**A \* Search Misplaced Tiles**

#include <bits/stdc++.h>

#include <chrono>

using namespace std;

using namespace chrono;

#define MATRIX\_SIZE 3

int moves = 0;

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

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

int targetMatrix[MATRIX\_SIZE][MATRIX\_SIZE] = {

        {0, 1, 2},

        {3, 4, 5},

        {6, 7, 8}

};

struct Tile {

    Tile \*ancestor;

    int layout[MATRIX\_SIZE][MATRIX\_SIZE];

    int x, y;

    int cost;

    int level;

};

int displayMatrix(int layout[MATRIX\_SIZE][MATRIX\_SIZE]) {

    for (int i = 0; i < MATRIX\_SIZE; i++) {

        for (int j = 0; j < MATRIX\_SIZE; j++)

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

        printf("\n");

    }

    return 0;

}

Tile \*createTile(int layout[MATRIX\_SIZE][MATRIX\_SIZE], int x, int y, int newX, int newY, int level, Tile \*ancestor) {

    Tile \*node = new Tile;

    node->ancestor = ancestor;

    for (int i = 0; i < MATRIX\_SIZE; i++) {

        for (int j = 0; j < MATRIX\_SIZE; j++) {

            node->layout[i][j] = layout[i][j];

        }

    }

    int temp = node->layout[x][y];

    node->layout[x][y] = node->layout[newX][newY];

    node->layout[newX][newY] = temp;

    node->cost = INT\_MAX;

    node->level = level;

    node->x = newX;

    node->y = newY;

    return node;

}

int computeCost(int layout[MATRIX\_SIZE][MATRIX\_SIZE], int target[MATRIX\_SIZE][MATRIX\_SIZE]) {

    int count = 0;

    for (int i = 0; i < MATRIX\_SIZE; i++)

        for (int j = 0; j < MATRIX\_SIZE; j++)

            if (layout[i][j] != target[i][j]) {

                count++;

            }

    return count;

}

int checkBounds(int x, int y) {

    return (x >= 0 && x < MATRIX\_SIZE && y >= 0 && y < MATRIX\_SIZE);

}

void showMatrix(Tile \*root) {

    if (root == NULL)

        return;

    showMatrix(root->ancestor);

    displayMatrix(root->layout);

    printf("\n");

}

struct comparison {

    bool operator()(const Tile \*lhs, const Tile \*rhs) const {

        return (lhs->cost + lhs->level) > (rhs->cost + rhs->level);

    }

};

// Function to convert the matrix into a string

std::string stringifyMatrix(int layout[MATRIX\_SIZE][MATRIX\_SIZE]) {

    std::ostringstream oss;

    for (int i = 0; i < MATRIX\_SIZE; i++) {

        for (int j = 0; j < MATRIX\_SIZE; j++) {

            oss << layout[i][j] << " ";

        }

    }

    return oss.str();

}

void solvePuzzle(int initial[MATRIX\_SIZE][MATRIX\_SIZE], int x, int y, int target[MATRIX\_SIZE][MATRIX\_SIZE]) {

    auto start\_time = high\_resolution\_clock::now(); // Record the start time

    priority\_queue<Tile \*, vector<Tile \*>, comparison> pq;

    unordered\_set<string> visitedStates; // Keep track of visited states

    Tile \*root = createTile(initial, x, y, x, y, 0, NULL);

    root->cost = computeCost(initial, target);

    pq.push(root);

    while (!pq.empty()) {

        Tile \*min = pq.top();

        pq.pop();

        if (min->cost == 0) {

            auto end\_time = high\_resolution\_clock::now(); // Record the end time

            auto duration = duration\_cast<milliseconds>(end\_time - start\_time);

            cout << "Solved state reached in " << moves << " moves.\n";

            cout << "Time taken: " << duration.count() << " milliseconds\n";

            showMatrix(min);

            return;

        }

        for (int i = 0; i < 4; i++) {

            if (checkBounds(min->x + rows[i], min->y + columns[i])) {

                Tile \*child = createTile(min->layout, min->x, min->y, min->x + rows[i], min->y + columns[i], min->level + 1, min);

                child->cost = computeCost(child->layout, target);

                if (visitedStates.find(stringifyMatrix(child->layout)) == visitedStates.end()) {

                    pq.push(child);

                    visitedStates.insert(stringifyMatrix(child->layout));

                    moves++;

                }

            }

        }

    }

}

int main() {

    int initial[MATRIX\_SIZE][MATRIX\_SIZE];

    cout << "\n\t\t----------------------------------------------------------------------------\n";

    cout << " Enter the starting state of puzzle in undermentioned form \n";

    cout << "\*\*\* 2 3 1 5 6 0 8 4 7 \*\*\*\n>> ";

    for(int i=0;i<3;i++)

        for(int j=0;j<3;j++)

            cin>>initial[i][j];

    cout << "Starting form of the puzzle is:  \n>> ";

    displayMatrix(initial);

    cout << "\n\t\t----------------------------------------------------------------------------\n";

    cout << "Solving the puzzle, please wait  \n>> ";

    int x = 1, y = 2;

    solvePuzzle(initial, x, y, targetMatrix);

    return 0;

}

**A\* Manhattan**

#include <iostream>

#include <vector>

#include <queue>

#include <unordered\_set>

using namespace std;

struct GameTile {

    vector<vector<int>> layout;

    int totalCost;

    int heuristicValue;

    int movesTaken;

    const GameTile\* prevTile;

    GameTile(const vector<vector<int>>& arrangement, int cost, int heuristic, int moves, const GameTile\* previous)

        : layout(arrangement), totalCost(cost), heuristicValue(heuristic), movesTaken(moves), prevTile(previous) {}

    bool operator==(const GameTile& other) const {

        return layout == other.layout;

    }

};

struct GameTileHash {

    size\_t operator()(const GameTile& tile) const {

        size\_t hash = 0;

        for (const auto& row : tile.layout) {

            for (int num : row) {

                hash ^= hash << 6 ^ hash >> 2 ^ size\_t(num) + 0x9e3779b9 + (hash << 14) + (hash >> 7);

            }

        }

        return hash;

    }

};

struct GameTileComparator {

    bool operator()(const GameTile& a, const GameTile& b) const {

        return a.totalCost + a.heuristicValue > b.totalCost + b.heuristicValue;

    }

};

void displayBoard(const vector<vector<int>>& arrangement) {

    for (const auto& row : arrangement) {

        for (int num : row) {

            cout << num << " ";

        }

        cout << endl;

    }

    cout << endl;

}

pair<int, int> locateNumber(const vector<vector<int>>& arrangement, int number) {

    for (int i = 0; i < 3; ++i) {

        for (int j = 0; j < 3; ++j) {

            if (arrangement[i][j] == number) {

                return {i, j};

            }

        }

    }

    return {-1, -1};

}

int computeManhattan(const GameTile& tile) {

    int distance = 0;

    for (int num = 1; num <= 8; ++num) {

        pair<int, int> currentPos = locateNumber(tile.layout, num);

        pair<int, int> goalPos = {(num - 1) / 3, (num - 1) % 3};

        distance += abs(currentPos.first - goalPos.first) + abs(currentPos.second - goalPos.second);

    }

    return distance;

}

bool isValid(int i, int j) {

    return i >= 0 && i < 3 && j >= 0 && j < 3;

}

vector<GameTile> generateTiles(const GameTile& tile);

void executeAStar(const vector<vector<int>>& initialLayout);

int main() {

    vector<vector<int>> startLayout = {{1, 2, 3}, {0, 4, 6}, {7, 5, 8}};

    cout << "Initial state:" << endl;

    displayBoard(startLayout);

    executeAStar(startLayout);

    return 0;

}

vector<GameTile> generateTiles(const GameTile& tile) {

    vector<GameTile> tiles;

    pair<int, int> emptyPosition = locateNumber(tile.layout, 0);

    const int moves[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    for (const auto& move : moves) {

        int newI = emptyPosition.first + move[0];

        int newJ = emptyPosition.second + move[1];

        if (isValid(newI, newJ)) {

            vector<vector<int>> newLayout = tile.layout;

            swap(newLayout[emptyPosition.first][emptyPosition.second], newLayout[newI][newJ]);

            int newCost = tile.totalCost + 1;

            int newHeuristic = computeManhattan({newLayout, 0, 0, 0, nullptr});

            int newMoves = tile.movesTaken + 1;

            tiles.emplace\_back(newLayout, newCost, newHeuristic, newMoves, &tile);

        }

    }

    return tiles;

}

void executeAStar(const vector<vector<int>>& initialLayout) {

    GameTile initialState{initialLayout, 0, computeManhattan({initialLayout, 0, 0, 0, nullptr}), 0, nullptr};

    priority\_queue<GameTile, vector<GameTile>, GameTileComparator> openTiles;

    unordered\_set<GameTile, GameTileHash> closedTiles;

    openTiles.push(initialState);

    while (!openTiles.empty()) {

        GameTile currentTile = openTiles.top();

        openTiles.pop();

        if (currentTile.heuristicValue == 0) {

            cout << "Goal state reached in " << currentTile.movesTaken << " moves." << endl;

            cout << "Solution:" << endl;

            while (currentTile.prevTile != nullptr) {

                displayBoard(currentTile.layout);

                currentTile = \*currentTile.prevTile;

            }

            displayBoard(initialState.layout);

            return;

        }

        closedTiles.insert(currentTile);

        vector<GameTile> successors = generateTiles(currentTile);

        for (const GameTile& successor : successors) {

            if (closedTiles.find(successor) == closedTiles.end()) {

                openTiles.push(successor);

            }

        }

    }

    cout << "Goal state not reachable." << endl;

}

**Greedy Misplaced Tiles**

#include <bits/stdc++.h>

#include <chrono>

using namespace std;

using namespace chrono;

#define SIZE\_OF\_MATRIX 3

int total\_moves = 0;

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

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

int target\_matrix[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX] = {

    {0, 1, 2},

    {3, 4, 5},

    {6, 7, 8}

};

struct BoardNode {

    BoardNode \*parent;

    int matrix[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX];

    int position\_row, position\_col;

    int cost;

    int level;

    int heuristic;

};

int displayMatrix(int matrix[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX]) {

    for (int i = 0; i < SIZE\_OF\_MATRIX; i++) {

        for (int j = 0; j < SIZE\_OF\_MATRIX; j++)

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

        printf("\n");

    }

    return 0;

}

BoardNode \*generateNode(int matrix[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX], int x, int y, int new\_x, int new\_y, int level, BoardNode \*parent) {

    BoardNode \*node = new BoardNode;

    node->parent = parent;

    for (int i = 0; i < SIZE\_OF\_MATRIX; i++) {

        for (int j = 0; j < SIZE\_OF\_MATRIX; j++) {

            node->matrix[i][j] = matrix[i][j];

        }

    }

    int temp = node->matrix[x][y];

    node->matrix[x][y] = node->matrix[new\_x][new\_y];

    node->matrix[new\_x][new\_y] = temp;

    node->cost = INT\_MAX;

    node->level = level;

    node->position\_row = new\_x;

    node->position\_col = new\_y;

    return node;

}

int calculateCost(int matrix[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX], int goal[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX]) {

    int count = 0;

    for (int i = 0; i < SIZE\_OF\_MATRIX; i++)

        for (int j = 0; j < SIZE\_OF\_MATRIX; j++)

            if (matrix[i][j] != goal[i][j])

                count++;

    return count;

}

int calculateHeuristic(int matrix[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX], int goal[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX]) {

    int heuristic = 0;

    for (int i = 0; i < SIZE\_OF\_MATRIX; i++)

        for (int j = 0; j < SIZE\_OF\_MATRIX; j++)

            if (matrix[i][j] != goal[i][j])

                heuristic++;

    return heuristic;

}

int isWithinBounds(int x, int y) {

    return (x >= 0 && x < SIZE\_OF\_MATRIX && y >= 0 && y < SIZE\_OF\_MATRIX);

}

void showBoard(BoardNode \*root) {

    if (root == NULL)

        return;

    showBoard(root->parent);

    displayMatrix(root->matrix);

    printf("\n");

}

struct CompareBoardNodes {

    bool operator()(const BoardNode \*lhs, const BoardNode \*rhs) const {

        return lhs->heuristic > rhs->heuristic;

    }

};

// Function to convert the puzzle matrix into a string

std::string matrixToString(int matrix[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX]) {

    std::ostringstream oss;

    for (int i = 0; i < SIZE\_OF\_MATRIX; i++) {

        for (int j = 0; j < SIZE\_OF\_MATRIX; j++) {

            oss << matrix[i][j] << " ";

        }

    }

    return oss.str();

}

void solvePuzzle(int initial[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX], int x, int y, int goal[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX]) {

    auto start\_time = high\_resolution\_clock::now();

    priority\_queue<BoardNode \*, vector<BoardNode \*>, CompareBoardNodes> pq;

    unordered\_set<string> visited;

    BoardNode \*root = generateNode(initial, x, y, x, y, 0, NULL);

    root->cost = calculateCost(initial, goal);

    root->heuristic = calculateHeuristic(initial, goal);

    pq.push(root);

    while (!pq.empty()) {

        BoardNode \*min = pq.top();

        pq.pop();

        if (min->cost == 0) {

            auto end\_time = high\_resolution\_clock::now();

            auto duration = duration\_cast<milliseconds>(end\_time - start\_time);

            cout << "Reached the goal state in " << total\_moves << " moves.\n";

            cout << "Time taken: " << duration.count() << " milliseconds\n";

            showBoard(min);

            return;

        }

        for (int i = 0; i < 4; i++) {

            if (isWithinBounds(min->position\_row + rows[i], min->position\_col + columns[i])) {

                BoardNode \*child = generateNode(min->matrix, min->position\_row, min->position\_col, min->position\_row + rows[i], min->position\_col + columns[i], min->level + 1, min);

                child->cost = calculateCost(child->matrix, goal);

                child->heuristic = calculateHeuristic(child->matrix, goal);

                if (visited.find(matrixToString(child->matrix)) == visited.end()) {

                    pq.push(child);

                    visited.insert(matrixToString(child->matrix));

                    total\_moves++;

                }

            }

        }

    }

}

int main() {

    int initial[SIZE\_OF\_MATRIX][SIZE\_OF\_MATRIX];

    cout << "\n\t\t----------------------------------------------------------------------------\n";

    cout << " Enter the initial state of the puzzle in this format \n";

    cout << "\*\*\* 2 3 1 5 6 0 8 4 7 \*\*\*\n>> ";

    for (int i = 0; i < SIZE\_OF\_MATRIX; i++)

        for (int j = 0; j < SIZE\_OF\_MATRIX; j++)

            cin >> initial[i][j];

    cout << "The entered initial puzzle is:  \n>> ";

    displayMatrix(initial);

    cout << "\n\t\t----------------------------------------------------------------------------\n";

    cout << "Solving the Puzzle  \n>> ";

    int start\_row = 1, start\_col = 2;

    solvePuzzle(initial, start\_row, start\_col, target\_matrix);

    return 0;

}

**Greedy Manhattan**

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include <unordered\_set>

using namespace std;

// Size of the grid puzzle

const int GRID\_SIZE = 3;

// Structure to represent a state of the grid puzzle

struct GridState {

    vector<vector<int>> grid;

    int heuristicValue;  // Heuristic value based on Manhattan distance

    // Constructor

    GridState(const vector<vector<int>>& puzzle) : grid(puzzle) {

        heuristicValue = computeHeuristic();

    }

    // Calculate the Manhattan distance heuristic

    int computeHeuristic() const {

        int distance = 0;

        for (int i = 0; i < GRID\_SIZE; ++i) {

            for (int j = 0; j < GRID\_SIZE; ++j) {

                if (grid[i][j] != 0) {

                    int targetRow = (grid[i][j] - 1) / GRID\_SIZE;

                    int targetCol = (grid[i][j] - 1) % GRID\_SIZE;

                    distance += abs(i - targetRow) + abs(j - targetCol);

                }

            }

        }

        return distance;

    }

    // Check if the current state is the goal state

    bool isGoalState() const {

        int targetValue = 0;

        for (int i = 0; i < GRID\_SIZE; ++i) {

            for (int j = 0; j < GRID\_SIZE; ++j) {

                if (grid[i][j] != targetValue) {

                    return false;

                }

                ++targetValue;

            }

        }

        return true;

    }

    // Check if two grid states are equal

    bool operator==(const GridState& other) const {

        return grid == other.grid;

    }

};

// Hash function for GridState (used for unordered\_set)

struct GridStateHash {

    size\_t operator()(const GridState& state) const {

        size\_t hashValue = 0;

        for (const auto& row : state.grid) {

            for (int val : row) {

                hashValue ^= hash<int>()(val) + 0x9e3779b9 + (hashValue << 6) + (hashValue >> 2);

            }

        }

        return hashValue;

    }

};

// Comparison function for priority\_queue

struct CompareGridState {

    bool operator()(const GridState& lhs, const GridState& rhs) const {

        return lhs.heuristicValue > rhs.heuristicValue;

    }

};

// Function to display the grid state

void displayGridState(const GridState& state) {

    for (const auto& row : state.grid) {

        for (int val : row) {

            cout << val << " ";

        }

        cout << endl;

    }

    cout << "Heuristic Value: " << state.heuristicValue << endl;

    cout << "-----------------" << endl;

}

// Function to perform the greedy search

void performGreedySearch(const GridState& initial) {

    priority\_queue<GridState, vector<GridState>, CompareGridState> priorityQueue;

    unordered\_set<GridState, GridStateHash> visited;

    priorityQueue.push(initial);

    while (!priorityQueue.empty()) {

        GridState current = priorityQueue.top();

        priorityQueue.pop();

        if (current.isGoalState()) {

            cout << "Goal state reached!" << endl;

            displayGridState(current);

            return;

        }

        if (visited.find(current) == visited.end()) {

            visited.insert(current);

            displayGridState(current);

            // Generate possible next states (left, right, up, down moves)

            vector<int> moves = {-1, 0, 1};

            for (int dx : moves) {

                for (int dy : moves) {

                    if (abs(dx) + abs(dy) == 1) {

                        int newX = 0, newY = 0;

                        // Find the position of the empty space (0 value)

                        for (int i = 0; i < GRID\_SIZE; ++i) {

                            for (int j = 0; j < GRID\_SIZE; ++j) {

                                if (current.grid[i][j] == 0) {

                                    newX = i + dx;

                                    newY = j + dy;

                                    break;

                                }

                            }

                        }

                        // Check if the new position is within bounds

                        if (newX >= 0 && newX < GRID\_SIZE && newY >= 0 && newY < GRID\_SIZE) {

                            // Create a new state by swapping the empty space and the adjacent tile

                            vector<vector<int>> newGrid = current.grid;

                            swap(newGrid[newX][newY], newGrid[newX - dx][newY - dy]);

                            GridState nextState(newGrid);

                            // Add the new state to the priority queue

                            priorityQueue.push(nextState);

                        }

                    }

                }

            }

        }

    }

    cout << "Goal state not reachable!" << endl;

}

int main() {

    // Initial grid state

    vector<vector<int>> initialGrid = {

        {8, 0, 6},

        {5, 4, 7},

        {2, 3, 1}

    };

    GridState initialGridState(initialGrid);

    cout << "Initial state:" << endl;

    displayGridState(initialGridState);

    cout << "Starting greedy search with Manhattan distance heuristic..." << endl;

    performGreedySearch(initialGridState);

    return 0;

}