**Name : Tanishq Thuse**

**Branch : CS(AI)**

**Year : SY**

**Div : B**

**Roll no. : 60**

**PRN : 12310237**

**Title : AI Assignment-3 : Implement Informed Search Algorithms**

**Q1) Implement A\* Algorithm**

**Code :**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <math.h>

#define ROWS 5

#define COLS 5

//defines the node structure

typedef struct Node {

struct Node\* parent;

int position[2];

int g;

int h;

int f;

} Node;

//creates a new node

//parent: the parent node, i.e the node from which this node is reached

//row: the row number of the node

//col: the column number of the node

Node\* createNode(Node\* parent, int row, int col) {

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

newNode->parent = parent;

newNode->position[0] = row;

newNode->position[1] = col;

newNode->g = 0; //distance from the start node

newNode->h = 0; //heuristic value

newNode->f = 0; //total cost

return newNode;

}

//checks if the given position is valid or not, i.e it is within the maze

bool isValid(int row, int col) {

return (row >= 0 && row < ROWS && col >= 0 && col < COLS);

}

//checks if the given position is an obstacle or not

bool isObstacle(int maze[ROWS][COLS], int row, int col) {

return (maze[row][col] == 1);

}

//checks if the given position is the destination or not

bool isDestination(int row, int col, int end[2]) {

return (row == end[0] && col == end[1]);

}

//calculates the heuristic value of the given position

//uses the Euclidean distance formula

//formula : sqrt((x1-x2)^2 + (y1-y2)^2)

int calculateHValue(int current[2], int end[2]) {

return (int)(pow((current[0] - end[0]), 2) + pow((current[1] - end[1]), 2));

}

//prints the path from the start node to the end node

//works when the end node is reached

void printPath(Node\* current) {

if (current == NULL) {

return;

}

printPath(current->parent);

printf("(%d, %d)\n", current->position[0], current->position[1]);

}

//implements the A\* algorithm

bool aStar(int maze[ROWS][COLS], int start[2], int end[2]) {

//creates first node with no parent, i.e the start node

//sets the g, h and f values to 0

Node\* startNode = createNode(NULL, start[0], start[1]);

startNode->g = startNode->h = startNode->f = 0;

//creates the end node with no parent

//sets the g, h and f values to 0

Node\* endNode = createNode(NULL, end[0], end[1]);

endNode->g = endNode->h = endNode->f = 0;

// Create a min-heap (priority queue)

//stores the nodes in the priority queue

//rows \* cols is the maximum number of nodes that is rows \* cols

Node\* priorityQueue[ROWS \* COLS];

int queueSize = 0;

// Add the start node to the priority queue

priorityQueue[queueSize++] = startNode;

while (queueSize > 0) {

// Find the node with the lowest f-value in the priority queue

// wont run 1st time bcuz 1<1 is false"

int minIndex = 0;

for (int i = 1; i < queueSize; i++) {

if (priorityQueue[i]->f < priorityQueue[minIndex]->f) {

minIndex = i;

}

}

Node\* currentNode = priorityQueue[minIndex];

// Remove the current node from the priority queue

for (int i = minIndex; i < queueSize - 1; i++) {

priorityQueue[i] = priorityQueue[i + 1];

}

queueSize--;

if (isDestination(currentNode->position[0], currentNode->position[1], end)) {

printf("Path:\n");

printPath(currentNode);

return true;

}

int directions[8][2] = {

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

{-1, -1}, {-1, 1}, {1, -1}, {1, 1}

};

for (int i = 0; i < 8; i++) {

int newRow = currentNode->position[0] + directions[i][0];

int newCol = currentNode->position[1] + directions[i][1];

if (isValid(newRow, newCol) && !isObstacle(maze, newRow, newCol)) {

Node\* newNode = createNode(currentNode, newRow, newCol);

newNode->g = currentNode->g + 1;

newNode->h = calculateHValue(newNode->position, end);

newNode->f = newNode->g + newNode->h;

// Add the new node to the priority queue

priorityQueue[queueSize++] = newNode;

}

}

}

printf("No path found.\n");

return false;

}

int main() {

int maze[ROWS][COLS] = {

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

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

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

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

{0, 0, 0, 0, 0}

};

//creates 2 arrays to store the start and end positions

int start[2], end[2];

printf("Enter the start position (row, column): ");

scanf("%d %d", &start[0], &start[1]);

printf("Enter the end position (row, column): ");

scanf("%d %d", &end[0], &end[1]);

if (!aStar(maze, start, end)) {

printf("No path found.\n");

}

return 0;

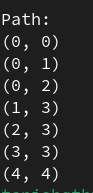
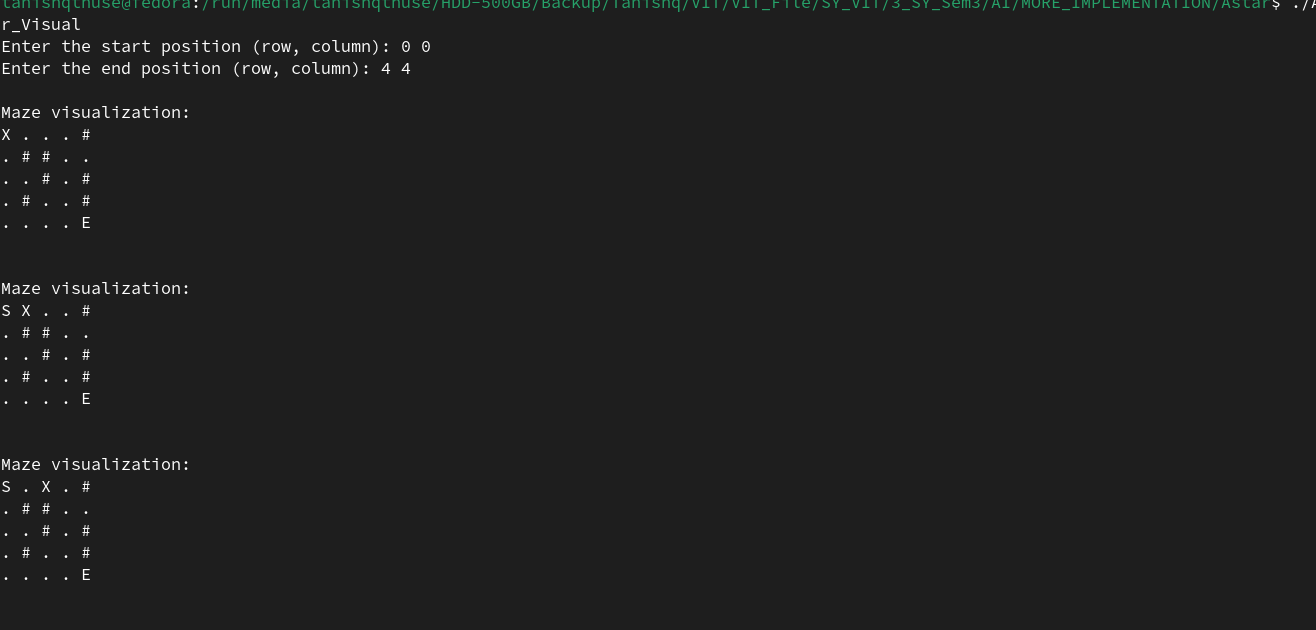
//Ex :

//Start : (0,0)

// End : (4,4)

}

**Output : (I have also included some visualization for how Astar works)**



**Q2) Implementation of AO\* Algorithms**

**Code :**

import java.util.\*;

class Code {

public static Map<String, Integer>

Cost(Map<String, Integer> H,

Map<String, List<String> > condition, int weight)

{

Map<String, Integer> cost = new HashMap<>();

if (condition.containsKey("AND")) {

List<String> AND\_nodes = condition.get("AND");

String Path\_A = String.join(" AND ", AND\_nodes);

int PathA

= AND\_nodes.stream()

.mapToInt(

node -> H.get(node) + weight)

.sum();

cost.put(Path\_A, PathA);

}

if (condition.containsKey("OR")) {

List<String> OR\_nodes = condition.get("OR");

String Path\_B = String.join(" OR ", OR\_nodes);

int PathB

= OR\_nodes.stream()

.mapToInt(

node -> H.get(node) + weight)

.min()

.getAsInt();

cost.put(Path\_B, PathB);

}

return cost;

}

public static Map<String, Map<String, Integer> >

UpdateCost(

Map<String, Integer> H,

Map<String, Map<String, List<String> > > Conditions,

int weight)

{

List<String> Main\_nodes

= new ArrayList<>(Conditions.keySet());

Collections.reverse(Main\_nodes);

Map<String, Map<String, Integer> > least\_cost

= new HashMap<>();

for (String key : Main\_nodes) {

Map<String, List<String> > condition

= Conditions.get(key);

System.out.printf("%s: %s >>> %s%n", key,

condition,

Cost(H, condition, weight));

Map<String, Integer> c

= Cost(H, condition, weight);

H.put(key, Collections.min(c.values()));

least\_cost.put(key, Cost(H, condition, weight));

}

return least\_cost;

}

public static String ShortestPath(

String Start,

Map<String, Map<String, Integer> > Updated\_cost,

Map<String, Integer> H)

{

String Path = Start;

if (Updated\_cost.containsKey(Start)) {

int Min\_cost = Collections.min(

Updated\_cost.get(Start).values());

List<String> key = new ArrayList<>(

Updated\_cost.get(Start).keySet());

List<Integer> values = new ArrayList<>(

Updated\_cost.get(Start).values());

int Index = values.indexOf(Min\_cost);

List<String> Next

= Arrays.asList(key.get(Index).split(" "));

if (Next.size() == 1) {

Start = Next.get(0);

Path += "<--"

+ ShortestPath(Start, Updated\_cost,

H);

}

else {

Path += "<--(" + key.get(Index) + ") ";

Start = Next.get(0);

Path += "["

+ ShortestPath(Start, Updated\_cost,

H)

+ " + ";

Start = Next.get(Next.size() - 1);

Path += ShortestPath(Start, Updated\_cost, H)

+ "]";

}

}

return Path;

}

public static void main(String[] args)

{

Map<String, Integer> H = new HashMap<>();

H.put("A", -1);

H.put("B", 5);

H.put("C", 2);

H.put("D", 4);

H.put("E", 7);

H.put("F", 9);

H.put("G", 3);

H.put("H", 0);

H.put("I", 0);

H.put("J", 0);

Map<String, Map<String, List<String> > > Conditions

= new HashMap<>();

Map<String, List<String> > aConditions

= new HashMap<>();

aConditions.put("OR", Arrays.asList("B"));

aConditions.put("AND", Arrays.asList("C", "D"));

Conditions.put("A", aConditions);

Map<String, List<String> > bConditions

= new HashMap<>();

bConditions.put("OR", Arrays.asList("E", "F"));

Conditions.put("B", bConditions);

Map<String, List<String> > cConditions

= new HashMap<>();

cConditions.put("OR", Arrays.asList("G"));

cConditions.put("AND", Arrays.asList("H", "I"));

Conditions.put("C", cConditions);

Map<String, List<String> > dConditions

= new HashMap<>();

dConditions.put("OR", Arrays.asList("J"));

Conditions.put("D", dConditions);

// weight

int weight = 1;

// Updated cost

System.out.println("Updated Cost :");

Map<String, Map<String, Integer> > Updated\_cost

= UpdateCost(H, Conditions, weight);

System.out.println("\*".repeat(75));

System.out.println("Shortest Path :");

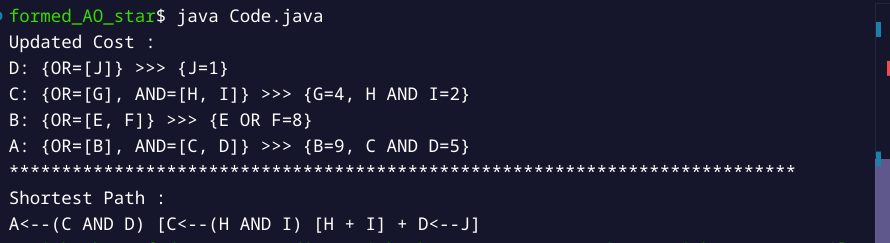
System.out.println(

ShortestPath("A", Updated\_cost, H));

}

}

**Output :**

  
 **Q3) Hill Climbing Algorithm**

**Code :**

//Hill climbing

#include <stdio.h>

#include <stdlib.h>

#define MAX\_NODES 100

typedef struct {

int state;

int func\_value;

} Node;

int find\_best\_neighbor(Node neighbors[], int num\_neighbors) {

int best\_index = 0;

for (int i = 1; i < num\_neighbors; i++) {

if (neighbors[i].func\_value > neighbors[best\_index].func\_value) {

best\_index = i;

}

}

return best\_index;

}

int hill\_climbing(Node nodes[], int num\_nodes, int start\_index) {

int current\_index = start\_index;

int num\_neighbors = 2;

Node neighbors[2];

while (1) {

int left\_index = -1, right\_index = -1;

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

if (nodes[i].state == nodes[current\_index].state - 1) {

left\_index = i;

}

if (nodes[i].state == nodes[current\_index].state + 1) {

right\_index = i;

}

}

if (left\_index == -1 && right\_index == -1) {

return nodes[current\_index].state; // No neighbors

}

int num\_valid\_neighbors = 0;

if (left\_index != -1) {

neighbors[num\_valid\_neighbors++] = nodes[left\_index];

}

if (right\_index != -1) {

neighbors[num\_valid\_neighbors++] = nodes[right\_index];

}

if (num\_valid\_neighbors == 0) {

return nodes[current\_index].state;

}

int best\_index = find\_best\_neighbor(neighbors, num\_valid\_neighbors);

if (neighbors[best\_index].func\_value <= nodes[current\_index].func\_value) {

return nodes[current\_index].state;

}

current\_index = (neighbors[best\_index].state == nodes[current\_index].state - 1) ? left\_index : right\_index;

}

}

int main() {

int num\_nodes;

printf("Enter the number of nodes: ");

scanf("%d", &num\_nodes);

if (num\_nodes > MAX\_NODES) {

printf("Error: Maximum number of nodes is %d\n", MAX\_NODES);

return 1;

}

Node nodes[MAX\_NODES];

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

printf("Enter state (value) and function value for node %d: ", i);

scanf("%d %d", &nodes[i].state, &nodes[i].func\_value);

}

int start\_index;

printf("Enter the index of the starting node: ");

scanf("%d", &start\_index);

if (start\_index < 0 || start\_index >= num\_nodes) {

printf("Error: Invalid starting index\n");

return 1;

}

int result = hill\_climbing(nodes, num\_nodes, start\_index);

printf("Result: %d\n", result);

return 0;

/\*\*

\* Enter the number of nodes: 5

Enter state (value) and function value for node 0: 0 1

Enter state (value) and function value for node 1: 1 3

Enter state (value) and function value for node 2: 2 5

Enter state (value) and function value for node 3: 3 2

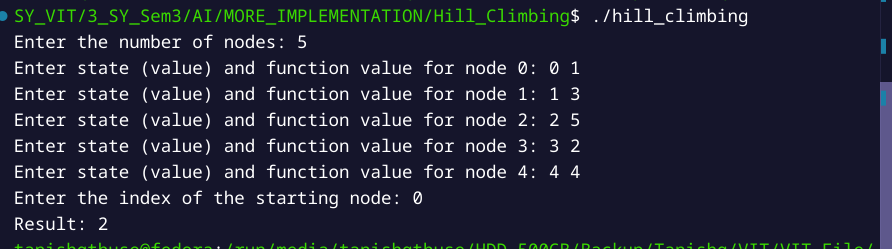
Enter state (value) and function value for node 4: 4 4

Enter the index of the starting node: 0

\*/

}

**Output :**

****