**A \* Search Misplaced Tiles**

 #include <bits/stdc++.h>

#include <chrono>

using namespace std;

using namespace chrono;

#define MATRIX\_SIZE 3

int totalMoves = 0;

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

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

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

{0, 1, 2},

{3, 4, 5},

{6, 7, 8}

};

struct PuzzleTile {

PuzzleTile \*ancestor;

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

int xPos, yPos;

int cost;

int level;

};

int displayPuzzleMatrix(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;

}

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

PuzzleTile \*node = new PuzzleTile;

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->xPos = newX;

node->yPos = newY;

return node;

}

int calculateCost(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 isWithinBounds(int x, int y) {

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

}

void displayPuzzleState(PuzzleTile \*root) {

if (root == NULL)

return;

displayPuzzleState(root->ancestor);

displayPuzzleMatrix(root->layout);

printf("\n");

}

struct PuzzleComparison {

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

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

}

};

std::string stringifyPuzzleMatrix(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 startTime = high\_resolution\_clock::now();

priority\_queue<PuzzleTile \*, vector<PuzzleTile \*>, PuzzleComparison> puzzleQueue;

unordered\_set<string> visitedStates;

PuzzleTile \*root = createPuzzleTile(initial, x, y, x, y, 0, NULL);

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

puzzleQueue.push(root);

while (!puzzleQueue.empty()) {

PuzzleTile \*minTile = puzzleQueue.top();

puzzleQueue.pop();

if (minTile->cost == 0) {

auto endTime = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(endTime - startTime);

cout << "Puzzle solved in " << totalMoves << " moves.\n";

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

displayPuzzleState(minTile);

return;

}

for (int i = 0; i < 4; i++) {

if (isWithinBounds(minTile->xPos + rowOffsets[i], minTile->yPos + colOffsets[i])) {

PuzzleTile \*childTile = createPuzzleTile(minTile->layout, minTile->xPos, minTile->yPos, minTile->xPos + rowOffsets[i], minTile->yPos + colOffsets[i], minTile->level + 1, minTile);

childTile->cost = calculateCost(childTile->layout, target);

if (visitedStates.find(stringifyPuzzleMatrix(childTile->layout)) == visitedStates.end()) {

puzzleQueue.push(childTile);

visitedStates.insert(stringifyPuzzleMatrix(childTile->layout));

totalMoves++;

}

}

}

}

}

int main() {

int initialPuzzle[MATRIX\_SIZE][MATRIX\_SIZE];

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

cout << " Enter the starting state of the puzzle in the following format: \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 >> initialPuzzle[i][j];

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

displayPuzzleMatrix(initialPuzzle);

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

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

int initialX = 1, initialY = 2;

solvePuzzle(initialPuzzle, initialX, initialY, goalMatrix);

return 0;

}

**A\* Manhattan**

#include <iostream>

#include <vector>

#include <queue>

#include <unordered\_set>

using namespace std;

struct PuzzlePiece {

vector<vector<int>> configuration;

int totalCost;

int heuristicValue;

int movesTaken;

const PuzzlePiece\* previousPiece;

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

: configuration(arrangement), totalCost(cost), heuristicValue(heuristic), movesTaken(moves), previousPiece(previous) {}

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

return configuration == other.configuration;

}

};

struct PuzzlePieceHash {

size\_t operator()(const PuzzlePiece& piece) const {

size\_t hash = 0;

for (const auto& row : piece.configuration) {

for (int num : row) {

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

}

}

return hash;

}

};

struct PuzzlePieceComparator {

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

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

}

};

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

for (const auto& row : arrangement) {

for (int num : row) {

cout << num << " ";

}

cout << endl;

}

cout << endl;

}

pair<int, int> findNumberPosition(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 calculateManhattan(const PuzzlePiece& piece) {

int distance = 0;

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

pair<int, int> currentPos = findNumberPosition(piece.configuration, 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 isPositionValid(int i, int j) {

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

}

vector<PuzzlePiece> generatePuzzlePieces(const PuzzlePiece& piece);

void runAStar(const vector<vector<int>>& initialConfiguration);

int main() {

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

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

displayPuzzle(startConfiguration);

runAStar(startConfiguration);

return 0;

}

vector<PuzzlePiece> generatePuzzlePieces(const PuzzlePiece& piece) {

vector<PuzzlePiece> pieces;

pair<int, int> emptyPosition = findNumberPosition(piece.configuration, 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 (isPositionValid(newI, newJ)) {

vector<vector<int>> newConfiguration = piece.configuration;

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

int newCost = piece.totalCost + 1;

int newHeuristic = calculateManhattan({newConfiguration, 0, 0, 0, nullptr});

int newMoves = piece.movesTaken + 1;

pieces.emplace\_back(newConfiguration, newCost, newHeuristic, newMoves, &piece);

}

}

return pieces;

}

void runAStar(const vector<vector<int>>& initialConfiguration) {

PuzzlePiece initialState{initialConfiguration, 0, calculateManhattan({initialConfiguration, 0, 0, 0, nullptr}), 0, nullptr};

priority\_queue<PuzzlePiece, vector<PuzzlePiece>, PuzzlePieceComparator> openPieces;

unordered\_set<PuzzlePiece, PuzzlePieceHash> closedPieces;

openPieces.push(initialState);

while (!openPieces.empty()) {

PuzzlePiece currentPiece = openPieces.top();

openPieces.pop();

if (currentPiece.heuristicValue == 0) {

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

cout << "Solution:" << endl;

while (currentPiece.previousPiece != nullptr) {

displayPuzzle(currentPiece.configuration);

currentPiece = \*currentPiece.previousPiece;

}

displayPuzzle(initialState.configuration);

return;

}

closedPieces.insert(currentPiece);

vector<PuzzlePiece> successors = generatePuzzlePieces(currentPiece);

for (const PuzzlePiece& successor : successors) {

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

openPieces.push(successor);

}

}

}

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

}

**Greedy Misplaced Tiles**

#include <bits/stdc++.h>

#include <chrono>

using namespace std;

using namespace chrono;

#define MATRIX\_SIZE 3

int totalMoves = 0;

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

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

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

{0, 1, 2},

{3, 4, 5},

{6, 7, 8}

};

struct PuzzleNode {

PuzzleNode \*parent;

int matrix[MATRIX\_SIZE][MATRIX\_SIZE];

int positionRow, positionCol;

int cost;

int level;

int heuristic;

};

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

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

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

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

printf("\n");

}

return 0;

}

PuzzleNode \*generateNode(int matrix[MATRIX\_SIZE][MATRIX\_SIZE], int x, int y, int newX, int newY, int level, PuzzleNode \*parent) {

PuzzleNode \*node = new PuzzleNode;

node->parent = parent;

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

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

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

}

}

int temp = node->matrix[x][y];

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

node->matrix[newX][newY] = temp;

node->cost = INT\_MAX;

node->level = level;

node->positionRow = newX;

node->positionCol = newY;

return node;

}

int calculateCost(int matrix[MATRIX\_SIZE][MATRIX\_SIZE], int goal[MATRIX\_SIZE][MATRIX\_SIZE]) {

int count = 0;

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

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

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

count++;

return count;

}

int calculateHeuristic(int matrix[MATRIX\_SIZE][MATRIX\_SIZE], int goal[MATRIX\_SIZE][MATRIX\_SIZE]) {

int heuristic = 0;

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

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

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

heuristic++;

return heuristic;

}

int isWithinBounds(int x, int y) {

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

}

void showPuzzle(PuzzleNode \*root) {

if (root == NULL)

return;

showPuzzle(root->parent);

displayMatrix(root->matrix);

printf("\n");

}

struct PuzzleNodeComparator {

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

return lhs->heuristic > rhs->heuristic;

}

};

std::string matrixToString(int matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

std::ostringstream oss;

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

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

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

}

}

return oss.str();

}

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

auto startTime = high\_resolution\_clock::now();

priority\_queue<PuzzleNode \*, vector<PuzzleNode \*>, PuzzleNodeComparator> pq;

unordered\_set<string> visited;

PuzzleNode \*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()) {

PuzzleNode \*min = pq.top();

pq.pop();

if (min->cost == 0) {

auto endTime = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(endTime - startTime);

cout << "Goal state reached in " << totalMoves << " moves.\n";

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

showPuzzle(min);

return;

}

for (int i = 0; i < 4; i++) {

if (isWithinBounds(min->positionRow + rowOffsets[i], min->positionCol + colOffsets[i])) {

PuzzleNode \*child = generateNode(min->matrix, min->positionRow, min->positionCol, min->positionRow + rowOffsets[i], min->positionCol + colOffsets[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));

totalMoves++;

}

}

}

}

}

int main() {

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

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 < MATRIX\_SIZE; i++)

for (int j = 0; j < MATRIX\_SIZE; 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 startRow = 1, startCol = 2;

solvePuzzle(initial, startRow, startCol, goalMatrix);

return 0;

}

**Greedy Manhattan**

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include <unordered\_set>

using namespace std;

// Size of the puzzle grid

const int PUZZLE\_SIZE = 3;

// Structure to represent a state of the puzzle grid

struct PuzzleState {

vector<vector<int>> grid;

int heuristicValue; // Heuristic value based on Manhattan distance

// Constructor

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

heuristicValue = calculateHeuristic();

}

// Calculate the Manhattan distance heuristic

int calculateHeuristic() const {

int distance = 0;

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

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

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

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

int targetCol = (grid[i][j] - 1) % PUZZLE\_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 < PUZZLE\_SIZE; ++i) {

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

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

return false;

}

++targetValue;

}

}

return true;

}

// Check if two puzzle states are equal

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

return grid == other.grid;

}

};

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

struct PuzzleStateHash {

size\_t operator()(const PuzzleState& 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 ComparePuzzleState {

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

return lhs.heuristicValue > rhs.heuristicValue;

}

};

// Function to display the puzzle state

void displayPuzzleState(const PuzzleState& 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 PuzzleState& initial) {

priority\_queue<PuzzleState, vector<PuzzleState>, ComparePuzzleState> priorityQueue;

unordered\_set<PuzzleState, PuzzleStateHash> visited;

priorityQueue.push(initial);

while (!priorityQueue.empty()) {

PuzzleState current = priorityQueue.top();

priorityQueue.pop();

if (current.isGoalState()) {

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

displayPuzzleState(current);

return;

}

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

visited.insert(current);

displayPuzzleState(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 < PUZZLE\_SIZE; ++i) {

for (int j = 0; j < PUZZLE\_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 < PUZZLE\_SIZE && newY >= 0 && newY < PUZZLE\_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]);

PuzzleState nextState(newGrid);

// Add the new state to the priority queue

priorityQueue.push(nextState);

}

}

}

}

}

}

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

}

int main() {

// Initial puzzle state

vector<vector<int>> initialPuzzle = {

{8, 0, 6},

{5, 4, 7},

{2, 3, 1}

};

PuzzleState initialPuzzleState(initialPuzzle);

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

displayPuzzleState(initialPuzzleState);

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

performGreedySearch(initialPuzzleState);

return 0;

}