Practical 1

Aim: Implementation of Tic-Tac-Toe game Algorithm using python AI code.

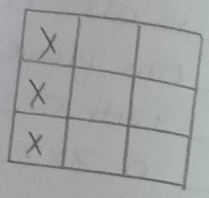
Theory: Tic-Tac-Toe is a very simple two-player game. So only two players can play at a time. This game is also known as Noughts and Crosses or X's or O's game one player plays with X and the other player plays with O. In this game we have a board consisting of a 3X3 grid. The number of grids may be increased.

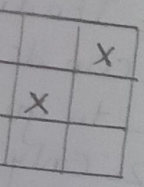
Game Rules / Constrains:

1. Traditionally the first player plays with "X". So you can decide who wants to go with "X" and who wants to 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 watch 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.

State: Start State:-
We start with an empty 3X3 Board and mark position

```
            break
        if is_board_full(board):
            print("Tie!")
            break
        choice = get_computer_move(board, "O")
        if board[choice] == " ":
            board[choice] = "O"
        else:
            print("Sorry, that space is not empty!")
        if is_winner(board, "O"):
            print_board()
            print("O wins! Congratulations")
            break
        if is_board_full(board):
            print("Tie!")
            break
```





Output:

```
   |   |
   |   |
   |   |
Please choose an empty space for X. 1
 X |   |
   | O |
   |   |
Please choose an empty space for X. 9
 X | O |
   | O |
   |   | X
Please choose an empty space for X. 8
 X | O |
   | O | O
   | X | X
Please choose an empty space for X. 7
 X | O |
   | O | O
 X | X | X
X wins! Congratulations
```

from each player, here player vs AI.

Winning condition/ Goal state:
Whoever places three respective marks (X or O) horizontally, vertically or diagonally will be the winner.

Algorithm:
The AI's Algorithm will have the following steps:
1. First, see if there's a move the computer can make that will win the game.
If there is, take that move. otherwise, go to step 2.
2. See if there's a move the player can make that will cause the computer to lose the game. If there is, move there to block the player. Otherwise, go to step 3.
3. Check if any of the corner spaces (spaces 1, 3, 7 or 9) are free. If so, move there. If no corner piece is free, then go to step 4.
4. Check if the centre is free. If so, move there.
If it isn't, then go to step 5.
5. Move on any of the side pieces (spaces 2, 4, 6 or 8).
There are no more steps, because if the execution reaches step 5 the side spaces are the only spaces left.

Result:-
The Tic-Tac-Toe game Algorithm using Python AI code is implemented successfully.

Practical 2

Aim: Implementation of Depth First Search (DFS), Breadth First Search (BFS) using python code.

Theory:

Depth First Search (DFS):

Depth First Search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking. It uses last in-first-out strategy and hence it is implemented using a stack.

Breadth First Search (BFS):-

Breadth First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It is implemented using a queue.

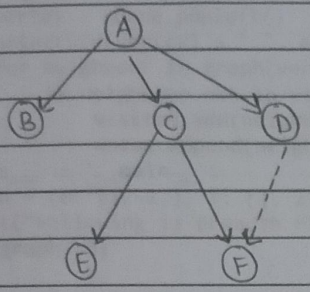
Start State:-

We start with the node of the given Graph/Tree and consider as the current state, till the next best state is not found.

Goal State:-

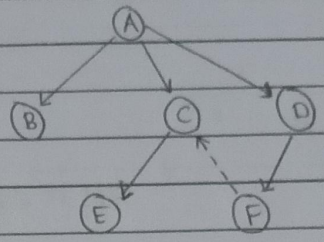
When we reach the goal state with either of the path the function will end itself automatically.

BFS



ABCDEF

DFS



ADFCEB

Algorithm:

Depth First Search (DFS):

1. Create a recursive function that takes the index of the node and a visited array.
2. Mark the current node as visited and print the node.
3. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

Breadth First Search (BFS):

1. Declare a queue and insert the starting vertex.
2. Initialize a visited array and mark the starting vertex as visited.
3. Follow the below process till the queue becomes empty:
   - Remove the first vertex of the queue.
   - Mark that vertex as visited.
   - Insert all the unvisited neighbours of the vertex into the queue.

Program:

```
import collections

def bfs(graph, root):
    visited, queue = set(), collections.deque([root])
    visited.add(root)
    while queue:
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)
if __name__ == '__main__':
    graph = {0: [1, 2,], 1: [], 2: [4, 5], 3: [5], 4: [], 5: []}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

Output:

```
Following is Breadth First Traversal:
0 1 2 3 4 5
```

Program:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end="")
    for next in graph[start] - visited:
        print("->", end="")
        dfs(graph, next, visited)
    return visited

graph = {'0': set(['1', '2', '3']), '1': set([]), '2': set(['4', '5']),
         '3': set(['5']), '4': set([]), '5': set([])}
print("Following is Depth First Traversal: ")
dfs(graph, '0')
```

Output:

```
Following is Depth First Traversal:
0->3->5->2->4->1
```

Result: The Depth First Search (DFS), Breadth First Search (BFS) algorithms is successfully implemented using Python code.

Practical 3

Aim: Implementation of 8-puzzle problem using python AI code.

Theory: The 8 puzzle problem solution is covered in this article. A 3 by 3 board with 8 tiles and a single empty space is provided. The goal is to use the vacant space to arrange the numbers on the tiles such that they match the final arrangement. Four neighboring tiles can be moved into the available area.

Rules/ constraints:

1. The tiles with numbers are to be shuffled with the blank block.
2. We start with the initial state and shuffle the blank block so as to reach the goal state with minimum number of swaps.
3. The blank block can be shuffled only with the neighbouring numbered block, ie. either Move UP, Move DOWN, Move LEFT, Move RIGHT.

Algorithm:

Branch and Bound:

The search for an answer node can be often be speeded by using an "intelligent" ranking function, also called an approximate cost function to avoid searching in subtrees that do not contain an answer node.

It is similar to the backtracking technique but uses a BFS-like search.
There are basically three types of nodes involved in Branch and Bound:
1. Live node is a node that has been generated but whose children have not yet been generated.
2. E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
3. Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

The ideal cost function for an 8-puzzle Algorithm:
We assume that moving one tile in any direction will have a 1 unit cost. Keeping that in mind, we define a cost function for the 8-puzzle algorithm as below:

$$c(x) = f(x) + h(x) \text{ where .....}$$

$f(x)$ - is the length of the path from root to x.
$h(x)$ - is the number of non-blank tiles not in their goal position. There are at least $h(x)$ moves to transform state x to a goal state.

Final Algorithm:
- In order to maintain the list of live nodes, algorithm LC search employs the functions Least() and Add().
- Least() identifies a live node with the least $c(y)$, removes it from the list and returns it.

Program:

```
import copy
from heapq import heappush, heappop
n = 3
rows = [1, 0, -1, 0]
cols = [0, -1, 0, 1]

class priorityQueue:
    def __init__(self):
        self.heap = []
    def push(self, key):
        heappush(self.heap, key)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

class nodes:
    def __init__(self, parent, mats, empty_tile_posi, costs, levels):
        self.parent = parent
        self.mats = mats
        self.empty_tile_posi = empty_tile_posi
        self.costs = costs
        self.levels = levels
    def __lt__(self, nxt):
        return self.costs < nxt.costs

def calculateCosts(mats, final) -> int:
    count = 0
    for i in range(n):
        for j in range(n):
            if ((mats[i][j]) and
                    (mats[i][j] != final[i][j])):
                count += 1
    return count

def newNodes(mats, empty_tile_posi, new_empty_tile_posi,
             levels, parent, final) -> nodes:
    new_mats = copy.deepcopy(mats)
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]
    costs = calculateCosts(new_mats, final)
    new_nodes = nodes(parent, new_mats, new_empty_tile_posi,
                      costs, levels)
    return new_nodes
def printMatsrix(mats):
    for i in range(n):
        for j in range(n):
            print("%d " % (mats[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)
    printMatsrix(root.mats)
    print()

def solve(initial, empty_tile_posi, final):
    pq = priorityQueue()
    costs = calculateCosts(initial, final)
    root = nodes(None, initial, empty_tile_posi, costs, 0)
    pq.push(root)
    while not pq.empty():
        minimum = pq.pop()
        if minimum.costs == 0:
            printPath(minimum)
            return
        for i in range(n):
            new_tile_posi = [
                minimum.empty_tile_posi[0] + rows[i],
                minimum.empty_tile_posi[1] + cols[i], ]
            if isSafe(new_tile_posi[0], new_tile_posi[1]):
                child = newNodes(minimum.mats,
                                 minimum.empty_tile_posi,
                                 new_tile_posi,
                                 minimum.levels + 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_posi = [1, 2]
solve(initial, empty_tile_posi, final)
print("Solution for given 8-puzzle problem is as follows:")
```

Output:

```
Solution for given 8-puzzle problem is as follows:
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
```

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

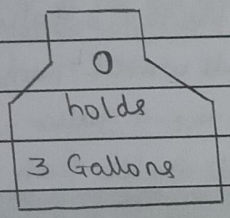
Result: The 8-puzzle problem using python AI code is successfully implemented.

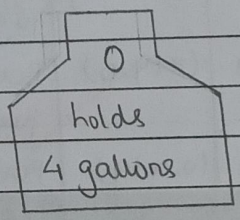
Practical 4

Aim: Implementation of water jug problem using python AI code.

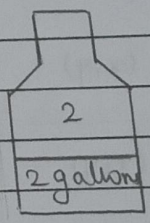
Theory: In the water jug problem in Artificial Intelligence, we are provided with two jugs: one having the capacity to hold 3 gallon of water and the other has the capacity of 4 gallon of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent's task here is to fill the 4 gallon of jug with 2 gallons of water by using only these two jugs and no other material.
Initially, both our jugs are empty.

Start State





Goal State



State space:
Jug 1: Amount of water (0, 1, 2, 3, 4)

Jug 2: Amount of water (0, 1, 2, 3)

start state: (Jug1, Jug2) = (0, 0)

Goal state: (Jug1, Jug2) = (2, 0).

Rule No.	Initial State	Condition	Final State	Description of action taken.
1.	(x,y)	if y<4	(4,y)	Fill the 4 gallon jug completely.
2.	(x,y)	if y<3	(x,3)	Fill the 3 gallon jug completely.
3.	(x,y)	if x>0	(x-d,y)	Pour some part from the 4 gallon jug.
4.	(x,y)	if y>0	(x,y-d)	Pour some part from the 3 gallon jug.
5.	(x,y)	if x>0	(0,y)	Empty the 4 gallon jug
6.	(x,y)	if y>0	(x,0)	Empty the 3 gallon jug.
7.	(x,y)	if (x+y)<7	(4,y-[4-x])	Pour some water from 3 gallon jug to fill the four gallon jug.
8.	(x,y)	if(x+y)<7	(x-[3-y],y)	Pour some water from 4 gallon jug to fill the 3 gallon jug.
9.	(x,y)	if(x+y)<4	(x+y,0)	Pour all water from 3 gallon jug to the 4 gallon jug.
10.	(x,y)	if(x+y)<3	(0,x+y)	Pour all water from the 4 gallon jug to the 3 gallon jug.

Program:

```
from collections import defaultdict

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print("  ", amt1, "\t", amt2)
        return True
    if visited[(amt1, amt2)] == False:
        print("  ", amt1, "\t", amt2)
        visited[(amt1, amt2)] = True
        return (waterJugSolver(0, amt2) or
                waterJugSolver(amt1, 0) or
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1 - amt1)),
                               amt2 - min(amt2, (jug1 - amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2 - amt2)),
                               amt2 + min(amt1, (jug2 - amt2))))
    else:
        return False

jug1 = 4
jug2 = 3
aim = 2
print("Capacity of Jug1 is 4 and Capacity of Jug2 is 3")
print("Amount of water at the end needed is 2")
print("Steps for required solution")
print("Jug 1\tJug2")
waterJugSolver(0, 0)
```

Output:

```
Capacity of Jug1 is 4 and Capacity of Jug2 is 3
Amount of water at the end needed is 2
Steps for required solution
Jug 1   Jug2
   0      0
   4      0
   4      3
   0      3
   3      0
   3      3
   4      2
   0      2
```

Algorithm:-
We have two jugs. "i" litre jug and "j" litre jug (0 < i < j). Both jugs will initially be empty, and they don't have marking to measure small quantities. Now we need to measure d litres of water by using these two jugs where d < j. We use the following three operator to measure small quantities by using the two jars:

1. Empty a jug.
2. Fill a jug.
3. We pour the water of one jug into another one until one of them is either full or empty.

Result: The Water Jug problem using Python AI code is successfully implemented.