8 puzzle

import copy

from heapq import heappush, heappop

n = 3

rows = [ 1, 0, -1, 0 ]

cols = [ 0, -1, 0, 1 ]

class priorityQueue:

def \_init\_(self):

self.heap = []

def push(self, key):

heappush(self.heap, key)

def pop(self):

return heappop(self.heap)

def empty(self):

if not self.heap:

return True

else:

return False

class nodes:

def \_init\_(self, parent, mats, empty\_tile\_posi,

costs, levels):

self.parent = parent

self.mats = mats

self.empty\_tile\_posi = empty\_tile\_posi

self.costs = costs

self.levels = levels

def \_lt\_(self, nxt):

return self.costs < nxt.costs

def calculateCosts(mats, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mats[i][j]) and

(mats[i][j] != final[i][j])):

count += 1

return count

def newNodes(mats, empty\_tile\_posi, new\_empty\_tile\_posi,

levels, parent, final) -> nodes:

new\_mats = copy.deepcopy(mats)

x1 = empty\_tile\_posi[0]

y1 = empty\_tile\_posi[1]

x2 = new\_empty\_tile\_posi[0]

y2 = new\_empty\_tile\_posi[1]

new\_mats[x1][y1], new\_mats[x2][y2] = new\_mats[x2][y2], new\_mats[x1][y1]

costs = calculateCosts(new\_mats, final)

new\_nodes = nodes(parent, new\_mats, new\_empty\_tile\_posi,

costs, levels)

return new\_nodes

def printMatsrix(mats):

for i in range(n):

for j in range(n):

print("%d " % (mats[i][j]), end = " ")

print()

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatsrix(root.mats)

print()

def solve(initial, empty\_tile\_posi, final):

pq = priorityQueue()

costs = calculateCosts(initial, final)

root = nodes(None, initial,

empty\_tile\_posi, costs, 0)

pq.push(root)

while not pq.empty():

minimum = pq.pop()

if minimum.costs == 0:

printPath(minimum)

return

for i in range(n):

new\_tile\_posi = [

minimum.empty\_tile\_posi[0] + rows[i],

minimum.empty\_tile\_posi[1] + cols[i], ]

if isSafe(new\_tile\_posi[0], new\_tile\_posi[1]):

child = newNodes(minimum.mats,

minimum.empty\_tile\_posi,

new\_tile\_posi,

minimum.levels + 1,

minimum, final,)

pq.push(child)

initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

empty\_tile\_posi = [ 1, 2 ]

solve(initial, empty\_tile\_posi, final)

8 queen

print ("Enter the number of queens")

N = int(input())

# here we create a chessboard

# NxN matrix with all elements set to 0

board = [[0]\*N for \_ in range(N)]

def attack(i, j):

#checking vertically and horizontally

for k in range(0,N):

if board[i][k]==1 or board[k][j]==1:

return True

#checking diagonally

for k in range(0,N):

for l in range(0,N):

if (k+l==i+j) or (k-l==i-j):

if board[k][l]==1:

return True

return False

def N\_queens(n):

if n==0:

return True

for i in range(0,N):

for j in range(0,N):

if (not(attack(i,j))) and (board[i][j]!=1):

board[i][j] = 1

if N\_queens(n-1)==True:

return True

board[i][j] = 0

return False

N\_queens(N)

for i in board:

print (i)

A\*

import heapq

def heuristic(node, goal):

# Replace this with an appropriate heuristic function

return 0

def astar(graph, start, goal):

open\_list = [(0, start)]

came\_from = {}

g\_score = {node: float('inf') for node in graph}

g\_score[start] = 0

while open\_list:

current\_cost, current\_node = heapq.heappop(open\_list)

if current\_node == goal:

path = []

while current\_node in came\_from:

path.insert(0, current\_node)

current\_node = came\_from[current\_node]

path.insert(0, start)

return path

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

tentative\_g\_score = g\_score[current\_node] + cost

if tentative\_g\_score < g\_score[neighbor]:

came\_from[neighbor] = current\_node

g\_score[neighbor] = tentative\_g\_score

f\_score = tentative\_g\_score + heuristic(neighbor, goal)

heapq.heappush(open\_list, (f\_score, neighbor))

return None

# Example usage:

graph = {

'A': {'B': 1, 'C': 3},

'B': {'A': 1, 'C': 1, 'D': 4},

'C': {'A': 3, 'B': 1, 'D': 1},

'D': {'B': 4, 'C': 1},

}

start\_node = 'A'

goal\_node = 'D'

path = astar(graph, start\_node, goal\_node)

if path:

print("Shortest path:", path)

else:

print("No path found.")

BFS

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = []

queue = []

def bfs(visited, graph, node):

visited.append(node)

queue.append(node)

while queue:

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

print("Following is the Breadth-First Search")

bfs(visited, graph, '5')

DFS

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = []

queue = []

def bfs(visited, graph, node):

visited.append(node)

queue.append(node)

while queue:

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

print("Following is the Breadth-First Search")

bfs(visited, graph, '5')

DECISION TREE

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

iris = load\_iris()

X = iris.data

y = iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

dt\_classifier = DecisionTreeClassifier()

dt\_classifier.fit(X\_train, y\_train)

y\_pred = dt\_classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

BEST FIRST

% Heuristic function (replace with your own heuristic)

heuristic(start, 5).

heuristic(a, 4).

heuristic(b, 2).

heuristic(c, 3).

heuristic(d, 1).

heuristic(goal, 0).

% Define the transition rules (replace with your own problem)

transition(start, a, 1).

transition(start, b, 2).

transition(a, c, 3).

transition(a, d, 2).

transition(b, d, 4).

transition(c, goal, 2).

transition(d, goal, 3).

% Best-First Search

best\_first\_search(Start, Path) :-

best\_first\_search\_internal([node(Start, 0, 0, [])], [], Path).

best\_first\_search\_internal([node(State, \_, \_, Path) | \_], \_, Path) :-

goal(State).

best\_first\_search\_internal([node(State, G, \_, Path) | Rest], Visited, FinalPath) :-

findall(

node(Child, NewG, H, [State | Path]),

(transition(State, Child, Cost),

\+ member(Child, Visited),

NewG is G + Cost,

heuristic(Child, H)),

Children

),

append(Children, Rest, UpdatedQueue),

sort\_queue(UpdatedQueue, SortedQueue),

best\_first\_search\_internal(SortedQueue, [State | Visited], FinalPath).

% Helper predicate to sort the queue based on heuristic values

sort\_queue(Queue, SortedQueue) :-

predsort(compare\_nodes, Queue, SortedQueue).

compare\_nodes(Order, node(\_, \_, H1, \_), node(\_, \_, H2, \_)) :-

compare(Order, H1, H2).

CRIPT ARTHEMATIC

print("Crypt-Arithmetic problem")

def is\_solution\_valid(mapping, words, result):

word\_values = [int(''.join(str(mapping[c]) for c in word)) for word in words]

result\_value = int(''.join(str(mapping[c]) for c in result))

return sum(word\_values) == result\_value

def solve\_cryptarithmetic(words, result):

all\_letters = set(''.join(words + [result]))

if len(all\_letters) > 10:

return None

unique\_letters = sorted(all\_letters)

permutations = range(10)

from itertools import permutations as permute

for perm in permute(permutations, len(unique\_letters)):

mapping = {letter: digit for letter, digit in zip(unique\_letters, perm)}

if mapping[result[0]] == 0:

continue

if is\_solution\_valid(mapping, words, result):

return mapping

return None

f = str(input("enter 1st word: "))

s = str(input("enter 2nd word: "))

result = str(input("enter result: "))

words = [f, s]

solution = solve\_cryptarithmetic(words, result)

if solution:

print("Solution found:")

print("{} = {}".format(letter, digit))

else:

print("No solution found for the given cryptarithmetic problem.")

DIETING SYSTEM

% Facts

has\_disease(john, diabetes).

has\_disease(mary, hypertension).

has\_disease(susan, celiac\_disease).

has\_disease(jim, obesity).

% Rules

diet\_recommendation(Person, Recommendation) :-

( has\_disease(Person, Disease) ->

( Disease = diabetes ->

Recommendation = 'Follow a balanced diet with controlled carbohydrates.'

; Disease = hypertension ->

Recommendation = 'Limit sodium intake and focus on fruits, vegetables, and lean proteins.'

; Disease = celiac\_disease ->

Recommendation = 'Avoid gluten-containing foods like wheat, barley, and rye.'

; Disease = obesity ->

Recommendation = 'Adopt a calorie-controlled diet with regular exercise.'

; Recommendation = 'No specific diet recommendation available for this disease.'

)

; has\_disease(Name, Disease) ->

( Name = john ->

Recommendation = 'John has diabetes.'

; Name = mary ->

Recommendation = 'Mary has hypertension.'

; Name = susan ->

Recommendation = 'Susan has celiac disease.'

; Name = jim ->

Recommendation = 'Jim has obesity.'

; Recommendation = 'No information available for this person.'

)

; Recommendation = 'Person or disease not found.'

).

MAP COLORING

print("Map Coloring")

def is\_valid\_coloring(graph, node, color, coloring):

for neighbor in graph[node]:

if coloring[neighbor] == color:

return False

return True

def map\_coloring(graph, colors, coloring, node):

if node == len(graph):

return True

for color in colors:

if is\_valid\_coloring(graph, node, color, coloring):

coloring[node] = color

if map\_coloring(graph, colors, coloring, node + 1):

return True

coloring[node] = None

return False

def solve\_map\_coloring(graph, colors):

num\_nodes = len(graph)

coloring = [None] \* num\_nodes

if map\_coloring(graph, colors, coloring, 0):

return coloring

return None

graph = {

0: [1, 2],

1: [0, 2, 3],

2: [0, 1, 3],

3: [1, 2]

}

print("graph:",graph)

colors = ['Red', 'Green', 'Blue']

coloring = solve\_map\_coloring(graph, colors)

if coloring:

for node, color in enumerate(coloring):

print(f"Node {node}: {color}")

else:

print("No solution found.")

MEDICAL DIAGONSIS

% Define symptoms associated with diseases

symptom(flue, fever).

symptom(flue, cough).

symptom(cold, headache).

symptom(cold, sore\_throat).

symptom(strep\_throat, sore\_throat).

symptom(strep\_throat, fever).

symptom(strep\_throat, swollen\_glands).

% Query to retrieve symptoms of a disease

symptoms\_of(Disease, Symptom) :-

symptom(Disease, Symptom).

PLANET DB

planet(mercury, rocky, small, hot, no\_moons).

planet(venus, rocky, medium, very\_hot, no\_moons).

planet(earth, rocky, medium, moderate, one\_moon).

planet(mars, rocky, small, cold, two\_moons).

planet(jupiter, gas\_giant, large, very\_cold, '79\_moons').

planet(saturn, gas\_giant, large, very\_cold, '82\_moons').

planet(uranus, ice\_giant, medium, extremely\_cold, '27\_moons').

planet(neptune, ice\_giant, medium, extremely\_cold, '14\_moons').

planet\_characteristics(Name, Type, Size, Temperature, Moons) :-

planet(Name, Type, Size, Temperature, Moons).

SUM 1 TO N

sum(1, 1).

sum(N, Sum) :-

N > 1,

N1 is N - 1,

sum(N1, Sum1),

Sum is Sum1 + N.

TIC TAC TOE

import numpy as np

import random

from time import sleep

def create\_board():

return(np.array([[0, 0, 0],

[0, 0, 0],

[0, 0, 0]]))

def possibilities(board):

l = []

for i in range(len(board)):

for j in range(len(board)):

if board[i][j] == 0:

l.append((i, j))

return(l)

def random\_place(board, player):

selection = possibilities(board)

current\_loc = random.choice(selection)

board[current\_loc] = player

return(board)

def row\_win(board, player):

for x in range(len(board)):

win = True

for y in range(len(board)):

if board[x, y] != player:

win = False

continue

if win == True:

return(win)

return(win)

def col\_win(board, player):

for x in range(len(board)):

win = True

for y in range(len(board)):

if board[y][x] != player:

win = False

continue

if win == True:

return(win)

return(win)

def diag\_win(board, player):

win = True

y = 0

for x in range(len(board)):

if board[x, x] != player:

win = False

if win:

return win

win = True

if win:

for x in range(len(board)):

y = len(board) - 1 - x

if board[x, y] != player:

win = False

return win

def evaluate(board):

winner = 0

for player in [1, 2]:

if (row\_win(board, player) or

col\_win(board, player) or

diag\_win(board, player)):

winner = player

if np.all(board != 0) and winner == 0:

winner = -1

return winner

def play\_game():

board, winner, counter = create\_board(), 0, 1

print(board)

sleep(2)

while winner == 0:

for player in [1, 2]:

board = random\_place(board, player)

print("Board after " + str(counter) + " move")

print(board)

sleep(2)

counter += 1

winner = evaluate(board)

if winner != 0:

break

return(winner)

print("Winner is: " + str(play\_game()))

TOWERS OF HANOI

% Predicate to solve the Towers of Hanoi problem

hanoi(N) :- move(N, left, center, right).

% Predicate to move N disks from A to B using C as a temporary peg

move(0, \_, \_, \_) :- !.

move(N, A, B, C) :-

M is N - 1,

move(M, A, C, B),

write('Move disk from '), write(A), write(' to '), write(B), nl,

move(M, C, B, A).

WATER JUG

print("water jug problrm")

from collections import defaultdict

jug1, jug2, aim = 4, 3, 2

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):

if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

print(amt1, amt2)

return True

if visited[(amt1, amt2)] == False:

print(amt1, amt2)

visited[(amt1, amt2)] = True

return (waterJugSolver(0, amt2) or

waterJugSolver(amt1, 0) or

waterJugSolver(jug1, amt2) or

waterJugSolver(amt1, jug2) or

waterJugSolver(amt1 + min(amt2, (jug1-amt1)),

amt2 - min(amt2, (jug1-amt1))) or

waterJugSolver(amt1 - min(amt1, (jug2-amt2)),

amt2 + min(amt1, (jug2-amt2))))

else:

return False

print("Steps: ")

waterJugSolver(0, 0)

VACUUM CLEANER

print("vaccum cleaner")

import random

def display(room):

print(room)

room = [[1, 1, 1, 1],[1, 1, 1, 1],]

print("All the rooom are dirty")

display(room)

x =0

y= 0

while x < 2:

while y < 2:

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

y+=1

x+=1

y=0

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

display(room)

x =0

y= 0

z=0

while x < 2:

while y < 2:

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

print("Vaccum in this location now,",x, y)

room[x][y] = 0

print("cleaned", x, y)

z+=1

y+=1

x+=1

y=0

pro= (100-((z/8)\*100))

print("Room is clean now, Thanks for using : 3710933")

display(room)

print('performance=',pro,'%')

FRUITS AND ITS COLOUR

% Define the fruits and their colors

fruit\_color(apple, red).

fruit\_color(banana, yellow).

fruit\_color(grape, purple).

fruit\_color(orange, orange).

fruit\_color(watermelon, green).

fruit\_color\_query(Fruit, Color) :-

fruit\_color(Fruit, Color).

FEED FORWARD NEURAL NETWORK

print("Feed forward neural Network")

import numpy as np

def relu(n):

return max(0, n)

def feedforward(input\_data, weights):

node0 = relu(np.dot(input\_data, weights[0]))

node1 = relu(np.dot(input\_data, weights[1]))

node2 = relu(np.dot(np.array([node0, node1]), weights[2]))

node3 = relu(np.dot(np.array([node0, node1]), weights[3]))

output = relu(np.dot(np.array([node2, node3]), weights[4]))

return output

inp = np.array([[2, -1], [2, 2], [3, 3]])

weights = [np.array([3, 3]), np.array([1, 5]), np.array([3, 3]), np.array([1, 5]), np.array([2, -1])]

for x in inp:

output = feedforward(x, weights)

print(f"Input: {x}, Output: {output}")

MAX MIN FOR GAMING

print("MIN MAX Algorithm")

import math

def minimax (curDepth, nodeIndex,

maxTurn, scores,

targetDepth):

if (curDepth == targetDepth):

return scores[nodeIndex]

if (maxTurn):

return max(minimax(curDepth + 1, nodeIndex \* 2,

False, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,

False, scores, targetDepth))

else:

return min(minimax(curDepth + 1, nodeIndex \* 2,

True, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,

True, scores, targetDepth))

scores = [3, 5, 2, 9, 12, 5, 23, 23]

print(scores)

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")

print(minimax(0, 0, True, scores, treeDepth))

FORWARD CHAINING

% Facts and Rules

parent(john, mary).

parent(john, tom).

parent(mary, ann).

parent(mary, pat).

parent(pat, jim).

ancestor(X, Y) :-

parent(X, Y).

ancestor(X, Y) :-

parent(X, Z),

ancestor(Z, Y).

% Query to find ancestors of a given person

find\_ancestors(Person) :-

ancestor(X, Person),

write(X), nl,

fail.

find\_ancestors(\_).

% Query to check if one person is an ancestor of another

is\_ancestor(Person1, Person2) :-

ancestor(Person1, Person2).

% Queries

% Query 1: Is John an ancestor of Jim?

query1 :- is\_ancestor(john, jim).

% Query 2: Who are the ancestors of Ann?

query2 :- find\_ancestors(ann).

% Query 3: Are there any descendants of Mary?

query3 :- ancestor(mary, X), write(X), nl, fail.

% Query 4: Is Pat a descendant of John?

query4 :- is\_ancestor(john, pat).

% Query 5: Who are the descendants of John?

query5 :- ancestor(john, X), write(X), nl, fail.

% Query 6: Are there any ancestors of Tom?

query6 :- ancestor(X, tom), write(X), nl, fail.

% Query 7: Is Ann a parent of Pat?

query7 :- parent(ann, pat).

% Query 8: Find ancestors of a given person (e.g., Mary)

query\_for\_input :-

write('Enter the name of the person: '),

read(Person),

find\_ancestors(Person).

BIRD CAN FLY

% Facts

can\_fly(crow).

can\_fly(sparrow).

can\_fly(eagle).

cannot\_fly(penguin).

cannot\_fly(ostrich).

% Rules

bird\_can\_fly(Bird) :- can\_fly(Bird), format('~w can fly.~n', [Bird]).

bird\_can\_fly(Bird) :- cannot\_fly(Bird), format('~w cannot fly.~n', [Bird]).

TRAVELLING SALES MAN PROBLEM

import itertools

def distance(city1, city2):

return abs(city1[0] - city2[0]) + abs(city1[1] - city2[1])

def total\_distance(path, cities):

total = 0

for i in range(len(path) - 1):

total += distance(cities[path[i]], cities[path[i + 1]])

total += distance(cities[path[-1]], cities[path[0]]) # Return to starting city

return total

def brute\_force\_tsp(cities):

num\_cities = len(cities)

min\_distance = float('inf')

best\_path = []

for path in itertools.permutations(range(num\_cities)):

d = total\_distance(path, cities)

if d < min\_distance:

min\_distance = d

best\_path = path

return best\_path, min\_distance

# Example usage

cities = [(0, 0), (1, 3), (4, 2), (3, 6)]

best\_path, min\_distance = brute\_force\_tsp(cities)

print("Best Path:", best\_path)

print("Min Distance:", min\_distance)

STUD TEAC CODE

teaches\_subject(teacher1, math, 'MAT101').

teaches\_subject(teacher2, physics, 'PHY201').

teaches\_subject(teacher3, computer\_science, 'CSC301').

enrolled\_in\_subject(mahesh, 'MAT101').

enrolled\_in\_subject(lakshmi, 'PHY201').

enrolled\_in\_subject(yach, 'CSC301').

enrolled\_in\_subject(prakash, 'MAT101').

teacher\_of\_subject(SubjectCode, Teacher) :-

teaches\_subject(Teacher, \_, SubjectCode).

students\_in\_subject(SubjectCode, Students) :-

findall(Student, enrolled\_in\_subject(Student, SubjectCode), Students).

WORD PRESS

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>My WordPress Blog</title>

</head>

<body>

<header>

<h1>Welcome to My Blog</h1>

</header>

<nav>

<ul>

<li><a href="#section1">Jump to Section 1</a></li>

<li><a href="#section2">Jump to Section 2</a></li>

</ul>

</nav>

<main>

<article>

<h2 id="section1">Section 1</h2>

<p>This is the content of section 1.</p>

<h2 id="section2">Section 2</h2>

<p>This is the content of section 2.</p>

</article>

</main>

<footer>

<p>&copy; 2024 My WordPress Blog. All rights reserved.</p>

</footer>

</body>

</html>

BACK WARD CHANING

/\* Facts \*/

parent(john, jim).

parent(john, ann).

parent(jane, jim).

parent(jane, ann).

parent(jim, jill).

parent(ann, jill).

/\* Rules \*/

father(X, Y) :- parent(X, Y), male(X).

mother(X, Y) :- parent(X, Y), female(X).

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

/\* Predicates to check for male and female \*/

male(john).

male(jim).

female(jane).

female(ann).

female(jill).

/\* Backward Chaining Rule \*/

is\_grandfather(X) :- grandfather(X, \_).

/\* New Rule to find and print grandparents \*/

print\_grandparents(X) :-

grandparents(X, GP),

write('The grandparents of '), write(X), write(' are: '), write(GP), nl.

/\* Rule to find grandparents \*/

grandparents(X, GP) :- father(X, P), parent(P, GP).

MONKEY BANANA

% Initial state: monkey at floor, box at floor, banana hanging from ceiling

at(floor, monkey).

at(floor, box).

at(ceiling, banana).

% Actions available to the monkey

action(grasp, monkey, banana).

action(climb, monkey, box).

action(push, monkey, box).

% Define the state transition rules

% Grasping the banana puts it in the monkey's possession

result(grasp, State, Result) :-

at(ceiling, banana),

at(floor, monkey),

at(floor, box),

subtract(State, [at(ceiling, banana)], TempState),

append(TempState, [in\_hand(monkey, banana)], Result).

% Climbing the box puts the monkey on top of the box

result(climb, State, Result) :-

at(floor, monkey),

at(floor, box),

subtract(State, [at(floor, monkey)], TempState),

append(TempState, [on\_box(monkey)], Result).

% Pushing the box moves it from the floor to the ceiling

result(push, State, Result) :-

at(floor, monkey),

at(floor, box),

subtract(State, [at(floor, box)], TempState),

append(TempState, [at(ceiling, box)], Result).

% Define the goal state

goal\_state(State) :-

member(in\_hand(monkey, banana), State).

% Define the state transition function

transition(State, Action, Result) :-

action(Action, \_, \_),

result(Action, State, Result).

% Define the solve predicate using depth-first search

solve(State, Actions) :-

goal\_state(State),

Actions = [].

solve(State, Actions) :-

transition(State, Action, Result),

not(member(Result, State)),

solve(Result, RestOfActions),

Actions = [Action | RestOfActions].

MISSIONARY CANNIBALS

from collections import deque

# Define the initial and goal states

initial\_state = (3, 3, 1) # (missionaries on the left, cannibals on the left, boat position)

goal\_state = (0, 0, 0) # (missionaries on the left, cannibals on the left, boat position)

def is\_valid\_state(state):

missionaries\_left, cannibals\_left, boat = state

missionaries\_right = 3 - missionaries\_left

cannibals\_right = 3 - cannibals\_left

# Check if missionaries are outnumbered by cannibals on either side

if (missionaries\_left > 0 and missionaries\_left < cannibals\_left) or \

(missionaries\_right > 0 and missionaries\_right < cannibals\_right):

return False

return True

def generate\_next\_states(state):

possible\_moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

next\_states = []

for move in possible\_moves:

missionaries\_left, cannibals\_left, boat = state

if boat == 1:

missionaries\_left -= move[0]

cannibals\_left -= move[1]

else:

missionaries\_left += move[0]

cannibals\_left += move[1]

missionaries\_right = 3 - missionaries\_left

cannibals\_right = 3 - cannibals\_left

if 0 <= missionaries\_left <= 3 and 0 <= cannibals\_left <= 3 and is\_valid\_state((missionaries\_left, cannibals\_left, 1 - boat)):

next\_states.append((missionaries\_left, cannibals\_left, 1 - boat))

return next\_states

def solve\_missionaries\_cannibals(initial\_state, goal\_state):

visited = set()

queue = deque([(initial\_state, [])])

while queue:

current\_state, path = queue.popleft()

if current\_state == goal\_state:

return path + [current\_state]

if current\_state not in visited:

visited.add(current\_state)

for next\_state in generate\_next\_states(current\_state):

if next\_state not in visited:

queue.append((next\_state, path + [current\_state]))

return None

result = solve\_missionaries\_cannibals(initial\_state, goal\_state)

if result:

print("Solution found:")

for state in result:

print(state)

else:

print("No solution found.")

DOB

dob(john, '1990-05-15').

dob(amy, '1985-09-28').

dob(mike, '1995-02-10').

dob(lisa, '1980-12-03').

get\_dob(Person, DateOfBirth ) :-

dob(Person, DateOfBirth).