P. R. Pote (Patil) Education & welfare Trust's, Group of Institution,
College of Engineering & Management, Amravati

Name :- Shreyash R. Namdurkar
Roll No. AIDSU2106S
Class:- 2nd Yr (AI & DS)
Subject: AI
Semester :- 4th
Year :- 2022-23

INDEX

Sr.No.	Name of Experiment	Date	Page No.	Remark
1.	WAP to implement BFS In Python	04/04/23	1	KK 11/04/23
2.	WAP to implement DFS in Python	11/04/23	3	KK 17/04/2023
3.	WAP to implement Tic-Tac-Toe Game using Python	25/04/23	5	KK 09/05/23
4.	Writ a program to implement 8-Puzzle Problem.	9/5/23	7	KK 09/05/23
5.	WAP to implement Water-Jug Problem	9/5/23	9	KK 09/05/23
6.	WAP to implement Travelling Salesman Problem	16/5/23	10	KK 16/05/23
7.	WAP to implement Tower of Hanoi problem.	18/5/23	12	KK 16/05/23
8.	WAP to implement Money Bonana Problem.	23/5/23	14	KK 23/05/23

04 | 04 | 23

Practical No. 1

Aim : Write a program to implement breadth first search using Python.

Software Use : Jupyter Notebook.

Theory :

The BFS algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at tree's root node traverse & visit all nodes at the current depth level before moving on to the node at the next depth level. BFS can be used to solve many problems in graph theory.

Algorithm :

Step 1: A standard BFS Implementation puts each vertex of the graph into one of 2 catagories
1] Visited 2] Not Visited.

Step 2: Start by putting any one of the graph vertices at the back of a queue.

Step 3: Take the front item of the queue & add it to the visited list.

Step 4: Create a list of that vertex adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

Step 5: Keep repeating steps 2 & 3 until the queue is empty

Program Output:

BFS is:

a e b d c

Conclusion :- In this way we successfully perform the BFS.

KKS
14/04/2023

A	B	C	D	F	Sign
Pre-lab test	In-lab Performance	Post Lab test	Record	Total	
✓	✓	✓	✓	15	KKS

In [4]:

```
# BFS Algorithm

graph={
    'a':['e','b'],
    'b':['c','a'],
    'c':['d','b'],
    'd':['e','c'],
    'e':['a','d']
}

visited=[]
queue=[]

def bfs(visited,graph,node):
    visited.append(node)
    queue.append(node)

    while queue:
        m=queue.pop(0)
        print(m,end=" ")
        for n in graph[m]:
            if n not in visited:
                visited.append(n)
                queue.append(n)

print ("BFS is :")
bfs(visited, graph,'a')
```

```
BFS is :
a e b d c
```

11	04	23

Practical No. 2

Aim : Write a Program to implement Depth first Search in Python.

Theory :

The DFS is a recursive algorithm that uses the concept of back tracking. It involves through searches of all the nodes by going ahead if potential else by tracking. Here, the word backtrack means once you are moving forward and there are ~~moving force~~ not any more nodes along the present path you progressing to be visited on the current path unit all the unvisited nodes are traversed, after which subsequent paths are going to be selected.

Requirements : Jupyter Notebook.

Algorithm :

Step 1 : Add the root/start node to the ~~Queue~~ Stack

Step 2 : After that take top item of the stack & add it to visited list of vertex

Step 3 : Create a list of the vertex adjacent node add those which are not within visited list to the stack.

4) Keep continuely steps two and three till the stack is empty.

Conclusion :- This program we have learnt DFS in Python.

A	B	C	D	E	Sign
Pre-Lab Test	In-lab Performance	Post-lab test	Record	total	
✓	✓	✓	✓	15	KKS

KKS
17/04/2023

In [7]:
```
graph = {
  'a' : ['b', 'c'],
  'b' : ['d', 'e'],
  'c' : ['f'],
  'd' : [],
  'e' : [],
  'f' : []
}

visited = [] # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.append(node)
        for n in graph[node]:
            dfs(visited, graph, n)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, 'a')
```

```
Following is the Depth-First Search
a
b
d
e
c
f
```

Practical No.3

Aim : Write a program to create a Tic-Tac-Toe game by using python.

Software Required : Jupyter Notebook

Theory :

Tic-tact-toe game is a very popular game. So lets implement an automatic Tic-tac-toe game using python. The game is automatically played by the program and hence, no user input is needed. Still developing an automatic game will be lots of fun.

Numpy and random python library is used to build this game. Instead of asking a user to put a mark on the board, the code randomly choses a place on the board put the mark on the board.

Algorithm :

PlayGame() is main function to perform following task :

1) Create Board : Create 3×3 board and initializes with 0.
2) For each player, calls the random_place() function to randomly choose a location on board and mark that loc. with the player number alternatively.

3) Print the board after each move.

4) Evaluate board after each move to check whether a row or column or diagonal has the same player no. If so, display the winner name.

Conclusion :- Thus, we created Tic-Tac-Toe Game using Python.

Pre-Lab Test	Post-Lab Test	In-Lab Performance	Record	Total	Sign
✓	✓	✓	✓	15	KG

KG
29/05/23

In [ ]:

```
import os
import time

board = [" ", " ", " ", " ", " ", " ", " ", " ", " ", " "]
player = 1

########win Flags##########
Win = 1
Draw = -1
Running = 0
Stop = 1
###########################
Game = Running
Mark = "X"

# This Function Draws Game Board
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("   |   |   ")


# 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(3)
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")
```

```
Tic-Tac-Toe Game Designed By Sourabh Somani
Player 1 [X] --- Player 2 [O]



Please Wait...
   |   |
___|___|___
   |   |
___|___|___
   |   |
   |   |
Player 1's chance
Enter the position between [1-9] where you want to mark : 1
```

```
 X |   |
___|___|___
   |   |
___|___|___
   |   |
   |   |
Player 2's chance
Enter the position between [1-9] where you want to mark : 2
 X | O |
___|___|___
   |   |
___|___|___
   |   |
   |   |
Player 1's chance
Enter the position between [1-9] where you want to mark : 3
 X | O | X
___|___|___
   |   |
___|___|___
   |   |
   |   |
Player 2's chance
Enter the position between [1-9] where you want to mark : 4
 X | O | X
___|___|___
 O |   |
___|___|___
   |   |
   |   |
Player 1's chance
```

Practical No. 4

Aim : WAP to Solve 8-Puzzle Problem using Python.

Software Requirement : Jupiter Notebook

Theory :

In this 8 puzzle problem, Given a 3×3 board with 8 tiles and one empty space. The objective is to place the no. of tiles to match the final configuration using the empty space.

The goal is to use the vacant space to arrange the no. on the tiles such that they match the final arrangement. In this problem, we can do four move left, right, above or below.

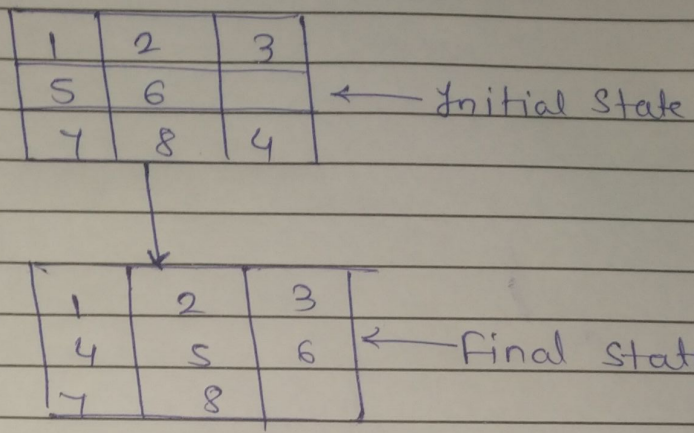
Algorithm :

1) In order to maintain the list of live nodes, algorithm LCSearch employs the functions Least() and Add().

2) Least() identifies a live node with the least c(y), removes it from the list and returns it.

3) Add(y) adds y to the list of live nodes.

4) Add(y) implements the list of live nodes as a min-heap.



Conclusion :- Hence, we implemented 8 puzzle problem and solved it using Python.

Pre-Lab Test	Post-Lab Test	In-Lab Performance	Record	Total	Sign
✓	✓	✓	✓	15	KLS

KLS
09/05/23

In [5]:

```
import copy

from heapq import heappush, heappop


n = 3

# bottom, left, top, right
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]

# A class for Priority Queue
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(4):
            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)
```

```
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
```

Practical No. 5

Aim : WAP to implement water - jug - problem.

Software Use : Jupyter Notebook.

Theory :

- Water Jug problem is a classic problem in AI that involves finding a way to measure specific amounts of water using two jugs with different capacities.
- The goal is to reach a specific target amount of water in one of the jugs, without exceeding its capacity, by transferring water from one jug to another.

Algorithm:

1. Represent the problem as a state space graph with nodes as states and edges as transitions.

2. Initialize to open list with the initial state (0,0)

3. Repeat the following steps until the open list is empty:
   a. Choose a state from the open list and remove it from the list
   b. If the state is goal state, return the solution
   c. Otherwise, generate all possible successor states using the production rules and add

add them to the open list if they have not been visited before

4. If the goal state is not reached, return failure.

Conclusion :- Hence we implemented water-jug problem.

Pre-Lab Test	Post-Lab Test	In-Lab Performance	Record	Total	Sign
✓	✓	✓	✓	15	KIY

KIY
09/05/23

In [1]:
```
from collections import deque
def Solution(a, b, target):
    m = {}
    isSolvable = False
    path = []

    q = deque()

    #Initializing with jugs being empty
    q.append((0, 0))

    while (len(q) > 0):

        # Current state
        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 == a and c >= 0) or d == b):
                q.append([c, d])
```

```
            q.append([a, 0])

            q.append([0, b])

        if (not isSolvable):
            print("Solution not possible")

if __name__ == '__main__':

    Jug1, Jug2, target = 4, 3, 2
    print("Path from initial state "
        "to solution state ::")

    Solution(Jug1, Jug2, target)
```

```
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 )
```

In [ ]:

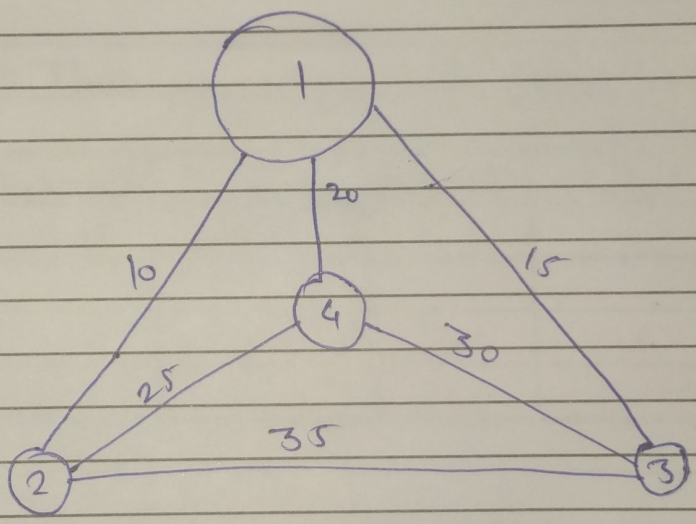
Practical No. 6

Aim : WAP to implement travelling Salesman Problem

Requirements : Jupyter Notebook

Theory :

Given a set of cities and the distance betn every pair of cities, the problem is to find shortest possible route that visits every city exactly once and returns to ~~be~~ the starting point.



Algorithm :

(1) Travelling Salesman problem takes a graph G (V, E) as an i/p and declare another graph as the o/p (say G) which will record the path the salesman is going to take from one node to another.

2] The algorithm begins by sorting all the edges in the i/p graph G from the least distance to largest distance.

3) The first edge adjacent edges of the node other than the origin node (B), find the least ~~code~~ cost edge add it onto o/p graph.

4] Continue the process with further nodes making sure there are no cycles in the o/p graph & the path reaches back to origin node A.

5) However, if the origin is mentioned in given problem, then the soln must always start from that node only.

Conclusion :- Thus we implement travelling ~~salesman~~ prob.

Pre-Lab Test	Post-Lab Test	In Lab Performance	Record	Total	Sign
✓	✓	✓	✓	10	✓

16/05/22

In [2]:

```
from sys import maxsize
from itertools import permutations
V = 4

# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)

    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:

        # store current Path weight(cost)
        current_pathweight = 0

        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        min_path = min(min_path, current_pathweight)

    return min_path


# Driver Code
if __name__ == "__main__":

    # matrix representation of graph
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
             [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))
```

```
80
```

Practical No.1

Aim : WAP to implement Tower of Hanoi Problem.

Requirements : Jupyter Notebook

Theory :

Tower of Hanoi is a mathematical puzzle where we have three rods (A, B and C) and No disks. Initially, all the disks are stacked in decreasing value of diameter. i.e. the smallest disk is placed on the top and they are on rod A. The objective of puzzle is to move entire stack to another rod.

Algorithm :

1) First we move smaller disk to aux peg.
2) Then we move the larger disk to destination peg.
3) Finally we move smaller disk from aux to destination peg.

Conclusion : Thus, we implement tower of Hanoi problem using Python.

Pre Lab Test	Post Lab Test	In-Lab Performance	Record	Total	Sign
✓	✓	✓	✓	15	KK

KK 16/05/23

In [2]:

```
def TowerOfHanoi(n, from_rod, to_rod, aux_rod):
    if n == 0:
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print("Move disk", n, "from rod", from_rod, "to rod", to_rod)
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)


# Driver code
N = 3

# A, C, B are the name of rods
TowerOfHanoi(N, 'A', 'C', 'B')
```

```
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
```

Practical No. 8

Aim : WAP to implement Monkey Bonana Problem.

Requirement : Jupyter Notebook

Theory :

Monkey is in a room with a bonana and a chair, and the monkey cannot reach the bonana until he moves the chair close to bonana then climbs the chair & gets the bonana.

- A hungry monkey is in a room, and he is near the door
- The monkey is on the floor.
- Bonanas have been hung from the center of the ceiling of the room.
- There is a chair present near the window
- The monkey wants the bonana, but cannot reach it.

Algorithm :

- When the block is at the middle, and monkey is on the top of the block and monkey does not have the bonana, then using the grasp action, it will change from has not state to have state

- From the floor, it can move to the top of the block, by performing action climb.

- the push or drag operation moves the block from one place to another.

- Monkey can move from one place to another using walk or move clauses.

Conclusion :- Thus, we have implemented monkey banana problem using python

Pre-Lab Test	Post-Lab Test	In-Lab Performance	Record	Total	Sign
✓	✓	✓	✓	LT KLY	KLY

KLY
23/05/23

In [1]:
```
def monkey_banana(height, distance, monkey_speed, crate_weight):
    # Calculate the time it takes the monkey to climb up the tree
    climb_time = height / monkey_speed

    # Calculate the time it takes the monkey to walk to the crate
    walk_time = distance / monkey_speed

    # Calculate the time it takes the monkey to climb down the tree with the crat
    climb_down_time = (height + crate_weight) / monkey_speed

    # Calculate the total time it takes for the monkey to get the banana
    total_time = climb_time + walk_time + climb_down_time

    return total_time
# Example usage
height = 10
distance = 5
monkey_speed = 5
crate_weight = 3
total_time = monkey_banana(height, distance, monkey_speed, crate_weight)
print("Total time taken: ", total_time)
```

```
Total time taken:  5.6
```

In [ ]: