#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

#define N 3

#define MAX\_STATES 362880 // 9!

// Structure to represent a state in the search space

typedef struct {

int board[N][N];

int blank\_row;

int blank\_col;

int cost;

} State;

// Priority Queue Node

typedef struct {

State\* state;

int priority;

} PQNode;

// Priority Queue

typedef struct {

PQNode\* nodes[MAX\_STATES];

int size;

} PriorityQueue;

// Function to initialize the initial state

State\* initializeState(int initial[N][N]) {

State\* state = (State\*)malloc(sizeof(State));

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

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

state->board[i][j] = initial[i][j];

if (initial[i][j] == 0) {

state->blank\_row = i;

state->blank\_col = j;

}

}

}

state->cost = 0;

return state;

}

// Function to check if the current state is the goal state

bool isGoalState(State\* state) {

int count = 1;

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

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

if (state->board[i][j] != count % (N\*N)) {

return false;

}

count++;

}

}

return true;

}

// Function to calculate Hamming priority

int calculateHammingPriority(State\* state) {

int priority = 0;

int count = 1;

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

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

if (state->board[i][j] != count % (N\*N) && state->board[i][j] != 0) {

priority++;

}

count++;

}

}

return priority;

}

// Function to calculate Manhattan distance

int calculateManhattanDistance(int value, int row, int col) {

int goal\_row = (value - 1) / N;

int goal\_col = (value - 1) % N;

return abs(goal\_row - row) + abs(goal\_col - col);

}

// Function to calculate Manhattan priority

int calculateManhattanPriority(State\* state) {

int priority = 0;

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

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

if (state->board[i][j] != 0) {

priority += calculateManhattanDistance(state->board[i][j], i, j);

}

}

}

return priority;

}

// Function to print the board

void printBoard(int board[N][N]) {

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

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

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

}

printf("\n");

}

printf("\n");

}

// Function to swap two elements on the board

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to create a new priority queue

PriorityQueue\* createPriorityQueue() {

PriorityQueue\* pq = (PriorityQueue\*)malloc(sizeof(PriorityQueue));

pq->size = 0;

return pq;

}

// Function to push a state into the priority queue

void push(PriorityQueue\* pq, State\* state, int priority) {

PQNode\* newNode = (PQNode\*)malloc(sizeof(PQNode));

newNode->state = state;

newNode->priority = priority;

pq->nodes[pq->size++] = newNode;

}

// Function to pop the state with the highest priority from the priority queue

State\* pop(PriorityQueue\* pq) {

int highestPriority = INT\_MAX;

int idx = -1;

for (int i = 0; i < pq->size; i++) {

if (pq->nodes[i]->priority < highestPriority) {

highestPriority = pq->nodes[i]->priority;

idx = i;

}

}

State\* state = pq->nodes[idx]->state;

free(pq->nodes[idx]);

for (int i = idx; i < pq->size - 1; i++) {

pq->nodes[i] = pq->nodes[i + 1];

}

pq->size--;

return state;

}

// Function to check if the priority queue is empty

bool isEmpty(PriorityQueue\* pq) {

return pq->size == 0;

}

// Function to perform A\* search

void aStarSearch(State\* initialState) {

PriorityQueue\* openSet = createPriorityQueue();

push(openSet, initialState, initialState->cost);

while (!isEmpty(openSet)) {

State\* currentState = pop(openSet);

if (isGoalState(currentState)) {

printf("Goal state reached!\n");

printf("Total cost: %d\n", currentState->cost);

printf("Steps to reach the goal state:\n");

printBoard(currentState->board);

return;

}

// Generate successor states

for (int dr = -1; dr <= 1; dr++) {

for (int dc = -1; dc <= 1; dc++) {

if ((dr == 0 && dc == 0) || (dr != 0 && dc != 0)) {

continue; // Skip invalid moves

}

int new\_blank\_row = currentState->blank\_row + dr;

int new\_blank\_col = currentState->blank\_col + dc;

if (new\_blank\_row >= 0 && new\_blank\_row < N && new\_blank\_col >= 0 && new\_blank\_col < N) {

State\* successor = (State\*)malloc(sizeof(State));

\*successor = \*currentState;

swap(&successor->board[currentState->blank\_row][currentState->blank\_col],

&successor->board[new\_blank\_row][new\_blank\_col]);

successor->blank\_row = new\_blank\_row;

successor->blank\_col = new\_blank\_col;

successor->cost = currentState->cost + 1;

int hammingPriority = calculateHammingPriority(successor);

int manhattanPriority = calculateManhattanPriority(successor);

int totalPriority = successor->cost + manhattanPriority; // A\* heuristic function

push(openSet, successor, totalPriority);

}

}

}

// Free the current state

free(currentState);

}

printf("Goal state could not be reached!\n");

// Free the priority queue

free(openSet);

}

int main() {

int initial[N][N] = {

{1, 2, 3},

{4, 0, 5},

{6, 7, 8}

};

State\* initialState = initializeState(initial);

if (isGoalState(initialState)) {

printf("Initial state is the goal state!\n");

return 0;

}

printf("Initial state:\n");

printBoard(initialState->board);

int hammingPriority = calculateHammingPriority(initialState);

printf("Hamming Priority: %d\n", hammingPriority);

int manhattanPriority = calculateManhattanPriority(initialState);

printf("Manhattan Priority: %d\n", manhattanPriority);

aStarSearch(initialState);

free(initialState);

return 0;

}