

Slip 1

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

GRAPH:-

PYTHON CODE:-

```
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:-

2 Write a program to simulate 4-Queen problem.

PYTHON CODE:-

```
class QueenChessBoard:
def __init__(self, size):
# board has dimensions size x size
self.size = size
# columns[r] is a number c if a queen is placed at row r and column c.
# columns[r] is out of range if no queen is placed in row r.
# Thus after all queens are placed, they will be at positions
# (columns[0], 0), (columns[1], 1), ... (columns[size - 1], size - 1)
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):
    # index of next row
    row = len(self.columns)
    # check column
    for queen_column in self.columns:
        if column == queen_column:
            return False
    # check diagonal
    for queen_row, queen_column in enumerate(self.columns):
        if queen_column - queen_row == column - row:
            return False
    # check other diagonal
    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):
    """Display a chessboard for each possible configuration of placing n queens on an n x n
    chessboard and print the number of such configurations."""
    board = QueenChessBoard(size)
    number_of_solutions = 0
    row = 0
    column = 0
    # iterate over rows of board
    while True:
        # place queen in next row
        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 could not find column to place in or if board is full
if (column == size or row == size):
# if board is full, we have a solution
if row == size:
board.display()
print()
number_of_solutions += 1
# small optimization:
# In a board that already has queens placed in all rows except
# the last, we know there can only be at most one position in
# the last row where a queen can be placed. In this case, there
# is a valid position in the last row. Thus we can backtrack two
# times to reach the second last row.
board.remove_in_current_row()
row -= 1
# now backtrack
try:
prev_column = board.remove_in_current_row()
except IndexError:
# all queens removed
# thus no more possible configurations
break
# try previous row again
row -= 1
# start checking at column = (1 + value of column in previous row)
column = 1 + prev_column
print('Number of solutions:', number_of_solutions)
n = int(input('Enter n: '))
solve_queen(n)
OUTPUT:-

```

Slip 2

1 Write a program to implement breadth first search algorithm.

GRAPH:-

PYTHON CODE:-

```

# sample graph implemented as a dictionary
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'])}

```

```

}
#Implement Logic of BFS
def bfs(start):
    queue = [start]
    levels={} #This Dict Keeps track of levels
    levels[start]=0 #Depth of start node is 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) #print graph level
        return visited
    print(str(bfs('A'))) #print graph node
#For Finding Breadth First Search Path
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)# [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
#For finding shortest path
def shortest_path(graph, start, goal):
    if neighbor not in visited:
        queue.append(neighbor)
        visited.add(neighbor)
        levels[neighbor]= levels[node]+1
        print(levels) #print graph level
        return visited
    print(str(bfs('A'))) #print graph node
#For Finding Breadth First Search Path
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)# [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
#For finding shortest path
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)# ['A', 'C', 'F']
try:
return next(bfs_paths(graph, start, goal))
except StopIteration:
return None
result1=shortest_path(graph, 'A', 'F')
print(result1)# ['A', 'C', 'F']

```

OUTPUT:-

2State the water jug problem. Write a program to solve water jug problem.

Aim:-

Write a program to solve water jug problem.

Diagram:-

Python Code:-

```

# 3 water jugs capacity -> (x,y,z) where x>y>z
# initial state (12,0,0)
# final state (6,6,0)
capacity = (12,8,5)
# Maximum capacities of 3 jugs -> x,y,z
x = capacity[0]
y = capacity[1]
z = capacity[2]
# to mark visited states
memory = { }
# store solution path
ans = []
def get_all_states(state):
# Let the 3 jugs be called a,b,c
a = state[0]

```

```

b = state[1]
c = state[2]
if(a==6 and b==6):
    ans.append(state)
    return True
# if current state is already visited earlier
if((a,b,c) in memory):
    return False
memory[(a,b,c)] = 1
#empty jug a
if(a>0):
    #empty a into b
    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
    #empty a into c
    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
    #empty jug b
    if(b>0):
        #empty b into a
        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
        #empty b into c
        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
#empty jug c
if(c>0):
#empty c into a
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
#empty c into b
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:-

```

Slip3

1Write a program to simulate N-Queen problem.

Slip no. 1 Q NO. 2

2Solve travelling salesman problem using artificial intelligence technique.

?

SLIP 4

1 Write a program to solve tower of Hanoi problem.

AIM:-

Write a program to solve tower of Hanoi problem.

DIAGRAM:

PYTHON CODE:

```
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:

-

2 Solve the block of World problem.

Aim:-

Implementation Of Constraints Satisfaction Problem

PYTHON CODE:

```
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] # expect the value of the neighbors to be different
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),
```



```

print(" %c | %c | %c " % (board[7],board[8],board[9]))
print(" || ")
#This Function Checks position is empty or not
def CheckPosition(x):
    if(board[x] == ' '):
        return True
    else:
        return False
#This Function Checks player has won or not
def CheckWin():
    global Game
    #Horizontal winning condition
    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
    #Vertical Winning Condition
    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
    #Diagonal Winning Condition
    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
    #Match Tie or Draw Condition
    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")

```

NOTE:-

Game Rules

1. Traditionally the first player plays with "X". So you can decide who wants to go with "X" and who wants go with "O".
2. Only one player can play at a time.
3. If any of the players have filled a square then the other player and the same player cannot override that square.
4. There are only two conditions that may match will be draw or may win.
5. The player that succeeds in placing three respective marks (X or O) in a horizontal, vertical or diagonal row wins the game.

OUTPUT:-

2 Solve constraint satisfaction problem. (e.g. Map coloring)

??????????????

SLIP 6

1 Write a program to implement alpha beta search

AIM:-

Write a program to implement alpha beta search.

PYTHON CODE

```
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

2 State the water jug problem. Write a program to solve water jug problem.

SLIP2 Q NO. 2

SLIP 7

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

SLIP5 Q NO. 1

2 Write a program to solve tower of Hanoi problem.

SLIP4 Q NO. 1

SLIP8

1 Write a program for Hill climbing problem.

AIM:-

Write a program for Hill climbing problem.

DIAGRAM:-**PYTHON CODE:**

```
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]]
    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, ' ', 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
#print i, ' ', round(startingPoint[0], 2), round(startingPoint[1], 2)
i+=1
else:
flag = False

```

OUTPUT

2 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.
 ?????????????????????

SLIP9

1 Write a program to solve Missionaries and Cannibals problem.

Aim:-

Write a program to solve Missionaries and Cannibals problem.

Diagram:-

Python Code:-

```

import math
# Missionaries and Cannibals Problem
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):
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)
## Two missionaries cross left to right.
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)

```



```

## Two cannibals cross left to right.
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)
## One missionary and one cannibal cross left to right.
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)
## One missionary crosses left to right.
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)
## One cannibal crosses left to right.
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)
## Two missionaries cross right to left.
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)
## Two cannibals cross right to left.
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)
## One missionary and one cannibal cross right to left.
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)
## One missionary crosses right to left.
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)
## One cannibal crosses right to left.
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 called from the command line, call main()
if __name__ == "__main__":
main()

```

OUTPUT

2 Write a program to derive the predicate for the following:-
?

SLIP 10

1 Write a program to shuffle Deck of cards.

Aim:-

Write a program to shuffle Deck of cards.

Diagram:-

Python Code:-

#first let's import random procedures since we will be shuffling

import random

#next, let's start building list holders so we can place our cards in there:

cardfaces = []

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

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

deck = []

#now, let's start using loops to add our content:

for i in range(2,11):

cardfaces.append(str(i)) **#this adds numbers 2-10 and converts them to string data**

for j in range(4):

append(royals[j]) **#this will add the royal faces to the cardbase**

for k in range(4):

```

for l in range(13):
    card = (cardfaces[l] + " of " + suits[k])
    #this makes each card, cycling through suits, but first through faces
    deck.append(card)
    #this adds the information to the "full deck" we want to make
#now let's shuffle our deck!
    random.shuffle(deck)
#now let's see the cards!
    for m in range(52):
        print(deck[m])
OR
# Python program to shuffle a deck of card using the module random and
draw 5 cards
# import modules
import itertools, random
# make a deck of cards
deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))
# shuffle the cards
random.shuffle(deck)
# draw five cards
print("You got:")
for i in range(5):
    print(deck[i][0], "of", deck[i][1])

```

Output:-

2 Write a program to solve Missionaries and Cannibals problem.

SLIP 9 Q NO. 1

SLIP 11

1 Write a program to implement A* algorithm

Aim:-

Write a program to implement A* algorithm.

Note:

Install 2 package in python scripts directory using pip command.

1. pip install simpleai

2. pip install pydot flask

PYTHON CODE:-

```
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):
        # how far are we from the goal?
        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:-

2 Write a program to solve tower of Hanoi problem.

SLIP 4 Q 1

SLIP 12

1 Write a program to implement alpha beta search.

SLIP 6 Q 1

2 Write a program to implement breadth first search algorithm.

SLIP 2 Q 1

SLIP 13

1 Design an application to simulate number puzzle problem.

AIM:-

Design an application to simulate number puzzle problem.

PYHTON CODE:-

'''

8 puzzle problem, a smaller version of the fifteen puzzle:

States are defined as string representations of the pieces on the puzzle.

Actions denote what piece will be moved to the empty space.

States must allways be immutable. We will use strings, but internally most of the time we will convert those strings to lists, which are easier to handle.

For example, the state (string):

'1-2-3

4-5-6

7-8-e'

will become (in lists):

[['1', '2', '3'],

['4', '5', '6'],

['7', '8', 'e']]

'''

```
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):
    """Find the location of a piece in the puzzle.
    Returns a tuple: row, column"""
    for ir, row in enumerate(rows):
        for ic, element in enumerate(row):
            if element == element_to_find:
                return ir, ic
# we create a cache for the goal position of each piece, so we don't have to
# recalculate them every time
    goal_positions = { }
    rows_goal = string_to_list(GOAL)
    for number in '12345678e':
        goal_positions[number] = find_location(rows_goal, number)
class EighthPuzzleProblem(SearchProblem):
    def actions(self, state):
        """Returns a list of the pieces we can move to the empty space."""
        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):
        """Return the resulting state after moving a piece to the empty space.
        (the "action" parameter contains the piece to move)
        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):
    """Returns true if a state is the goal state."""
    return state == GOAL
def cost(self, state1, action, state2):
    """Returns the cost of performing an action. No useful on this problem, i
    but needed.
    """
    return 1
def heuristic(self, state):
    """Returns an *estimation* of the distance from a state to the goal.
    We are using the manhattan distance.
    """
    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:

```

2 Write a program to shuffle Deck of cards.

Aim:-

Write a program to shuffle Deck of cards.

Diagram:-

Python Code:-

#first let's import random procedures since we will be shuffling

```
import random
```

#next, let's start building list holders so we can place our cards in there:

```
cardfaces = []
```

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

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



```

deck = []
#now, let's start using loops to add our content:
for i in range(2,11):
    cardfaces.append(str(i)) #this adds numbers 2-10 and converts them to string data
for j in range(4):
    append(royals[j]) #this will add the royal faces to the cardbase
for k in range(4):
    for l in range(13):
        card = (cardfaces[l] + " of " + suits[k])
        #this makes each card, cycling through suits, but first through faces
        deck.append(card)
        #this adds the information to the "full deck" we want to make
#now let's shuffle our deck!
random.shuffle(deck)
#now let's see the cards!
for m in range(52):
    print(deck[m])
OR
# Python program to shuffle a deck of card using the module random and
draw 5 cards
# import modules
import itertools, random
# make a deck of cards
deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))
# shuffle the cards
random.shuffle(deck)
# draw five cards
print("You got:")
for i in range(5):
    print(deck[i][0], "of", deck[i][1])

```

Output:-

SLIP 14

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.

??????

2 Write a program to implement depth first search algorithm.

SLIP 1 Q NO . 1

SLIP 15

1 Write a program to implement breadth first search algorithm.

SLIP 2 Q NO . 1

2 Write a program to implement A* algorithm.

SLIP 10 Q NO. 1

SLIP 16

1 Derive the expressions based on Distributive law

??????????????

2 Design an application to simulate number puzzle problem.

SLIP 12 Q NO. 1

SLIP 17

1 Solve traveling salesman problem using artificial intelligence technique.

??????????????

2 Derive the expressions based on Associative law.

????????????????

Slip 18

1 Solve constraint satisfaction problem. (e.g. Map coloring)

????????????????

2 Solve traveling salesman problem using artificial intelligence technique.

Slip 19

1 Write a program to solve water jug problem.

Slip 6 Q NO. 2

2 Solve the block of World problem

Aim:-

Implementation Of Constraints Satisfaction Problem

PYTHON CODE:

```
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] # expect the value of the neighbors to be different
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:-

SLIP 20

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

SLIP 5 Q NO 1

2 Write a program to simulate 4-Queen problem.

SLIP 1 Q NO 2

SLIP 21

1 Write a program to solve Missionaries and Cannibals problem.

SLIP 9 Q NO. 1

2 Solve constraint satisfaction problem. (e.g. Map coloring)

??????????

SLIP 22

1 Write a program to solve tower of Hanoi problem.

SLIP 4 Q NO. 1

2 Write a program to implement breadth first search algorithm.

SLIP 2 Q NO. 1

SLIP 23

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

SLIP 5 Q NO. 1

2 Write a program to solve tower of Hanoi problem.

SLIP 4 Q NO. 1

SLIP 24

1 Write a program to solve tower of Hanoi problem.

SLIP 4 Q NO. 1

2 Design an application to simulate number puzzle problem.

SLIP 13 Q NO. 1

SLIP 25

1 Solve constraint satisfaction problem. (e.g. Map coloring).

????????????????????

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

SLIP 5 Q NO. 1

SLIP 26

1 Write a program to solve water jug problem.

Slip 6 Q NO. 2

2 Design an application to simulate number puzzle problem.

Slip 13 Q NO. 1

SLIP 27

1 Write a program to derive the predicate for the following:-

????????????????

2 Write a program to simulate 4-Queen problem.

SLIP 1 Q NO 2

SLIP 28

1 Write a program to solve water jug problem.

Slip 6 Q NO. 2

2 Write a program to derive the predicate for the following the genealogy tree. We will define a series of relationships – father, mother, sibling, brother, sister, aunt, uncle, grandmother, grandfather and ancestor.

??????????????

SLIP 29

1 Write a program to derive the predicate for the following family tree:-

????????????????

2 Write a program to implement depth first search algorithm.

SLIP 1 Q NO . 1

SLIP 30

1 WRITE a program to implement breadth first search algorithm.

SLIP 2 Q 1

2-Write a program to implement A* algorithm.

SLIP 11 Q 1