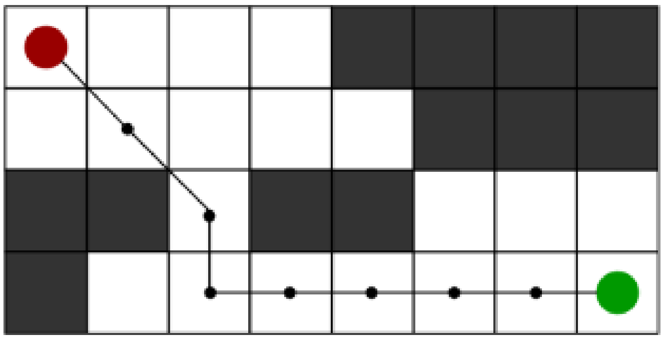
Name: Gourav Kumar Shaw
Enrollment No. : 2020CSB010
Subject : Artificial Intelligence Laboratory.
Semester: 8th

Assignment - 5

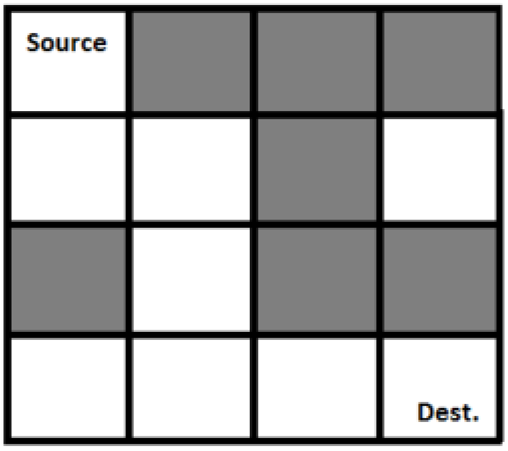
Qn1. In a spatial context defined by a square grid featuring numerous obstacles, a task is presented wherein a starting cell and a target cell are specified. The objective is to efficiently traverse from the starting cell to the target cell, optimizing for expeditious navigation. In this scenario, the A* Search algorithm proves instrumental. The A* Search algorithm operates by meticulously selecting nodes within the grid, employing a parameter denoted as 'f.' This parameter, critical to the decision-making process, is the summation of two distinct parameters – 'g' and 'h.' At each iterative step, the algorithm strategically identifies the node with the lowest 'f' value and progresses the exploration accordingly. The parameters 'g' and 'h' are delineated as follows: - 'g': Represents the cumulative movement cost incurred in traversing the path from the designated starting point to the current square on the grid, factoring in the path generated for that journey. - 'h': Constitutes the estimated movement cost anticipated for the traversal from the current square on the grid to the specified destination. This element, often denoted as the heuristic, embodies an intelligent estimation. The true distance remains unknown until the path is discovered, given potential obstacles like walls or bodies of water. The A* Search algorithm, distinguished by its ability to efficiently find optimal or near- optimal paths amidst obstacles, holds significant applicability in diverse domains such as robotics, gaming, and route planning.



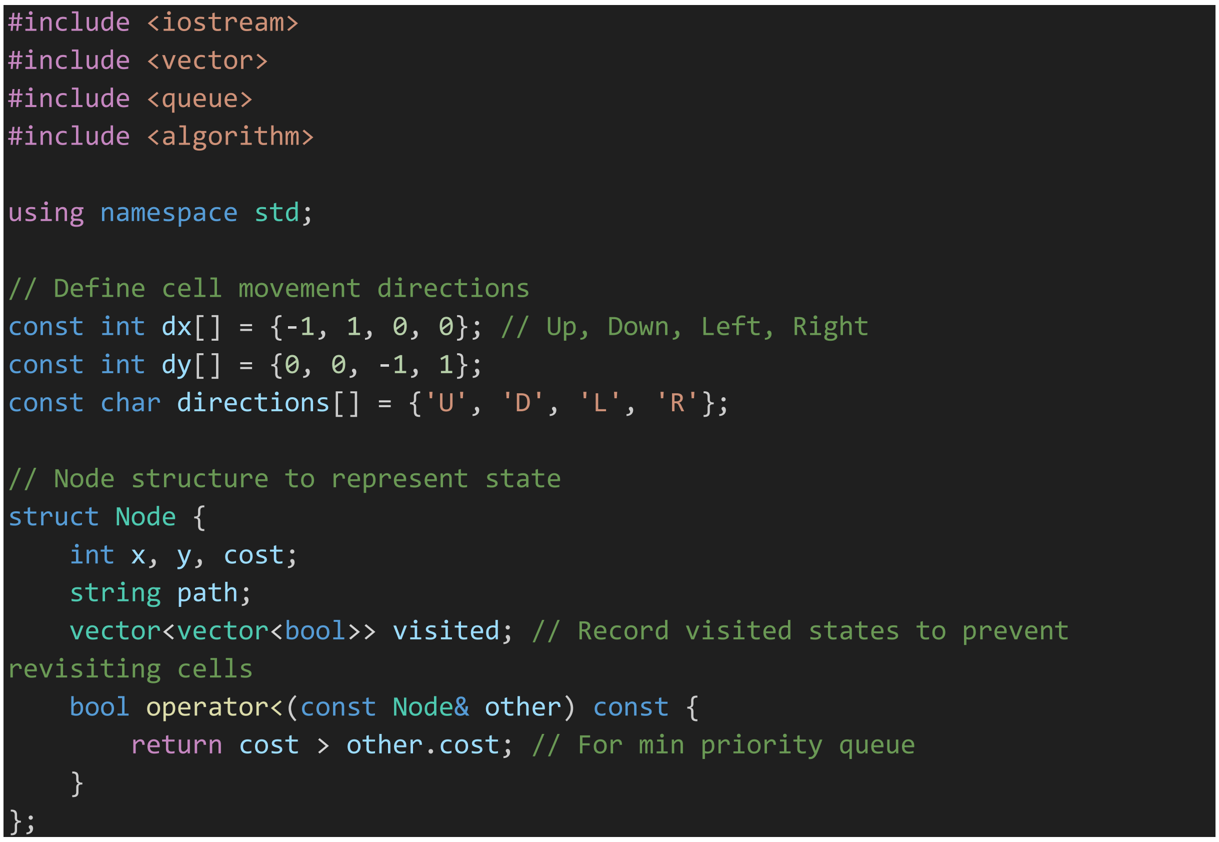
Code:

```cpp
#include <bits/stdc++.h>
#include <vector>
#include <queue>
#include <cmath>
#include <functional> // Include functional for std::function

using namespace std;

// Define a structure to represent a cell in the grid
struct Cell {
    int x, y; // Coordinates of the cell
    int f, g, h; // A* parameters
    Cell* parent; // Pointer to the parent cell
};

// Function to calculate the Manhattan distance heuristic
int heuristic(Cell* a, Cell* b) {
    return abs(a->x - b->x) + abs(a->y - b->y);
}

// Function to check if a cell is within the grid and is traversable
bool isValid(int x, int y, int rows, int cols, vector<vector<int>>& grid) {
    return (x >= 0 && x < rows && y >= 0 && y < cols && grid[x][y] == 0);
}

// Function to implement A* search algorithm
vector<Cell*> astar(vector<vector<int>>& grid, Cell* start, Cell* end) {
    // Define directions: up, down, left, right
    int dx[] = {-1, 1, 0, 0, -1, -1, 1, 1};
    int dy[] = {0, 0, -1, 1, -1, 1, -1, 1};

    int rows = grid.size();
    int cols = grid[0].size();

    // Create a priority queue for open cells
    priority_queue<Cell*, vector<Cell*>, function<bool(Cell*, Cell*)>>
openList([](Cell* a, Cell* b) { return a->f > b->f; });

    // Initialize start cell parameters
    start->g = 0;
    start->h = heuristic(start, end);
    start->f = start->g + start->h;
    openList.push(start);

    // Create a 2D vector to track visited cells
    vector<vector<bool>> visited(rows, vector<bool>(cols, false));
    visited[start->x][start->y] = true;

    // A* search algorithm
    while (!openList.empty()) {
        Cell* current = openList.top();
        openList.pop();

        // Check if the current cell is the destination
        if (current->x == end->x && current->y == end->y) {
            vector<Cell*> path;
            while (current != nullptr) {
                path.push_back(current);
                current = current->parent;
            }
            reverse(path.begin(), path.end());
            return path;
        }

        // Explore neighbors
        for (int i = 0; i < 8; ++i) {
            int nextX = current->x + dx[i];
            int nextY = current->y + dy[i];

            // Check if the neighbor is valid
            if (isValid(nextX, nextY, rows, cols, grid) &&
!visited[nextX][nextY]) {
                Cell* neighbor = new Cell();
                neighbor->x = nextX;
                neighbor->y = nextY;
                neighbor->parent = current;
                neighbor->g = current->g + 1; // Assuming each move has a cost
of 1
                neighbor->h = heuristic(neighbor, end);
                neighbor->f = neighbor->g + neighbor->h;
                openList.push(neighbor);
                visited[nextX][nextY] = true;
            }
        }
    }

    // No path found
    return {};
}

int main() {
    int rows, cols;
    // Get user input for grid dimensions
    cout << "Enter number of rows and columns: ";
    cin >> rows >> cols;

    // Initialize grid with dimensions provided by user
    vector<vector<int>> grid(rows, vector<int>(cols));

    // Get user input for grid
    cout << "Enter grid (0 for traversable, 1 for obstacle):\n";
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            cin >> grid[i][j];
        }
    }

    // Get user input for start and end points
    int startX, startY, endX, endY;
    cout << "Enter start point (row column): ";
    cin >> startX >> startY;

    cout << "Enter end point (row column): ";
    cin >> endX >> endY;

    // Define start and end cells
    Cell* start = new Cell();
    start->x = startX;
    start->y = startY;

    Cell* end = new Cell();
    end->x = endX;
    end->y = endY;

    // Perform A* search
    vector<Cell*> path = astar(grid, start, end);

    // Output the path
    if (path.empty()) {
        cout << "No path found." << endl;
    } else {
        cout << "Path found:" << endl;
        for (auto cell : path) {
            cout << "(" << cell->x << ", " << cell->y << ")" << endl;
        }
    }

    return 0;
}
```

<u>Qn2.</u> In a spatial context defined by a square matrix of order N * N, a rat is situated at the starting point (0,0), aiming to reach the destination at (N-1, N-1). The task at hand is to enumerate all feasible paths that the rat can undertake to traverse from the source to the destination. The permissible directions for the rat's movement are denoted as 'U' (up), 'D' (down), 'L' (left), and 'R' (right). Within this matrix, a cell assigned the value 0 signifies an obstruction, rendering it impassable for the rat, while a value of 1 indicates a traversable cell. The objective is to furnish a list of paths in lexicographically increasing order, with the constraint that no cell can be revisited along the path. Moreover, if the source cell is assigned a value of 0, the rat is precluded from moving to any other cell. To accomplish this, the AO* Search algorithm is employed to systematically explore and evaluate all conceivable paths from source to destination. The algorithm dynamically adapts its heuristic function during the search, optimizing the exploration process. The resultant list of paths reflects a meticulous exploration of the matrix, ensuring lexicographical order and adherence to the specified constraints.

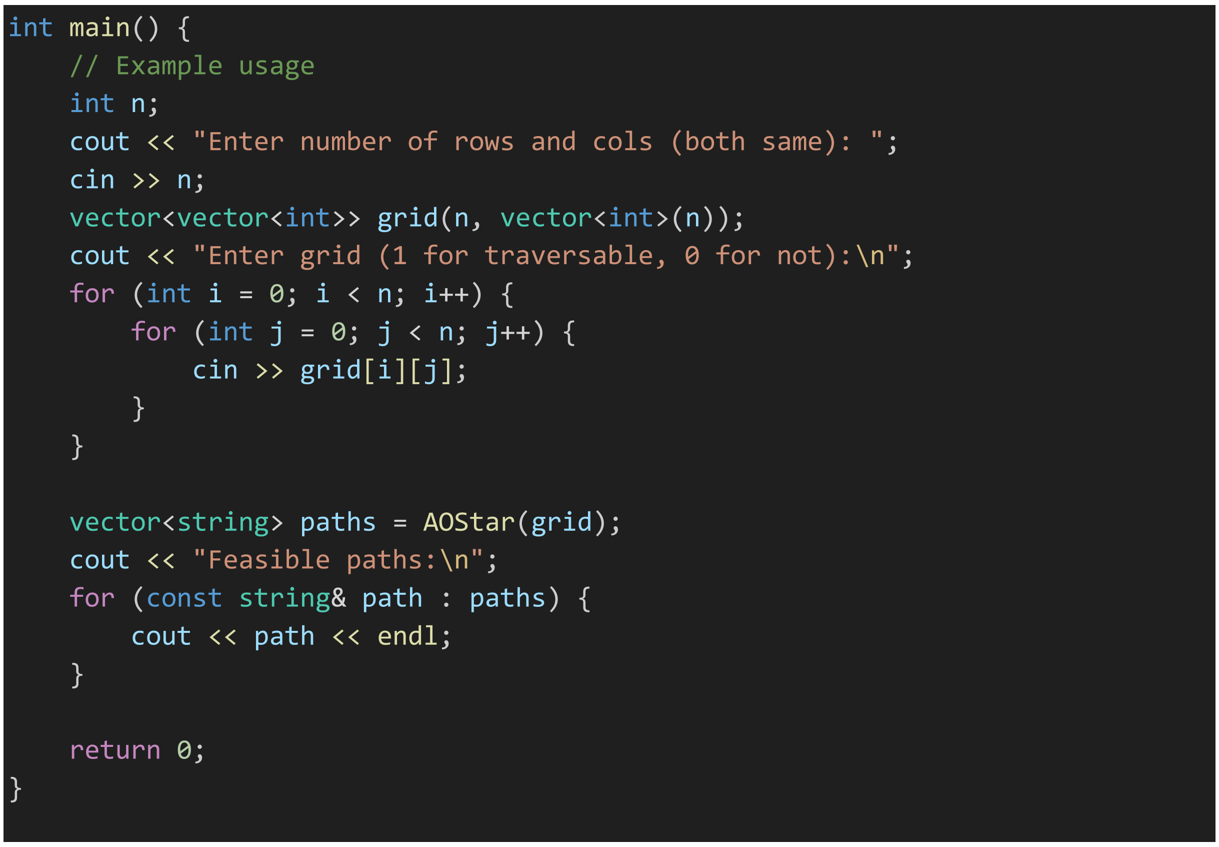


Code:



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

using namespace std;

// Define cell movement directions
const int dx[] = {-1, 1, 0, 0}; // Up, Down, Left, Right
const int dy[] = {0, 0, -1, 1};
const char directions[] = {'U', 'D', 'L', 'R'};

// Node structure to represent state
struct Node {
    int x, y, cost;
    string path;
    vector<vector<bool>> visited; // Record visited states to prevent
revisiting cells
    bool operator<(const Node& other) const {
        return cost > other.cost; // For min priority queue
    }
};
```

```cpp
// Function to check if the cell is within the grid and not an obstacle
bool isValid(int x, int y, int n, const vector<vector<int>>& grid, const
vector<vector<bool>>& visited) {
    return x >= 0 && x < n && y >= 0 && y < n && grid[x][y] != 0 &&
!visited[x][y];
}

// AO* search function
vector<string> AOStar(const vector<vector<int>>& grid) {
    int n = grid.size();
    vector<vector<int>> heuristic(n, vector<int>(n, 0)); // Since we're
exploring all paths, heuristic is not used
    priority_queue<Node> pq;
    vector<string> paths;

    pq.push({0, 0, 0, "", vector<vector<bool>>(n, vector<bool>(n, false))});

    while (!pq.empty()) {
        Node cur = pq.top();
        pq.pop();

        // Check if current cell is the destination
        if (cur.x == n - 1 && cur.y == n - 1) {
            paths.push_back(cur.path);
            continue;
        }

        // Mark the current cell as visited
        cur.visited[cur.x][cur.y] = true;

        for (int i = 0; i < 4; ++i) {
            int nx = cur.x + dx[i];
            int ny = cur.y + dy[i];

            if (isValid(nx, ny, n, grid, cur.visited)) {
                int new_cost = cur.cost + 1; // Cost of moving to adjacent
cell

                // Push new state to priority queue
                pq.push({nx, ny, new_cost, cur.path + directions[i],
cur.visited});
            }
        }
    }
    return paths;
}
```



```cpp
int main() {
    // Example usage
    int n;
    cout << "Enter number of rows and cols (both same): ";
    cin >> n;
    vector<vector<int>> grid(n, vector<int>(n));
    cout << "Enter grid (1 for traversable, 0 for not):\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> grid[i][j];
        }
    }

    vector<string> paths = AOStar(grid);
    cout << "Feasible paths:\n";
    for (const string& path : paths) {
        cout << path << endl;
    }

    return 0;
}
```

Qn3. Connect-4 is a strategic two-player game where participants choose a disc color and take turns dropping their colored discs into a seven-column, six-row grid. Victory is achieved by forming a line of four discs horizontally, vertically, or diagonally. Several winning strategies enhance gameplay:

a. Middle Column Placement:

The player initiating the game benefits from placing the first disc in the middl column. This strategic move maximizes the possibilities for vertical, diagonal, and horizontal connections, totaling five potential ways to win.

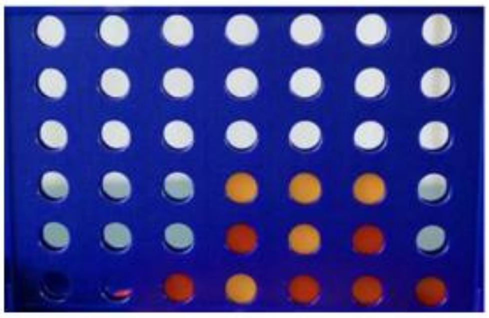
b. Trapping Opponents:

To prevent losses, players strategically block their opponent's potential winning paths. For instance, placing a disc adjacent to an opponent's three-disc line disrupts their progression and protects the player from falling into traps set by the opponent.

c. "7" Formation:

Employing a "7" trap involves arranging discs to resemble the shape of a 7 on the board. This strategic move, which can be configured in various orientations, provides players with multiple directions to achieve a connect-four, adding versatility to their gameplay. Connect-4 Implementation using Mini-Max Algorithm:

In this scenario, a user engages in a game against the computer, and the Mini-Max algorithm is employed to generate game states. Mini-Max, a backtracking algorithm widely used in decision-making and game theory, determines the optimal move for a player under the assumption that the opponent also plays optimally. Two players, the maximizer and the minimizer, aim to achieve the highest and lowest scores, respectively. A heuristic function calculates the values associated with each board state, representing the advantage of one player over the other. Connect-4 Implementation using Alpha-Beta Pruning: To optimize the Mini-Max algorithm, the Alpha-Beta Pruning technique is applied. Alpha-Beta Pruning involves passing two additional parameters, alpha and beta, to the Mini-Max function, reducing the number of evaluated nodes in the game tree. By introducing these parameters, the algorithm searches more efficiently, reaching greater depths in the game tree. Alpha-Beta Pruning accelerates the search process by eliminating the need to evaluate unnecessary branches when a superior move has been identified, resulting in significant computational time savings.

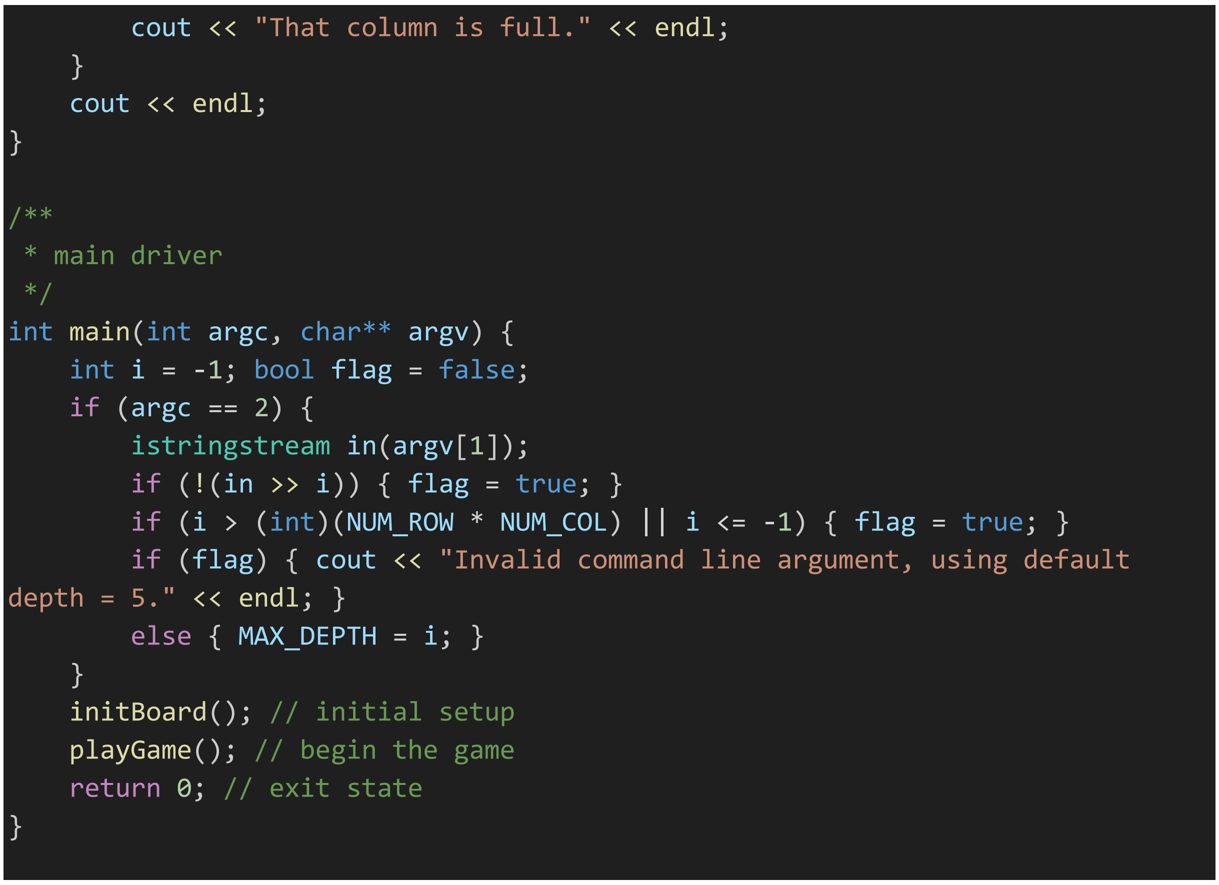


Code:

```cpp
#include <stdio.h>
#include <iostream>
#include <vector>
#include <limits.h>
#include <array>
#include <sstream>

#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))

using namespace std;

// function declarations
void printBoard(vector<vector<int> >&);
int userMove();
void makeMove(vector<vector<int> >&, int, unsigned int);
void errorMessage(int);
int aiMove();
vector<vector<int> > copyBoard(vector<vector<int> >);
bool winningMove(vector<vector<int> >&, unsigned int);
int scoreSet(vector<unsigned int>, unsigned int);
int tabScore(vector<vector<int> >, unsigned int);
array<int, 2> miniMax(vector<vector<int> >&, unsigned int, int, int, unsigned int);
int heurFunction(unsigned int, unsigned int, unsigned int);

// I'll be real and say this is just to avoid magic numbers
unsigned int NUM_COL = 7; // how wide is the board
unsigned int NUM_ROW = 6; // how tall
unsigned int PLAYER = 1; // player number
unsigned int COMPUTER = 2; // AI number
unsigned int MAX_DEPTH = 5; // the default "difficulty" of the computer controlled AI

bool gameOver = false; // flag for if game is over
unsigned int turns = 0; // count for # turns
unsigned int currentPlayer = PLAYER; // current player

vector<vector<int>> board(NUM_ROW, vector<int>(NUM_COL)); // the game board

/**
 * game playing function
 * loops between players while they take turns
 */
void playGame() {
    printBoard(board); // print initial board
    while (!gameOver) { // while no game over state
        if (currentPlayer == COMPUTER) { // AI move
            makeMove(board, aiMove(), COMPUTER);
        }
        else if (currentPlayer == PLAYER) { // player move
            makeMove(board, userMove(), PLAYER);
        }
        else if (turns == NUM_ROW * NUM_COL) { // if max number of turns
reached
            gameOver = true;
        }
        gameOver = winningMove(board, currentPlayer); // check if player won
        currentPlayer = (currentPlayer == 1) ? 2 : 1; // switch player
        turns++; // iterate number of turns
        cout << endl;
        printBoard(board); // print board after successful move
    }
    if (turns == NUM_ROW * NUM_COL) { // if draw condition
        cout << "Draw!" << endl;
    }
    else { // otherwise, someone won
        cout << ((currentPlayer == PLAYER) ? "AI Wins!" : "Player Wins!") <<
endl;
    }
}

/**
 * function that makes the move for the player
 * @param b - the board to make move on
 * @param c - column to drop piece into
 * @param p - the current player
 */
void makeMove(vector<vector<int> >& b, int c, unsigned int p) {
    // start from bottom of board going up
    for (unsigned int r = 0; r < NUM_ROW; r++) {
        if (b[r][c] == 0) { // first available spot
            b[r][c] = p; // set piece
            break;
        }
    }
}

/**
 * prompts the user for their move
 * and ensures valid user input
 * @return the user chosen column
 */
int userMove() {
    int move = -1; // temporary
    while (true) { // repeat until proper input given
        cout << "Enter a column: ";
        cin >> move; // init move as input
        if (!cin) { // if non-integer
            cin.clear();
            cin.ignore(INT_MAX, '\n');
            errorMessage(1); // let user know
        }
        else if (!((unsigned int)move < NUM_COL && move >= 0)) { // if out of
bounds
            errorMessage(2); // let user know
        }
        else if (board[NUM_ROW - 1][move] != 0) { // if full column
            errorMessage(3); // let user know
        }
        else { // if it gets here, input valid
            break;
        }
        cout << endl << endl;
    }
    return move;
}

/**
 * AI "think" algorithm
 * uses minimax to find ideal move
 * @return - the column number for best move
 */
int aiMove() {
    cout << "AI is thinking about a move..." << endl;
    return miniMax(board, MAX_DEPTH, 0 - INT_MAX, INT_MAX, COMPUTER)[1];
}

/**
 * Minimax implementation using alpha-beta pruning
 * @param b - the board to perform MM on
 * @param d - the current depth
 * @param alf - alpha
 * @param bet - beta
 * @param p - current player
 */
array<int, 2> miniMax(vector<vector<int> > &b, unsigned int d, int alf, int
bet, unsigned int p) {
    /**
     * if we've reached minimal depth allowed by the program
     * we need to stop, so force it to return current values
     * since a move will never (theoretically) get this deep,
     * the column doesn't matter (-1) but we're more interested
     * in the score
     *
     * as well, we need to take into consideration how many moves
     * ie when the board is full
     */
    if (d == 0 || d >= (NUM_COL * NUM_ROW) - turns) {
        // get current score to return
        return array<int, 2>{tabScore(b, COMPUTER), -1};
    }
    if (p == COMPUTER) { // if AI player
        array<int, 2> moveSoFar = {INT_MIN, -1}; // since maximizing, we want
lowest possible value
        if (winningMove(b, PLAYER)) { // if player about to win
            return moveSoFar; // force it to say it's worst possible score, so
it knows to avoid move
        } // otherwise, business as usual
        for (unsigned int c = 0; c < NUM_COL; c++) { // for each possible move
            if (b[NUM_ROW - 1][c] == 0) { // but only if that column is non-
full
                vector<vector<int> > newBoard = copyBoard(b); // make a copy
of the board
                makeMove(newBoard, c, p); // try the move
                int score = miniMax(newBoard, d - 1, alf, bet, PLAYER)[0]; //
find move based on that new board state
                if (score > moveSoFar[0]) { // if better score, replace it,
and consider that best move (for now)
                    moveSoFar = {score, (int)c};
                }
                alf = max(alf, moveSoFar[0]);
                if (alf >= bet) { break; } // for pruning
            }
        }
        return moveSoFar; // return best possible move
    }
    else {
        array<int, 2> moveSoFar = {INT_MAX, -1}; // since PLAYER is minimized,
we want moves that diminish this score
        if (winningMove(b, COMPUTER)) {
            return moveSoFar; // if about to win, report that move as best
        }
        for (unsigned int c = 0; c < NUM_COL; c++) {
            if (b[NUM_ROW - 1][c] == 0) {
                vector<vector<int> > newBoard = copyBoard(b);
                makeMove(newBoard, c, p);
                int score = miniMax(newBoard, d - 1, alf, bet, COMPUTER)[0];
                if (score < moveSoFar[0]) {
                    moveSoFar = {score, (int)c};
                }
                bet = min(bet, moveSoFar[0]);
                if (alf >= bet) { break; }
            }
        }
        return moveSoFar;
    }
}

/**
 * function to tabulate current board "value"
 * @param b - the board to evaluate
 * @param p - the player to check score of
 * @return - the board score
 */
int tabScore(vector<vector<int> > b, unsigned int p) {
    int score = 0;
    vector<unsigned int> rs(NUM_COL);
    vector<unsigned int> cs(NUM_ROW);
    vector<unsigned int> set(4);
    /**
     * horizontal checks, we're looking for sequences of 4
     * containing any combination of AI, PLAYER, and empty pieces
     */
    for (unsigned int r = 0; r < NUM_ROW; r++) {
        for (unsigned int c = 0; c < NUM_COL; c++) {
            rs[c] = b[r][c]; // this is a distinct row alone
        }
        for (unsigned int c = 0; c < NUM_COL - 3; c++) {
            for (int i = 0; i < 4; i++) {
                set[i] = rs[c + i]; // for each possible "set" of 4 spots from
that row
            }
            score += scoreSet(set, p); // find score
        }
    }
    // vertical
    for (unsigned int c = 0; c < NUM_COL; c++) {
        for (unsigned int r = 0; r < NUM_ROW; r++) {
            cs[r] = b[r][c];
        }
        for (unsigned int r = 0; r < NUM_ROW - 3; r++) {
            for (int i = 0; i < 4; i++) {
                set[i] = cs[r + i];
            }
            score += scoreSet(set, p);
        }
    }
    // diagonals
    for (unsigned int r = 0; r < NUM_ROW - 3; r++) {
        for (unsigned int c = 0; c < NUM_COL; c++) {
            rs[c] = b[r][c];
        }
        for (unsigned int c = 0; c < NUM_COL - 3; c++) {
            for (int i = 0; i < 4; i++) {
                set[i] = b[r + i][c + i];
            }
            score += scoreSet(set, p);
        }
    }
    for (unsigned int r = 0; r < NUM_ROW - 3; r++) {
        for (unsigned int c = 0; c < NUM_COL; c++) {
            rs[c] = b[r][c];
        }
        for (unsigned int c = 0; c < NUM_COL - 3; c++) {
            for (int i = 0; i < 4; i++) {
                set[i] = b[r + 3 - i][c + i];
            }
            score += scoreSet(set, p);
        }
    }
    return score;
}

/**
 * function to find the score of a set of 4 spots
 * @param v - the row/column to check
 * @param p - the player to check against
 * @return - the score of the row/column
 */
int scoreSet(vector<unsigned int> v, unsigned int p) {
    unsigned int good = 0; // points in favor of p
    unsigned int bad = 0; // points against p
    unsigned int empty = 0; // neutral spots
    for (unsigned int i = 0; i < v.size(); i++) { // just enumerate how many
of each
        good += (v[i] == p) ? 1 : 0;
        bad += (v[i] == PLAYER || v[i] == COMPUTER) ? 1 : 0;
        empty += (v[i] == 0) ? 1 : 0;
    }
    // bad was calculated as (bad + good), so remove good
    bad -= good;
    return heurFunction(good, bad, empty);
}

/**
```

```
 * my """heuristic function""" is pretty bad, but it seems to work
 * it scores 2s in a row and 3s in a row
 * @param g - good points
 * @param b - bad points
 * @param z - empty spots
 * @return - the score as tabulated
 */
int heurFunction(unsigned int g, unsigned int b, unsigned int z) {
    int score = 0;
    if (g == 4) { score += 500001; } // preference to go for winning move vs.
block
    else if (g == 3 && z == 1) { score += 5000; }
    else if (g == 2 && z == 2) { score += 500; }
    else if (b == 2 && z == 2) { score -= 501; } // preference to block
    else if (b == 3 && z == 1) { score -= 5001; } // preference to block
    else if (b == 4) { score -= 500000; }
    return score;
}

/**
 * function to determine if a winning move is made
 * @param b - the board to check
 * @param p - the player to check against
 * @return - whether that player can have a winning move
 */
bool winningMove(vector<vector<int> > &b, unsigned int p) {
    unsigned int winSequence = 0; // to count adjacent pieces
    // for horizontal checks
    for (unsigned int c = 0; c < NUM_COL - 3; c++) { // for each column
        for (unsigned int r = 0; r < NUM_ROW; r++) { // each row
            for (int i = 0; i < 4; i++) { // recall you need 4 to win
                if ((unsigned int)b[r][c + i] == p) { // if not all pieces
match
                    winSequence++; // add sequence count
                }
                if (winSequence == 4) { return true; } // if 4 in row
            }
            winSequence = 0; // reset counter
        }
    }
    // vertical checks
    for (unsigned int c = 0; c < NUM_COL; c++) {
        for (unsigned int r = 0; r < NUM_ROW - 3; r++) {
            for (int i = 0; i < 4; i++) {
                if ((unsigned int)b[r + i][c] == p) {
                    winSequence++;
                }
                if (winSequence == 4) { return true; }
            }
            winSequence = 0;
        }
    }
    // the below two are diagonal checks
    for (unsigned int c = 0; c < NUM_COL - 3; c++) {
        for (unsigned int r = 3; r < NUM_ROW; r++) {
            for (int i = 0; i < 4; i++) {
                if ((unsigned int)b[r - i][c + i] == p) {
                    winSequence++;
                }
                if (winSequence == 4) { return true; }
            }
            winSequence = 0;
        }
    }
    for (unsigned int c = 0; c < NUM_COL - 3; c++) {
        for (unsigned int r = 0; r < NUM_ROW - 3; r++) {
            for (int i = 0; i < 4; i++) {
                if ((unsigned int)b[r + i][c + i] == p) {
                    winSequence++;
                }
                if (winSequence == 4) { return true; }
            }
            winSequence = 0;
        }
    }
    return false; // otherwise no winning move
}

/**
 * sets up the board to be filled with empty spaces
 * also used to reset the board to this state
 */
void initBoard() {
    for (unsigned int r = 0; r < NUM_ROW; r++) {
        for (unsigned int c = 0; c < NUM_COL; c++) {
            board[r][c] = 0; // make sure board is empty initially
        }
    }
}

/**
 * function to copy board state to another 2D vector
 * ie. make a duplicate board; used for mutating copies rather
 * than the original
 * @param b - the board to copy
 * @return - said copy
```

```cpp
 */
vector<vector<int> > copyBoard(vector<vector<int> > b) {
    vector<vector<int>> newBoard(NUM_ROW, vector<int>(NUM_COL));
    for (unsigned int r = 0; r < NUM_ROW; r++) {
        for (unsigned int c = 0; c < NUM_COL; c++) {
            newBoard[r][c] = b[r][c]; // just straight copy
        }
    }
    return newBoard;
}

/**
 * prints the board to console out
 * @param b - the board to print
 */
void printBoard(vector<vector<int> > &b) {
    for (unsigned int i = 0; i < NUM_COL; i++) {
        cout << " " << i;
    }
    cout << endl << "---------------" << endl;
    for (unsigned int r = 0; r < NUM_ROW; r++) {
        for (unsigned int c = 0; c < NUM_COL; c++) {
            cout << "|";
            switch (b[NUM_ROW - r - 1][c]) {
            case 0: cout << " "; break; // no piece
            case 1: cout << "O"; break; // one player's piece
            case 2: cout << "X"; break; // other player's piece
            }
            if (c + 1 == NUM_COL) { cout << "|"; }
        }
        cout << endl;
    }
    cout << "---------------" << endl;
    cout << endl;
}

/**
 * handler for displaying error messages
 * @param t - the type of error to display
 */
void errorMessage(int t) {
    if (t == 1) { // non-int input
        cout << "Use a value 0.." << NUM_COL - 1 << endl;
    }
    else if (t == 2) { // out of bounds
        cout << "That is not a valid column." << endl;
    }
    else if (t == 3) { // full column
```



```
        cout << "That column is full." << endl;
    }
    cout << endl;
}

/**
 * main driver
 */
int main(int argc, char** argv) {
    int i = -1; bool flag = false;
    if (argc == 2) {
        istringstream in(argv[1]);
        if (!(in >> i)) { flag = true; }
        if (i > (int)(NUM_ROW * NUM_COL) || i <= -1) { flag = true; }
        if (flag) { cout << "Invalid command line argument, using default
depth = 5." << endl; }
        else { MAX_DEPTH = i; }
    }
    initBoard(); // initial setup
    playGame(); // begin the game
    return 0; // exit state
}
```

<u>Qn4.</u> Design a program to facilitate the safe transfer of a family comprising 5 individuals across a river using a boat. The boat has a maximum capacity of carrying 2 people at a time. Each family member has a specific travel time: 1 second, 3 seconds, 6 seconds, 8 seconds, and 12 seconds. Notably, if two people are on the boat simultaneously, the boat will travel at the speed of the slower person. The objective is to transport the entire family across the river within a time constraint of 30 seconds

<u>Code:</u>

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

using namespace std;

int main() {
    // Define travel times for each family member
    vector<int> travelTimes = {1, 3, 6, 8, 12};

    // Initialize variables to track time and family members on each side of
the river
    int timeElapsed = 0;
    vector<int> leftBank = travelTimes;
    vector<int> rightBank;

    // Output initial configuration
    cout << "Initial configuration:\n";
    cout << "Time: " << timeElapsed << "s, Left bank: ";
    for (const auto& person : leftBank) {
        cout << person << " ";
    }
    cout << ", Right bank: ";
    for (const auto& person : rightBank) {
        cout << person << " ";
    }
    cout << endl;

    // Cross 1-second and 3-second members together
    timeElapsed += 3;
    leftBank.erase(find(leftBank.begin(), leftBank.end(), 1));
    leftBank.erase(find(leftBank.begin(), leftBank.end(), 3));
    rightBank.push_back(1);
    rightBank.push_back(3);

    // Output configuration after crossing
    cout << "\nAfter crossing 1-second and 3-second members:\n";
    cout << "Time: " << timeElapsed << "s, Left bank: ";
    for (const auto& person : leftBank) {
        cout << person << " ";
    }
```

```cpp
    cout << ", Right bank: ";
    for (const auto& person : rightBank) {
        cout << person << " ";
    }
    cout << endl;

    // Send 1-second member back
    timeElapsed += 1;
    leftBank.push_back(1);
    rightBank.erase(find(rightBank.begin(), rightBank.end(), 1));

    // Output configuration after sending 1-second member back
    cout << "\nAfter sending 1-second member back:\n";
    cout << "Time: " << timeElapsed << "s, Left bank: ";
    for (const auto& person : leftBank) {
        cout << person << " ";
    }
    cout << ", Right bank: ";
    for (const auto& person : rightBank) {
        cout << person << " ";
    }
    cout << endl;

    // Cross 8-second and 12-second members together
    timeElapsed += 12;
    leftBank.erase(find(leftBank.begin(), leftBank.end(), 8));
    leftBank.erase(find(leftBank.begin(), leftBank.end(), 12));
    rightBank.push_back(8);
    rightBank.push_back(12);

    // Output configuration after crossing 8-second and 12-second members
    cout << "\nAfter crossing 8-second and 12-second members:\n";
    cout << "Time: " << timeElapsed << "s, Left bank: ";
    for (const auto& person : leftBank) {
        cout << person << " ";
    }
    cout << ", Right bank: ";
    for (const auto& person : rightBank) {
        cout << person << " ";
    }
    cout << endl;

    // Send 3-second member back
    timeElapsed += 3;
    leftBank.push_back(3);
    rightBank.erase(find(rightBank.begin(), rightBank.end(), 3));

    // Output configuration after sending 3-second member back
    cout << "\nAfter sending 3-second member back:\n";
    cout << "Time: " << timeElapsed << "s, Left bank: ";
    for (const auto& person : leftBank) {
        cout << person << " ";
    }
    cout << ", Right bank: ";
    for (const auto& person : rightBank) {
        cout << person << " ";
    }
    cout << endl;

    // Cross 1-second and 6-second members together
    timeElapsed += 6;
    leftBank.erase(find(leftBank.begin(), leftBank.end(), 1));
    leftBank.erase(find(leftBank.begin(), leftBank.end(), 6));
    rightBank.push_back(1);
    rightBank.push_back(6);

    // Output configuration after crossing 1-second and 6-second members
    cout << "\nAfter crossing 1-second and 6-second members:\n";
    cout << "Time: " << timeElapsed << "s, Left bank: ";
    for (const auto& person : leftBank) {
        cout << person << " ";
    }
    cout << ", Right bank: ";
    for (const auto& person : rightBank) {
        cout << person << " ";
    }
    cout << endl;

    // Send 1-second member back
    timeElapsed += 1;
    leftBank.push_back(1);
    rightBank.erase(find(rightBank.begin(), rightBank.end(), 1));

    // Output configuration after sending 1-second member back
    cout << "\nAfter sending 1-second member back:\n";
    cout << "Time: " << timeElapsed << "s, Left bank: ";
    for (const auto& person : leftBank) {
        cout << person << " ";
    }
    cout << ", Right bank: ";
    for (const auto& person : rightBank) {
        cout << person << " ";
    }
    cout << endl;

    // Cross 1-second and 3-second members together again
    timeElapsed += 3;
    leftBank.erase(find(leftBank.begin(), leftBank.end(), 1));
    leftBank.erase(find(leftBank.begin(), leftBank.end(), 3));
    rightBank.push_back(1);
    rightBank.push_back(3);

    // Output final configuration
    cout << "\nFinal configuration:\n";
    cout << "Time: " << timeElapsed << "s, Left bank: ";
    for (const auto& person : leftBank) {
        cout << person << " ";
    }
    cout << ", Right bank: ";
    for (const auto& person : rightBank) {
        cout << person << " ";
    }
    cout << endl;

    return 0;
}
```