Practical – 1

1. Write a program to implement depth first search algorithm :

graph = {

'A': set(['B', 'C']),

'B': set(['A', 'D', 'E']),

'C': set(['A', 'F']),

'D': set(['B']),

'E': set(['B', 'F']),

'F': set(['C', 'E'])

}

def dfs(graph, start):

visited, stack = set(), [start]

while stack:

vertex = stack.pop()

if vertex not in visited:

visited.add(vertex)

stack.extend(graph[vertex] - visited)

return visited

print(dfs(graph, 'A'))

def dfs\_paths(graph, start, goal):

stack = [(start, [start])]

while stack:

(vertex, path) = stack.pop()

for next in graph[vertex] - set(path):

if next == goal:

yield path + [next]

else:

stack.append((next, path + [next]))

print(list(dfs\_paths(graph, 'A', 'E')))

def short(graph, start, goal):

try:

return next(dfs\_paths(graph, start, goal))

except StopIteration:

return None

print(short(graph, 'A', 'E'))

1. Write a program to implement breadth first search algorithm :

graph = {

'A': set(['B', 'C']),

'B': set(['A', 'D', 'E']),

'C': set(['A', 'F']),

'D': set(['B']),

'E': set(['B', 'F']),

'F': set(['C', 'E'])

}

def bfs(graph, start):

visited, queue = set(), [start]

while queue:

vertex = queue.pop(0)

if vertex not in visited:

visited.add(vertex)

queue.extend(graph[vertex] - visited)

return visited

print(bfs(graph, 'C'))

def bfs\_paths(graph, start, goal):

queue = [(start, [start])]

while queue:

(vertex, path) = queue.pop(0)

for next in graph[vertex] - set(path):

if next == goal:

yield path + [next]

else:

queue.append((next, path + [next]))

print(list(bfs\_paths(graph, 'A', 'F')))

def short(graph, start, goal):

try:

return next(bfs\_paths(graph, start, goal))

except StopIteration:

return None

print(short(graph, 'A', 'F'))

**Practical – 2**

1. **N-Queens problem**

from math import \*

import sys

x = {}

n = int(4)

def clear\_future\_blocks(k):

for i in range(k, n + 1):

x[i] = None

def place(k, i):

for j in range(1, k):

if x[j] == i:

return False

j = 1

while j < k:

if abs(x[j] - i) == abs(j - k):

return False

j += 1

return True

def NQueens(k):

for i in range(1, n + 1):

clear\_future\_blocks(k)

if place(k, i):

x[k] = i

if k == n:

for row in range(1, n + 1):

print(x[row], end=' ')

print()

for row in range(1, n + 1):

for col in range(1, n + 1):

if col == x[row]:

print('Q', end='')

else:

print('.', end='')

print()

print('--------------------')

else:

NQueens(k + 1)

NQueens(1)

1. Write a program to solve tower of Hanoi problem.

def moveTower(height, fromPole, toPole, withPole):

if height >= 1:

moveTower(height - 1, fromPole, withPole, toPole)

moveDisk(fromPole, toPole)

moveTower(height - 1, withPole, toPole, fromPole)

def moveDisk(fp, tp):

print("Moving disk from ", fp, " to ", tp)

moveTower(3, "A", "B", "C")

PRACTICAL NO.: - 05

1. Write a program to solve water jug problem.

capacity = (12, 8, 5)

x = capacity[0]

y = capacity[1]

z = capacity[2]

memory = {}

ans = []

def get\_all\_states(state):

a, b, c = state

if a == 6 and b == 6:

ans.append(state)

return True

if (a, b, c) in memory:

return False

memory[(a, b, c)] = 1

pour\_operations = [

(0, 1), (0, 2),

(1, 0), (1, 2),

(2, 0), (2, 1)

]

levels = [a, b, c]

caps = [x, y, z]

for src\_idx, dest\_idx in pour\_operations:

src\_level = levels[src\_idx]

dest\_level = levels[dest\_idx]

dest\_cap = caps[dest\_idx]

if src\_level > 0 and dest\_level < dest\_cap:

pour\_amount = min(src\_level, dest\_cap - dest\_level)

new\_levels = list(levels)

new\_levels[src\_idx] -= pour\_amount

new\_levels[dest\_idx] += pour\_amount

new\_state = tuple(new\_levels)

if get\_all\_states(new\_state):

ans.append(state)

return True

return False

initial\_state = (12, 0, 0)

get\_all\_states(initial\_state)

ans.reverse()

for i in ans:

print(i)

1. Design the simulation of tic – tac – toe game using min-max algorithm :

import os

import time

board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']

player = 1

win = 1

draw = -1

running = 0

stop = 1

game = running

mark = 'X'

def DrawBoard():

print(" \_ \_ \_")

print("| %c | %c | %c |" % (board[1], board[2], board[3]))

print("|\_\_\_|\_\_\_|\_\_\_|")

print("| %c | %c | %c |" % (board[4], board[5], board[6]))

print("|\_\_\_|\_\_\_|\_\_\_|")

print("| %c | %c | %c |" % (board[7], board[8], board[9]))

print(" \_ \_ \_")

def CheckPosition(x):

if (board[x] == ' '):

return True

else:

return False

def CheckWin():

global game

if (board[1] == board[2] and board[2] == board[3] and board[1] != ' '):

game = win

elif (board[4] == board[5] and board[5] == board[6] and board[4] != ' '):

game = win

elif (board[7] == board[8] and board[8] == board[9] and board[7] != ' '):

game = win

elif (board[1] == board[4] and board[4] == board[7] and board[1] != ' '):

game = win

elif (board[2] == board[5] and board[5] == board[8] and board[2] != ' '):

game = win

elif (board[3] == board[6] and board[6] == board[9] and board[3] != ' '):

game = win

elif (board[1] == board[5] and board[5] == board[9] and board[5] != ' '):

game = win

elif (board[3] == board[5] and board[5] == board[7] and board[5] != ' '):

game = win

elif (board[1] != ' ' and board[2] != ' ' and board[3] != ' ' and board[4] != ' ' and board[5] != ' ' and board[6] != ' ' and board[7] != ' ' and board[8] != ' ' and board[9] != ' '):

game = draw

else:

game = running

print("Tic-Tac-Toe Game")

print("Player1[X]--Player2[O]\n")

print()

print()

print("Please Wait ....")

time.sleep(1)

while(game == running):

os.system('cls' if os.name == 'nt' else 'clear')

DrawBoard()

if (player % 2 != 0):

print("Player 1's Chance ")

mark = 'X'

else:

print("Player 2's Chance ")

mark = 'O'

try:

choice = int(input("Enter The Position Between [1-9] Where You Want To Mark: "))

if 1 <= choice <= 9:

if(CheckPosition(choice)):

board[choice] = mark

player += 1

CheckWin()

else:

print("That position is already taken or invalid. Try again.")

time.sleep(1.5)

else:

print("Invalid input. Please enter a number between 1 and 9.")

time.sleep(1.5)

except ValueError:

print("Invalid input. Please enter a number.")

time.sleep(1.5)

os.system('cls' if os.name == 'nt' else 'clear')

DrawBoard()

if (game == draw):

print("Game Draw")

elif(game == win):

# player was incremented after the winning move, so we decrease it to know the winner

player -= 1

if(player % 2 != 0):

print("Player 1 Won")

else:

print("Player 2 Won")

1. PRACTICAL NO - 6
2. B) Design an application to simulate number puzzle problem .

from \_\_future\_\_ import print\_function

from simpleai.search import astar, SearchProblem

from simpleai.search.viewers import WebViewer

GOAL = '''1-2-3

4-5-6

7-8-e'''

INITIAL = '''4-1-2

7-3-e

8-5-6'''

def list\_to\_string(list\_):

return '\n'.join(['-'.join(row) for row in list\_])

def string\_to\_list(string\_):

return [row.split('-') for row in string\_.split('\n')]

def find\_location(rows, element\_to\_find):

"""Find the location (row, column) of a piece in the puzzle."""

for ir, row in enumerate(rows):

for ic, element in enumerate(row):

if element == element\_to\_find:

return ir, ic

# Precompute goal positions for heuristic

goal\_positions = {}

rows\_goal = string\_to\_list(GOAL)

for number in '12345678e':

goal\_positions[number] = find\_location(rows\_goal, number)

class EightPuzzleProblem(SearchProblem):

def actions(self, state):

rows = string\_to\_list(state)

row\_e, col\_e = find\_location(rows, 'e')

actions = []

rows = string\_to\_list(state)

row\_e, col\_e = find\_location(rows, 'e')

actions = []

# possible moves (up, down, left, right)

if row\_e > 0:

actions.append((row\_e - 1, col\_e)) # move tile down

if row\_e < 2:

actions.append((row\_e + 1, col\_e)) # move tile up

if col\_e > 0:

actions.append((row\_e, col\_e - 1)) # move tile right

if col\_e < 2:

actions.append((row\_e, col\_e + 1)) # move tile left

return actions

def result(self, state, action):

rows = string\_to\_list(state)

row\_e, col\_e = find\_location(rows, 'e')

row\_n, col\_n = find\_location(rows, action)

# swap empty with the chosen tile

rows[row\_e][col\_e], rows[row\_n][col\_n] = rows[row\_n][col\_n], rows[row\_e][col\_e]

return list\_to\_string(rows)

def is\_goal(self, state):

return state == GOAL

def cost(self, state1, action, state2):

return 1

def heuristic(self, state):

rows = string\_to\_list(state)

distance = 0

for number in '12345678e':

row\_n, col\_n = find\_location(rows, number)

row\_goal, col\_goal = goal\_positions[number]

distance += abs(row\_n - row\_goal) + abs(col\_n - col\_goal)

return distance

# Run A\* search

result = astar(EightPuzzleProblem(INITIAL))

result = astar(EightPuzzleProblem(INITIAL))

for action, state in result.path():

print('Move number:', action)

print(state)

PRACTICAL NO.: - 07

1. Write a program to shuffle Deck of cards

5 cards :

import itertools.product

import random

deck=list(itertools.product(range(1,14), ['spade', 'heart', 'diamond', 'club']))

random.shuffle(deck)

print("You Got")

for i in range(5):

print(deck[i][0], 'of', deck[i][1])

52 cards :

import random

cardfaces = []

suits = ["Hearts", "Diamonds", "Clubs", "Spades"]

royals = ["J", "Q", "K", "A"]

deck = []

for i in range(2,11):

cardfaces.append(str(i))

for j in range(4):

cardfaces.append(royals[j])

for k in range(4):

for l in range(13):

card = cardfaces[l] + " of " + suits[k]

deck.append(card)

random.shuffle(deck)

for m in range(52):

print(deck[m])

prac 8 :

map coloring :

Based on the image, here is the converted text:

File Edit Format View Help

adjacent(1,2). adjacent(2,1).

adjacent(1,3). adjacent(3,1).

adjacent(1,4). adjacent(4,1).

adjacent(1,5). adjacent(5,1).

adjacent(2,3). adjacent(3,2).

adjacent(2,4). adjacent(4,2).

adjacent(3,4). adjacent(4,3).

adjacent(4,5). adjacent(5,4).

color(1,red,a). color(1,red,b).

color(2,blue,a). color(2,blue,b).

color(3,green,a). color(3,green,b).

color(4,yellow,a). color(4,blue,b).

color(5,blue,a). color(5,green,b).

conflict(Coloring) :-

adjacent(X,Y),

color(X,Color,Coloring),

color(Y,Color,Coloring).

conflict(R1,R2,Coloring) :-

adjacent(R1,R2),

color(R1,Color,Coloring),

color(R2,Color,Coloring).

Zebra :

solution(Houses):-

Houses = [

house(Nationality1,Color1,Pet1,Drink1,Smoke1),

house(Nationality2,Color2,Pet2,Drink2,Smoke2),

house(Nationality3,Color3,Pet3,Drink3,Smoke3),

house(Nationality4,Color4,Pet4,Drink4,Smoke4),

house(Nationality5,Color5,Pet5,Drink5,Smoke5)

],

Nationalities = [norwegian,englishman,spanish,ukrainian,japanese],

Colors = [red,green,white,yellow,blue],

Pets = [dog,snails,fox,horse,zebra],

Drinks = [coffee,tea,milk,orange\_juice,water],

Smokes = [old\_gold,kools,chesterfields,lucky\_strike,parliaments],

permutation(Nationalities,[Nationality1,Nationality2,Nationality3,Nationality4,Nationality5]),

permutation(Colors, [Color1,Color2,Color3,Color4,Color5]),

permutation(Pets, [Pet1,Pet2,Pet3,Pet4,Pet5]),

permutation(Drinks, [Drink1,Drink2,Drink3,Drink4,Drink5]),

permutation(Smokes, [Smoke1,Smoke2,Smoke3,Smoke4,Smoke5]),

member(house(englishman,red,\_,\_,\_),Houses),

member(house(spanish,\_,dog,\_,\_),Houses),

member(house(\_,green,\_,coffee,\_),Houses),

member(house(ukrainian,\_,\_,tea,\_),Houses),

left\_of(house(\_,green,\_,\_,\_),house(\_,white,\_,\_,\_),Houses),

member(house(\_,\_,snails,\_,old\_gold),Houses),

member(house(\_,yellow,\_,\_,kools),Houses),

Houses = [house(norwegian,\_,\_,\_,\_)|\_],

member(house(\_,\_,\_,milk,\_),Houses),

next\_to(house(norwegian,\_,\_,\_,\_),house(\_,blue,\_,\_,\_),Houses),

next\_to(house(\_,\_,\_,\_,kools),house(\_,horse,\_,\_,\_),Houses),

member(house(\_,\_,\_,orange\_juice,lucky\_strike),Houses),

member(house(japanese,\_,\_,\_,parliaments),Houses),

next\_to(house(norwegian,\_,\_,\_,\_),house(\_,blue,\_,\_,\_),Houses),

member(house(\_,\_,\_,water,\_),Houses),

member(house(\_,\_,zebra,\_,\_),Houses).

left\_of(A,B,List) :- append(\_, [A,B|\_], List).

next\_to(A,B,List) :- left\_of(A,B,List).

next\_to(A,B,List) :- left\_of(B,A,List).

Magic square :

def generate\_square(n):

mat = [[0] \* n for \_ in range(n)]

i = n // 2

j = n - 1

for num in range(1, n \* n + 1):

if i == -1 and j == n:

j = n - 2

i = 0

else:

if j == n:

j = 0

if i < 0:

i = n - 1

if mat[i][j]:

j = j - 2

i = i + 1

continue

else:

mat[i][j] = num

j += 1

i -= 1

return mat

n = 3

magic\_square = generate\_square(n)

for row in magic\_square:

print(" ".join(map(str, row)))

prac 10 :

1. Write a program which contains three predicates: male, female, parent. Make rules for following family relations: father, mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew and niece, cousin.

Question:

i. Draw FamilyTree.

ii. Define: Clauses, Facts, Predicates and Rules with conjunction and

disjunction :

Based on the image, here is the converted text:

male(john).

male(paul).

male(mike).

male(raj).

male(amit).

female(mary).

female(linda).

female(susan).

female(rita).

female(amita).

parent(john, paul).

parent(mary, paul).

parent(john, linda).

parent(mary, linda).

parent(paul, mike).

parent(susan, mike).

parent(paul, rita).

parent(susan, rita).

parent(linda, raj).

parent(amit, raj).

parent(linda, amita).

parent(amit, amita).

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

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

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

grandmother(X, Y) :- female(X), parent(X, Z), parent(Z, Y).

brother(X, Y) :- male(X), parent(Z, X), parent(Z, Y), X \= Y.

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

uncle(X, Y) :- male(X), parent(Z, Y), brother(X, Z).

aunt(X, Y) :- female(X), parent(Z, Y), sister(X, Z).

nephew(X, Y) :- male(X), (uncle(Y, X); aunt(Y, X)).

niece(X, Y) :- female(X), (uncle(Y, X); aunt(Y, X)).

cousin(X, Y) :-

parent(P1, X), parent(P2, Y),

(brother(P1, P2); sister(P1, P2)),

X \= Y.