Pro

likes(sachin,cricket).

likes(saurav,cricket).

likes(raj,football).

likes(bhavik,badminton).

likes(karan,chess).

likes(parth,X):-likes(saurav,X).

2

%facts

male(michel).

male(john).

male(robert).

male(boby).

male(sunny).

female(mary).

female(linda).

female(sophia).

female(micky).

%relationships

father(michel,john).

father(michel,robert).

father(john,boby).

father(robert,micky).

father(robert,sunny).

mother(mary,john).

mother(mary,robert).

mother(linda,boby).

mother(sophia,sunny).

mother(sophia,micky).

wife(mary,michel).

wife(linda,john).

wife(sophia,robert).

parent(michel,mary,john).

parent(michel,mary,robert).

parent(john,linda,boby).

parent(robert,sophia,sunny).

parent(robert,sophia,micky).

%rules

sister(X,Y):-parent(A,B,X),parent(A,B,Y),female(X),X\==Y.

brother(X,Y):-parent(A,B,X),parent(A,B,Y),X\==Y.

siblings(X,Y):-parent(A,B,X),parent(A,B,Y),X\==Y.

husband(X,Y):-wife(Y,X).

son(X,Y):-father(Y,X);mother(Y,X),male(X).

daughter(X,Y):-father(Y,X);mother(Y,X),female(X).

grandfather(X,Y):-father(X,Z),father(Z,Y).

grandmother(X,Y):-mother(X,Z),mother(Z,Y).

uncle(X,Y):-brother(Z,X),father(Z,Y).

aunt(X,Y):-husband(Z,X),uncle(Z,Y).

cousin(X,Y):-uncle(Z,X),father(Z,Y).

graph = {

    'A' : ['B','C'],

    'B' : ['D','E'],

    'C' : ['F'],

    'D' : [],

    'E' : ['F'],

    'F' : []

}

visited = set()

def dfs(visited,graph,node):

    if node not in visited:

        print(node)

        visited.add(node)

        for neighbour in graph[node]:

            dfs(visited,graph,neighbour)

dfs(visited,graph, 'A')

on(floor,monkey).

on(floor,box).

in(room,monkey).

in(room,box).

in(room,banana).

at(ceiling,banana).

strong(monkey).

grasp(monkey).

climb(monkey,box).

push(monkey,box):-

    strong(monkey).

under(banana,box):-

    push(monkey,box).

canreach(banana,monkey):-

    at(floor,banana);

    at(ceiling,banana);

    under(banana,box);

    climb(monkey,box).

canget(banana,monkey):-

    canreach(banana,monkey),grasp(monkey).

#include <iostream>

#include <vector>

#include <algorithm>

#include <climits>

using namespace std;

const int INF = INT\_MAX;

// Function to find factorial of a number

int factorial(int n) {

    if (n <= 1)

        return 1;

    return n \* factorial(n - 1);

}

// Function to find the shortest path using permutations

int tsp(vector< vector<int> >& graph, int src) {

    int n = graph.size();

    vector<int> vertices;

    for (int i = 0; i < n; ++i) {

        if (i != src)

            vertices.push\_back(i);

    }

    int min\_path = INF;

    do {

        int current\_pathweight = 0;

        int k = src;

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

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

            k = vertices[i];

        }

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

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

    } while (next\_permutation(vertices.begin(), vertices.end()));

    return min\_path;

}

int main() {

    int n; // Number of cities

    cout << "Enter the number of cities: ";

    cin >> n;

    vector< vector<int> > graph(n, vector<int>(n));

    cout << "Enter the adjacency matrix representing the distances between the cities:\n";

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            cin >> graph[i][j];

        }

    }

    int src = 0; // Starting city

    int min\_path = tsp(graph, src);

    cout << "The shortest possible route in which the salesman visits all the cities and returns to the origin is: " << min\_path << endl;

    return 0;

}

graph = {

    'A' : ['B','C'],

    'B' : ['D','E'],

    'C' : ['F'],

    'D' : [],

    'E' : ['F'],

    'F' : []

}

visited = []

queue = []

def bfs(visited,graph,node):

    visited.append(node)

    queue.append(node)

    while queue:

        s = queue.pop(0)

        print (s, end = " ")

        for neighbour in graph[s]:

            if neighbour not in visited:

                visited.append(neighbour)

                queue.append(neighbour)

bfs(visited, graph, 'A')

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

struct State {

    int jugA, jugB;

    State(int a, int b) : jugA(a), jugB(b) {}

};

bool isVisited[101][101] = {false};

void BFS(int jugACapacity, int jugBCapacity, int target) {

    queue<State> q;

    q.push(State(0, 0));

    isVisited[0][0] = true;

    while (!q.empty()) {

        State current = q.front();

        q.pop();

        if (current.jugA == target || current.jugB == target) {

            cout << "Target reached!" << endl;

            return;

        }

        // Fill jug A

        if (!isVisited[jugACapacity][current.jugB]) {

            q.push(State(jugACapacity, current.jugB));

            isVisited[jugACapacity][current.jugB] = true;

        }

        // Fill jug B

        if (!isVisited[current.jugA][jugBCapacity]) {

            q.push(State(current.jugA, jugBCapacity));

            isVisited[current.jugA][jugBCapacity] = true;

        }

        // Empty jug A

        if (!isVisited[0][current.jugB]) {

            q.push(State(0, current.jugB));

            isVisited[0][current.jugB] = true;

        }

        // Empty jug B

        if (!isVisited[current.jugA][0]) {

            q.push(State(current.jugA, 0));

            isVisited[current.jugA][0] = true;

        }

        // Pour jug A to jug B

        int pourAmount = min(current.jugA, jugBCapacity - current.jugB);

        if (!isVisited[current.jugA - pourAmount][current.jugB + pourAmount]) {

            q.push(State(current.jugA - pourAmount, current.jugB + pourAmount));

            isVisited[current.jugA - pourAmount][current.jugB + pourAmount] = true;

        }

        // Pour jug B to jug A

        pourAmount = min(current.jugB, jugACapacity - current.jugA);

        if (!isVisited[current.jugA + pourAmount][current.jugB - pourAmount]) {

            q.push(State(current.jugA + pourAmount, current.jugB - pourAmount));

            isVisited[current.jugA + pourAmount][current.jugB - pourAmount] = true;

        }

    }

    cout << "Target cannot be reached!" << endl;

}

int main() {

    int jugACapacity, jugBCapacity, target;

    cout << "Enter the capacity of jug A: ";

    cin >> jugACapacity;

    cout << "Enter the capacity of jug B: ";

    cin >> jugBCapacity;

    cout << "Enter the target amount of water: ";

    cin >> target;

    BFS(jugACapacity, jugBCapacity, target);

    return 0;

}

#include <iostream>

#define N 4

using namespace std;

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

{

    cout<<"N Queen Solution\n\n";

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++)

           if(board[i][j])

            cout << "Q ";

           else cout<<". ";

        printf("\n");

    }

    cout<<"............\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 solveNQ(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 (solveNQ(board, col + 1))

                return true;

            board[i][col] = 0;

        }

    }

    return false;

}

bool solveNQP()

{

    int board[N][N] = {

    {0,0,0,0},

    {0,0,0,0},

    {0,0,0,0},

    {0,0,0,0}

    };

    if (solveNQ(board, 0) == false) {

        cout << "Solution does not exist";

        return false;

    }

    printSol(board);

    return true;

}

int main(){

    solveNQP();

    return 0;

}

# Python3 program to implement traveling salesman

# problem using naive approach.

from sys import maxsize

from itertools import permutations

V = 4

# implementation of traveling Salesman Problem

def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex

    vertex = []

    for i in range(V):

        if i != s:

            vertex.append(i)

    # store minimum weight Hamiltonian Cycle

    min\_path = maxsize

    next\_permutation=permutations(vertex)

    for i in next\_permutation:

        # store current Path weight(cost)

        current\_pathweight = 0

        # compute current path weight

        k = s

        for j in i:

            current\_pathweight += graph[k][j]

            k = j

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

        # update minimum

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

    return min\_path

# Driver Code

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

    # matrix representation of graph

    graph = [[0, 10, 15, 20], [10, 0, 35, 25],

            [15, 35, 0, 30], [20, 25, 30, 0]]

    s = 0

    print(travellingSalesmanProblem(graph, s))

#include<iostream>

using namespace std;

void TowerOfHanoi(int n, char source, char destination, char a)

{

if (n == 1)

{

cout<<"Move disk 1 from rod "<<source<<" to rod "<<destination<< endl;

return;

}

else

{

TowerOfHanoi(n - 1, source, a, destination);

cout<<"Move disk "<<n<<" from rod "<<source<<" to rod "<<destination<< endl;

TowerOfHanoi(n - 1, a, destination,source);

}

}

int main()

{

int n=0;

cout<<"Enter the no of disk from user:"<<endl;

cin>>n;

TowerOfHanoi(n, 'A', 'C', 'B');

return 0;

}

import heapq

def best\_first\_search(graph, start, goal):

    explored = set()

    frontier = [(heuristic(start, goal), start, [start])]

    while frontier:

        current\_cost, current\_node, path = heapq.heappop(frontier)

        if current\_node == goal:

            return True, path

        explored.add(current\_node)

        for neighbor, cost in graph[current\_node].items():

            if neighbor not in explored:

                new\_path = path + [neighbor]

                heapq.heappush(frontier, (heuristic(neighbor, goal), neighbor, new\_path))

    return False, []

def heuristic(node, goal):

    return abs(node - goal)

graph = {

    1: {2: 1, 3: 4},

    2: {4: 3},

    3: {5: 5},

    4: {5: 2},

    5: {}

}

start\_node = 1

goal\_node = 5

path\_exists, path = best\_first\_search(graph, start\_node, goal\_node)

if path\_exists:

    print("Path found:", path)

else:

    print("No path found.")

Prolog for 8 puzzle

test(Plan):-

write('Initial State:'),nl,

Init=[at(tile4,1),at(tile3,2),at(tile8,3),at(empty,4),at(tile2,5),at(tile6,6),at(tile5,7),at(tile1,8),at(tile7,9)],

write\_sol(Init),

Goal=[at(tile4,1),at(tile3,2),at(tile3,3),at(tile4,4),at(empty,5),at(tile5,6),at(tile6,7),at(tile7,8),at(tile8,9)],

nl,write('Goal state:'),nl,

write(Goal),nl,nl,

solve(Init,Goal,Plan).

solve(State,Goal,Plan):-

solve(State,Goal,[],Plan).

is\_movable(X1,Y1):-(1 is X1-Y1);(-1 is X1-Y1);(3 is X1-Y1);(-3 is X1-Y1).

solve(State,Goal,Plan,Plan):-

is\_subset(Goal,State),nl,

write\_sol(Plan).

solve(State,Goal,Sofar,Plan):-

act(Action,Preconditions,Delete,Add),

is\_subset(Preconditions,State),

\+member(Action,Sofar),

delete\_list(Delete,State,Remainder),

append(Add,Remainder,NewState),

solve(NewState,Goal,[Action|Sofar],Plan).

act(move(X,Y,Z),

[at(X,Y),at(empty,Z),is\_movable(Y,Z)],

[at(X,Y),at(empty,Z)],

[at(X,Z),at(empty,Y)]).

is\_subset([H|T],Set):-

member(H,Set),

is\_subset(T,Set).

is\_subset([],\_).

delete\_list([H|T],Curstate,Newstate):-

remove(H,Curstate,Remainder),

delete\_list(T,Remainder,Newstate).

delete\_list([],Curstate,Curstate).

remove(X,[X|T],T).

remove(X,[H|T],[H|R]):-

remove(X,T,R).

write\_sol([]).

write\_sol([H|T]):-

write\_sol(T),

write(H),nl.

append([H|T],L1,[H|L2]):-

append(T,L1,L2).

append([],L,L).

member(X,[X|\_]).

member(X,[\_|T]):-

member(X,T).

Output:

test(Plan).

Initial State:

at(tile7,9)

at(tile1,8)

at(tile5,7)

at(tile6,6)

at(tile2,5)

at(empty,4)

at(tile8,3)

at(tile3,2)

at(tile4,1)

Goal state:

[at(tile4,1), at(tile3,2), at(tile3,3), at(tile4,4), at(empty,5), at(tile5,6), at(tile6,7), at(tile7,8), at(tile8,9)]

False

1)Write a LISP program to display your name

(write-line "harshali")

output:-harshali

2)write a program for addition in LISP

;set value 1 to 400

; set value 2 to 600

(setq val1 400)

(setq val2 600)

(print (+ val1 val2))

output:-1000

test(Plan):-

write('Initial state:'),nl,

Init= [at(tile1,1), at(tile2,2),at(tile3,3), at(tile4,4), at(empty,5), at(tile5,6), at(tile6,7), at(tile7,8), at(tile8,9)],

write\_sol(Init),

Goal= [at(tile1,1), at(tile2,2), at(tile3,3), at(tile4,4), at(tile5,5), at(tile8,6), at(tile6,7), at(empty,8), at(tile7,9)],

nl,write('Goal state:'),nl,

write(Goal),nl,nl,

solve(Init,Goal,Plan).

solve(State, Goal, Plan):-

solve(State, Goal, [], Plan).

is\_movable(X1,Y1) :-

(1 is X1 - Y1) ;

(-1 is X1 - Y1) ;

(3 is X1 - Y1) ;

(-3 is X1 - Y1).

solve(State, Goal, Plan, Plan):-

is\_subset(Goal, State), nl,

write\_sol(Plan).

solve(State, Goal, Sofar, Plan):-

act(Action, Preconditions, Delete, Add),

is\_subset(Preconditions, State),

\+ member(Action, Sofar),

delete\_list(Delete, State, Remainder),

append(Add, Remainder, NewState),

solve(NewState, Goal, [Action|Sofar], Plan).

act(move(X,Y,Z),

[at(X,Y), at(empty,Z), is\_movable(Y,Z)],

[at(X,Y), at(empty,Z)],

[at(X,Z), at(empty,Y)]).

is\_subset([H|T], Set):-

member(H, Set),

is\_subset(T, Set).

is\_subset([], \_).

delete\_list([H|T], Curstate, Newstate):-

remove(H, Curstate, Remainder),

delete\_list(T, Remainder, Newstate).

delete\_list([], Curstate, Curstate).

remove(X, [X|T], T).

remove(X, [H|T], [H|R]):-

remove(X, T, R).

write\_sol([]).

write\_sol([H|T]):-

write\_sol(T),

write(H), nl.

append([H|T], L1, [H|L2]):-

append(T, L1, L2).

append([], L, L).

member(X, [X|\_]).

member(X, [\_|T]):-

member(X, T).

query:-

test(plan).

output:-

Initial state:

at(tile8,9)

at(tile7,8)

at(tile6,7)

at(tile5,6)

at(empty,5)

at(tile4,4)

at(tile3,3)

at(tile2,2)

at(tile1,1)

Goal state:

[at(tile1,1), at(tile2,2), at(tile3,3), at(tile4,4), at(tile5,5), at(tile8,6), at(tile6,7), at(empty,8), at(tile7,9)]

False