**Index**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr no.** | **Title** | **Page** | **Date** | **Sign** |
| 1.a) | Write a program to implement depth first search algorithm. | 2 | 30/06/22 |  |
| 1.b) | Write a program to implement breadth first search algorithm. | 3 | 30/06/22 |  |
| 2.a) | Write a program to simulate 4-Queen / N-Queen problem. | 5 | 07/07/22 |  |
| 2.b) | Write a program to solve tower of Hanoi problem. | 7 | 07/07/22 |  |
| 3.a) | Write a program to implement alpha beta search. | 8 | 28/07/22 |  |
| 3.b) | Write a program for Hill climbing problem. | 10 | 14/07/22 |  |
| 4) | Write a program to implement A\* algorithm. | 12 | 28/07/22 |  |
| 5.a) | Write a program to solve water jug problem. | 13 | 14/07/22 |  |
| 5.b) | Design the simulation of tic – tac – toe game using min-max algorithm. | 15 | 28/07/22 |  |
| 6.a) | Write a program to solve Missionaries and Cannibals problem. | 18 | 04/08/22 |  |
| 6.b) | Design an application to simulate number puzzle problem. | 22 | 04/08/22 |  |
| 7) | Write a program to shuffle Deck of cards. | 25 | 11/08/22 |  |
| 8) | Solve constraint satisfaction problem | 27 | 15/09/22 |  |

**Practical 1.A)**

**Aim:** Write a program to implement depth first search algorithm.

Program:

graph1 = {

 '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, node, visited):

    if node not in visited:

        visited.append(node)

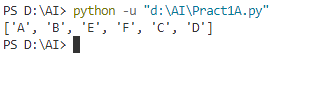
        for n in graph[node]:

            dfs(graph,n, visited)

    return visited

visited = dfs(graph1,'A', [])

print(visited)

Output:

**Practical 1.B)**

**Aim:** Write a program to implement breadth first search algorithm.

Program:

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(start):

    queue = [start]

    levels={}

    levels[start]=0

    visited = set(start)

    while queue:

        node = queue.pop(0)

        neighbours=graph[node]

        for neighbor in neighbours:

            if neighbor not in visited:

                queue.append(neighbor)

                visited.add(neighbor)

                levels[neighbor]= levels[node]+1

    print(levels)

    return visited

print(str(bfs('A')))

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]))

result=list(bfs\_paths(graph, 'A', 'F'))

print(result)

def shortest\_path(graph, start, goal):

    try:

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

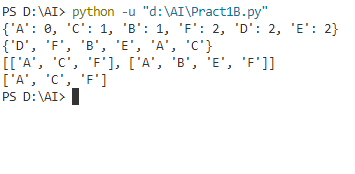
    except StopIteration:

        return None

result1=shortest\_path(graph, 'A', 'F')

print(result1)

OUTPUT:



**Practical 2.A)**

**Aim:** Write a program to simulate 4-Queen / N-Queen problem.

Program:

class QueenChessBoard:

    def \_\_init\_\_(self, size):

        self.size = size

        self.columns = []

    def place\_in\_next\_row(self, column):

        self.columns.append(column)

    def remove\_in\_current\_row(self):

        return self.columns.pop()

    def is\_this\_column\_safe\_in\_next\_row(self, column):

        row = len(self.columns)

        for queen\_column in self.columns:

            if column == queen\_column:

                return False

        for queen\_row, queen\_column in enumerate(self.columns):

                if queen\_column - queen\_row == column - row:

                    return False

        for queen\_row, queen\_column in enumerate(self.columns):

                if ((self.size - queen\_column) - queen\_row == (self.size - column) - row):

                    return False

        return True

    def display(self):

        for row in range(self.size):

            for column in range(self.size):

                if column == self.columns[row]:

                    print('Q', end=' ')

                else:

                    print('.', end=' ')

            print()

def solve\_queen(size):

    board = QueenChessBoard(size)

    number\_of\_solutions = 0

    row = 0

    column = 0

    while True:

        while column < size:

            if board.is\_this\_column\_safe\_in\_next\_row(column):

                board.place\_in\_next\_row(column)

                row += 1

                column = 0

                break

            else:

                column += 1

        if (column == size or row == size):

            if row == size:

                board.display()

                print()

                number\_of\_solutions += 1

                board.remove\_in\_current\_row()

                row -= 1

            try:

                prev\_column = board.remove\_in\_current\_row()

            except IndexError:

                break

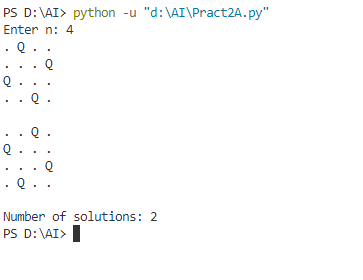
            row -= 1

            column = 1 + prev\_column

    print('Number of solutions:', number\_of\_solutions)

n = int(input('Enter n: '))

solve\_queen(n)

OUTPUT

**Practical 2.B)**

**Aim:** Write a program to solve tower of Hanoi problem.

Program:

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

    if height >= 1:

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

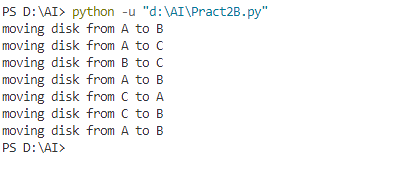
        moveDisk(fromPole,toPole)

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

def moveDisk(fp,tp):

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

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

OUTPUT

**Practical 3.A)**

**Aim:** Write a program to implement alpha beta search.

Program:

tree = [[[5, 1, 2], [8, -8, -9]], [[9, 4, 5], [-3, 4, 3]]]

root = 0

pruned = 0

def children(branch, depth, alpha, beta):

    global tree

    global root

    global pruned

    i = 0

    for child in branch:

        if type(child) is list:

            (nalpha, nbeta) = children(child, depth + 1, alpha, beta)

            if depth % 2 == 1:

                beta = nalpha if nalpha < beta else beta

            else:

                alpha = nbeta if nbeta > alpha else alpha

            branch[i] = alpha if depth % 2 == 0 else beta

            i += 1

        else:

            if depth % 2 == 0 and alpha < child:

                alpha = child

            if depth % 2 == 1 and beta > child:

                beta = child

            if alpha >= beta:

                pruned += 1

                break

    if depth == root:

        tree = alpha if root == 0 else beta

    return (alpha, beta)

def alphabeta(in\_tree=tree, start=root, upper=-15, lower=15):

    global tree

    global pruned

    global root

    (alpha, beta) = children(tree, start, upper, lower)

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

        print("(alpha, beta): ", alpha, beta)

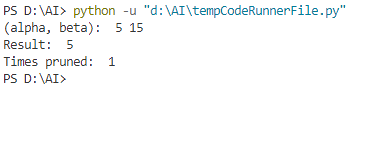
        print("Result: ", tree)

        print("Times pruned: ", pruned)

    return (alpha, beta, tree, pruned)

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

    alphabeta(None)

OUTPUT

**Practical 3.B)**

**Aim:** Write a program for Hill climbing problem.

Program:

import math

increment = 0.1

startingPoint = [1, 1]

point1 = [1,5]

point2 = [6,4]

point3 = [5,2]

point4 = [2,1]

def distance(x1, y1, x2, y2):

    dist = math.pow(x2-x1, 2) + math.pow(y2-y1, 2)

    return dist

def sumOfDistances(x1, y1, px1, py1, px2, py2, px3, py3, px4, py4):

    d1 = distance(x1, y1, px1, py1)

    d2 = distance(x1, y1, px2, py2)

    d3 = distance(x1, y1, px3, py3)

    d4 = distance(x1, y1, px4, py4)

    return d1 + d2 + d3 + d4

def newDistance(x1, y1, point1, point2, point3, point4):

    d1 = [x1, y1]

    d1temp = sumOfDistances(x1, y1, point1[0],point1[1], point2[0],point2[1],

    point3[0],point3[1], point4[0],point4[1] )

    d1.append(d1temp)

    return d1

minDistance = sumOfDistances(startingPoint[0], startingPoint[1],

point1[0],point1[1], point2[0],point2[1],

point3[0],point3[1], point4[0],point4[1] )

flag = True

def newPoints(minimum, d1, d2, d3, d4):

    if d1[2] == minimum:

        return [d1[0], d1[1]]

    elif d2[2] == minimum:

        return [d2[0], d2[1]]

    elif d3[2] == minimum:

        return [d3[0], d3[1]]

    elif d4[2] == minimum:

        return [d4[0], d4[1]]

i = 1

while flag:

    d1 = newDistance(startingPoint[0]+increment, startingPoint[1], point1, point2, point3, point4)

    d2 = newDistance(startingPoint[0]-increment, startingPoint[1], point1, point2, point3, point4)

    d3 = newDistance(startingPoint[0], startingPoint[1]+increment, point1, point2, point3, point4)

    d4 = newDistance(startingPoint[0], startingPoint[1]-increment, point1, point2, point3, point4)

    print (i,'\t', round(startingPoint[0], 2), round(startingPoint[1], 2))

    minimum = min(d1[2], d2[2], d3[2], d4[2])

    if minimum < minDistance:

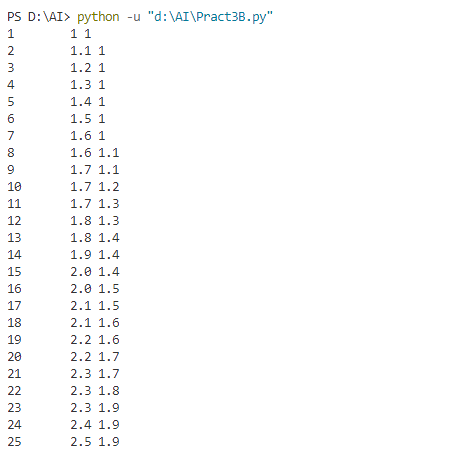
        startingPoint = newPoints(minimum, d1, d2, d3, d4)

        minDistance = minimum

        i+=1

    else:

        flag = False

OUTPUT

**Practical 4)**

**Aim:** Write a program to implement A\* algorithm.

Setup:

Install 2 package in python scripts directory using pip command.

1. pip install simpleai

2. pip install pydot flask

Program:

from simpleai.search import SearchProblem, astar

GOAL = 'HELLO WORLD'

class HelloProblem(SearchProblem):

    def actions(self, state):

        if len(state) < len(GOAL):

            return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')

        else:

            return []

    def result(self, state, action):

        return state + action

    def is\_goal(self, state):

        return state == GOAL

    def heuristic(self, state):

        wrong = sum([1 if state[i] != GOAL[i] else 0

                for i in range(len(state))])

        missing = len(GOAL) - len(state)

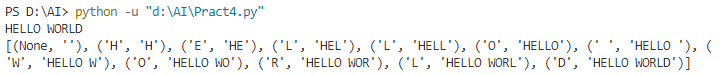
        return wrong + missing

problem = HelloProblem(initial\_state='')

result = astar(problem)

print(result.state)

print(result.path())

OUTPUT

**Practical 5.A)**

**Aim:** Write a program to solve water jug problem.

Program:

capacity = (12,8,5)

x = capacity[0]

y = capacity[1]

z = capacity[2]

memory = {}

ans = []

def get\_all\_states(state):

    a = state[0]

    b = state[1]

    c = state[2]

    if(a==6 and b==6):

        ans.append(state)

        return True

    if((a,b,c) in memory):

        return False

    memory[(a,b,c)] = 1

    if(a>0):

        if(a+b<=y):

            if( get\_all\_states((0,a+b,c)) ):

                ans.append(state)

                return True

        else:

            if( get\_all\_states((a-(y-b), y, c)) ):

                ans.append(state)

                return True

        if(a+c<=z):

            if( get\_all\_states((0,b,a+c)) ):

                ans.append(state)

                return True

        else:

            if( get\_all\_states((a-(z-c), b, z)) ):

                ans.append(state)

                return True

    if(b>0):

        if(a+b<=x):

            if( get\_all\_states((a+b, 0, c))):

                ans.append(state)

                return True

        else:

            if( get\_all\_states((x, b-(x-a), c)) ):

                ans.append(state)

                return True

        if(b+c<=z):

            if( get\_all\_states((a, 0, b+c)) ):

                ans.append(state)

                return True

        else:

            if( get\_all\_states((a, b-(z-c), z)) ):

                ans.append(state)

                return True

    if(c>0):

        if(a+c<=x):

            if( get\_all\_states((a+c, b, 0)) ):

                ans.append(state)

                return True

        else:

            if( get\_all\_states((x, b, c-(x-a))) ):

                ans.append(state)

                return True

        if(b+c<=y):

            if( get\_all\_states((a, b+c, 0)) ):

                ans.append(state)

                return True

        else:

            if( get\_all\_states((a, y, c-(y-b))) ):

                ans.append(state)

                return True

    return False

initial\_state = (12,0,0)

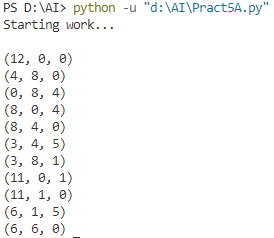
print("Starting work...\n")

get\_all\_states(initial\_state)

ans.reverse()

for i in ans:

    print(i)

OUTPUT

**Practical 5.B)**

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

Program:

import os

import time

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

player = 1

Win = 1

Draw = -1

Running = 0

Stop = 1

Game = Running

Mark = 'X'

def DrawBoard():

    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("Player 1 [X] --- Player 2 [O]\n")

print()

print()

print("Please Wait...")

time.sleep(1)

while(Game == Running):

    os.system('cls')

    DrawBoard()

    if(player % 2 != 0):

        print("Player 1's chance")

        Mark = 'X'

    else:

        print("Player 2's chance")

        Mark = 'O'

    choice = int(input("Enter the position between [1-9] where you want to mark : "))

    if(CheckPosition(choice)):

        board[choice] = Mark

        player+=1

        CheckWin()

os.system('cls')

DrawBoard()

if(Game==Draw):

    print("Game Draw")

elif(Game==Win):

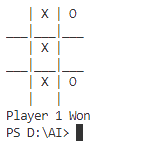
    player-=1

    if(player%2!=0):

        print("Player 1 Won")

    else:

        print("Player 2 Won")

OUTPUT

**Practical 6.A)**

**Aim:** Write a program to solve Missionaries and Cannibals problem.

Program:

import math

class State():

    def \_\_init\_\_(self, cannibalLeft, missionaryLeft, boat, cannibalRight, missionaryRight):

        self.cannibalLeft = cannibalLeft

        self.missionaryLeft = missionaryLeft

        self.boat = boat

        self.cannibalRight = cannibalRight

        self.missionaryRight = missionaryRight

        self.parent = None

    def is\_goal(self):

        if self.cannibalLeft == 0 and self.missionaryLeft == 0:

            return True

        else:

            return False

    def is\_valid(self):

        if self.missionaryLeft >= 0 and self.missionaryRight >= 0 and self.cannibalLeft >= 0 and self.cannibalRight >= 0 and (self.missionaryLeft == 0 or self.missionaryLeft >= self.cannibalLeft) and (self.missionaryRight == 0 or self.missionaryRight >= self.cannibalRight):

            return True

        else:

            return False

    def \_\_eq\_\_(self, other):

        return self.cannibalLeft == other.cannibalLeft and self.missionaryLeft == other.missionaryLeft and self.boat == other.boat and self.cannibalRight == other.cannibalRight and self.missionaryRight == other.missionaryRight

    def \_\_hash\_\_(self):

        return hash((self.cannibalLeft, self.missionaryLeft, self.boat, self.cannibalRight, self.missionaryRight))

def successors(cur\_state):

    children = [];

    if cur\_state.boat == 'left':

        new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft -2, 'right', cur\_state.cannibalRight, cur\_state.missionaryRight + 2)

        if new\_state.is\_valid():

            new\_state.parent = cur\_state

            children.append(new\_state)

        new\_state = State(cur\_state.cannibalLeft - 2, cur\_state.missionaryLeft, 'right', cur\_state.cannibalRight + 2, cur\_state.missionaryRight)

        if new\_state.is\_valid():

            new\_state.parent = cur\_state

            children.append(new\_state)

        new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft - 1, 'right', cur\_state.cannibalRight + 1, cur\_state.missionaryRight + 1)

        if new\_state.is\_valid():

            new\_state.parent = cur\_state

            children.append(new\_state)

        new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 1, 'right', cur\_state.cannibalRight, cur\_state.missionaryRight + 1)

        if new\_state.is\_valid():

            new\_state.parent = cur\_state

            children.append(new\_state)

        new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft, 'right', cur\_state.cannibalRight + 1, cur\_state.missionaryRight)

        if new\_state.is\_valid():

            new\_state.parent = cur\_state

            children.append(new\_state)

    else:

        new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 2, 'left', cur\_state.cannibalRight, cur\_state.missionaryRight - 2)

        if new\_state.is\_valid():

            new\_state.parent = cur\_state

            children.append(new\_state)

        new\_state = State(cur\_state.cannibalLeft + 2, cur\_state.missionaryLeft, 'left', cur\_state.cannibalRight - 2, cur\_state.missionaryRight)

        if new\_state.is\_valid():

            new\_state.parent = cur\_state

            children.append(new\_state)

        new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft + 1, 'left', cur\_state.cannibalRight - 1, cur\_state.missionaryRight - 1)

        if new\_state.is\_valid():

            new\_state.parent = cur\_state

            children.append(new\_state)

        new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 1, 'left',cur\_state.cannibalRight, cur\_state.missionaryRight - 1)

        if new\_state.is\_valid():

            new\_state.parent = cur\_state

            children.append(new\_state)

        new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft, 'left',cur\_state.cannibalRight - 1, cur\_state.missionaryRight)

        if new\_state.is\_valid():

            new\_state.parent = cur\_state

            children.append(new\_state)

    return children

def breadth\_first\_search():

    initial\_state = State(3,3,'left',0,0)

    if initial\_state.is\_goal():

        return initial\_state

    frontier = list()

    explored = set()

    frontier.append(initial\_state)

    while frontier:

        state = frontier.pop(0)

        if state.is\_goal():

            return state

        explored.add(state)

        children = successors(state)

        for child in children:

            if (child not in explored) or (child not in frontier):

                frontier.append(child)

    return None

def print\_solution(solution):

    path = []

    path.append(solution)

    parent = solution.parent

    while parent:

        path.append(parent)

        parent = parent.parent

    for t in range(len(path)):

        state = path[len(path) - t - 1]

        print ("(" + str(state.cannibalLeft) + "," + str(state.missionaryLeft) + "," + state.boat + "," + str(state.cannibalRight) + "," + str(state.missionaryRight) + ")")

def main():

    solution = breadth\_first\_search()

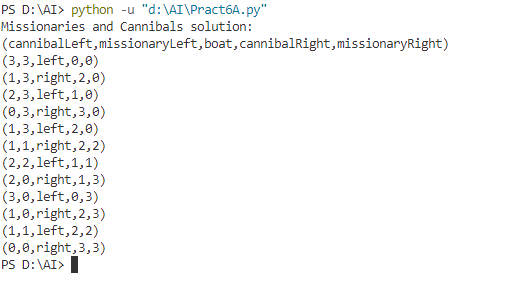
    print ("Missionaries and Cannibals solution:")

    print ("(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)")

    print\_solution(solution)

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

    main()

OUTPUT

**Practical 6.B)**

**Aim:** Design an application to simulate number puzzle problem.

Program:

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-e-3

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):

    for ir, row in enumerate(rows):

        for ic, element in enumerate(row):

            if element == element\_to\_find:

                return ir, ic

goal\_positions = {}

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

for number in '12345678e':

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

class EigthPuzzleProblem(SearchProblem):

    def actions(self, state):

        rows = string\_to\_list(state)

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

        actions = []

        if row\_e > 0:

            actions.append(rows[row\_e - 1][col\_e])

        if row\_e < 2:

            actions.append(rows[row\_e + 1][col\_e])

        if col\_e > 0:

            actions.append(rows[row\_e][col\_e - 1])

        if col\_e < 2:

            actions.append(rows[row\_e][col\_e + 1])

        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)

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

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

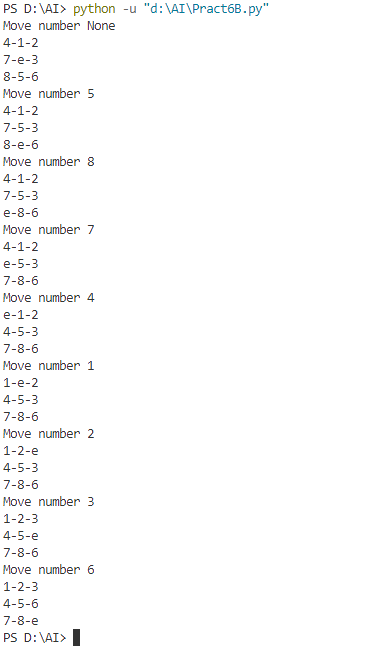
        return distance

result = astar(EigthPuzzleProblem(INITIAL))

for action, state in result.path():

    print('Move number', action)

    print(state)

OUTPUT

**Practical 7)**

**Aim:** Write a program to shuffle Deck of cards.

**Program:**

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):

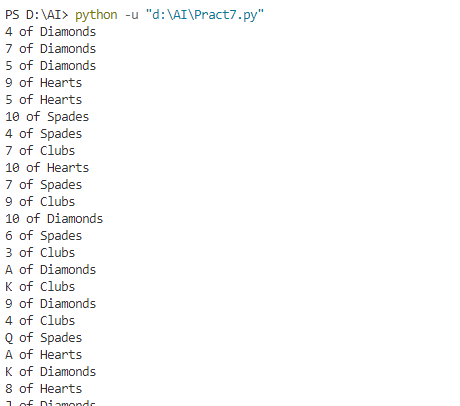
        card = (cardfaces[l] + " of " + suits[k])

        deck.append(card)

random.shuffle(deck)

for m in range(52):

    print(deck[m])

OUTPUT

**Program** to shuffle a deck of card using the module random and draw 5 cards

import itertools, random

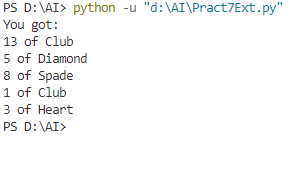
deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))

random.shuffle(deck)

print("You got:")

for i in range(5):

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

OUTPUT

**Practical 8)**

**Aim:** Solve constraint satisfaction problem

**Program:**

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

from simpleai.search import CspProblem, backtrack, min\_conflicts, MOST\_CONSTRAINED\_VARIABLE, HIGHEST\_DEGREE\_VARIABLE, LEAST\_CONSTRAINING\_VALUE

variables = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T')

domains = dict((v, ['red', 'green', 'blue']) for v in variables)

def const\_different(variables, values):

    return values[0] != values[1]

constraints = [

    (('WA', 'NT'), const\_different),

    (('WA', 'SA'), const\_different),

    (('SA', 'NT'), const\_different),

    (('SA', 'Q'), const\_different),

    (('NT', 'Q'), const\_different),

    (('SA', 'NSW'), const\_different),

    (('Q', 'NSW'), const\_different),

    (('SA', 'V'), const\_different),

    (('NSW', 'V'), const\_different),

]

my\_problem = CspProblem(variables, domains, constraints)

print(backtrack(my\_problem))

print(backtrack(my\_problem,variable\_heuristic=MOST\_CONSTRAINED\_VARIABLE))

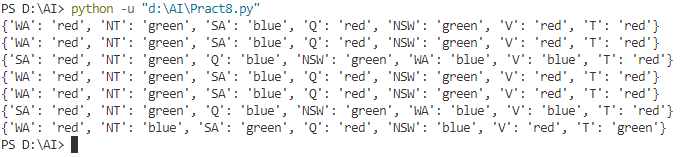
print(backtrack(my\_problem,variable\_heuristic=HIGHEST\_DEGREE\_VARIABLE))

print(backtrack(my\_problem,value\_heuristic=LEAST\_CONSTRAINING\_VALUE))

print(backtrack(my\_problem,variable\_heuristic=MOST\_CONSTRAINED\_VARIABLE,value\_heuristic=LEAST\_CONSTRAINING\_VALUE))

print(backtrack(my\_problem,variable\_heuristic=HIGHEST\_DEGREE\_VARIABLE,value\_heuristic=LEAST\_CONSTRAINING\_VALUE))

print(min\_conflicts(my\_problem))

OUTPUT