MAHARISHI DAYANAND UNIVERSITY

Delhi Global Institute of Technology

Artificial Intelligence Lab

Submitted By: Bazgha Razi

Subject Code : LC-CSE-326G

Subject Name: Artificial Intelligence Lab using Python

Registration Number :*191380214*

INDEX

S.No.	Program	Date	Remaks
1.	Write a program to implement Breadth First Search using Python.	30/03/22	
2.	Write a program to implement Depth First Search using Python.	05/04/22	
3.	Write a program to implement Tic Tac Toe using Python.	08/04/22	
4.	Write a program to implement 8-Puzzle problem using Python.	22/04/22	
5.	Write a program to implement Water-Jug problem using Python.	29/04/22	
6.	Write a program to implement Travelling Salesman problem using Python.	05/05/22	
7.	Write a program to implement Tower of Hanoi using Python.	13/05/22	
8.	Write a program to implement Monkey Banana problem using Python.	03/06/22	
9.	Write a program to implement Missionaries-Cannibals problem using Python.	03/06/22	
10.	Write a program to implement N-Queens problem using Python.	10/06/22	

Program 1: Write a program to implement Breadth First Search using python.

Code:

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def BFS(self, s):
        visited = [False] * (len(self.graph))
        queue = []
        queue.append(s)
        visited[s] = True
        while queue:
            s = queue.pop(0)
            print (s, end = " ")
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print ("Following is Breadth First Traversal"" (starting from vertex 1)")
g.BFS(1)
```

Output:

```
Following is Breadth First Traversal (starting from vertex 1)
1 2 0 3
```

Program 2: Write a program to implement Depth First Search using python.

Code:

```
def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}
visited = set()
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output:

```
Following is the Depth-First Search
5
3
2
4
8
7
```

Program 3: Write a program to implement Tic-Tac-Toe using python.

Code:

```
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()))
```

Output:

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move
[[0 0 0]
 [0 0 0]
 [0 0 1]]
Board after 2 move
[[0 0 0]
 [2 0 0]
 [0 0 1]]
Board after 3 move
[[0 1 0]
 [2 0 0]
 [0 0 1]]
Board after 4 move
[[0 1 0]
 [2 0 0]
 [2 0 1]]
Board after 5 move
[[0 1 1]
 [2 0 0]
 [2 0 1]]
Board after 6 move
[[0 1 1]
 [2 0 0]
 [2 2 1]]
Board after 7 move
[[0 1 1]
 [2 0 1]
 [2 2 1]]
Winner is: 1
```

Program 4: Write a program to implement 8-Puzzle problem using python.

Code:

```
import copy
from heapq import heappush, heappop
n = 3
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]
class priorityQueue:
    def __init__(self):
        self.heap = []
    def push(self, k):
        heappush(self.heap, k)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

class node:

    def __init__(self, parent, mat, empty_tile_pos,
                cost, level):
        self.parent = parent
        self.mat = mat
        self.empty_tile_pos = empty_tile_pos
        self.cost = cost
        self.level = level
    def __lt__(self, nxt):
        return self.cost < nxt.cost
def calculateCost(mat, final) -> int:

    count = 0
    for i in range(n):
        for j in range(n):
            if ((mat[i][j]) and
                (mat[i][j] != final[i][j])):
                count += 1

    return count
```

```
def newNode(mat, empty_tile_pos, new_empty_tile_pos,
                        level, parent, final) -> node:
        new_mat = copy.deepcopy(mat)
        x1 = empty_tile_pos[0]
        y1 = empty_tile_pos[1]
        x2 = new_empty_tile_pos[0]
        y2 = new_empty_tile_pos[1]
        new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2],
new_mat[x1][y1]
        cost = calculateCost(new_mat, final)
        new_node = node(parent, new_mat, new_empty_tile_pos,
                                        cost, level)
        return new_node
def printMatrix(mat):
        for i in range(n):
                for j in range(n):
                        print("%d " % (mat[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)
        printMatrix(root.mat)
        print()
def solve(initial, empty_tile_pos, final):
        pq = priorityQueue()
        cost = calculateCost(initial, final)
        root = node(None, initial,
                                empty_tile_pos, cost, 0)
        pq.push(root)
        while not pq.empty():
                minimum = pq.pop()
                if minimum.cost == 0:
                        printPath(minimum)
                        return
                for i in range(n):
                        new_tile_pos = [
                                minimum.empty_tile_pos[0] + row[i],
                                minimum.empty_tile_pos[1] + col[i], ]
                        if isSafe(new_tile_pos[0], new_tile_pos[1]):
                                child = newNode(minimum.mat,
```

```
                                        minimum.empty_tile_pos,
                                        new_tile_pos,
                                        minimum.level + 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_pos = [ 1, 2 ]

solve(initial, empty_tile_pos, final)
```

Output:

```
1   2   3
5   6   0
7   8   4

1   2   3
5   0   6
7   8   4

1   2   3
5   8   6
7   0   4

1   2   3
5   8   6
0   7   4
```

Program 5: Write a program to implement Water-Jug problem using python.

Code:

```
from collections import deque
def BFS(a, b, target):
    m = {}
    isSolvable = False
    path = []
    q = deque()
    q.append((0, 0))
    while (len(q) > 0):
        u = q.popleft()
        if ((u[0], u[1]) in m):
            continue
        if ((u[0] > a or u[1] > b or
            u[0] < 0 or u[1] < 0)):
            continue
        path.append([u[0], u[1]])
        m[(u[0], u[1])] = 1
        if (u[0] == target or u[1] == target):
            isSolvable = True
            if (u[0] == target):
                if (u[1] != 0):
                    path.append([u[0], 0])
            else:
                if (u[0] != 0):
                    path.append([0, u[1]])
            sz = len(path)
            for i in range(sz):
                print("(", path[i][0], ",",
                        path[i][1], ")")
            break
        q.append([u[0], b]) # Fill Jug2
        q.append([a, u[1]]) # Fill Jug1

        for ap in range(max(a, b) + 1):
            c = u[0] + ap
            d = u[1] - ap
            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])
            c = u[0] - ap
            d = u[1] + ap
            if ((c == 0 and c >= 0) or d == b):
```

```
                    q.append([c, d])
            q.append([a, 0])
            q.append([0, b])
        if (not isSolvable):
            print ("No solution")

if __name__ == '__main__':
    Jug1, Jug2, target = 4, 3, 2
    print("Path from initial state ""to solution state ::")
    BFS(Jug1, Jug2, target)
```

Output:

```
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 0 , 2 )
```

Program 6: Write a program to implement Travelling Salesman problem using python.

Code:

```
from sys import maxsize
from itertools import permutations
V = 4
def travellingSalesmanProblem(graph, s):
	vertex = []
	for i in range(V):
		if i != s:
			vertex.append(i)
	min_path = maxsize
	next_permutation=permutations(vertex)
	for i in next_permutation:
		current_pathweight = 0
		k = s
		for j in i:
			current_pathweight += graph[k][j]
			k = j
		current_pathweight += graph[k][s]
		min_path = min(min_path, current_pathweight)
	return min_path


if __name__ == "__main__":
	graph = [[0, 10, 25, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 10, 0]]
	s = 0
	print(travellingSalesmanProblem(graph, s))
```

Output:

```
60
```

Program 7: Write a program to implement Tower of Hanoi using python.

Code:

```
def TowerOfHanoi(n , source, destination, auxiliary):
	if n==1:
		print ("Move disk 1 from source",source,"to destination",destination)
		return
	TowerOfHanoi(n-1, source, auxiliary, destination)
	print ("Move disk",n,"from source",source,"to destination",destination)
	TowerOfHanoi(n-1, auxiliary, destination, source)

n = 4
TowerOfHanoi(n,'A','B','C')
```

Output:

```
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from source A to destination C
Move disk 4 from source A to destination B
Move disk 1 from source C to destination B
Move disk 2 from source C to destination A
Move disk 1 from source B to destination A
Move disk 3 from source C to destination B
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
```

Program 8: Write a program to implement Monkey Banana problem using python.

Code:

```
from poodle import Object, schedule
from typing import Set
class Position(Object):
    def str (self):
        if not hasattr(self, "locname"): return "unknown"
        return self.locname
class HasHeight(Object):
    height: int
class HasPosition(Object):
    at: Position
class Monkey(HasHeight, HasPosition): pass
class PalmTree(HasHeight, HasPosition):
    def init (self, *args, **kwargs):
        super(). init (*args, **kwargs)
        self.height = 2
class Box(HasHeight, HasPosition): pass
class Banana(HasHeight, HasPosition):
    owner: Monkey
    attached: PalmTree

class World(Object):
    locations: Set[Position]
p1 = Position()
p1.locname = "Position A"
p2 = Position()
p2.locname = "Position B"
p3 = Position()
p3.locname = "Position C"

w = World()
w.locations.add(p1)
w.locations.add(p2)
w.locations.add(p3)
m = Monkey()
m.height = 0 # ground
m.at = p1
box = Box()
```

```
box.height = 2
box.at = p2
p = PalmTree()
p.at = p3
b = Banana()
b.attached = p
def go(monkey: Monkey, where: Position):
    assert where in w.locations
    assert monkey.height < 1, "Monkey can only move while on the ground"
    monkey.at = where
    return f"Monkey moved to {where}"
def push(monkey: Monkey, box: Box, where: Position):
    assert monkey.at == box.at
    assert where in w.locations
    assert monkey.height < 1, "Monkey can only move the box while on the
ground"

    monkey.at = where
    box.at = where
    return f"Monkey moved box to {where}"

def climb_up(monkey: Monkey, box: Box):
    assert monkey.at == box.at
    monkey.height += box.height
    return "Monkey climbs the box"
def grasp(monkey: Monkey, banana: Banana):
    assert monkey.height == banana.height
    assert monkey.at == banana.at
    banana.owner = monkey
    return "Monkey takes the banana"
def infer_owner_at(palmtree: PalmTree, banana: Banana):
    assert banana.attached == palmtree
    banana.at = palmtree.at
    return "Remembered that if banana is on palm tree, its location is where palm
tree is"
def infer_banana_height(palmtree: PalmTree, banana: Banana):
    assert banana.attached == palmtree
    banana.height = palmtree.height
    return "Remembered that if banana is on the tree, its height equals tree's
height"
print('\n'.join(x() for x in schedule(
[go, push, climb_up, grasp, infer_banana_height, infer_owner_at],
```

[w,p1,p2,p3,m,box,p,b],
goal=lambda: b.owner == m)))

Output:

```
$ pip install poodle

$ python ./monkey.py

Monkey moved to Position B

Remembered that if banana is on the tree, its height equals tree's height

Remembered that if banana is on palm tree, its location is where palm tree is

Monkey moved box to Position C

Monkey climbs the box

Monkey takes the banana
```

Program 9: Write a program to implement Missionaries-Cannibals problem using python.

Code:

```
from copy import deepcopy
from collections import deque
import sys
import time
class State(object):
  def __init__(self, missionaries, cannibals , boats):
    self.missionaries = missionaries
    self.cannibals = cannibals
    self.boats = boats
  def successors(self):
    if self.boats == 1:
      sgn = -1
      direction = "from the original shore to the new shore"
    else:
      sgn = 1
      direction = "back from the new shore to the original shore"
    for m in range(3):
      for c in range(3):
        newState = State(self.missionaries+sgn*m, self.cannibals+sgn*c,
self.boats+sgn*1);
        if m+c >= 1 and m+c <= 2 and newState.isValid():
          action = "take %d missionaries and %d cannibals %s. %r" % ( m, c,
direction, newState)
          yield action, newState
  def isValid(self):
    if self.missionaries < 0 or self.cannibals < 0 or self.missionaries > 3 or
self.cannibals > 3 or (self.boats != 0 and self.boats != 1):
      return False
    if self.cannibals > self.missionaries and self.missionaries > 0:
      return False
    if self.cannibals < self.missionaries and self.missionaries < 3:
      return False
    return True

  def is_goal_state(self):
    return self.cannibals == 0 and self.missionaries == 0 and self.boats == 0
```

```
  def __repr__(self):
    return "< State (%d, %d, %d) >" % (self.missionaries, self.cannibals,
self.boats)
class Node(object):
  def __init__(self, parent_node, state, action, depth):
    self.parent_node = parent_node
    self.state = state
    self.action = action
    self.depth = depth
  def expand(self):
    for (action, succ_state) in self.state.successors():
      succ_node = Node(
                  parent_node=self,
                  state=succ_state,
                  action=action,
                  depth=self.depth + 1)
      yield succ_node
  def extract_solution(self):
    solution = []
    node = self
    while node.parent_node is not None:
      solution.append(node.action)
      node = node.parent_node
    solution.reverse()
    return solution
def breadth_first_tree_search(initial_state):
  initial_node = Node(
                 parent_node=None,
                 state=initial_state,
                 action=None,
                 depth=0)
  fifo = deque([initial_node])
  num_expansions = 0
  max_depth = -1
  while True:
    if not fifo:
      print ("%d expansions" % num_expansions)
      return None
    node = fifo.popleft()
    if node.depth > max_depth:
      max_depth = node.depth
      print ("[depth = %d] %.2fs" % (max_depth, time.clock()))
    if node.state.is_goal_state():
```

```
      print ("%d expansions" % num_expansions)
      solution = node.extract_solution()
      return solution
    num_expansions += 1
    fifo.extend(node.expand())
def usage():
  print >> sys.stderr, "usage:"
  print >> sys.stderr, "    %s" % sys.argv[0]
  raise SystemExit(2)
def main():
  initial_state = State(3,3,1)
  solution = breadth_first_tree_search(initial_state)
  if solution is None:
    print ("no solution")
  else:
    print ("solution (%d steps):" % len(solution))
    for step in solution:
      print ("%s" % step)
  print ("elapsed time: %.2fs" % time.clock())
if __name__ == "__main__":
  main()
```

Output:

```
take 0 missionaries and 2 cannibals from the original shore to the new shore. < State (3, 1, 0) >
take 0 missionaries and 1 cannibals back from the new shore to the original shore. < State (3, 2, 1) >
take 0 missionaries and 2 cannibals from the original shore to the new shore. < State (3, 0, 0) >
take 0 missionaries and 1 cannibals back from the new shore to the original shore. < State (3, 1, 1) >
take 2 missionaries and 0 cannibals from the original shore to the new shore. < State (1, 1, 0) >
take 1 missionaries and 1 cannibals back from the new shore to the original shore. < State (2, 2, 1) >
take 2 missionaries and 0 cannibals from the original shore to the new shore. < State (0, 2, 0) >
take 0 missionaries and 1 cannibals back from the new shore to the original shore. < State (0, 3, 1) >
take 0 missionaries and 2 cannibals from the original shore to the new shore. < State (0, 1, 0) >
take 0 missionaries and 1 cannibals back from the new shore to the original shore. < State (0, 2, 1) >
take 0 missionaries and 2 cannibals from the original shore to the new shore. < State (0, 0, 0) >
```

Program 10: Write a program to implement N-Queens problem using python.

Code:

```
global N
N = 4
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print (board[i][j],end=' ')
        print()
def isSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True
def solveNQUtil(board, col):
    if col >= N:
        return True
    for i in range(N):

        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQUtil(board, col + 1) == True:
                return True
            board[i][col] = 0
    return False
def solveNQ():
    board = [ [0, 0, 0, 0],
            [0, 0, 0, 0],
            [0, 0, 0, 0],
            [0, 0, 0, 0]
```

```
        ]
    if solveNQUtil(board, 0) == False:
        print ("Solution does not exist")
        return False
    printSolution(board)
    return True
solveNQ()
```

Output:

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```