U18CO018
Shubham Shekhaliya
Lab Assignment 5
AIML

Implement 8 Puzzle problem using below algorithms in Python. Compare the complexity of both algorithms.

Code:-

```
from collections import deque
import time

#goal state
goal_state = [1,2,3,4,5,6,7,8,0]

class Node:
    def __init__(self, state, parent, operator, depth, cost):
        self.state = state
        self.parent = parent
        self.operator = operator  # i.e. left , right , up, down
        self.depth = depth
        self.cost = cost
        self.heuristic=None

def display_board(state):
    print( "-------------")
    print( "| %i | %i | %i |" % (state[0], state[1], state[2]))
    print( "-------------")
    print( "| %i | %i | %i |" % (state[3], state[4], state[5]))
    print( "-------------")
    print( "| %i | %i | %i |" % (state[6], state[7], state[8]))
    print( "-------------")

def heuristic(state, goal):     # here heuristic is number of misplaced tiles
    not_match = 0
    for i in range(0,9):
        if state.state[i] != goal[i]:
            not_match += 1
    state.heuristic = not_match

def create_node(state,parent,operator,depth,cost):
```

```python
    return Node(state,parent,operator,depth,cost)

def move_left(state):
    new_state = state[:]
    index = new_state.index(0)

    if index not in [0,3,6]:
        new_state[index-1],new_state[index] = new_state[index],new_state[index-1]
        return new_state
    return None

def move_right(state):
    new_state = state[:]
    index = new_state.index(0)

    if index not in [2,5,8]:
        new_state[index+1],new_state[index] = new_state[index],new_state[index+1]
        return new_state
    return None

def move_up(state):
    new_state = state[:]
    index = new_state.index(0)

    if index not in [0,1,2]:
        new_state[index],new_state[index-3] = new_state[index-3],new_state[index]
        return new_state
    return None

def move_down(state):
    new_state = state[:]
    index = new_state.index(0)

    if index not in [6,7,8]:
        new_state[index],new_state[index + 3] = new_state[index + 3],new_state[in
dex]
        return new_state
    return None

def expand_node(node):
    expanded_nodes = []
    expanded_nodes.append(create_node(move_up(node.state),node,'u'
                                                       ,node.depth+1,0))
    expanded_nodes.append(create_node(move_down(node.state),node,'d',
                                                       node.depth+1,0))
```

```
    expanded_nodes.append(create_node(move_left(node.state),node,'l'
                                                      ,node.depth+1,0))
    expanded_nodes.append(create_node(move_right(node.state),node,'r'
                                                      ,node.depth+1,0))

    expanded_nodes  = [node for node in expanded_nodes if node.state != None]
    return expanded_nodes
```

1. Breadth First Search

```
def bfs(start,goal):
    start_time = time.time()
    start_node = create_node(start,None,None,0,0)

    queue = deque()
    current = start_node

    path = []
    while current.state != goal:
        temp = expand_node(current)
        for item in temp:
            queue.append(item)
        current = queue.popleft()

    while (current.parent != None):
        path.insert(0,current.operator)
        current = current.parent

    print(path)
    print("--- %s seconds ---" % (time.time() - start_time))
```

2. Depth First Search

```
def dfsHelper(list,goal):
    start_time = time.time()
    temp_node = create_node(list,None,None,0,0)

    def dfs(start_node,goal,depth):

        if depth>10:        # recursion limit
            return [False,None]

        if(start_node.state == goal):
            return [True,[]]

        temp = expand_node(start_node)
        for item in temp:
            [ans,path] = dfs(item,goal,depth+1)
            if(ans == True):
                if(item.operator != None):
                    path.append(item.operator)
                return [True,path]

        return [False,None]

    [a,b] = dfs(temp_node,goal,0)
    if(a == True):
        print(b[::-1])
    else:
        print("No Solution Exists")
    print("--- %s seconds ---" % (time.time() - start_time))
```

3. Uniform Cost Search

```python
def ucs(start,goal):
    start_time = time.time()
    start_node = create_node(start,None,None,0,0)

    pq = []
    path = []

    current = start_node

    while current.state != goal:
        temp = expand_node(current)
        for item in temp:
            item.depth += current.depth
            pq.append(item)
        pq.sort(key = lambda x:x.depth)     #sort according to depth
        current = pq.pop(0)

    while (current.parent != None):
        path.insert(0,current.operator)
        current = current.parent

    print(path)
    print("--- %s seconds ---" % (time.time() - start_time))
```

4. Greedy Best First Search

```python
def greedy(start,goal):
    start_time = time.time()
    start_node = create_node(start,None,None,0,0)

    pq = []
    path = []

    current = start_node

    while current.state != goal:
        temp = expand_node(current)
        for item in temp:
            heuristic(item,goal)
            pq.append(item)
        pq.sort(key = lambda x:x.heuristic)     #heuristic value wise sort
        current = pq.pop(0)
```

```python
    while (current.parent != None):
        path.insert(0,current.operator)
        current = current.parent

    print(path)
    print("--- %s seconds ---" % (time.time() - start_time))
```

```python
if __name__ == "__main__":
    list=[1,0,2,4,5,3,7,8,6]
    display_board(list)
    print("\n\nUsing dfs")
    dfsHelper(list,goal_state)
    print("\n\nUsing bfs")
    bfs(list,goal_state)
    print("\n\nUsing Uniform cost search")
    ucs(list,goal_state)
    print("\n\nUsing Greedy Best first search")
    greedy(list,goal_state)
```

Output :-

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: Cod

```
D:\xampp\htdocs\Assignments>python -u "d:\xampp\htdocs\Assignments\AIML\Assignment-5.py"
-------------
| 1 | 0 | 2 |
-------------
| 4 | 5 | 3 |
-------------
| 7 | 8 | 6 |
-------------


Using dfs
['d', 'u', 'd', 'u', 'd', 'u', 'r', 'd', 'd']
--- 0.0010290145874023438 seconds ---


Using bfs
['r', 'd', 'd']
--- 0.0009646415710449219 seconds ---


Using Uniform cost search
['r', 'd', 'd']
--- 0.0020413398742675578 seconds ---


Using Greedy Best first search
['r', 'd', 'd']
--- 0.0 seconds ---

D:\xampp\htdocs\Assignments>
```

Which algorithm is best suited for implementing 8 Puzzle problem and why?

- Greedy Best Search is most suitable algorithm as it use heuristic value an explore fewer node for traversal.
- It is more efficient than that of BFS and DFS.
- Time complexity of Best first search is much less than Breadth first search. The Best first search allows us to switch between paths by gaining the benefits of both breadth first and depth first search