**AI LAB ASSIGNMENT 3**

**Name: Atharva Salitri PRN: 12310120**

**Div : CSAI B Batch: 2 Roll No: 37**

**Implement Informed Search**

**1) A\* Algorithm**

**CODE:**

#include <iostream>

#include <vector>

#include <queue>

#include <cmath>

#include <climits>

#include <algorithm>

using namespace std;

struct Node {

int x, y;

int g, h;

Node\* parent;

bool operator>(const Node& other) const {

return (g + h) > (other.g + other.h);

}

};

vector<pair<int, int>> getNeighbors(int x, int y) {

return {{x + 1, y}, {x - 1, y}, {x, y + 1}, {x, y - 1}};

}

int heuristic(int x1, int y1, int x2, int y2) {

return abs(x1 - x2) + abs(y1 - y2);

}

bool isValid(int x, int y, int rows, int cols, const vector<vector<int>>& grid) {

return x >= 0 && x < rows && y >= 0 && y < cols && grid[x][y] == 0;

}

void printPath(Node\* node) {

if (node) {

printPath(node->parent);

cout << "(" << node->x << ", " << node->y << ") ";

}

}

void aStar(const vector<vector<int>>& grid, pair<int, int> start, pair<int, int> goal) {

int rows = grid.size();

int cols = grid[0].size();

priority\_queue<Node, vector<Node>, greater<Node>> openSet;

vector<vector<int>> gCost(rows, vector<int>(cols, INT\_MAX));

vector<vector<bool>> closedSet(rows, vector<bool>(cols, false));

Node\* startNode = new Node{start.first, start.second, 0, heuristic(start.first, start.second, goal.first, goal.second), nullptr};

openSet.push(\*startNode);

gCost[start.first][start.second] = 0;

while (!openSet.empty()) {

Node current = openSet.top();

openSet.pop();

if (current.x == goal.first && current.y == goal.second) {

cout << "Path Found"<<endl;

cout << "Path: ";

printPath(&current);

cout << endl;

return;

}

closedSet[current.x][current.y] = true;

for (auto& neighbor : getNeighbors(current.x, current.y)) {

int nx = neighbor.first;

int ny = neighbor.second;

if (!isValid(nx, ny, rows, cols, grid) || closedSet[nx][ny]) {

continue;

}

int tentativeG = current.g + 1;

if (tentativeG < gCost[nx][ny]) {

Node\* neighborNode = new Node{nx, ny, tentativeG, heuristic(nx, ny, goal.first, goal.second), new Node(current)};

gCost[nx][ny] = tentativeG;

openSet.push(\*neighborNode);

}

}

}

cout << "No path found." << endl;

}

int main() {

vector<vector<int>> grid = {

{0, 0, 0, 0, 0},

{0, 1, 1, 1, 0},

{0, 0, 0, 0, 0},

{0, 1, 1, 1, 0},

{0, 0, 0, 0, 0}

};

pair<int, int> start = {0, 0};

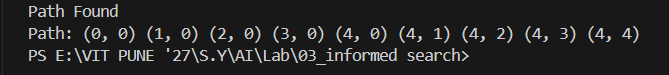
pair<int, int> goal = {4, 4};

aStar(grid, start, goal);

return 0;

}

**OUTPUT:**



**2)Ao\* Algorithm**

**CODE:**

#include <iostream>

#include <vector>

#include <queue>

#include <unordered\_map>

#include <cmath>

#include <climits>

#include <algorithm>

using namespace std;

struct Node {

int x, y;

int g, h;

Node\* parent;

bool operator>(const Node& other) const {

return (g + h) > (other.g + other.h);

}

};

vector<pair<int, int>> getNeighbors(int x, int y) {

return {{x + 1, y}, {x - 1, y}, {x, y + 1}, {x, y - 1}};

}

int heuristic(int x1, int y1, int x2, int y2) {

return abs(x1 - x2) + abs(y1 - y2);

}

bool isValid(int x, int y, int rows, int cols, const vector<vector<int>>& grid) {

return x >= 0 && x < rows && y >= 0 && y < cols && grid[x][y] == 0;

}

void printPath(Node\* node) {

if (node) {

printPath(node->parent);

cout << "(" << node->x << ", " << node->y << ") ";

}

}

void aoStar(const vector<vector<int>>& grid, pair<int, int> start, pair<int, int> goal) {

int rows = grid.size();

int cols = grid[0].size();

priority\_queue<Node, vector<Node>, greater<Node>> openSet;

unordered\_map<int, unordered\_map<int, int>> gCost;

unordered\_map<int, unordered\_map<int, bool>> closedSet;

Node\* startNode = new Node{start.first, start.second, 0, heuristic(start.first, start.second, goal.first, goal.second), nullptr};

openSet.push(\*startNode);

gCost[start.first][start.second] = 0;

while (!openSet.empty()) {

Node current = openSet.top();

openSet.pop();

if (current.x == goal.first && current.y == goal.second) {

cout << "Path: ";

printPath(&current);

cout << endl;

return;

}

closedSet[current.x][current.y] = true;

for (auto& neighbor : getNeighbors(current.x, current.y)) {

int nx = neighbor.first;

int ny = neighbor.second;

if (!isValid(nx, ny, rows, cols, grid) || closedSet[nx][ny]) {

continue;

}

int tentativeG = current.g + 1;

if (tentativeG < gCost[nx][ny] || gCost[nx][ny] == 0) {

Node\* neighborNode = new Node{nx, ny, tentativeG, heuristic(nx, ny, goal.first, goal.second), new Node(current)};

gCost[nx][ny] = tentativeG;

openSet.push(\*neighborNode);

}

}

}

cout << "No path found." << endl;

}

int main() {

vector<vector<int>> grid = {

{0, 0, 0, 0, 0, 0},

{0, 1, 1, 0, 0, 0},

{0, 0, 0, 0, 1, 0},

{0, 1, 1, 0, 1, 0},

{0, 0, 0, 0, 0, 0},

{0, 0, 1, 1, 1, 0}

};

pair<int, int> start = {0, 0};

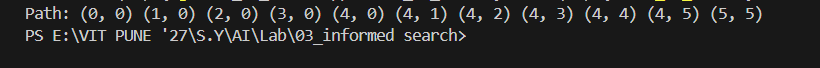
pair<int, int> goal = {5, 5};

aoStar(grid, start, goal);

return 0;

}

**OUTPUT:**



**3)Hill Climbing Algorithm**

**CODE:**

#include <iostream>

#include <vector>

#include <algorithm>

#include <ctime>

#include <cstdlib>

using namespace std;

#define NUM\_CITIES 4

// Distance matrix representing distances between cities

int distance\_matrix[NUM\_CITIES][NUM\_CITIES] = {

{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}

};

int total\_distance(const std::vector<int>& path) {

// Calculate the total distance traveled in the given path

int total = 0;

for (size\_t i = 0; i < path.size() - 1; i++) {

total += distance\_matrix[path[i]][path[i + 1]];

}

total += distance\_matrix[path.back()][path[0]]; // Return to starting city

return total;

}

void hill\_climbing\_tsp(int num\_cities, int max\_iterations) {

vector<int> current\_path(num\_cities); // Initial solution, visiting cities in order

for (int i = 0; i < num\_cities; i++) {

current\_path[i] = i;

}

int current\_distance = total\_distance(current\_path);

for (int it = 0; it < max\_iterations; it++) {

// Generate a neighboring solution by swapping two random cities

std::vector<int> neighbor\_path = current\_path;

int i = rand() % num\_cities;

int j = rand() % num\_cities;

swap(neighbor\_path[i], neighbor\_path[j]);

int neighbor\_distance = total\_distance(neighbor\_path);

// If the neighbor solution is better, move to it

if (neighbor\_distance < current\_distance) {

current\_path = neighbor\_path;

current\_distance = neighbor\_distance;

}

}

cout << "Optimal path: ";

for (int city : current\_path) {

cout << city << " ";

}

cout << endl;

cout << "Total distance: " << current\_distance << endl;

}

int main() {

srand(time(NULL));

int max\_iterations = 10000;

hill\_climbing\_tsp(NUM\_CITIES, max\_iterations);

return 0;

}

**OUTPUT:**

