**Artificial Intelligence: Assignment 2**

**Implement A star Algorithm for any game search problem.**

#include<iostream>

#include<cmath> // For abs() function

#include<limits.h> // For INT\_MAX (maximum integer value)

using namespace std;

// Global variable to keep track of the number of moves taken

int g = 0;

// Function to print the 8-puzzle grid

void Print(int puzzle[]) {

for(int i = 0; i < 9; i++) {

if(i % 3 == 0) cout << '\n'; // Start a new line after every 3 elements

if(puzzle[i] == -1) cout << "\_ "; // Print '\_' for empty tile

else cout << puzzle[i] << " "; // Print the tile number

}

cout << "\n\n";

}

// Function to move the empty tile left (swap with the left tile)

void moveLeft(int start[], int position) {

swap(start[position], start[position - 1]);

}

// Function to move the empty tile right (swap with the right tile)

void moveRight(int start[], int position) {

swap(start[position], start[position + 1]);

}

// Function to move the empty tile up (swap with the tile above)

void moveUp(int start[], int position) {

swap(start[position], start[position - 3]);

}

// Function to move the empty tile down (swap with the tile below)

void moveDown(int start[], int position) {

swap(start[position], start[position + 3]);

}

// Function to copy one puzzle state into another

void Copy(int temp[], int real[]) {

for(int i = 0; i < 9; i++) temp[i] = real[i];

}

/\*

\* Heuristic function (Manhattan Distance)

\* For each tile, calculate the sum of the vertical and horizontal distances

\* between its position in the start state and its position in the goal state.

\*/

int heuristic(int start[], int goal[]) {

int h = 0; // Initialize heuristic cost

for(int i = 0; i < 9; i++) { // Loop through each tile in the start state

for(int j = 0; j < 9; j++) { // Loop through each tile in the goal state

if (start[i] == goal[j] && start[i] != -1) {

// Compute Manhattan Distance and add to heuristic cost

h += abs((j - i) / 3) + abs((j - i) % 3);

}

}

}

return h + g; // Return f(n) = g(n) + h(n)

}

// Function to determine the best move based on heuristic values

void moveTile(int start[], int goal[]) {

int emptyAt = 0; // Position of the empty tile (-1)

// Find the position of the empty tile in the puzzle

for(int i = 0; i < 9; i++) {

if(start[i] == -1) {

emptyAt = i;

break;

}

}

// Temporary arrays to store possible moves

int t1[9], t2[9], t3[9], t4[9];

int f1

= INT\_MAX, f2 = INT\_MAX, f3 = INT\_MAX, f4 = INT\_MAX;

// Copy the current puzzle state to each temporary array

Copy(t1, start);

Copy(t2, start);

Copy(t3, start);

Copy(t4, start);

// Get row and column of empty tile

int row = emptyAt / 3;

int col = emptyAt % 3;

// If moving left is possible, compute heuristic

if(col - 1 >= 0) {

moveLeft(t1, emptyAt);

f1 = heuristic(t1, goal);

}

// If moving right is possible, compute heuristic

if(col + 1 < 3) {

moveRight(t2, emptyAt);

f2 = heuristic(t2, goal);

}

// If moving down is possible, compute heuristic

if(row + 1 < 3) {

moveDown(t3, emptyAt);

f3 = heuristic(t3, goal);

}

// If moving up is possible, compute heuristic

if(row - 1 >= 0) {

moveUp(t4, emptyAt);

f4 = heuristic(t4, goal);

}

// Find the move with the least heuristic value and perform that move

if(f1 <= f2 && f1 <= f3 && f1 <= f4) {

moveLeft(start, emptyAt);

}

else if(f2 <= f1 && f2 <= f3 && f2 <= f4) {

moveRight(start, emptyAt);

}

else if(f3 <= f1 && f3 <= f2 && f3 <= f4) {

moveDown(start, emptyAt);

}

else {

moveUp(start, emptyAt);

}

}

// Function to solve the 8-puzzle using A\* algorithm

void solveEight(int start[], int goal[]) {

g++; // Increment move count

moveTile(start, goal); // Choose the best move

Print(start); // Print the current puzzle state

// Compute heuristic after the move

int f = heuristic(start, goal);

// If f(n) == g(n), puzzle is solved

if(f == g) {

cout << "Solved in " << f << " moves\n";

return;

}

// Recursively solve the puzzle

solveEight(start, goal);

}

/\*

\* Function to check if the puzzle is solvable

\* Count the number of inversions (pairs where a larger number precedes a smaller number).

\* If the number of inversions is odd, the puzzle is unsolvable.

\*/

bool solvable(int start[]) {

int invrs = 0;

for(int i = 0; i < 9; i++) {

if(start[i] <= 1) continue; // Skip empty tile and 1

for(int j = i + 1; j < 9; j++) {

if(start[j] == -1) continue; // Skip empty tile

if(start[i] > start[j]) invrs++; // Count inversions

}

}

return invrs % 2 == 0; // If inversions are even, puzzle is solvable

}

int main() {

int start[9]; // Array to store the initial puzzle state

int goal[9]; // Array to store the goal puzzle state

// Take user input for start state

cout << "Enter the start state (Enter -1 for empty tile): ";

for(int i = 0; i < 9; i++) {

cin >> start[i];

}

// Take user input for goal state

cout << "Enter the goal state (Enter -1 for empty tile): ";

for(int i = 0; i < 9; i++) {

cin >> goal[i];

}

// Print the initial state

Print(start);

// Check if the puzzle is solvable

if(solvable(start)) {

solveEight(start, goal); // Start solving the puzzle

}

else {

cout << "\nImpossible To Solve\n"; // If not solvable, print message

}

return 0; // Exit program

}