**A\* Manhattan**

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include <unordered\_set>

using namespace std;

// Size of the grid

const int GRID\_SIZE = 3;

// Structure to represent a grid state

struct GridState {

vector<vector<int>> layout;

int heuristic; // Manhattan distance heuristic value

int steps; // Steps taken to reach this state

// Constructor

GridState(const vector<vector<int>>& l, int s) : layout(l), steps(s) {

heuristic = computeHeuristic();

}

// Calculate the Manhattan distance heuristic

int computeHeuristic() const {

int totalDistance = 0;

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

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

int value = layout[i][j];

if (value != 0) {

int targetRow = (value - 1) / GRID\_SIZE;

int targetCol = (value - 1) % GRID\_SIZE;

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

}

}

}

return totalDistance;

}

// Check if the current state is the goal state

bool isGoal() const {

int targetValue = 0;

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

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

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

return false;

}

++targetValue;

}

}

return true;

}

// Check if two grid states are equal

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

return layout == other.layout;

}

};

// 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.layout) {

for (int val : row) {

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

}

}

return hashValue;

}

};

// Compare function for priority\_queue

struct CompareGridState {

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

return (lhs.steps + lhs.heuristic) > (rhs.steps + rhs.heuristic);

}

};

// Function to print the grid state

void printGridState(const GridState& state) {

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

for (int val : row) {

cout << val << " ";

}

cout << endl;

}

cout << "Heuristic: " << state.heuristic << " Steps: " << state.steps << endl;

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

}

// Function to perform the A\* search with Manhattan distance heuristic

void aStarSearch(const GridState& initial) {

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

unordered\_set<GridState, GridStateHash> visited;

pq.push(initial);

while (!pq.empty()) {

GridState current = pq.top();

pq.pop();

if (current.isGoal()) {

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

printGridState(current);

return;

}

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

visited.insert(current);

printGridState(current);

// Generate possible next states

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 zero (empty space)

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

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

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

newX = i + dx;

newY = j + dy;

break;

}

}

}

// Check if the new position is within the bounds

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

// Create a new state by swapping the zero and the adjacent tile

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

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

GridState nextState(newLayout, current.steps + 1);

// Add the new state to the priority queue

pq.push(nextState);

}

}

}

}

}

}

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

}

int main() {

// Initial grid state

vector<vector<int>> initialLayout = {

{8, 0, 6},

{5, 4, 7},

{2, 3, 1}

};

GridState initialGridState(initialLayout, 0);

cout << "Initial state:" << endl;

printGridState(initialGridState);

cout << "Starting A\* search with Manhattan distance heuristic..." << endl;

aStarSearch(initialGridState);

return 0;

}

**A\* Misplaced**

#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;

}

**Greedy Manhattan**

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include <unordered\_set>

using namespace std;

// Size of the game board

const int BOARD\_SIZE = 3;

// Structure to represent a state of the game

struct GameState {

vector<vector<int>> tiles;

int heuristicValue; // Manhattan distance heuristic value

// Constructor

GameState(const vector<vector<int>>& t) : tiles(t) {

heuristicValue = calculateHeuristic();

}

// Calculate the Manhattan distance heuristic

int calculateHeuristic() const {

int distance = 0;

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

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

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

int targetRow = (tiles[i][j] - 1) / BOARD\_SIZE;

int targetCol = (tiles[i][j] - 1) % BOARD\_SIZE;

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

}

}

}

return distance;

}

// Check if the current state is the goal state

bool isGoal() const {

int targetValue = 0;

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

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

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

return false;

}

++targetValue;

}

}

return true;

}

// Check if two game states are equal

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

return tiles == other.tiles;

}

};

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

struct GameStateHash {

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

size\_t hashValue = 0;

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

for (int val : row) {

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

}

}

return hashValue;

}

};

// Compare function for priority\_queue

struct CompareGameState {

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

return lhs.heuristicValue > rhs.heuristicValue;

}

};

// Function to print the game state

void printGameState(const GameState& state) {

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

for (int val : row) {

cout << val << " ";

}

cout << endl;

}

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

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

}

// Function to perform the greedy search

void performGreedySearch(const GameState& initial) {

priority\_queue<GameState, vector<GameState>, CompareGameState> pq;

unordered\_set<GameState, GameStateHash> visited;

pq.push(initial);

while (!pq.empty()) {

GameState current = pq.top();

pq.pop();

if (current.isGoal()) {

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

printGameState(current);

return;

}

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

visited.insert(current);

printGameState(current);

// Generate possible next states

// In this implementation, only left, right, up, and down moves are considered

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 zero (empty space)

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

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

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

newX = i + dx;

newY = j + dy;

break;

}

}

}

// Check if the new position is within the bounds

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

// Create a new state by swapping the zero and the adjacent tile

vector<vector<int>> newTiles = current.tiles;

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

GameState nextState(newTiles);

// Add the new state to the priority queue

pq.push(nextState);

}

}

}

}

}

}

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

}

int main() {

// Initial game state

vector<vector<int>> initialTiles = {

{8, 0, 6},

{5, 4, 7},

{2, 3, 1}

};

GameState initialGameState(initialTiles);

cout << "Initial state:" << endl;

printGameState(initialGameState);

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

performGreedySearch(initialGameState);

return 0;

}

**Greedy Misplaced**

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include <unordered\_set>

using namespace std;

// Dimension of the puzzle

const int PuzzleSize = 3;

// Structure representing the puzzle's state

struct GameState {

vector<vector<int>> gameBoard;

int heuristicValue; // Evaluation based on misplaced tiles

// Constructor

GameState(const vector<vector<int>>& board) : gameBoard(board) {

heuristicValue = calculateHeuristic();

}

// Calculating heuristic based on misplaced tiles

int calculateHeuristic() const {

int misplacedTiles = 0;

int target = 0;

for (int i = 0; i < PuzzleSize; ++i) {

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

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

++misplacedTiles;

}

++target;

}

}

return misplacedTiles;

}

// Checking if the current state is the goal state

bool isGoalState() const {

int target = 0;

for (int i = 0; i < PuzzleSize; ++i) {

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

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

return false;

}

++target;

}

}

return true;

}

// Comparison for equality between two game states

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

return gameBoard == other.gameBoard;

}

};

// Hash function for GameState (used in unordered\_set)

struct GameStateHash {

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

size\_t hashValue = 0;

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

for (int val : row) {

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

}

}

return hashValue;

}

};

// Comparison function for priority\_queue

struct CompareGameState {

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

return lhs.heuristicValue > rhs.heuristicValue;

}

};

// Function to display the puzzle's state

void displayGameState(const GameState& state) {

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

for (int val : row) {

cout << val << " ";

}

cout << endl;

}

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

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

}

// Function to perform a greedy search

void performGreedySearch(const GameState& initial) {

priority\_queue<GameState, vector<GameState>, CompareGameState> priorityQueue;

unordered\_set<GameState, GameStateHash> visitedStates;

priorityQueue.push(initial);

while (!priorityQueue.empty()) {

GameState currentState = priorityQueue.top();

priorityQueue.pop();

if (currentState.isGoalState()) {

cout << "Reached the goal state!" << endl;

displayGameState(currentState);

return;

}

if (visitedStates.find(currentState) == visitedStates.end()) {

visitedStates.insert(currentState);

displayGameState(currentState);

// Generating possible next states

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;

// Finding the position of the zero (empty space)

for (int i = 0; i < PuzzleSize; ++i) {

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

if (currentState.gameBoard[i][j] == 0) {

newX = i + dx;

newY = j + dy;

break;

}

}

}

// Checking if the new position is within the bounds

if (newX >= 0 && newX < PuzzleSize && newY >= 0 && newY < PuzzleSize) {

// Creating a new state by swapping the zero and the adjacent tile

vector<vector<int>> newBoard = currentState.gameBoard;

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

GameState nextState(newBoard);

// Adding the new state to the priority queue

priorityQueue.push(nextState);

}

}

}

}

}

}

cout << "The goal state is not reachable!" << endl;

}

int main() {

// Initial state of the puzzle

vector<vector<int>> initialPuzzle = {

{8, 0, 6},

{5, 4, 7},

{2, 3, 1}

};

GameState initialGameState(initialPuzzle);

cout << "Initial state:" << endl;

displayGameState(initialGameState);

cout << "Starting the greedy search..." << endl;

performGreedySearch(initialGameState);

return 0;

}