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):

    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[index]

        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**

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**

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)

    while (current.parent != None):

        path.insert(0,current.operator)

        current = current.parent

    print(path)

    print("--- %s seconds ---" % (time.time() - start\_time))

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)

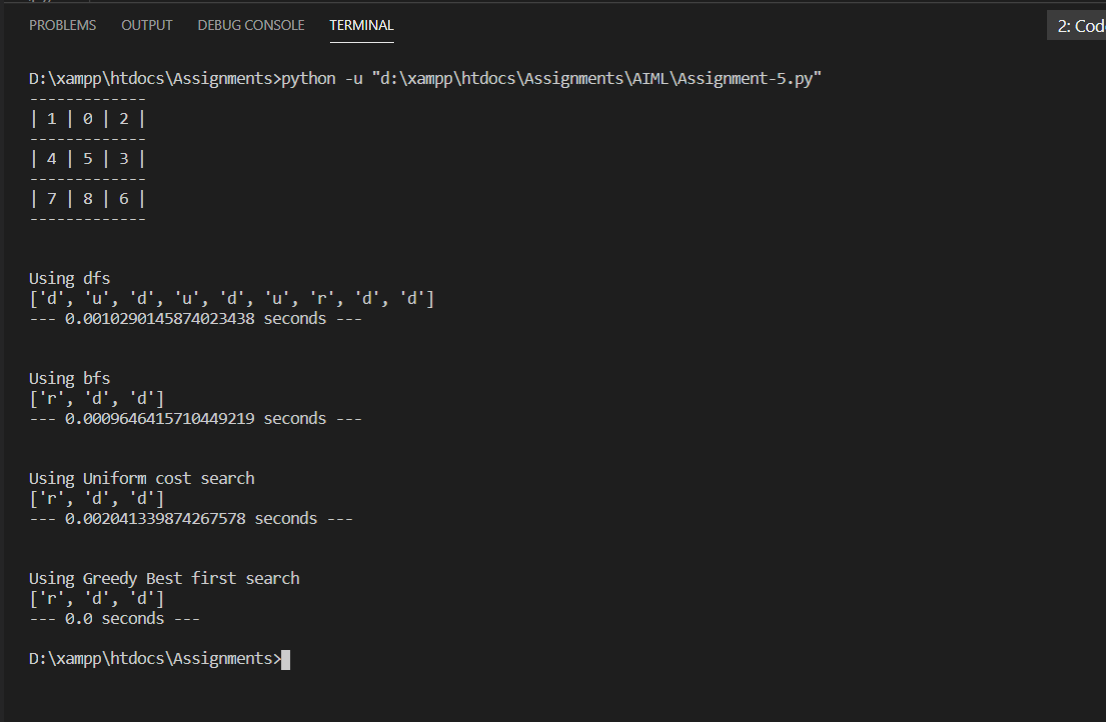
    print("\n\nUsing Uniform cost search")

    ucs(list,goal\_state)

    print("\n\nUsing Greedy Best first search")

    greedy(list,goal\_state)

**Output :-**

****

**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