**BFS DFS**

#include <stdio.h>

#define MAX 100

void bfs\_traversal(int graph[MAX][MAX], int start\_vertex, int num\_vertices);

void dfs\_traversal(int graph[MAX][MAX], int start\_vertex, int num\_vertices, int visited[]);

int main() {

int num\_vertices, graph[MAX][MAX], visited[MAX] = {0};

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

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

printf("Enter the adjacency matrix:\n");

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

for (int j = 0; j < num\_vertices; j++) {

scanf("%d", &graph[i][j]);

}

}

// Count the number of edges in the graph

int num\_edges = 0;

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

for (int j = 0; j < num\_vertices; j++) {

if (graph[i][j] == 1) {

num\_edges++;

}

}

}

// Determine whether BFS or DFS is better suited for the graph

if (num\_edges > num\_vertices) {

printf("The input graph is better suited for DFS traversal.\n");

printf("Enter the starting vertex: ");

int start\_vertex;

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

dfs\_traversal(graph, start\_vertex, num\_vertices, visited);

} else {

printf("The input graph is better suited for BFS traversal.\n");

printf("Enter the starting vertex: ");

int start\_vertex;

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

bfs\_traversal(graph, start\_vertex, num\_vertices);

}

return 0;

}

void bfs\_traversal(int graph[MAX][MAX], int start\_vertex, int num\_vertices) {

int visited[MAX] = {0};

int queue[MAX];

int front = -1, rear = -1;

visited[start\_vertex] = 1;

queue[++rear] = start\_vertex;

printf("BFS order of traversal starting from vertex %d:\n", start\_vertex);

while (front != rear) {

int current\_vertex = queue[++front];

printf("%d ", current\_vertex);

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

if (graph[current\_vertex][i] == 1 && visited[i] == 0) {

visited[i] = 1;

queue[++rear] = i;

}

}

}

printf("\n");

}

void dfs\_traversal(int graph[MAX][MAX], int start\_vertex, int num\_vertices, int visited[]) {

visited[start\_vertex] = 1;

printf("DFS order of traversal starting from vertex %d:\n", start\_vertex);

printf("%d ", start\_vertex);

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

if (graph[start\_vertex][i] == 1 && visited[i] == 0) {

dfs\_traversal(graph, i, num\_vertices, visited);

}

}

printf("\n");

}

**MIN MAX**

MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer,

values, alpha, beta):

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

False, values, alpha, beta)

best = max(best, val)

alpha = max(alpha, best)

if beta <= alpha:

break

return best

else:

best = MAX

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

True, values, alpha, beta)

best = min(best, val)

beta = min(beta, best)

if beta <= alpha:

break

return best

if \_\_name\_\_ == "\_\_main\_\_":

values = []

for i in range(0, 8):

x = int(input(f"Enter Value {i} : "))

values.append(x)

print ("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

**UNIFICATION**

def get\_index\_comma(string):

index\_list = list()

par\_count = 0

for i in range(len(string)):

if string[i] == ',' and par\_count == 0:

index\_list.append(i)

elif string[i] == '(':

par\_count += 1

elif string[i] == ')':

par\_count -= 1

return index\_list

def is\_variable(expr):

for i in expr:

if i == '(' or i == ')':

return False

return True

def process\_expression(expr):

expr = expr.replace(' ', '')

index = None

for i in range(len(expr)):

if expr[i] == '(':

index = i

break

predicate\_symbol = expr[:index]

expr = expr.replace(predicate\_symbol, '')

expr = expr[1:len(expr) - 1]

arg\_list = list()

indices = get\_index\_comma(expr)

if len(indices) == 0:

arg\_list.append(expr)

else:

arg\_list.append(expr[:indices[0]])

for i, j in zip(indices, indices[1:]):

arg\_list.append(expr[i + 1:j])

arg\_list.append(expr[indices[len(indices) - 1] + 1:])

return predicate\_symbol, arg\_list

def get\_arg\_list(expr):

\_, arg\_list = process\_expression(expr)

flag = True

while flag:

flag = False

for i in arg\_list:

if not is\_variable(i):

flag = True

\_, tmp = process\_expression(i)

for j in tmp:

if j not in arg\_list:

arg\_list.append(j)

arg\_list.remove(i)

return arg\_list

def check\_occurs(var, expr):

arg\_list = get\_arg\_list(expr)

if var in arg\_list:

return True

return False

def unify(expr1, expr2):

if is\_variable(expr1) and is\_variable(expr2):

if expr1 == expr2:

return 'Null'

else:

return False

elif is\_variable(expr1) and not is\_variable(expr2):

if check\_occurs(expr1, expr2):

return False

else:

tmp = str(expr2) + '/' + str(expr1)

return tmp

elif not is\_variable(expr1) and is\_variable(expr2):

if check\_occurs(expr2, expr1):

return False

else:

tmp = str(expr1) + '/' + str(expr2)

return tmp

else:

predicate\_symbol\_1, arg\_list\_1 = process\_expression(expr1)

predicate\_symbol\_2, arg\_list\_2 = process\_expression(expr2)

# Step 2

if predicate\_symbol\_1 != predicate\_symbol\_2:

return False

# Step 3

elif len(arg\_list\_1) != len(arg\_list\_2):

return False

else:

# Step 4: Create substitution list

sub\_list = list()

# Step 5:

for i in range(len(arg\_list\_1)):

tmp = unify(arg\_list\_1[i], arg\_list\_2[i])

if not tmp:

return False

elif tmp == 'Null':

pass

else:

if type(tmp) == list:

for j in tmp:

sub\_list.append(j)

else:

sub\_list.append(tmp)

# Step 6

return sub\_list

if \_\_name\_\_ == '\_\_main\_\_':

n = int(input('Enter the number of expressions: '))

expressions = []

for i in range(n):

expressions.append(input(f'Enter expression {i+1}: '))

result = expressions[0]

for i in range(1, n):

result = unify(result, expressions[i])

if not result:

print('The process of Unification failed!')

else:

print('The process of Unification successful!')

print(result)

**FUZZY**

#include <iostream>

#include <cmath>

#include <cstring>

const double cdMinimumPrice = 0;

const double cdMaximumPrice = 70;

using namespace std;

class CFuzzyFunction

{

protected:double dLeft, dRight;

char cType;

char \*sName;

public:CFuzzyFunction ()

{

};

virtual ~ CFuzzyFunction ()

{

delete[]sName;

sName = NULL;

}

virtual void setInterval (double l, double r)

{

dLeft = l;

dRight = r;

}

virtual void setMiddle (double dL = 0, double dR = 0) = 0;

virtual void setType (char c)

{

cType = c;

}

virtual void setName (const char \*s)

{

sName = new char[strlen (s) + 1];

strcpy (sName, s);

}

bool isDotInInterval (double t)

{

if ((t >= dLeft) && (t <= dRight))

return true;

else

return false;

}

char getType (void) const

{

return cType;

}

void getName () const

{

cout << sName << endl;

}

virtual double getValue (double t) = 0;

};

class CTriangle:public CFuzzyFunction

{

private:double dMiddle;

public:void setMiddle (double dL, double dR)

{

dMiddle = dL;

}

double getValue (double t)

{

if (t <= dLeft)

return 0;

else if (t < dMiddle)

return (t - dLeft) / (dMiddle - dLeft);

else if (t == dMiddle)

return 1.0;

else if (t < dRight)

return (dRight - t) / (dRight - dMiddle);

else

return 0;

}

};

class CTrapezoid:public CFuzzyFunction

{

private:double dLeftMiddle, dRightMiddle;

public:void setMiddle (double dL, double dR)

{

dLeftMiddle = dL;

dRightMiddle = dR;

}

double getValue (double t)

{

if (t <= dLeft)

return 0;

else if (t < dLeftMiddle)

return (t - dLeft) / (dLeftMiddle - dLeft);

else if (t <= dRightMiddle)

return 1.0;

else if (t < dRight)

return (dRight - t) / (dRight - dRightMiddle);

else

return 0;

}

};

int

main (void)

{

CFuzzyFunction \*FuzzySet[3];

FuzzySet[0] = new CTrapezoid;

FuzzySet[1] = new CTriangle;

FuzzySet[2] = new CTrapezoid;

FuzzySet[0]->setInterval (-5, 30);

FuzzySet[0]->setMiddle (0, 20);

FuzzySet[0]->setType ('r');

FuzzySet[0]->setName ("low\_price");

FuzzySet[1]->setInterval (25, 45);

FuzzySet[1]->setMiddle (35, 35);

FuzzySet[1]->setType ('t');

FuzzySet[1]->setName ("good\_price");

FuzzySet[2]->setInterval (40, 75);

FuzzySet[2]->setMiddle (50, 70);

FuzzySet[2]->setType ('r');

FuzzySet[2]->setName ("to\_expensive");

double dValue;

do

{

cout << "\nInput the value->";

cin >> dValue;

if (dValue < cdMinimumPrice)

continue;

if (dValue > cdMaximumPrice)

continue;

for (int i = 0; i < 3; i++)

{

cout << "\nThe dot=" << dValue << endl;

if (FuzzySet[i]->isDotInInterval (dValue))

cout << "In the interval";

else

cout << "Not in the interval";

cout << endl;

cout << "The name of function is" << endl;

FuzzySet[i]->getName ();

cout << "and the membership is=";

cout << FuzzySet[i]->getValue (dValue);

}

}

while (true);

return EXIT\_SUCCESS;

}

**BEST FIRST SEARCH**

import java.util.\*;

public class BestFirstSearch {

private int V; // number of vertices

private LinkedList<Node>[] adj; // adjacency list

public BestFirstSearch(int v) {

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList<Node>();

}

}

public void addEdge(int v, Node w) {

adj[v].add(w);

}

public void bestFirstSearch(int s, int goal) {

PriorityQueue<Node> queue = new PriorityQueue<Node>(new NodeComparator());

boolean[] visited = new boolean[V];

visited[s] = true;

queue.add(new Node(s, 0));

while (!queue.isEmpty()) {

Node curr = queue.poll();

System.out.print(curr.vertex + " ");

if (curr.vertex == goal) {

return;

}

Iterator<Node> i = adj[curr.vertex].listIterator();

while (i.hasNext()) {

Node next = i.next();

if (!visited[next.vertex]) {

visited[next.vertex] = true;

queue.add(next);

}

}

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of vertices: ");

int V = scanner.nextInt();

BestFirstSearch g = new BestFirstSearch(V);

System.out.print("Enter the number of edges: ");

int E = scanner.nextInt();

System.out.println("Enter the edges: ");

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

int v = scanner.nextInt();

int w = scanner.nextInt();

int cost = scanner.nextInt();

g.addEdge(v, new Node(w, cost));

}

System.out.print("Enter the starting vertex: ");

int s = scanner.nextInt();

System.out.print("Enter the goal vertex: ");

int goal = scanner.nextInt();

System.out.print("Best-First Search traversal starting from vertex " + s + ": ");

g.bestFirstSearch(s, goal);

scanner.close();

}

}

class Node {

int vertex;

int cost;

public Node(int v, int c) {

vertex = v;

cost = c;

}

}

class NodeComparator implements Comparator<Node> {

public int compare(Node n1, Node n2) {

return n1.cost - n2.cost;

}

}