**Name – Indranil Bain DEPT. – Computer Science & Technology**

**Enrolment No. – 2020CSB039, Assignment - 05**

**Subject – Artificial Intelligence LAB**

**1. 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.**

* #include <bits/stdc++.h>
* using namespace std;
* // Define the possible movements (up, down, left, right, diagonal)
* const vector<pair<int, int>> MOVES = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}, {1, 1}, {1, -1}, {-1, 1}, {-1, -1}};
* // Define a hash function for pair to be used in unordered\_map
* struct pair\_hash {
* template <class T1, class T2>
* std::size\_t operator()(const std::pair<T1, T2> &p) const {
* auto h1 = std::hash<T1>{}(p.first);
* auto h2 = std::hash<T2>{}(p.second);
* return h1 ^ h2;
* }
* };
* int heuristic(pair<int, int> current, pair<int, int> target) {
* // Manhattan distance heuristic
* return abs(current.first - target.first) + abs(current.second - target.second);
* }
* vector<pair<int, int>> astar(vector<vector<int>> &grid, pair<int, int> start, pair<int, int> target) {
* // Initialize the open set as a priority queue
* priority\_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>>, greater<pair<int, pair<int, int>>>> open\_set;
* unordered\_map<pair<int, int>, pair<int, int>, pair\_hash> came\_from;
* unordered\_map<pair<int, int>, int, pair\_hash> g\_score;
* // Initialize the starting node
* open\_set.push({0, start});
* g\_score[start] = 0;
* while (!open\_set.empty()) {
* // Get the node with the lowest f score
* pair<int, int> current = open\_set.top().second;
* open\_set.pop();
* // Check if we've reached the target
* if (current == target) {
* // Reconstruct the path
* vector<pair<int, int>> path;
* while (came\_from.find(current) != came\_from.end()) {
* path.push\_back(current);
* current = came\_from[current];
* }
* path.push\_back(start);
* reverse(path.begin(), path.end());
* return path;
* }
* // Explore neighbors
* for (auto &move : MOVES) {
* pair<int, int> neighbor = {current.first + move.first, current.second + move.second};
* // Skip if the neighbor is out of bounds or an obstacle
* if (neighbor.first < 0 || neighbor.first >= grid.size() || neighbor.second < 0 || neighbor.second >= grid[0].size() || grid[neighbor.first][neighbor.second] == 1) {
* continue;
* }
* // Calculate tentative g score
* int tentative\_g\_score = g\_score[current] + 1;
* // Update g score if this is a better path
* if (g\_score.find(neighbor) == g\_score.end() || tentative\_g\_score < g\_score[neighbor]) {
* came\_from[neighbor] = current;
* g\_score[neighbor] = tentative\_g\_score;
* int f\_score = tentative\_g\_score + heuristic(neighbor, target);
* open\_set.push({f\_score, neighbor});
* }
* }
* }
* // If no path is found
* return {};
* }
* // Example usage
* int main() {
* vector<vector<int>> grid = {
* {0, 0, 0, 0, 0},
* {0, 1, 1, 1, 0},
* {0, 0, 0, 0, 0},
* {0, 1, 1, 1, 0},
* {0, 0, 0, 0, 0}};
* pair<int, int> start = {0, 0};
* pair<int, int> target = {(int)grid.size() - 1, (int)grid[0].size() - 1};
* vector<pair<int, int>> path = astar(grid, start, target);
* cout << "Path:";
* for (auto &point : path) {
* cout << " (" << point.first << "," << point.second << ")";
* }
* cout << endl;
* return 0;
* }

**2. 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.**

* #include <iostream>
* #include <vector>
* #include <queue>
* #include <algorithm>
* using namespace std;
* // Define possible movements (up, down, left, right)
* const vector<pair<int, int>> MOVES = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
* // Node structure for A\* search
* struct Node {
* int x, y; // coordinates
* vector<char> path; // current path
* Node(int \_x, int \_y, const vector<char>& \_path) : x(\_x), y(\_y), path(\_path) {}
* };
* // Heuristic function (Manhattan distance)
* int heuristic(int x, int y, int N) {
* return (N - 1 - x) + (N - 1 - y);
* }
* // Custom comparator for priority queue
* struct CompareNodes {
* bool operator()(const pair<int, Node>& a, const pair<int, Node>& b) {
* return a.first > b.first; // Compare based on heuristic value
* }
* };
* // Function to check if cell is valid and not visited
* bool isValid(int x, int y, int N, const vector<vector<int>>& grid, const vector<vector<bool>>& visited) {
* return x >= 0 && y >= 0 && x < N && y < N && grid[x][y] && !visited[x][y];
* }
* // Function to find all feasible paths from source to destination
* vector<vector<char>> findPaths(const vector<vector<int>>& grid) {
* int N = grid.size();
* vector<vector<bool>> visited(N, vector<bool>(N, false));
* vector<vector<char>> paths;
* // Priority queue for A\* search
* priority\_queue<pair<int, Node>, vector<pair<int, Node>>, CompareNodes> pq;
* pq.push({heuristic(0, 0, N), Node(0, 0, {})});
* // Perform A\* search
* while (!pq.empty()) {
* int x = pq.top().second.x;
* int y = pq.top().second.y;
* vector<char> path = pq.top().second.path;
* pq.pop();
* // Mark current cell as visited
* visited[x][y] = true;
* // Destination reached, add path to result
* if (x == N - 1 && y == N - 1) {
* paths.push\_back(path);
* continue;
* }
* // Explore all possible movements
* for (const auto& move : MOVES) {
* int new\_x = x + move.first;
* int new\_y = y + move.second;
* char direction;
* if (move == make\_pair(-1, 0)) direction = 'U';
* else if (move == make\_pair(1, 0)) direction = 'D';
* else if (move == make\_pair(0, -1)) direction = 'L';
* else direction = 'R';
* // If valid cell and not visited, add to priority queue
* if (isValid(new\_x, new\_y, N, grid, visited)) {
* vector<char> new\_path = path;
* new\_path.push\_back(direction);
* pq.push({heuristic(new\_x, new\_y, N) + new\_path.size(), Node(new\_x, new\_y, new\_path)});
* }
* }
* }
* return paths;
* }
* // Function to print paths
* void printPaths(const vector<vector<char>>& paths) {
* for (const auto& path : paths) {
* for (char c : path) {
* cout << c << " ";
* }
* cout << endl;
* }
* }
* int main() {
* vector<vector<int>> grid = {
* {1, 0, 1},
* {1, 1, 0},
* {1, 1, 1}
* };
* vector<vector<char>> paths = findPaths(grid);
* if (paths.empty()) {
* cout << "No feasible paths exist." << endl;
* } else {
* cout << "Feasible paths:" << endl;
* printPaths(paths);
* }
* return 0;
* }

**3. 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 middle**

**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.**

* #include <climits>
* #include <iostream>
* #include <vector>
* using namespace std;
* const int ROWS = 6;
* const int COLS = 7;
* const int MAX\_DEPTH = 5; // Maximum depth to search in Mini-Max algorithm
* // Function to print the current board
* void printBoard(vector<vector<char>> &board) {
* for (int i = 0; i < ROWS; ++i) {
* for (int j = 0; j < COLS; ++j) {
* cout << board[i][j] << " ";
* }
* cout << endl;
* }
* }
* // Function to check if the move is valid
* bool isValidMove(vector<vector<char>> &board, int col) {
* return col >= 0 && col < COLS && board[0][col] == '-';
* }
* // Function to make a move
* void makeMove(vector<vector<char>> &board, int col, char player) {
* for (int i = ROWS - 1; i >= 0; --i) {
* if (board[i][col] == '-') {
* board[i][col] = player;
* break;
* }
* }
* }
* // Function to check if the board is full
* bool isBoardFull(vector<vector<char>> &board) {
* for (int i = 0; i < ROWS; ++i) {
* for (int j = 0; j < COLS; ++j) {
* if (board[i][j] == '-') {
* return false;
* }
* }
* }
* return true;
* }
* // Function to check if there is a winner
* bool checkWinner(vector<vector<char>> &board, char player) {
* // Check horizontally
* for (int i = 0; i < ROWS; ++i) {
* for (int j = 0; j <= COLS - 4; ++j) {
* if (board[i][j] == player && board[i][j + 1] == player && board[i][j + 2] == player && board[i][j + 3] == player) {
* return true;
* }
* }
* }
* // Check vertically
* for (int j = 0; j < COLS; ++j) {
* for (int i = 0; i <= ROWS - 4; ++i) {
* if (board[i][j] == player && board[i + 1][j] == player && board[i + 2][j] == player && board[i + 3][j] == player) {
* return true;
* }
* }
* }
* // Check diagonally (bottom-left to top-right)
* for (int i = 0; i <= ROWS - 4; ++i) {
* for (int j = 0; j <= COLS - 4; ++j) {
* if (board[i][j] == player && board[i + 1][j + 1] == player && board[i + 2][j + 2] == player && board[i + 3][j + 3] == player) {
* return true;
* }
* }
* }
* // Check diagonally (top-left to bottom-right)
* for (int i = 3; i < ROWS; ++i) {
* for (int j = 0; j <= COLS - 4; ++j) {
* if (board[i][j] == player && board[i - 1][j + 1] == player && board[i - 2][j + 2] == player && board[i - 3][j + 3] == player) {
* return true;
* }
* }
* }
* return false;
* }
* // Function to evaluate the board
* int evaluateBoard(vector<vector<char>> &board) {
* if (checkWinner(board, 'X'))
* return 1000; // Maximizer wins
* else if (checkWinner(board, 'O'))
* return -1000; // Minimizer wins
* else
* return 0; // Draw
* }
* // Mini-Max function with Alpha-Beta pruning
* int miniMax(vector<vector<char>> &board, int depth, bool isMaximizer, int alpha, int beta) {
* int score = evaluateBoard(board);
* if (depth == MAX\_DEPTH || score != 0) {
* return score;
* }
* if (isBoardFull(board))
* return 0; // Draw
* if (isMaximizer) {
* int maxScore = INT\_MIN;
* for (int col = 0; col < COLS; ++col) {
* if (isValidMove(board, col)) {
* makeMove(board, col, 'X');
* int currentScore = miniMax(board, depth + 1, false, alpha, beta);
* maxScore = max(maxScore, currentScore);
* alpha = max(alpha, currentScore);
* board[ROWS - 1][col] = '-'; // Undo the move
* if (beta <= alpha)
* break; // Beta cut-off
* }
* }
* return maxScore;
* } else {
* int minScore = INT\_MAX;
* for (int col = 0; col < COLS; ++col) {
* if (isValidMove(board, col)) {
* makeMove(board, col, 'O');
* int currentScore = miniMax(board, depth + 1, true, alpha, beta);
* minScore = min(minScore, currentScore);
* beta = min(beta, currentScore);
* board[ROWS - 1][col] = '-'; // Undo the move
* if (beta <= alpha)
* break; // Alpha cut-off
* }
* }
* return minScore;
* }
* }
* // Function to find the best move using Mini-Max with Alpha-Beta pruning
* int findBestMove(vector<vector<char>> &board) {
* int bestMove = -1;
* int bestScore = INT\_MIN;
* int alpha = INT\_MIN;
* int beta = INT\_MAX;
* for (int col = 0; col < COLS; ++col) {
* if (isValidMove(board, col)) {
* makeMove(board, col, 'X');
* int currentScore = miniMax(board, 0, false, alpha, beta);
* if (currentScore > bestScore) {
* bestScore = currentScore;
* bestMove = col;
* }
* board[ROWS - 1][col] = '-'; // Undo the move
* }
* }
* return bestMove;
* }
* int main() {
* vector<vector<char>> board(ROWS, vector<char>(COLS, '-'));
* int movesCount = 0;
* while (true) {
* printBoard(board);
* // Player's turn
* int playerMove;
* cout << "Your turn! Enter column number (0-6): ";
* cin >> playerMove;
* if (isValidMove(board, playerMove)) {
* makeMove(board, playerMove, 'O');
* movesCount++;
* if (checkWinner(board, 'O')) {
* cout << "Congratulations! You win!" << endl;
* break;
* }
* if (isBoardFull(board)) {
* cout << "It's a draw!" << endl;
* break;
* }
* } else {
* cout << "Invalid move! Try again." << endl;
* continue;
* }
* // Computer's turn
* if (movesCount < ROWS \* COLS) {
* int computerMove = findBestMove(board);
* makeMove(board, computerMove, 'X');
* cout << "Computer's move: " << computerMove << endl;
* if (checkWinner(board, 'X')) {
* cout << "Computer wins!" << endl;
* break;
* }
* if (isBoardFull(board)) {
* cout << "It's a draw!" << endl;
* break;
* }
* } else {
* cout << "It's a draw!" << endl;
* break;
* }
* }
* printBoard(board);
* return 0;
* }

**4. 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.**

* ([#include <iostream>
* #include <vector>
* #include <algorithm>
* using namespace std;
* // Function to find the minimum time required for a group to cross the river
* int minCrossingTime(vector<int>& times, vector<pair<int, int>>& solution) {
* sort(times.begin(), times.end());
* int n = times.size();
* if (n <= 2) {
* solution.push\_back({times.back(), -1}); // If there are 2 or fewer people, return the time of the slowest person
* return times.back();
* }
* int minTime = 0;
* while (n > 3) {
* // Send the two fastest people across
* int fastestTime = times[0] + times[1];
* int slowestTime = times[1];
* minTime += fastestTime;
* solution.push\_back({times[0], times[1]});
* times.erase(times.begin());
* times.erase(times.begin());
* // Bring back the fastest person
* minTime += slowestTime;
* solution.push\_back({-1, slowestTime});
* times.push\_back(slowestTime);
* sort(times.begin(), times.end());
* n -= 2;
* }
* if (n == 3) {
* minTime += times[0] + times[1] + times[2]; // If there are 3 people left, send them all across together
* solution.push\_back({times[0], times[1]});
* solution.push\_back({times[0], -1});
* solution.push\_back({times[0], times[2]});
* } else if (n == 2) {
* minTime += times[1]; // If there are 2 people left, send them across together
* solution.push\_back({times[0], times[1]});
* } else {
* minTime += times[0]; // If there is only 1 person left, send them across
* }
* return minTime;
* }
* // Function to check if the family can be transported within the time constraint
* bool canTransportFamily(vector<int>& times, int timeConstraint, vector<pair<int, int>>& solution) {
* return minCrossingTime(times, solution) <= timeConstraint;
* }
* int main() {
* vector<int> familyTimes = {1, 3, 6, 8, 12}; // Travel times for each family member
* int timeConstraint = 30; // Time constraint in seconds
* vector<pair<int, int>> solution;
* if (canTransportFamily(familyTimes, timeConstraint, solution)) {
* cout << "The family can be safely transported across the river within the time constraint." << endl;
* cout << "Solution:" << endl;
* for (const auto& step : solution) {
* if (step.first != -1) cout << "Person " << step.first << " crosses the river." << endl;
* if (step.second != -1) cout << "Person " << step.second << " crosses the river." << endl;
* cout << "Boat returns." << endl;
* }
* } else {
* cout << "It's not possible to transport the family across the river within the time constraint." << endl;
* }
* return 0;
* }