8 puzzle

#include <bits/stdc++.h>

using namespace std;

#define N 3

struct Node{

Node\* parent;

int mat[N][N];

int x, y;

int cost;

int level;

};

int printMatrix(int mat[N][N])

{

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

{

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

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

printf("\n");

}

}

Node\* newNode(int mat[N][N], int x, int y, int newX,int newY, int level, Node\* parent){

Node\* node = new Node;

node->parent = parent;

memcpy(node->mat, mat, sizeof node->mat);

swap(node->mat[x][y], node->mat[newX][newY]);

node->cost = INT\_MAX;

node->level = level;

node->x = newX;

node->y = newY;

return node;

}

int row[] = { 1, 0, -1, 0 };

int col[] = { 0, -1, 0, 1 };

int calculateCost(int initial[N][N], int final[N][N])

{

int count = 0;

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

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

if (initial[i][j] && initial[i][j] != final[i][j])

count++;

return count;

}

int isSafe(int x, int y)

{

return (x >= 0 && x < N && y >= 0 && y < N);

}

void printPath(Node\* root)

{

if (root == NULL)

return;

printPath(root->parent);

printMatrix(root->mat);

printf("\n");

}

struct comp

{

bool operator()(const Node\* lhs, const Node\* rhs) const

{

return (lhs->cost + lhs->level) > (rhs->cost + rhs->level);

}

};

void solve(int initial[N][N], int x, int y,int final[N][N]){

priority\_queue<Node\*, std::vector<Node\*>, comp> pq;

Node\* root = newNode(initial, x, y, x, y, 0, NULL);

root->cost = calculateCost(initial, final);

pq.push(root);

while (!pq.empty()){

Node\* min = pq.top();

pq.pop();

if (min->cost == 0){

printPath(min);

return;

}

for (int i = 0; i < 4; i++){

if (isSafe(min->x + row[i], min->y + col[i])){

Node\* child = newNode(min->mat, min->x,

min->y, min->x + row[i],

min->y + col[i],

min->level + 1, min);

child->cost = calculateCost(child->mat, final);

pq.push(child);

}

}

}

}

int main(){

int initial[N][N] =

{

{1, 2, 3},

{5, 6, 0},

{7, 8, 4}

};

int final[N][N] ={

{1, 2, 3},

{5, 8, 6},

{0, 7, 4}

};

int x = 1, y = 2;

solve(initial, x, y, final);

return 0;

}

Bfs

#include <bits/stdc++.h>

using namespace std;

//DFS Function

void BFS(int start , int v, vector<int> adj[], vector<bool>& visited){

queue<int> q;

q.push(start);

visited[start] = true;

while(!q.empty())

{

int curr = q.front();

q.pop();

cout<<curr<<" ";

for(int i=0;i<adj[curr].size();i++)

if(visited[adj[curr][i]] == false)

{

q.push(adj[curr][i]);

visited[adj[curr][i]] = true;

}

}

}

int main()

{

int v,e;

cin>>v>>e;

//creating a 2d matrix to form a adjagency matrix

vector<int> adj[v];

//adding an edge

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

{

int x,y;

cin>>x>>y;

//undirected graph

adj[x].push\_back(y);

adj[y].push\_back(x);

//directedgraph

// adj[x].push\_back(y);

}

// array to get if the node is visited oer not

vector<bool> visited(v, false);

//DFS function

BFS(0, v, adj, visited);

}

Crypto arithmetic

#include <bits/stdc++.h>

using namespace std;

vector<int> use(10);

struct node

{

char c;

int v;

};

int check(node\* nodeArr, const int count, string s1,

string s2, string s3)

{

int val1 = 0, val2 = 0, val3 = 0, m = 1, j, i;

for (i = s1.length() - 1; i >= 0; i--)

{

char ch = s1[i];

for (j = 0; j < count; j++)

if (nodeArr[j].c == ch)

break;

val1 += m \* nodeArr[j].v;

m \*= 10;

}

m = 1;

for (i = s2.length() - 1; i >= 0; i--)

{

char ch = s2[i];

for (j = 0; j < count; j++)

if (nodeArr[j].c == ch)

break;

val2 += m \* nodeArr[j].v;

m \*= 10;

}

m = 1;

for (i = s3.length() - 1; i >= 0; i--)

{

char ch = s3[i];

for (j = 0; j < count; j++)

if (nodeArr[j].c == ch)

break;

val3 += m \* nodeArr[j].v;

m \*= 10;

}

if (val3 == (val1 + val2))

return 1;

return 0;

}

bool permutation(const int count, node\* nodeArr, int n,string s1, string s2, string s3){

if (n == count - 1){

for (int i = 0; i < 10; i++){

if (use[i] == 0){

nodeArr[n].v = i;

if (check(nodeArr, count, s1, s2, s3) == 1){

cout << "\nSolution found: ";

for (int j = 0; j < count; j++)

cout << " " << nodeArr[j].c << " = "

<< nodeArr[j].v;

return true;

}

}

}

return false;

}

for (int i = 0; i < 10; i++){

if (use[i] == 0){

nodeArr[n].v = i;

use[i] = 1;

if (permutation(count, nodeArr, n + 1, s1, s2, s3))

return true;

use[i] = 0;

}

}

return false;

}

bool solveCryptographic(string s1, string s2,string s3){

int count = 0;

int l1 = s1.length();

int l2 = s2.length();

int l3 = s3.length();

vector<int> freq(26);

for (int i = 0; i < l1; i++)

++freq[s1[i] - 'A'];

for (int i = 0; i < l2; i++)

++freq[s2[i] - 'A'];

for (int i = 0; i < l3; i++)

++freq[s3[i] - 'A'];

for (int i = 0; i < 26; i++)

if (freq[i] > 0)

count++;

if (count > 10)

{

cout << "Invalid strings";

return 0;

}

node nodeArr[count];

for (int i = 0, j = 0; i < 26; i++)

{

if (freq[i] > 0)

{

nodeArr[j].c = char(i + 'A');

j++;

}

}

return permutation(count, nodeArr, 0, s1, s2, s3);

}

int main()

{

string s1 = "SEND";

string s2 = "MORE";

string s3 = "MONEY";

if (solveCryptographic(s1, s2, s3) == false)

cout << "No solution";

return 0;

}

Dfs

#include <bits/stdc++.h>

using namespace std;

//DFS Function

void DFS(int start , int v, int \*\*adj, vector<bool>& visited){

cout << start << " ";

visited[start] = true;

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

if (adj[start][i] == 1 && (!visited[i])) {

DFS(i, v, adj, visited);

}

}

}

int main()

{

int v,e;

//v -> no of vertices

//e -> no of edges

cin>>v>>e;

//creating a 2d matrix to form a adjagency matrix

int \*adj = new int[v];

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

adj[i] = new int[v];

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

adj[i][j] = 0;

}

}

//adding an edge

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

{

int x,y;

cin>>x>>y;

//undirected graph

adj[x][y] = 1;

adj[y][x] = 1;

//directedgraph

//adj[x][y] = 1;

}

// array to get if the node is visited oer not

vector<bool> visited(v, false);

//DFS function

DFS(0, v, adj, visited);

}

Graph coloring

#include<bits/stdc++.h>

using namespace std;

#define V 4

void printSolution(int color[]);

bool isSafe(bool graph[V][V], int color[]){

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

for (int j = i + 1; j < V; j++)

if (graph[i][j] && color[j] == color[i])

return false;

return true;

}

bool graphColoring(bool graph[V][V], int m, int i,int color[V]){

if (i == V) {

if (isSafe(graph, color)) {

printSolution(color);

return true;

}

return false;

}

for (int j = 1; j <= m; j++) {

color[i] = j;

if (graphColoring(graph, m, i + 1, color))

return true;

color[i] = 0;

}

return false;

}

void printSolution(int color[])

{

cout << "Solution Exists:" " Following are the assigned colors \n";

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

cout << " " << color[i];

cout << "\n";

}

int main()

{

bool graph[V][V] = {

{ 0, 1, 1, 1 },

{ 1, 0, 1, 0 },

{ 1, 1, 0, 1 },

{ 1, 0, 1, 0 },

};

int m = 3;

int color[V];

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

color[i] = 0;

if (!graphColoring(graph, m, 0, color))

cout << "Solution does not exist";

return 0;

}

N queen

#include <bits/stdc++.h>

#define N 4

using namespace std;

void printSolution(int board[N][N])

{

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

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

cout << " " << board[i][j] << " ";

printf("\n");

}

}

bool isSafe(int board[N][N], int row, int col){

int i, j;

for (i = 0; i < col; i++)

if (board[row][i])

return false;

for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j])

return false;

for (i = row, j = col; j >= 0 && i < N; i++, j--)

if (board[i][j])

return false;

return true;

}

bool solveNQUtil(int board[N][N], int col)

{

if (col >= N)

return true;

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

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

bool solveNQ()

{

int board[N][N] = { { 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 } };

if (solveNQUtil(board, 0) == false) {

cout << "Solution does not exist";

return false;

}

printSolution(board);

return true;

}

int main()

{

solveNQ();

return 0;

}

Tic tac toe

#include<bits/stdc++.h>

using namespace std;

char board [3][3];

int counter = 0;

char turn = 'x';

int choice = 0;

pair<int, int> getrownumber(int x){

switch(x){

case 1:

return {0,0};

case 2:

return {0,1};

case 3:

return {0,2};

case 4:

return {1,0};

case 5:

return {1,1};

case 6:

return {1,2};

case 7:

return {2,0};

case 8:

return {2,1};

case 9:

return {2,2};

default:

return {-1,-1};

}

}

bool checkwin(int i, int j){

if(board[i][0] == turn && board[i][1] == turn && board[i][2] == turn)

return true;

if(board[0][j] == turn && board[1][j] == turn && board[2][j] == turn)

return true;

if(board[0][0] == turn && board[1][1] == turn && board[2][2] == turn)

return true;

if(board[0][2] == turn && board[1][1] == turn && board[2][0] == turn)

return true;

return false;

}

bool game(){

if(turn == 'x')

cout<<"Player 1 Enter Value: \n";

if(turn == 'o')

cout<<"player 2 Enter Value: \n";

cout<<"Enter Your Choice: ";

cin>>choice;

pair <int, int> p = getrownumber(choice);

int i=p.first, j=p.second;

if(board[i][j] != '-'){

cout<<"Already taken number /n";

game();

}

else{

board[i][j] = turn;

if(checkwin(i,j) == true)

return true;

if(turn == 'x')

turn = 'o';

else if(turn == 'o')

turn = 'x';

}

return false;

}

void printBoard(){

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

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

cout<<board[i][j]<<" ";

cout<<endl;

}

}

int main() {

bool isdraw = false;

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

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

board[i][j] = '-';

cout<<"New game"<<endl;

printBoard();

while(true){

if(counter >= 9){

cout<<"Match is Draw";

isdraw = true;

break;

}

bool istrue = game();

printBoard();

if(istrue)

break;

}

if(!isdraw)

cout<<"Player "<<turn<<" wins"<<endl;

return 0;

}

Tsp

#include <bits/stdc++.h>

using namespace std;

#define V 4

int travllingSalesmanProblem(int graph[][V], int s)

{

vector<int> vertex;

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

if (i != s)

vertex.push\_back(i);

int min\_path = INT\_MAX;

do {

int current\_pathweight = 0;

int k = s;

for (int i = 0; i < vertex.size(); i++) {

current\_pathweight += graph[k][vertex[i]];

k = vertex[i];

}

current\_pathweight += graph[k][s];

min\_path = min(min\_path, current\_pathweight);

} while (

next\_permutation(vertex.begin(), vertex.end()));

return min\_path;

}

int main()

{

int graph[][V] = { { 0, 10, 15, 20 },

{ 10, 0, 35, 25 },

{ 15, 35, 0, 30 },

{ 20, 25, 30, 0 } };

int s = 0;

cout << travllingSalesmanProblem(graph, s) << endl;

return 0;

}

Water jug

#include <bits/stdc++.h>

using namespace std;

typedef pair<int,int> pii;

void printpath(map<pii,pii>mp ,pii u)

{

if(u.first==0 &&u.second==0)

{

cout<<0<<" "<<0<<endl;

return ;

}

printpath(mp,mp[u]);

cout<<u.first<<" "<<u.second<<endl;

}

void BFS(int a ,int b, int target)

{

map<pii, int>m;

bool isSolvable =false;

vector<tuple<int ,int ,int>>path;

map<pii, pii>mp;

queue<pii>q;

q.push(make\_pair(0,0));

while(!q.empty())

{

auto u =q.front();

// cout<<u.first<<" "<<u.second<<endl;

q.pop();

if(m[u]==1)

continue;

if ((u.first > a || u.second > b || u.first < 0 || u.second < 0))

continue;

// cout<<u.first<<" "<<u.second<<endl;

m[{u.first,u.second}]=1;

if(u.first == target || u.second==target)

{

isSolvable = true;

printpath(mp,u);

if (u.first == target) {

if (u.second != 0)

cout<<u.first<<" "<<0<<endl;

}

else {

if (u.first != 0)

cout<<0<<" "<<u.second<<endl;

}

return;

}

// completely fill the jug 2

if(m[{u.first,b}]!=1)

{q.push({u.first,b});

mp[{u.first,b}]=u;}

// completely fill the jug 1

if(m[{a,u.second}]!=1)

{ q.push({a,u.second});

mp[{a,u.second}]=u;}

//transfer jug 1 -> jug 2

int d = b - u.second;

if(u.first >= d)

{

int c = u.first - d;

if(m[{c,b}]!=1)

{q.push({c,b});

mp[{c,b}]=u;}

}

else

{

int c = u.first + u.second;

if(m[{0,c}]!=1)

{q.push({0,c});

mp[{0,c}]=u;}

}

//transfer jug 2 -> jug 1

d = a - u.first;

if(u.second >= d)

{

int c = u.second - d;

if(m[{a,c}]!=1)

{q.push({a,c});

mp[{a,c}]=u;}

}

else

{

int c = u.first + u.second;

if(m[{c,0}]!=1)

{q.push({c,0});

mp[{c,0}]=u;}

}

// empty the jug 2

if(m[{u.first,0}]!=1)

{ q.push({u.first,0});

mp[{u.first,0}]=u;}

// empty the jug 1

if(m[{0,u.second}]!=1)

{q.push({0,u.second});

mp[{0,u.second}]=u;}

}

if (!isSolvable)

cout << "No solution";

}

int main()

{

int Jug1 = 4, Jug2 = 3, target = 2;

cout << "Path from initial state "

"to solution state ::\n";

BFS(Jug1, Jug2, target);

return 0;

}

Implimentation of bfs

// BFS algorithm in C++

#include <iostream>

#include <list>

using namespace std;

class Graph {

int numVertices;

list<int>\* adjLists;

bool\* visited;

public:

Graph(int vertices);

void addEdge(int src, int dest);

void BFS(int startVertex);

};

// Create a graph with given vertices,

// and maintain an adjacency list

Graph::Graph(int vertices) {

numVertices = vertices;

adjLists = new list<int>[vertices];

}

// Add edges to the graph

void Graph::addEdge(int src, int dest) {

adjLists[src].push\_back(dest);

adjLists[dest].push\_back(src);

}

// BFS algorithm

void Graph::BFS(int startVertex) {

visited = new bool[numVertices];

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

visited[i] = false;

list<int> queue;

visited[startVertex] = true;

queue.push\_back(startVertex);

list<int>::iterator i;

while (!queue.empty()) {

int currVertex = queue.front();

cout << "Visited " << currVertex << " ";

queue.pop\_front();

for (i = adjLists[currVertex].begin(); i != adjLists[currVertex].end(); ++i) {

int adjVertex = \*i;

if (!visited[adjVertex]) {

visited[adjVertex] = true;

queue.push\_back(adjVertex);

}

}

}

}

int main() {

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

g.BFS(2);

return 0;

}

A star algo

#include <bits/stdc++.h>

using namespace std;

#define ROW 9

#define COL 10

// Creating a shortcut for int, int pair type

typedef pair<int, int> Pair;

// Creating a shortcut for pair<int, pair<int, int>> type

typedef pair<double, pair<int, int> > pPair;

// A structure to hold the necessary parameters

struct cell {

// Row and Column index of its parent

// Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1

int parent\_i, parent\_j;

// f = g + h

double f, g, h;

};

// A Utility Function to check whether given cell (row, col)

// is a valid cell or not.

bool isValid(int row, int col)

{

// Returns true if row number and column number

// is in range

return (row >= 0) && (row < ROW) && (col >= 0)

&& (col < COL);

}

// A Utility Function to check whether the given cell is

// blocked or not

bool isUnBlocked(int grid[][COL], int row, int col)

{

// Returns true if the cell is not blocked else false

if (grid[row][col] == 1)

return (true);

else

return (false);

}

// A Utility Function to check whether destination cell has

// been reached or not

bool isDestination(int row, int col, Pair dest)

{

if (row == dest.first && col == dest.second)

return (true);

else

return (false);

}

// A Utility Function to calculate the 'h' heuristics.

double calculateHValue(int row, int col, Pair dest)

{

// Return using the distance formula

return ((double)sqrt(

(row - dest.first) \* (row - dest.first)

+ (col - dest.second) \* (col - dest.second)));

}

// A Utility Function to trace the path from the source

// to destination

void tracePath(cell cellDetails[][COL], Pair dest)

{

printf("\nThe Path is ");

int row = dest.first;

int col = dest.second;

stack<Pair> Path;

while (!(cellDetails[row][col].parent\_i == row

&& cellDetails[row][col].parent\_j == col)) {

Path.push(make\_pair(row, col));

int temp\_row = cellDetails[row][col].parent\_i;

int temp\_col = cellDetails[row][col].parent\_j;

row = temp\_row;

col = temp\_col;

}

Path.push(make\_pair(row, col));

while (!Path.empty()) {

pair<int, int> p = Path.top();

Path.pop();

printf("-> (%d,%d) ", p.first, p.second);

}

return;

}

// A Function to find the shortest path between

// a given source cell to a destination cell according

// to A\* Search Algorithm

void aStarSearch(int grid[][COL], Pair src, Pair dest)

{

// If the source is out of range

if (isValid(src.first, src.second) == false) {

printf("Source is invalid\n");

return;

}

// If the destination is out of range

if (isValid(dest.first, dest.second) == false) {

printf("Destination is invalid\n");

return;

}

// Either the source or the destination is blocked

if (isUnBlocked(grid, src.first, src.second) == false

|| isUnBlocked(grid, dest.first, dest.second)

== false) {

printf("Source or the destination is blocked\n");

return;

}

// If the destination cell is the same as source cell

if (isDestination(src.first, src.second, dest)

== true) {

printf("We are already at the destination\n");

return;

}

// Create a closed list and initialise it to false which

// means that no cell has been included yet This closed

// list is implemented as a boolean 2D array

bool closedList[ROW][COL];

memset(closedList, false, sizeof(closedList));

// Declare a 2D array of structure to hold the details

// of that cell

cell cellDetails[ROW][COL];

int i, j;

for (i = 0; i < ROW; i++) {

for (j = 0; j < COL; j++) {

cellDetails[i][j].f = FLT\_MAX;

cellDetails[i][j].g = FLT\_MAX;

cellDetails[i][j].h = FLT\_MAX;

cellDetails[i][j].parent\_i = -1;

cellDetails[i][j].parent\_j = -1;

}

}

// Initialising the parameters of the starting node

i = src.first, j = src.second;

cellDetails[i][j].f = 0.0;

cellDetails[i][j].g = 0.0;

cellDetails[i][j].h = 0.0;

cellDetails[i][j].parent\_i = i;

cellDetails[i][j].parent\_j = j;

/\*

Create an open list having information as-

<f, <i, j>>

where f = g + h,

and i, j are the row and column index of that cell

Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1

This open list is implemented as a set of pair of

pair.\*/

set<pPair> openList;

// Put the starting cell on the open list and set its

// 'f' as 0

openList.insert(make\_pair(0.0, make\_pair(i, j)));

// We set this boolean value as false as initially

// the destination is not reached.

bool foundDest = false;

while (!openList.empty()) {

pPair p = \*openList.begin();

// Remove this vertex from the open list

openList.erase(openList.begin());

// Add this vertex to the closed list

i = p.second.first;

j = p.second.second;

closedList[i][j] = true;

/\*

Generating all the 8 successor of this cell

N.W N N.E

\ | /

\ | /

W----Cell----E

/ | \

/ | \

S.W S S.E

Cell-->Popped Cell (i, j)

N --> North (i-1, j)

S --> South (i+1, j)

E --> East (i, j+1)

W --> West (i, j-1)

N.E--> North-East (i-1, j+1)

N.W--> North-West (i-1, j-1)

S.E--> South-East (i+1, j+1)

S.W--> South-West (i+1, j-1)\*/

// To store the 'g', 'h' and 'f' of the 8 successors

double gNew, hNew, fNew;

//----------- 1st Successor (North) ------------

// Only process this cell if this is a valid one

if (isValid(i - 1, j) == true) {

// If the destination cell is the same as the

// current successor

if (isDestination(i - 1, j, dest) == true) {

// Set the Parent of the destination cell

cellDetails[i - 1][j].parent\_i = i;

cellDetails[i - 1][j].parent\_j = j;

printf("The destination cell is found\n");

tracePath(cellDetails, dest);

foundDest = true;

return;

}

// If the successor is already on the closed

// list or if it is blocked, then ignore it.

// Else do the following

else if (closedList[i - 1][j] == false

&& isUnBlocked(grid, i - 1, j)

== true) {

gNew = cellDetails[i][j].g + 1.0;

hNew = calculateHValue(i - 1, j, dest);

fNew = gNew + hNew;

// If it isn’t on the open list, add it to

// the open list. Make the current square

// the parent of this square. Record the

// f, g, and h costs of the square cell

// OR

// If it is on the open list already, check

// to see if this path to that square is

// better, using 'f' cost as the measure.

if (cellDetails[i - 1][j].f == FLT\_MAX

|| cellDetails[i - 1][j].f > fNew) {

openList.insert(make\_pair(

fNew, make\_pair(i - 1, j)));

// Update the details of this cell

cellDetails[i - 1][j].f = fNew;

cellDetails[i - 1][j].g = gNew;

cellDetails[i - 1][j].h = hNew;

cellDetails[i - 1][j].parent\_i = i;

cellDetails[i - 1][j].parent\_j = j;

}

}

}

//----------- 2nd Successor (South) ------------

// Only process this cell if this is a valid one

if (isValid(i + 1, j) == true) {

// If the destination cell is the same as the

// current successor

if (isDestination(i + 1, j, dest) == true) {

// Set the Parent of the destination cell

cellDetails[i + 1][j].parent\_i = i;

cellDetails[i + 1][j].parent\_j = j;

printf("The destination cell is found\n");

tracePath(cellDetails, dest);

foundDest = true;

return;

}

// If the successor is already on the closed

// list or if it is blocked, then ignore it.

// Else do the following

else if (closedList[i + 1][j] == false

&& isUnBlocked(grid, i + 1, j)

== true) {

gNew = cellDetails[i][j].g + 1.0;

hNew = calculateHValue(i + 1, j, dest);

fNew = gNew + hNew;

// If it isn’t on the open list, add it to

// the open list. Make the current square

// the parent of this square. Record the

// f, g, and h costs of the square cell

// OR

// If it is on the open list already, check

// to see if this path to that square is

// better, using 'f' cost as the measure.

if (cellDetails[i + 1][j].f == FLT\_MAX

|| cellDetails[i + 1][j].f > fNew) {

openList.insert(make\_pair(

fNew, make\_pair(i + 1, j)));

// Update the details of this cell

cellDetails[i + 1][j].f = fNew;

cellDetails[i + 1][j].g = gNew;

cellDetails[i + 1][j].h = hNew;

cellDetails[i + 1][j].parent\_i = i;

cellDetails[i + 1][j].parent\_j = j;

}

}

}

//----------- 3rd Successor (East) ------------

// Only process this cell if this is a valid one

if (isValid(i, j + 1) == true) {

// If the destination cell is the same as the

// current successor

if (isDestination(i, j + 1, dest) == true) {

// Set the Parent of the destination cell

cellDetails[i][j + 1].parent\_i = i;

cellDetails[i][j + 1].parent\_j = j;

printf("The destination cell is found\n");

tracePath(cellDetails, dest);

foundDest = true;

return;

}

// If the successor is already on the closed

// list or if it is blocked, then ignore it.

// Else do the following

else if (closedList[i][j + 1] == false

&& isUnBlocked(grid, i, j + 1)

== true) {

gNew = cellDetails[i][j].g + 1.0;

hNew = calculateHValue(i, j + 1, dest);

fNew = gNew + hNew;

// If it isn’t on the open list, add it to

// the open list. Make the current square

// the parent of this square. Record the

// f, g, and h costs of the square cell

// OR

// If it is on the open list already, check

// to see if this path to that square is

// better, using 'f' cost as the measure.

if (cellDetails[i][j + 1].f == FLT\_MAX

|| cellDetails[i][j + 1].f > fNew) {

openList.insert(make\_pair(

fNew, make\_pair(i, j + 1)));

// Update the details of this cell

cellDetails[i][j + 1].f = fNew;

cellDetails[i][j + 1].g = gNew;

cellDetails[i][j + 1].h = hNew;

cellDetails[i][j + 1].parent\_i = i;

cellDetails[i][j + 1].parent\_j = j;

}

}

}

//----------- 4th Successor (West) ------------

// Only process this cell if this is a valid one

if (isValid(i, j - 1) == true) {

// If the destination cell is the same as the

// current successor

if (isDestination(i, j - 1, dest) == true) {

// Set the Parent of the destination cell

cellDetails[i][j - 1].parent\_i = i;

cellDetails[i][j - 1].parent\_j = j;

printf("The destination cell is found\n");

tracePath(cellDetails, dest);

foundDest = true;

return;

}

// If the successor is already on the closed

// list or if it is blocked, then ignore it.

// Else do the following

else if (closedList[i][j - 1] == false

&& isUnBlocked(grid, i, j - 1)

== true) {

gNew = cellDetails[i][j].g + 1.0;

hNew = calculateHValue(i, j - 1, dest);

fNew = gNew + hNew;

// If it isn’t on the open list, add it to

// the open list. Make the current square

// the parent of this square. Record the

// f, g, and h costs of the square cell

// OR

// If it is on the open list already, check

// to see if this path to that square is

// better, using 'f' cost as the measure.

if (cellDetails[i][j - 1].f == FLT\_MAX

|| cellDetails[i][j - 1].f > fNew) {

openList.insert(make\_pair(

fNew, make\_pair(i, j - 1)));

// Update the details of this cell

cellDetails[i][j - 1].f = fNew;

cellDetails[i][j - 1].g = gNew;

cellDetails[i][j - 1].h = hNew;

cellDetails[i][j - 1].parent\_i = i;

cellDetails[i][j - 1].parent\_j = j;

}

}

}

//----------- 5th Successor (North-East)

//------------

// Only process this cell if this is a valid one

if (isValid(i - 1, j + 1) == true) {

// If the destination cell is the same as the

// current successor

if (isDestination(i - 1, j + 1, dest) == true) {

// Set the Parent of the destination cell

cellDetails[i - 1][j + 1].parent\_i = i;

cellDetails[i - 1][j + 1].parent\_j = j;

printf("The destination cell is found\n");

tracePath(cellDetails, dest);

foundDest = true;

return;

}

// If the successor is already on the closed

// list or if it is blocked, then ignore it.

// Else do the following

else if (closedList[i - 1][j + 1] == false

&& isUnBlocked(grid, i - 1, j + 1)

== true) {

gNew = cellDetails[i][j].g + 1.414;

hNew = calculateHValue(i - 1, j + 1, dest);

fNew = gNew + hNew;

// If it isn’t on the open list, add it to

// the open list. Make the current square

// the parent of this square. Record the

// f, g, and h costs of the square cell

// OR

// If it is on the open list already, check

// to see if this path to that square is

// better, using 'f' cost as the measure.

if (cellDetails[i - 1][j + 1].f == FLT\_MAX

|| cellDetails[i - 1][j + 1].f > fNew) {

openList.insert(make\_pair(

fNew, make\_pair(i - 1, j + 1)));

// Update the details of this cell

cellDetails[i - 1][j + 1].f = fNew;

cellDetails[i - 1][j + 1].g = gNew;

cellDetails[i - 1][j + 1].h = hNew;

cellDetails[i - 1][j + 1].parent\_i = i;

cellDetails[i - 1][j + 1].parent\_j = j;

}

}

}

//----------- 6th Successor (North-West)

//------------

// Only process this cell if this is a valid one

if (isValid(i - 1, j - 1) == true) {

// If the destination cell is the same as the

// current successor

if (isDestination(i - 1, j - 1, dest) == true) {

// Set the Parent of the destination cell

cellDetails[i - 1][j - 1].parent\_i = i;

cellDetails[i - 1][j - 1].parent\_j = j;

printf("The destination cell is found\n");

tracePath(cellDetails, dest);

foundDest = true;

return;

}

// If the successor is already on the closed

// list or if it is blocked, then ignore it.

// Else do the following

else if (closedList[i - 1][j - 1] == false

&& isUnBlocked(grid, i - 1, j - 1)

== true) {

gNew = cellDetails[i][j].g + 1.414;

hNew = calculateHValue(i - 1, j - 1, dest);

fNew = gNew + hNew;

// If it isn’t on the open list, add it to

// the open list. Make the current square

// the parent of this square. Record the

// f, g, and h costs of the square cell

// OR

// If it is on the open list already, check

// to see if this path to that square is

// better, using 'f' cost as the measure.

if (cellDetails[i - 1][j - 1].f == FLT\_MAX

|| cellDetails[i - 1][j - 1].f > fNew) {

openList.insert(make\_pair(

fNew, make\_pair(i - 1, j - 1)));

// Update the details of this cell

cellDetails[i - 1][j - 1].f = fNew;

cellDetails[i - 1][j - 1].g = gNew;

cellDetails[i - 1][j - 1].h = hNew;

cellDetails[i - 1][j - 1].parent\_i = i;

cellDetails[i - 1][j - 1].parent\_j = j;

}

}

}

//----------- 7th Successor (South-East)

//------------

// Only process this cell if this is a valid one

if (isValid(i + 1, j + 1) == true) {

// If the destination cell is the same as the

// current successor

if (isDestination(i + 1, j + 1, dest) == true) {

// Set the Parent of the destination cell

cellDetails[i + 1][j + 1].parent\_i = i;

cellDetails[i + 1][j + 1].parent\_j = j;

printf("The destination cell is found\n");

tracePath(cellDetails, dest);

foundDest = true;

return;

}

// If the successor is already on the closed

// list or if it is blocked, then ignore it.

// Else do the following

else if (closedList[i + 1][j + 1] == false

&& isUnBlocked(grid, i + 1, j + 1)

== true) {

gNew = cellDetails[i][j].g + 1.414;

hNew = calculateHValue(i + 1, j + 1, dest);

fNew = gNew + hNew;

// If it isn’t on the open list, add it to

// the open list. Make the current square

// the parent of this square. Record the

// f, g, and h costs of the square cell

// OR

// If it is on the open list already, check

// to see if this path to that square is

// better, using 'f' cost as the measure.

if (cellDetails[i + 1][j + 1].f == FLT\_MAX

|| cellDetails[i + 1][j + 1].f > fNew) {

openList.insert(make\_pair(

fNew, make\_pair(i + 1, j + 1)));

// Update the details of this cell

cellDetails[i + 1][j + 1].f = fNew;

cellDetails[i + 1][j + 1].g = gNew;

cellDetails[i + 1][j + 1].h = hNew;

cellDetails[i + 1][j + 1].parent\_i = i;

cellDetails[i + 1][j + 1].parent\_j = j;

}

}

}

//----------- 8th Successor (South-West)

//------------

// Only process this cell if this is a valid one

if (isValid(i + 1, j - 1) == true) {

// If the destination cell is the same as the

// current successor

if (isDestination(i + 1, j - 1, dest) == true) {

// Set the Parent of the destination cell

cellDetails[i + 1][j - 1].parent\_i = i;

cellDetails[i + 1][j - 1].parent\_j = j;

printf("The destination cell is found\n");

tracePath(cellDetails, dest);

foundDest = true;

return;

}

// If the successor is already on the closed

// list or if it is blocked, then ignore it.

// Else do the following

else if (closedList[i + 1][j - 1] == false

&& isUnBlocked(grid, i + 1, j - 1)

== true) {

gNew = cellDetails[i][j].g + 1.414;

hNew = calculateHValue(i + 1, j - 1, dest);

fNew = gNew + hNew;

// If it isn’t on the open list, add it to

// the open list. Make the current square

// the parent of this square. Record the

// f, g, and h costs of the square cell

// OR

// If it is on the open list already, check

// to see if this path to that square is

// better, using 'f' cost as the measure.

if (cellDetails[i + 1][j - 1].f == FLT\_MAX

|| cellDetails[i + 1][j - 1].f > fNew) {

openList.insert(make\_pair(

fNew, make\_pair(i + 1, j - 1)));

// Update the details of this cell

cellDetails[i + 1][j - 1].f = fNew;

cellDetails[i + 1][j - 1].g = gNew;

cellDetails[i + 1][j - 1].h = hNew;

cellDetails[i + 1][j - 1].parent\_i = i;

cellDetails[i + 1][j - 1].parent\_j = j;

}

}

}

}

// When the destination cell is not found and the open

// list is empty, then we conclude that we failed to

// reach the destination cell. This may happen when the

// there is no way to destination cell (due to

// blockages)

if (foundDest == false)

printf("Failed to find the Destination Cell\n");

return;

}

// Driver program to test above function

int main()

{

/\* Description of the Grid-

1--> The cell is not blocked

0--> The cell is blocked \*/

int grid[ROW][COL]

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

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

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

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

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

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

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

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

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

// Source is the left-most bottom-most corner

Pair src = make\_pair(8, 0);

// Destination is the left-most top-most corner

Pair dest = make\_pair(0, 0);

aStarSearch(grid, src, dest);

return (0);

}

Fuzzy logic calculator

#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<<"\nImput 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;

}

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\_\_':

f1 **=** 'Q(a, g(x, a), f(y))'

f2 **=** 'Q(a, g(f(b), a), x)'

*# f1 = input('f1 : ')*

*# f2 = input('f2 : ')*

result **=** unify(f1, f2)

**if** **not** result:

print('The process of Unification failed!')

**else**:

print('The process of Unification successful!')

print(result)

resolution

import copy

import time

class Parameter:

variable\_count = 1

def \_\_init\_\_(self, name=None):

if name:

self.type = &quot;Constant&quot;

self.name = name

else:

self.type = &quot;Variable&quot;

self.name = &quot;v&quot; + str(Parameter.variable\_count)

Parameter.variable\_count += 1

def isConstant(self):

return self.type == &quot;Constant&quot;

def unify(self, type\_, name):

self.type = type\_

self.name = name

def \_\_eq\_\_(self, other):

return self.name == other.name

def \_\_str\_\_(self):

return self.name

class Predicate:

def \_\_init\_\_(self, name, params):

self.name = name

self.params = params

def \_\_eq\_\_(self, other):

return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

def \_\_str\_\_(self):

return self.name + &quot;(&quot; + &quot;,&quot;.join(str(x) for x in self.params) + &quot;)&quot;

def getNegatedPredicate(self):

return Predicate(negatePredicate(self.name), self.params)

class Sentence:

sentence\_count = 0

def \_\_init\_\_(self, string):

self.sentence\_index = Sentence.sentence\_count

Sentence.sentence\_count += 1

self.predicates = []

self.variable\_map = {}

local = {}

for predicate in string.split(&quot;|&quot;):

name = predicate[:predicate.find(&quot;(&quot;)]

params = []

for param in predicate[predicate.find(&quot;(&quot;) + 1: predicate.find(&quot;)&quot;)].split(&quot;,&quot;):

if param[0].islower():

if param not in local: # Variable

local[param] = Parameter()

self.variable\_map[local[param].name] = local[param]

new\_param = local[param]

else:

new\_param = Parameter(param)

self.variable\_map[param] = new\_param

params.append(new\_param)

self.predicates.append(Predicate(name, params))

def getPredicates(self):

return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):

return [predicate for predicate in self.predicates if predicate.name == name]

def removePredicate(self, predicate):

self.predicates.remove(predicate)

for key, val in self.variable\_map.items():

if not val:

self.variable\_map.pop(key)

def containsVariable(self):

return any(not param.isConstant() for param in self.variable\_map.values())

def \_\_eq\_\_(self, other):

if len(self.predicates) == 1 and self.predicates[0] == other:

return True

return False

def \_\_str\_\_(self):

return &quot;&quot;.join([str(predicate) for predicate in self.predicates])

class KB:

def \_\_init\_\_(self, inputSentences):

self.inputSentences = [x.replace(&quot; &quot;, &quot;&quot;) for x in inputSentences]

self.sentences = []

self.sentence\_map = {}

def prepareKB(self):

self.convertSentencesToCNF()

for sentence\_string in self.inputSentences:

sentence = Sentence(sentence\_string)

for predicate in sentence.getPredicates():

self.sentence\_map[predicate] = self.sentence\_map.get(

predicate, []) + [sentence]

def convertSentencesToCNF(self):

for sentenceIdx in range(len(self.inputSentences)):

# Do negation of the Premise and add them as literal

if &quot;=&gt;&quot; in self.inputSentences[sentenceIdx]:

self.inputSentences[sentenceIdx] = negateAntecedent(

self.inputSentences[sentenceIdx])

def askQueries(self, queryList):

results = []

for query in queryList:

negatedQuery = Sentence(negatePredicate(query.replace(&quot; &quot;, &quot;&quot;)))

negatedPredicate = negatedQuery.predicates[0]

prev\_sentence\_map = copy.deepcopy(self.sentence\_map)

self.sentence\_map[negatedPredicate.name] = self.sentence\_map.get(

negatedPredicate.name, []) + [negatedQuery]

self.timeLimit = time.time() + 40

try:

result = self.resolve([negatedPredicate], [

False]\*(len(self.inputSentences) + 1))

except:

result = False

self.sentence\_map = prev\_sentence\_map

if result:

results.append(&quot;TRUE&quot;)

else:

results.append(&quot;FALSE&quot;)

return results

def resolve(self, queryStack, visited, depth=0):

if time.time() &gt; self.timeLimit:

raise Exception

if queryStack:

query = queryStack.pop(-1)

negatedQuery = query.getNegatedPredicate()

queryPredicateName = negatedQuery.name

if queryPredicateName not in self.sentence\_map:

return False

else:

queryPredicate = negatedQuery

for kb\_sentence in self.sentence\_map[queryPredicateName]:

if not visited[kb\_sentence.sentence\_index]:

for kbPredicate in kb\_sentence.findPredicates(queryPredicateName):

canUnify, substitution = performUnification(

copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

if canUnify:

newSentence = copy.deepcopy(kb\_sentence)

newSentence.removePredicate(kbPredicate)

newQueryStack = copy.deepcopy(queryStack)

if substitution:

for old, new in substitution.items():

if old in newSentence.variable\_map:

parameter = newSentence.variable\_map[old]

newSentence.variable\_map.pop(old)

parameter.unify(

&quot;Variable&quot; if new[0].islower() else &quot;Constant&quot;, new)

newSentence.variable\_map[new] = parameter

for predicate in newQueryStack:

for index, param in enumerate(predicate.params):

if param.name in substitution:

new = substitution[param.name]

predicate.params[index].unify(

&quot;Variable&quot; if new[0].islower() else &quot;Constant&quot;, new)

for predicate in newSentence.predicates:

newQueryStack.append(predicate)

new\_visited = copy.deepcopy(visited)

if kb\_sentence.containsVariable() and len(kb\_sentence.predicates) &gt; 1:

new\_visited[kb\_sentence.sentence\_index] = True

if self.resolve(newQueryStack, new\_visited, depth + 1):

return True

return False

return True

def performUnification(queryPredicate, kbPredicate):

substitution = {}

if queryPredicate == kbPredicate:

return True, {}

else:

for query, kb in zip(queryPredicate.params, kbPredicate.params):

if query == kb:

continue

if kb.isConstant():

if not query.isConstant():

if query.name not in substitution:

substitution[query.name] = kb.name

elif substitution[query.name] != kb.name:

return False, {}

query.unify(&quot;Constant&quot;, kb.name)

else:

return False, {}

else:

if not query.isConstant():

if kb.name not in substitution:

substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {}

kb.unify(&quot;Variable&quot;, query.name)

else:

if kb.name not in substitution:

substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {}

return True, substitution

def negatePredicate(predicate):

return predicate[1:] if predicate[0] == &quot;~&quot; else &quot;~&quot; + predicate

def negateAntecedent(sentence):

antecedent = sentence[:sentence.find(&quot;=&gt;&quot;)]

premise = []

for predicate in antecedent.split(&quot;&amp;&quot;):

premise.append(negatePredicate(predicate))

premise.append(sentence[sentence.find(&quot;=&gt;&quot;) + 2:])

return &quot;|&quot;.join(premise)

def getInput(filename):

with open(filename, &quot;r&quot;) as file:

noOfQueries = int(file.readline().strip())

inputQueries = [file.readline().strip() for \_ in range(noOfQueries)]

noOfSentences = int(file.readline().strip())

inputSentences = [file.readline().strip()

for \_ in range(noOfSentences)]

return inputQueries, inputSentences

def printOutput(filename, results):

print(results)

with open(filename, &quot;w&quot;) as file:

for line in results:

file.write(line)

file.write(&quot;\n&quot;)

file.close()

if \_\_name\_\_ == &#39;\_\_main\_\_&#39;:

inputQueries\_, inputSentences\_ =

getInput(&#39;/home/ubuntu/environment/255/resolution.txt&#39;)

knowledgeBase = KB(inputSentences\_)

knowledgeBase.prepareKB()

results\_ = knowledgeBase.askQueries(inputQueries\_)

printOutput(&quot;output.txt&quot;, results\_)word

min max algo

#include<iostream>

#include<algorithm>

**using** **namespace** std;

**int** main()

{

// declaring pair to catch the return value

pair<**int**, **int**> mnmx;

// Using minmax(a, b)

mnmx = minmax(53, 23);

// printing minimum and maximum values

cout << "The minimum value obtained is : ";

cout << mnmx.first;

cout << "\nThe maximum value obtained is : ";

cout << mnmx.second ;

// Using minmax((array of elements)

mnmx = minmax({2, 5, 1, 6, 3});

// printing minimum and maximum values.

cout << "\n\nThe minimum value obtained is : ";

cout << mnmx.first;

cout << "\nThe maximum value obtained is : ";

cout << mnmx.second;

**return** 0;

}

Best first

def best\_first\_search(source, target, n):

visited = [0] \* n

visited[source] = True

pq = PriorityQueue()

pq.put((0, source))

while pq.empty() == False:

u = pq.get()[1]

print(u, end=" ")

if u == target:

break

for v, c in graph[u]:

if visited[v] == False:

visited[v] = True

pq.put((c, v))

print()

def addedge(x, y, cost):

graph[x].append((y, cost))

graph[y].append((x, cost))

G = nx.Graph()

v = int(input("Enter the number of nodes: "))

graph = [[] for i in range(v)]

e = int(input("Enter the number of edges: "))

print("Enter the edges along with their weights:")

for i in range(e):

x, y, z = list(map(int, input().split()))

addedge(x, y, z)

G.add\_edge(x, y, weight = z)

source = int(input("Enter the Source Node: "))

target = int(input("Enter the Target/Destination Node: ")) 1

print("\nPath: ", end = "")

best\_first\_search(source, target, v)

Knapsack problem

Problem statement:

Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack. An item if chosen, needs to be picked completely, that is, a fraction of the item cannot be picked.

Code:

**def** knapSack(W, wt, val, n):

**if** n **==** 0 **or** W **==** 0 :

**return** 0

**if** (wt[n**-**1] **>** W):

**return** knapSack(W, wt, val, n**-**1)

**else**:

**return** max(val[n**-**1] **+** knapSack(W**-**wt[n**-**1], wt, val, n**-**1), knapSack(W, wt, val, n**-**1))

val **=** [1, 2, 5, 6]

wt **=** [2, 3, 4, 5]

W **=** 8

n **=** len(val) print(knapSack(W, wt, val, n))



Time Complexity:

Time complexity of the given code is O(nW); where n is the number of items, and W is the capacity of the knapsack.

Uses:

The knapsack problems have a variety of uses in real life like financial modelling, production queues, inventory management, and design of queuing network models.

Vacuum Problem with 3 rooms:

Problem Statement:

Given a number of rooms with an array specifying if they are dirty or clean, we need to find the movement cost of cleaning all the rooms, and the performance of the cleaner.

The value 0 indicates that the room is dirty, and 1 indicates that the room Is clean.

Code:

import random

def display(room): print(room)

# 3 rooms

room = [0,0,0]# 0 for dirty 1 for clean print("Dirty room")

display(room)

x =0 y=0

while x < 3:

room[x] = random.choice([0,1]) x+=1

print("Before cleaning detect all of these random dirts in the room") display(room)

x =0 z=0

while x < 3:

if room[x] == 0: y+=1

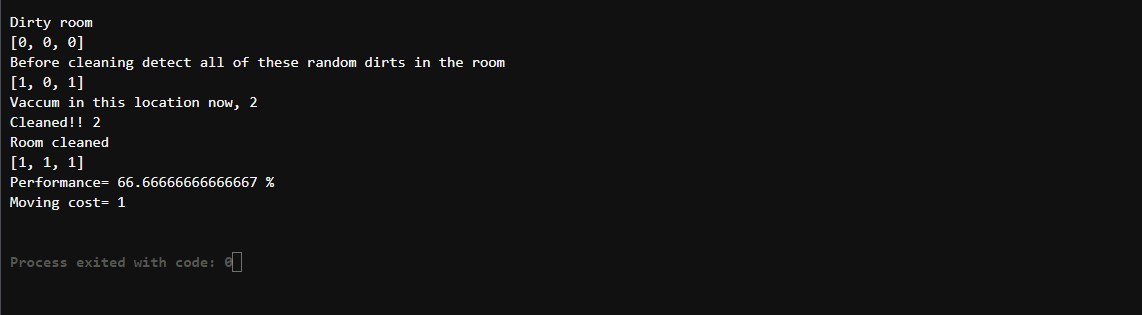
print("Vaccum in this location now,",x+1) room[x] = 1

print("Cleaned!!", x+1) z+=1

x+=1

pro= (100-((z/3)\*100))

print("Room cleaned") display(room) print('Performance=',pro,'%') print('Moving cost=',y)



Unification

Code:

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 == '(':

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):

# Step 1:

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 ': #

Data 1

#f1 = 'p(b(A), X, f(g(Z)))'

#f2 = 'p(Z, f(Y), f(Y))'

# Data 2

f1 = 'Q(a, g(x, a), f(y))'

f2 = 'Q(a, g(f(b), a), x)'

# Data 3

#f1 = 'Q(a, g(x, a, d), f(y))'

#f2 = 'Q(a, g(f(b), a), x)'

result = unify(f1, f2) if not result:

print('Unification failed!') else: print('Unification successfully!') print(result)

**Bayesian Belief**

#Import requiredpackages

import math

from pomegranate import \*

# Initially thedoor selectedby the guest is completely random

guest =DiscreteDistribution( { 'A': 1./3, 'B': 1./3, 'C': 1./3 } )

# The door containing the prize is also a random process

prize =DiscreteDistribution( { 'A': 1./3, 'B': 1./3, 'C': 1./3 } )

# The door Montypicks, depends on the choice of the guest and the prize

door

monty =ConditionalProbabilityTable(

[[ 'A', 'A', 'A', 0.0 ], [ 'A', 'A', 'B', 0.5 ], [ 'A', 'A', 'C', 0.5 ], [ 'A', 'B', 'A', 0.0 ], [ 'A', 'B', 'B', 0.0 ], [ 'A', 'B', 'C', 1.0 ], [ 'A', 'C', 'A', 0.0 ], [ 'A', 'C', 'B', 1.0 ], [ 'A', 'C', 'C', 0.0 ], [ 'B', 'A', 'A', 0.0 ], [ 'B', 'A', 'B', 0.0 ], [ 'B', 'A', 'C', 1.0 ], [ 'B', 'B', 'A', 0.5 ], [ 'B', 'B', 'B', 0.0 ], [ 'B', 'B', 'C', 0.5 ], [ 'B', 'C', 'A', 1.0 ], [ 'B', 'C', 'B', 0.0 ],

[ 'B', 'C', 'C', 0.0 ], [ 'C', 'A', 'A', 0.0 ], [ 'C', 'A', 'B', 1.0 ], [ 'C', 'A', 'C', 0.0 ], [ 'C', 'B', 'A', 1.0 ], [ 'C', 'B', 'B', 0.0 ], [ 'C', 'B', 'C', 0.0 ], [ 'C', 'C', 'A', 0.5 ], [ 'C', 'C', 'B', 0.5 ], [ 'C', 'C', 'C', 0.0 ]], [guest, prize] )

d1 = State( guest, name="guest" )

d2 = State(prize, name="prize" )

d3 = State( monty, name="monty" )

#Building the Bayesian Network

network = BayesianNetwork( "Solving the Monty Hall Problem With

Bayesian Networks" )

network.add\_states(d1, d2, d3)

network.add\_edge(d1,d3)

network.add\_edge(d2,d3)

network.bake()

beliefs = network.predict\_proba({'guest' : 'A', 'monty' : 'B'})

print("n".join( "{}t{}".format( state.name, str(belief) ) for state, belief in

zip( network.states, beliefs )))

beliefs = network.predict\_proba({ 'guest' : 'A' })

beliefs = map(str, beliefs)

print("n".join( "{}t{}".format( state.name, belief ) for state, belief in

zip( network.states,