NEPAL ENGINEERING COLLEGE

(Affiliated To Pokhara University)

Changunarayan,Bhaktpur

Report On:

AI Lab

SUBMITTED BY:

Name Narayan Gautam

Roll No: 019-357

SUBMITTED TO:

Computer Science & Engg

Lab 2

1. WAP to implement DFS / BFS on water jug problem. Given a 4l jug filled with water & an empty 3l jug. How can you obtain exactly 2l in 4l jug. There is no measuring mark on any of them

→ ~~Using DFS~~ Used Python 3.11.3

```
def pour(node, x2y=None):
    x,y = node
    if x2y:
        diff = min(x, 3-y)
        return x-diff, y+diff
    diff = min(y, 4-x)
    return x+diff, y-diff

def generate(node):
    x,y = node
    child1 = (4,y)  # fill x
    child2 = (x,3)  # " y
    child3 = (0,y)  # empty x
    child4 = (x,0)  # " y
    child5 = pour(node, x2y=True)  # pour x2y
    child6 = pour(node)            # " y2x

    children = [child1, child2, child3, child4, child5, child6]
    return list(set(children))

def bfs(initial_node, goal_node):
    queue = [initial_node]
    visited = set()

    while queue:
        node = queue.pop(0)
        if node in visited:
            continue
        print(node, '-->', end='')

        if node == goal_node:
            return True

        visited.add(node)
        for child in generate(node):
            if child not in visited:
                queue.append(child)

    return False
```

```
def dfs(initial_node, goal_node):
    stack = [initial_node]
    visited = set()
    while stack:
        node = stack.pop(0)
        print(node, '-->', end = '')
        if node == goal_node:
            return True
        if node not in visited:
            visited.add(node)
        visited.add(node)
        for child in generate(node):
            if child not in visited:
                stack.insert(0, child)
    return False

start = (4,0)
goal = (2,0)  # Assume (2,0) but can be (2,1),(2,2),(2,3)

print('\n')
#Using BFS
print(bfs(start,goal))
# Using DFS
print(dfs(start,goal))
```

Output:

BFS ⇒ (4,0) → (0,0) → (1,3) → (4,3) → (0,3) → (1,0) → (3,0).
→ (0,1) → (3,3) → (4,1) → (4,2) → (2,3) → (0,2) →
(2,0) --> True

DFS ⇒ (4,0) → (4,3) → (0,3) → (3,0) → (3,3) → (4,2) → (0,2) →
(2,0) → True

2. WAP to calculate heuristic value of the states for BLOCKS World problem.

```
def heuristic(current, goal):
    h_value = 0
    for i in range(4):
        if current[i] == goal[i]:
            h_value += 1
        else:
            h_value += 1
    return h_value

def main():
    current = ['A', 'D', 'B', 'C']
    goal = ['A', 'B', 'C', 'D']
    print("The current state is: ", end=" ")
    for i in current:
        print(i, end=" ")
    print(" ")
    h_value = heuristic(current, goal)
    print("Heuristic value: ", h_value)
```

3. WAP to calculate heuristic value of states for Tic-Tac-Toe

```
def heuristic(board, player, opponent):
    coins = [[0,1,2], [3,4,5], ....]
    h, flag = 0, 0
    player = {'X': 0, 'O': 1}
    for i in range(8)
        flag = 0
```

```
def heuristic(board):
    win_combination = [
        [(0,0), (0,1), (0,2)],
        [(1,0), (1,1), (1,2)],
        [(2,0), (2,1), (2,2)],
        [(0,0), (1,0), (2,0)],
        [(0,1), (1,1), (2,1)],
        [(0,2), (1,2), (2,2)],
        [(0,0), (1,1), (2,2)],
        [(0,2), (1,1), (2,0)],
    ]
```

```
# compute heuristic value:
    e_p = sum ( all ( board[x][y] == 'X' or board[x][y] == ''
                    for x,y in combination) for combination in
                    win_combinations)
    e_o = sum( all ( board[x][y] == 'O' or board[x][y] == ''
                    for x,y in combination) for combination in
                    win combinations)

    return e_p - e_o
```

~~toe~~ ~~tac~~

```
tic_tac_toe_board = [ ['X', '' , '' ],
                      ['' , 'X' , '' ],
                      ['' , '', 'X' ]
                    ]

print ('Heuristic value : ', heuristic ( tic_tac_toe_board ) )
```

5. Solve 8 puzzle problem using A* algorithm

```
class Node:
    def __init__ (self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child (self):
        x,y = self.find ('_')
        val_list = [ [x,y-1], [x,y+1], [x-1,y], [x+1,y] ]
        children = [ self.shuffle (x,y, i[0], i[1]) for i in
                    val_list if i[0] >=0 and i[0] < len (self.data)
                    and i[1] >=0 and i[1] < len (self.data) ]
        return [Node (child, self.level+1,0) for child in children
                    if child ]

def shuffle (self, x1, y1, x2, y2):
    temp_puz = [ i[:] for i in self.data]
    temp_puz [x2][y2], temp_puz[x1][y1] = temp_puz[x1][y1],
                                          temp_puz [x2][y2]
    return temp_puz
```

```
    def find(self,char):
        for i in range(len(self.data)):
            for j in range(len(self.data)):
                if self.data[i][j]==char:
                    return i,j

class Puzzle:
    def __init__(self,size):
        self.n=size
        self.open=[]

    def accept(self):
        return [input().split(' ') for _ in range(self.n)]

    def f(self,start,goal):
        return self.h(start.data,goal)+start.level

    def h(self,start,goal):
        return sum(start[i][j]!=goal[i][j] and start[i][j]!=
            '_' for i in range(self.n) for j in range(self.n))

    def process(self):
        print("Start State")
        start=self.accept()
        print("Goal State")
        goal=self.accept()

        start=Node(start,0,0)
        start.fval=self.f(start,goal)
        self.open.append(start)
        while True:
            cur=self.open[0]
            print("\n")
            for i in cur.data:
                for j in i:
                    print(j,end=" ")
                print(" ")

            if self.h(cur.data,goal)==0:
                break
            for i in cur.generate_child():
                i.fval=self.f(i,goal)
                self.open.append(i)
            del self.open[0]
            self.open.sort(key=lambda x:x.fval,reverse=False)
```

```
puz = Puzzle(3)
puz.process()
```

6. WAP to implement steepest ascent hill climbing for 8-puzzle problem. Develop a appropriate heuristic function.

→
```
import numpy as np
import copy

class Node:
    def __init__(self, state, parent):
        self.state = state
        self.parent = parent

def manhatten_distance(state):
    goal_state = np.array([[0,1,2],[3,4,5],[6,7,8]])
    return sum(abs(b%3 - g%3) + abs(b//3 - g//3)
               for b,g in (state[i,j], goal_state[i,j])
               for i in range(3) for j in range(3))
               if b!=0)

def get_neighbors(node):
    neighbors = []
    actions = [(0,-1), (-1,0), (0,1), (1,0)] # L, U, R, D
    zero_pos = tuple(map(int, np.where(node.state==0)))
    for action in actions:
        new_pos = tuple(sum(x) for x in zip(zero_pos, action))
        if 0 <= new_pos[0] < 3 and 0 <= new_pos[1] < 3:
            new_state = copy.deepcopy(node.state)
            new_state[zero_pos], new_state[new_pos] =
                new_state[new_pos], new_state[zero_pos]
            neighbors.append(Node(new_state, node))
    return neighbors

def steepest_ascent_hill_climbing(start_state):
    current_node = Node(start_state, None)
    while True:
        neighbors = get_neighbors(current_node)
        if not neighbors:
            break
        neighbor = min(neighbors, key=lambda node:
                       manhattan_distance(node.state))
        if manhattan_distance(neighbor.state) >=
                    " " (current_node.state):
            break
```

```
        current_node = neighbor

    return current_node.state


start_state = np.array([[7,2,4],[5,0,6],[8,3,1]])
print('Start State:')
print(start_state)
print(steepest_ascent_hill_climbing(start_state))
```