Shortest Path in an Unweighted Graph**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct {

    int items[MAX];

    int front, rear;

} Queue;

void initializeQueue(Queue \*q) {

    q->front = -1;

    q->rear = -1;

}

int isEmpty(Queue \*q) {

    return q->rear == -1;

}

void enqueue(Queue \*q, int value) {

    if (q->rear == MAX - 1) return;

    if (q->front == -1) q->front = 0;

    q->items[++q->rear] = value;

}

int dequeue(Queue \*q) {

    int item = q->items[q->front];

    if (q->front >= q->rear) q->front = q->rear = -1;

    else q->front++;

    return item;

}

void bfs(int graph[MAX][MAX], int start, int n) {

    int visited[MAX] = {0};

    Queue q;

    initializeQueue(&q);

    visited[start] = 1;

    enqueue(&q, start);

    while (!isEmpty(&q)) {

        int current = dequeue(&q);

        printf("%d ", current);

        for (int i = 0; i < n; i++) {

            if (graph[current][i] && !visited[i]) {

                visited[i] = 1;

                enqueue(&q, i);

            }

        }

    }

}

int main() {

    int graph[MAX][MAX] = {

        {0, 1, 1, 0, 0, 0},

        {1, 0, 0, 1, 1, 0},

        {1, 0, 0, 0, 1, 0},

        {0, 1, 0, 0, 0, 1},

        {0, 1, 1, 0, 0, 1},

        {0, 0, 0, 1, 1, 0}

    };

    int n = 6;  // Number of nodes

    int start = 0;  // Starting node for BFS

    printf("BFS starting from node %d:\n", start);

    bfs(graph, start, n);

    return 0;

}

SHORTEST PATH IN MAZE(MAZE SOLVER)

#include <stdio.h>

#include <stdlib.h>

#define ROW 5

#define COL 5

#define MAX (ROW \* COL)

typedef struct {

    int x, y;

} Point;

typedef struct {

    Point point;

    int dist;

} QueueNode;

typedef struct {

    QueueNode items[MAX];

    int front, rear;

} Queue;

void initializeQueue(Queue \*q) {

    q->front = q->rear = -1;

}

int isEmpty(Queue \*q) {

    return q->rear == -1;

}

void enqueue(Queue \*q, QueueNode value) {

    if (q->rear == MAX - 1) return;

    q->items[++q->rear] = value;

    if (q->front == -1) q->front = 0;

}

QueueNode dequeue(Queue \*q) {

    QueueNode item = q->items[q->front];

    if (q->front >= q->rear) q->front = q->rear = -1;

    else q->front++;

    return item;

}

int isValid(int row, int col) {

    return (row >= 0) && (row < ROW) && (col >= 0) && (col < COL);

}

int bfs(int maze[ROW][COL], Point start, Point end) {

    int visited[ROW][COL] = {0};

    visited[start.x][start.y] = 1;

    Queue q;

    initializeQueue(&q);

    enqueue(&q, (QueueNode){{start.x, start.y}, 0});

    while (!isEmpty(&q)) {

        QueueNode curr = dequeue(&q);

        Point pt = curr.point;

        if (pt.x == end.x && pt.y == end.y) return curr.dist;

        int rowNum[] = {-1, 0, 0, 1};

        int colNum[] = {0, -1, 1, 0};

        for (int i = 0; i < 4; i++) {

            int row = pt.x + rowNum[i];

            int col = pt.y + colNum[i];

            if (isValid(row, col) && maze[row][col] && !visited[row][col]) {

                visited[row][col] = 1;

                enqueue(&q, (QueueNode){{row, col}, curr.dist + 1});

            }

        }

    }

    return -1;

}

int main() {

    int maze[ROW][COL] = {

        {1, 0, 1, 1, 1},

        {1, 1, 1, 0, 1},

        {0, 1, 0, 1, 1},

        {1, 1, 1, 0, 1},

        {1, 0, 1, 1, 1}

    };

    Point start = {0, 0};

    Point end = {4, 4};

    int dist = bfs(maze, start, end);

    printf(dist != -1 ? "Shortest Path is %d\n" : "No path exists\n", dist);

    return 0;

}

VACCUUM WORLD USING BFS

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#define ROOMS 2

typedef struct {

    int room; // Room number

    bool cleaned; // Flag to indicate if the room is cleaned

} State;

typedef struct {

    State state;

    int moves; // Number of moves to reach this state

} Node;

typedef struct {

    Node items[1000];

    int front, rear;

} Queue;

void initializeQueue(Queue \*q) {

    q->front = q->rear = -1;

}

int isEmpty(Queue \*q) {

    return q->rear == -1;

}

void enqueue(Queue \*q, Node value) {

    if (q->rear == 999) return;

    if (q->front == -1) q->front = 0;

    q->items[++q->rear] = value;

}

Node dequeue(Queue \*q) {

    Node item = q->items[q->front];

    if (q->front >= q->rear) q->front = q->rear = -1;

    else q->front++;

    return item;

}

int bfs(State initial\_state) {

    Queue q;

    initializeQueue(&q);

    enqueue(&q, (Node){initial\_state, 0});

    while (!isEmpty(&q)) {

        Node current\_node = dequeue(&q);

        State current\_state = current\_node.state;

        if (!current\_state.cleaned) {

            // Clean the room

            printf("Clean Room %d\n", current\_state.room);

            current\_state.cleaned = true;

        }

        // Move left

        State left\_state = current\_state;

        left\_state.room = (current\_state.room - 1 + ROOMS) % ROOMS;

        if (!left\_state.cleaned) {

            printf("Move left to Room %d\n", left\_state.room);

            enqueue(&q, (Node){left\_state, current\_node.moves + 1});

        }

        // Move right

        State right\_state = current\_state;

        right\_state.room = (current\_state.room + 1) % ROOMS;

        if (!right\_state.cleaned) {

            printf("Move right to Room %d\n", right\_state.room);

            enqueue(&q, (Node){right\_state, current\_node.moves + 1});

        }

    }

    return -1; // No need to return distance in this problem

}

int main() {

    State initial\_state = {0, false}; // Starting in Room 0, not cleaned

    bfs(initial\_state);

    return 0;

}

DFS : NQUEEN

#include <stdio.h>

#include <stdbool.h>

#define N 4

void print\_board(int board[N][N]) {

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            printf("%d ", board[i][j]);

        }

        printf("\n");

    }

    printf("\n");

}

bool is\_safe(int board[N][N], 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])

            return false;

    }

    // Check upper left diagonal

    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {

        if (board[i][j])

            return false;

    }

    // Check upper right diagonal

    for (int i = row, j = col; i >= 0 && j < N; i--, j++) {

        if (board[i][j])

            return false;

    }

    return true;

}

bool solve\_n\_queens\_util(int board[N][N], int row) {

    if (row == N) {

        print\_board(board);

        return true;

    }

    bool res = false;

    for (int col = 0; col < N; col++) {

        if (is\_safe(board, row, col)) {

            board[row][col] = 1;

            res = solve\_n\_queens\_util(board, row + 1) || res;

            board[row][col] = 0;

        }

    }

    return res;

}

void solve\_n\_queens() {

    int board[N][N] = {0};

    if (!solve\_n\_queens\_util(board, 0)) {

        printf("No solution exists\n");

    }

}

int main() {

    solve\_n\_queens();

    return 0;

}

DFS : VACCUUM WORLD

#include <stdio.h>

#include <stdbool.h>

#define ROOMS 2

void dfs\_vacuum\_world(int room, bool cleaned[]) {

    // Clean the current room

    cleaned[room] = true;

    printf("Cleaned room %d\n", room);

    // Move left and clean if not already cleaned

    int left\_room = (room - 1 + ROOMS) % ROOMS;

    if (!cleaned[left\_room]) {

        printf("Move left to room %d\n", left\_room);

        dfs\_vacuum\_world(left\_room, cleaned);

    }

    // Move right and clean if not already cleaned

    int right\_room = (room + 1) % ROOMS;

    if (!cleaned[right\_room]) {

        printf("Move right to room %d\n", right\_room);

        dfs\_vacuum\_world(right\_room, cleaned);

    }

}

void solve\_vacuum\_world() {

    bool cleaned[ROOMS] = {false}; // Initialize all rooms as not cleaned

    int start\_room = 0; // Starting room

    dfs\_vacuum\_world(start\_room, cleaned);

}

int main() {

    solve\_vacuum\_world();

    return 0;

}

CSP: N QUEEN

#include <stdio.h>

#include <stdbool.h>

#define N 8

// Function to print the solution (board configuration)

void print\_solution(int board[N][N]) {

    static int solution\_count = 0;

    printf("Solution %d:\n", ++solution\_count);

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            printf("%d ", board[i][j]);

        }

        printf("\n");

    }

    printf("\n");

}

// Function to check if a queen can be placed at board[row][col]

bool is\_safe(int board[N][N], int row, int col) {

    int i, j;

    // Check this row on the left side

    for (i = 0; i < col; i++) {

        if (board[row][i]) {

            return false;

        }

    }

    // Check upper diagonal on left side

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {

        if (board[i][j]) {

            return false;

        }

    }

    // Check lower diagonal on left side

    for (i = row, j = col; j >= 0 && i < N; i++, j--) {

        if (board[i][j]) {

            return false;

        }

    }

    return true;

}

// Recursive function to solve N-Queens problem using backtracking

void solve\_n\_queens(int board[N][N], int col) {

    // Base case: If all queens are placed then print the solution

    if (col >= N) {

        print\_solution(board);

        return;

    }

    // Consider this column and try placing this queen in all rows one by one

    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on board[i][col]

        if (is\_safe(board, i, col)) {

            // Place this queen in board[i][col]

            board[i][col] = 1;

            // Recur to place the rest of the queens

            solve\_n\_queens(board, col + 1);

            // Backtrack and remove the queen to explore other possibilities

            board[i][col] = 0;

        }

    }

}

int main() {

    int board[N][N] = {0}; // Initialize all cells to 0

    solve\_n\_queens(board, 0); // Solve the N-Queens problem to find all solutions

    return 0;

}

HILL CLIMBING: N QUEEN

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#include <time.h>

#define N 8

// Function to print the board configuration

void print\_board(int board[N]) {

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            if (board[i] == j) {

                printf("Q ");

            } else {

                printf(". ");

            }

        }

        printf("\n");

    }

    printf("\n");

}

// Function to calculate the number of conflicts (queens attacking each other)

int calculate\_conflicts(int board[N]) {

    int conflicts = 0;

    for (int i = 0; i < N; i++) {

        for (int j = i + 1; j < N; j++) {

            if (board[i] == board[j] || abs(board[i] - board[j]) == j - i) {

                conflicts++;

            }

        }

    }

    return conflicts;

}

// Function to generate a random board configuration

void generate\_random\_board(int board[N]) {

    srand(time(NULL));

    for (int i = 0; i < N; i++) {

        board[i] = rand() % N;

    }

}

// Function to perform hill climbing to solve the N-Queens problem

void hill\_climbing(int board[N]) {

    int current\_conflicts = calculate\_conflicts(board);

    int moves = 0;

    while (current\_conflicts > 0) {

        // Find the column with the least conflicts if moved

        int min\_conflicts = current\_conflicts;

        int best\_move[N];

        for (int i = 0; i < N; i++) {

            int original\_row = board[i];

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

                board[i] = j;

                int new\_conflicts = calculate\_conflicts(board);

                if (new\_conflicts < min\_conflicts) {

                    min\_conflicts = new\_conflicts;

                    for (int k = 0; k < N; k++) {

                        best\_move[k] = board[k];

                    }

                }

            }

            board[i] = original\_row;

        }

        if (min\_conflicts >= current\_conflicts) {

            // Local minimum reached

            printf("Local minimum reached.\n");

            break;

        }

        for (int i = 0; i < N; i++) {

            board[i] = best\_move[i];

        }

        current\_conflicts = min\_conflicts;

        printf("Move %d:\n", ++moves);

        print\_board(board);

    }

    printf("Solution found in %d moves.\n", moves);

}

int main() {

    int board[N];

    generate\_random\_board(board);

    printf("Initial board:\n");

    print\_board(board);

    hill\_climbing(board);

    return 0;

}

A\* ALGORITHM:

PROLOG:

% Facts

male(john).

male(peter).

female(lisa).

female(emma).

parent(john, lisa).

parent(john, peter).

parent(lisa, emma).

% Rules

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

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

grandparent(X, Z) :- parent(X, Y), parent(Y, Z).

% Queries

% Is John a father of Lisa?

% father(john, lisa).

% Who are the grandparents of Emma?

% grandparent(X, emma).

**Minimal BFS Implementation for Grid Traversal**

#include <stdio.h>

#include <stdbool.h>

#define ROWS 3

#define COLS 3

// Directions for moving in the grid: right, down, left, up

int directionRow[] = {0, 1, 0, -1};

int directionCol[] = {1, 0, -1, 0};

// Queue implementation using a simple array

typedef struct {

    int row, col;

} Point;

void bfs\_grid(int grid[ROWS][COLS], int startRow, int startCol) {

    bool visited[ROWS][COLS] = {false};

    Point queue[ROWS \* COLS];

    int front = 0, rear = 0;

    // Enqueue starting point

    queue[rear++] = (Point){startRow, startCol};

    visited[startRow][startCol] = true;

    while (front < rear) {

        Point p = queue[front++];

        int row = p.row, col = p.col;

        printf("Visited cell (%d, %d)\n", row, col);

        // Explore neighbors

        for (int i = 0; i < 4; i++) {

            int newRow = row + directionRow[i];

            int newCol = col + directionCol[i];

            if (newRow >= 0 && newRow < ROWS && newCol >= 0 && newCol < COLS &&

                grid[newRow][newCol] == 1 && !visited[newRow][newCol]) {

                queue[rear++] = (Point){newRow, newCol};

                visited[newRow][newCol] = true;

            }

        }

    }

}

int main() {

    int grid[ROWS][COLS] = {

        {1, 1, 0},

        {0, 1, 1},

        {1, 0, 1}

    };

    bfs\_grid(grid, 0, 0);

    return 0;

}

### BFS Implementation for the Water Jug Problem

#include <stdio.h>

#include <stdbool.h>

typedef struct {

    int jug1, jug2;

} State;

#define MAX\_CAPACITY 100

#define ENQUEUE(s) (queue[rear++] = (s))

#define DEQUEUE() (queue[front++])

#define IS\_EMPTY() (front == rear)

void bfs(int jug1\_capacity, int jug2\_capacity, int target) {

    bool visited[MAX\_CAPACITY][MAX\_CAPACITY] = {false};

    State queue[MAX\_CAPACITY \* MAX\_CAPACITY];

    int front = 0, rear = 0;

    State start = {0, 0};

    ENQUEUE(start);

    visited[0][0] = true;

    while (!IS\_EMPTY()) {

        State current = DEQUEUE();

        if (current.jug1 == target || current.jug2 == target) {

            printf("Solution found: (%d, %d)\n", current.jug1, current.jug2);

            return;

        }

        State next\_states[] = {

            {jug1\_capacity, current.jug2}, {current.jug1, jug2\_capacity},

            {0, current.jug2}, {current.jug1, 0},

            {current.jug1 - (jug2\_capacity - current.jug2 > current.jug1 ? current.jug1 : jug2\_capacity - current.jug2),

             current.jug2 + (jug2\_capacity - current.jug2 > current.jug1 ? current.jug1 : jug2\_capacity - current.jug2)},

            {current.jug1 + (jug1\_capacity - current.jug1 > current.jug2 ? current.jug2 : jug1\_capacity - current.jug1),

             current.jug2 - (jug1\_capacity - current.jug1 > current.jug2 ? current.jug2 : jug1\_capacity - current.jug1)}

        };

        for (int i = 0; i < 6; i++) {

            if (!visited[next\_states[i].jug1][next\_states[i].jug2]) {

                ENQUEUE(next\_states[i]);

                visited[next\_states[i].jug1][next\_states[i].jug2] = true;

            }

        }

    }

    printf("No solution found.\n");

}

int main() {

    int jug1\_capacity = 4, jug2\_capacity = 3, target = 2;

    bfs(jug1\_capacity, jug2\_capacity, target);

    return 0;

}

**% Symptoms**

**symptom(feel\_fatigue).**

**symptom(cough).**

**symptom(fever).**

**symptom(body\_aches).**

**symptom(headache).**

**% Diseases and their associated symptoms**

**disease(flue, [cough, fever, body\_aches, headache]).**

**disease(cold, [cough, feel\_fatigue]).**

**% Rule to check if a person has a particular disease**

**has\_disease(Symptoms, Disease) :-**

**disease(Disease, DiseaseSymptoms),**

**subset(DiseaseSymptoms, Symptoms).**

**% Rule to provide a diagnosis based on symptoms**

**diagnose(Symptoms, Diagnosis) :-**

**has\_disease(Symptoms, Disease),**

**Diagnosis = 'You may have ' + Disease + '.'.**

**% Example usage:**

**% ?- diagnose([cough, fever], Diagnosis).**

**% Output: "You may have flue."**