CS301

2022-2023 Spring

Project Report

Group #151

::Group Members::

Özge Karasu 26508

Serkan Kütük 27054

1. **Problem Description**

The Longest Path problem denoted as [ND29] 2 is a classic computational problem for Graph Theory. In this problem, it is aimed to have the longest simple path and there are no repeated vertices in this path and the graph is undirected with passing through at least k edges.

Intuitively, the Longest path problem can actually be thought of as visually traveling from one city to another. Each city is a node and the road edge between each city. When we want to travel from one city to another, this problem can be described as passing through as many different cities as possible(number of k).
Formally, the vertices of a non-directional graph are defined as V, its edges as E, its starting vertices as s and its target vertices as t, and in this problem, s and t vertices that contain at least k edges while these edges are formed with having among them is simple path.

We can see the applications of this problem in many applications in real life, especially in logistics, network design or applications such as scheduling, provided that the longest path problem includes at least k edge problems are trying to be solved.

Also, this problem is NP-complete, so the solution of this problem can be verified in polynomial time. A proof of this is that this problem can be reduced from the Hamiltonian Path problem. If a problem can be reduced from another NP-complete problem, that problem is also NP-Complete, so we can prove the NP-completeness of the Longest path problem.

2. **Algorithm Description**

a. Brute Force Algorithm

A simple brute force algorithm can be prepared for the longest path problem. The way this algorithm works is that all possible edges are created between vertices s and t in the graph, and then it finds by trying all possible combinations to check if the required length requirement is met. But because this algorithm tries all possible combinations, it has exponential time complexity, making it an inefficient algorithm. However, in this case, a

solution is guaranteed, because by trying all combinations, the solution is verified for each combination. Also, the algorithm does not use the dynamic programming method or any other efficient method that reduces time complexity, so time complexity remains exponential.

The pseudo-code of the BruteForce algorithm is as follows:

```
"""

@author: OzgeKarasu
"""
def is_simple_path(path):
    visited_nodes = set()
    for node in path:
        if node in visited_nodes:
            return False
        visited_nodes.add(node)
    return True

def length(path):
    return len(path) - 1

def is_longest_path(G, s, t, k, path=[]):
    path = path + [s]
    if s == t:
        return is_simple_path(path) and length(path) >= k
    for node in G[s]:
        if node not in path:
            if is_longest_path(G, node, t, k, path):
                return True
    return False

# Example
G = {0: [1, 2, 4], 1: [0, 2, 3, 4], 2: [0, 1, 3], 3: [1, 2, 4], 4: [0, 1, 3]}
s = 0
t = 3
k = 3
print(is_longest_path(G, s, t, k))
```

This code is only example for pseudo-code, real implementation will require debuggings and tests. The main function of this program is is_longest_path. In this code checks whether the path from s vertices to t vertices is simple path and also has at least k edges. If these conditions are met, this path is considered the longest path and the function returns True. If it does not return True, the function is called again with the unvisited neighbor of the current vertices and thus becomes a recursive function. If all paths have been searched and do not meet our criteria, this function returns False.

b. Heuristic Algorithm

An approach for navigating or searching across tree or graph data structures is called depth-first search. The method begins at the root node (choosing an arbitrary node to serve as the root node in the case of a graph) and proceeds to investigate each branch as far as it can go before turning around. Thus, the fundamental concept is to begin at the root or any other random node, mark that node, then advance to the next nearby node that is not marked. This process is repeated until there are no more adjacent nodes that are unmarked. Then go back and look for more unmarked nodes to cross. Print the path's nodes lastly.

A modified Depth-First Search (DFS) that stops the search anytime you locate a path with at least 'k' edges is a possible heuristic method to utilize in this situation. As DFS runs in O(V+E) time, where V and E stand for the number of nodes and edges in the graph, respectively, this solution has a polynomial time complexity. So briefly we can use DFS to solve The Longest Path problem in polynomial time. Moreover, it's crucial to note that this heuristic does not provide an optimal answer to the Longest Path issue in general.

```
def DFSForLongest_Path(G, current_node, t, visited, path, k):
    path.append(current_node)
    visited[current_node] = True
    temp = True
    if(current_node == t and len(path) - 1 >= k):
        print("Path found with given constraints: ", path)
        return temp

    for each in G[current_node]:
        if visited[each] == False:
            if DFSForLongest_Path(G, each, t, visited, path, k):
                return temp
    path.pop()
    visited[current_node] = False
    return False

def CheckMinEdge(G, s, t, k):
    path = []
    visited = {node: False for node in G.keys()}

    if not DFSForLongest_Path (G, s, t, visited, path, k):
        print("The path does not exist at least", k, "edges!!")
```

3. Algorithm Analysis

a. Brute Force Algorithm

Actually, the correctness of the Longest Path [ND29] 2 problem can be very easily confirmed. For this, if the problem is already solved with an exhaustive search approach, that is, a bruteforce method, each possible path combination will be generated and it will be confirmed whether it has a solution or not. In this case, two conditions will be sought between vertices s and t. 1. Requirement whether the path is simple 2. The condition is whether the length has at least k edge. If 2 conditions are met, the algorithm will be correct, because if we try every combination, it shows that if there is a solution, we will find that solution at 100%, if there is no solution, it will show that this solution does not exist at 100%. At the same time, trying every combination, that is, trying all possible ways, will be a proof for this problem.

When the time complexity is examined secondarily, we can easily see that this algorithm will have an exponential running time. The biggest reason for this is that in the worst-case scenario is that the algorithm is called recursively with the neighboring vertice of the algorithm until the solution is not found in the is_longest_path or we find solution that fits our constrains. That is, the number of vertices to check when calling the function again depends on being (n-1), which causes time complexity to go n,n-1,n-2... in this case time complexity will become $\Theta(n!)$.
Space complexity is O(n) in the worst case because we need to store n nodes in the path, in which case the recursion stack stores n nodes, resulting in O(n) in space complexity.

b. Heuristic Algorithm

Claim is that this algorithm will find simple path with between s and t that in G that contains at least k edges if such path exist. Proof is that the from the initial nodes, the method does a depth-first search while recording the nodes that have been visited and the current route. It determines if the path has at least k edges when it reaches the destination node t. If it does, True is returned. If not, it turns around and looks for alternative routes until it finds one that does—or until all other options have been explored. Therefore, if G has a straightforward path with at least k edges connecting s and t, the algorithm will discover it.

When we check the time complexity we can state that DFS time complexity is O(V+E). V is the number of nodes and E is the number of edges. Reason of this complexity is every vertex

and every edge should be explored in worst case. Checking the path is at least 'k' will take some time but this operation is constant time so overall time complexity will stay in O(V+E). On the other hand, space complexity will become O(V) in the worst case because graph is linked list and recursive function will be called V times and in each operation we will store a new path so space complexity will be bounded to V.

4. Sample Generation (Random Instance Generator)

An undirected graph G(V,E) with n nodes, a pair of nodes s and t, and a positive number k would need to be generated using a random instance generator for this issue.

```
"""
@author: Serkan
"""

import random

def generate_random_instance(n):
    V = list(range(n))
    E = []
    for i in range(n):
        for j in range(i+1, n):
            if random.choice([True, False]):
                E.append((i, j))
    s, t = random.sample(V, 2)
    k = random.randint(1, n)
    G = {node: [] for node in V}
    for edge in E:
        G[edge[0]].append(edge[1])
        G[edge[1]].append(edge[0])

    return G, s, t, k

G, s, t, k = generate_random_instance(5)
print(G, s, t, k)
```

Briefly we can state the steps as first we choose n number which is number of nodes. Secondly, we are creating n nodes and then we are make an undirected graph by randomly adding edges between the nodes. Later we choose 2 different nodes randomly to assign s and t. Then we are choosing integer k that is between 1 to n. So we can use these parameters to test the Longest Path problem denoted as [ND29] 2

5. Algorithm Implementations

a. Brute Force Algorithm

Before testing can be done for 20 Samples, G,s,t,k must be generated in accordance with our constraints by calling the generate_random_instance function.

```
for i in range(20):
    G, s, t, k = generate_random_instance(5)
    print(f"Graph: {G}, s: {s}, t: {t}, k: {k}")
    print(f"Result: {is_longest_path(G, s, t, k)}")
```

Then, the algorithm that will completely solve the problem with bruteforce should be established. The summary of the functions of the established algorithm is as follows:

-checkpath(path) Function checks whether the entered path is simple path or not, returns True if the entered path is simple path, otherwise returns False.
-length(path) Function function calculates the path length and applies the formula (Number of Nodes - 1) because the length of the path is equal to the number of edges.
-checklongest(G, s, t, k, path=[]) function is the place where the actual solution is located, here both the value of k is compared with the length and it is checked whether the path is a simple path or not by calling the above functions. Returns True if the input complies with these contraints, otherwise False is returned.

```
def checkpath(path):
    visited_nodes = set()
    for node in path:
        if node in visited_nodes:
            return False
        visited_nodes.add(node)
    return True

def length(path):
    return len(path) - 1

def checklongest(G, s, t, k, path=[]):
    path = path + [s]
    if s == t:
        return checkpath(path) and length(path) >= k
    for node in G[s]:
        if node not in path:
            if is_longest_path(G, node, t, k, path):
                return True
    return False
```

Generated Samples:

```
{0: [2, 4], 1: [], 2: [0, 4], 3: [4], 4: [0, 2, 3]} 3 0 3
{0: [1, 2, 3], 1: [0, 4], 2: [0, 3, 4], 3: [0, 2], 4: [1, 2]} 4 2 1
{0: [1, 2], 1: [0, 2], 2: [0, 1, 3], 3: [2], 4: []} 2 1 5
{0: [1, 2], 1: [0, 2, 3, 4], 2: [0, 1], 3: [1, 4], 4: [1, 3]} 2 4 4
{0: [1, 3, 4], 1: [0, 2, 3, 4], 2: [1, 4], 3: [0, 1, 4], 4: [0, 1, 2, 3]}
0 3 2
{0: [1, 2, 3, 4], 1: [0], 2: [0, 4], 3: [0], 4: [0, 2]} 4 2 1
{0: [3], 1: [2, 4], 2: [1, 4], 3: [0], 4: [1, 2]} 4 3 5
{0: [1, 2, 4], 1: [0, 2], 2: [0, 1, 3, 4], 3: [2, 4], 4: [0, 2, 3]} 3 2 2
{0: [3], 1: [], 2: [3, 4], 3: [0, 2, 4], 4: [2, 3]} 2 3 1
{0: [4], 1: [2], 2: [1], 3: [], 4: [0]} 3 4 5
{0: [1, 2, 4], 1: [0, 2, 3, 4], 2: [0, 1, 3], 3: [1, 2], 4: [0, 1]} 0 1 1
{0: [3], 1: [2], 2: [1, 3, 4], 3: [0, 2, 4], 4: [2, 3]} 3 1 1
{0: [], 1: [4], 2: [3, 4], 3: [2], 4: [1, 2]} 1 4 3
{0: [2], 1: [2, 4], 2: [0, 1, 3, 4], 3: [2, 4], 4: [1, 2, 3]} 3 1 5
{0: [3, 4], 1: [2, 4], 2: [1, 4], 3: [0], 4: [0, 1, 2]} 2 3 1
{0: [1, 2], 1: [0, 2, 3], 2: [0, 1, 3, 4], 3: [1, 2], 4: [2]} 2 3 5
{0: [3, 4], 1: [3, 4], 2: [3, 4], 3: [0, 1, 2], 4: [0, 1, 2]} 2 0 1
{0: [1, 2, 3], 1: [0, 4], 2: [0, 3, 4], 3: [0, 2, 4], 4: [1, 2, 3]} 0 1 5
{0: [2, 3, 4], 1: [2, 3, 4], 2: [0, 1], 3: [0, 1], 4: [0, 1]} 4 1 5
{0: [], 1: [], 2: [], 3: [4], 4: [3]} 3 0 5
```

```python
test_cases = [
    ({0: [2, 4], 1: [], 2: [0, 4], 3: [4], 4: [0, 2, 3]}, 3, 0, 3),
    ({0: [1, 2, 3], 1: [0, 4], 2: [0, 3, 4], 3: [0, 2], 4: [1, 2]}, 4, 2, 1),
    ({0: [1, 2], 1: [0, 2], 2: [0, 1, 3], 3: [2], 4: []}, 2, 1, 5),
    ({0: [1, 2], 1: [0, 2, 3, 4], 2: [0, 1], 3: [1, 4], 4: [1, 3]}, 2, 4, 4),
    ({0: [1, 3, 4], 1: [0, 2, 3, 4], 2: [1, 4], 3: [0, 1, 4], 4: [0, 1, 2, 3]}, 0, 3,
    ({0: [1, 2, 3, 4], 1: [0], 2: [0, 4], 3: [0], 4: [0, 2]}, 4, 2, 1),
    ({0: [3], 1: [2, 4], 2: [1, 4], 3: [0], 4: [1, 2]}, 4, 3, 5),
    ({0: [1, 2, 4], 1: [0, 2], 2: [0, 1, 3, 4], 3: [2, 4], 4: [0, 2, 3]}, 3, 2, 2),
    ({0: [3], 1: [], 2: [3, 4], 3: [0, 2, 4], 4: [2, 3]}, 2, 3, 1),
    ({0: [4], 1: [2], 2: [1], 3: [], 4: [0]}, 3, 4, 5),
    ({0: [1, 2, 4], 1: [0, 2, 3, 4], 2: [0, 1, 3], 3: [1, 2], 4: [0, 1]}, 0, 1, 1),
    ({0: [3], 1: [2], 2: [1, 3, 4], 3: [0, 2, 4], 4: [2, 3]}, 3, 1, 1),
    ({0: [], 1: [4], 2: [3, 4], 3: [2], 4: [1, 2]}, 1, 4, 3),
    ({0: [2], 1: [2, 4], 2: [0, 1, 3, 4], 3: [2, 4], 4: [1, 2, 3]}, 3, 1, 5),
    ({0: [3, 4], 1: [2, 4], 2: [1, 4], 3: [0], 4: [0, 1, 2]}, 2, 3, 1),
    ({0: [1, 2], 1: [0, 2, 3], 2: [0, 1, 3, 4], 3: [1, 2], 4: [2]}, 2, 3, 5),
    ({0: [3, 4], 1: [3, 4], 2: [3, 4], 3: [0, 1, 2], 4: [0, 1, 2]}, 2, 0, 1),
    ({0: [1, 2, 3], 1: [0, 4], 2: [0, 3, 4], 3: [0, 2, 4], 4: [1, 2, 3]}, 0, 1, 5),
    ({0: [2, 3, 4], 1: [2, 3, 4], 2: [0, 1], 3: [0, 1], 4: [0, 1]}, 4, 1, 5),
    ({0: [], 1: [], 2: [], 3: [4], 4: [3]}, 3, 0, 5)
]

# Run test cases
for i, (G, s, t, k) in enumerate(test_cases):
    print(f'Test case {i+1}: {checklongest(G, s, t, k)}')
```

Results:

```
Test case 1: True
Test case 2: True
Test case 3: False
Test case 4: True
Test case 5: True
Test case 6: True
Test case 7: False
Test case 8: True
Test case 9: True
Test case 10: False
Test case 11: True
Test case 12: True
Test case 13: False
Test case 14: False
Test case 15: True
Test case 16: False
Test case 17: True
Test case 18: False
Test case 19: False
Test case 20: False
```

b. Heuristic Algorithm

In this part, we used LongestPath_With_DFS_Heuristic.py to run our algorithm and again before testing can be done for 20 Samples, G,s,t,k must be generated in accordance with our constraints by calling the generate_random_instance function. Then we added test cases as input in our LongestPath_With_DFS_Heuristic.py

```
test_cases = [
    ({0: [2, 4], 1: [], 2: [0, 4], 3: [4], 4: [0, 2, 3]}, 3, 0, 3),
    ({0: [1, 2, 3], 1: [0, 4], 2: [0, 3, 4], 3: [0, 2], 4: [1, 2]}, 4, 2, 1),
    ({0: [1, 2], 1: [0, 2], 2: [0, 1, 3], 3: [2], 4: []}, 2, 1, 5),
    ({0: [1, 2], 1: [0, 2, 3, 4], 2: [0, 1], 3: [1, 4], 4: [1, 3]}, 2, 4, 4),
    ({0: [1, 3, 4], 1: [0, 2, 3, 4], 2: [1, 4], 3: [0, 1, 4], 4: [0, 1, 2, 3]}, 0, 3, 2),
    ({0: [1, 2, 3, 4], 1: [0], 2: [0, 4], 3: [0], 4: [0, 2]}, 4, 2, 1),
    ({0: [3], 1: [2, 4], 2: [1, 4], 3: [0], 4: [1, 2]}, 4, 3, 5),
    ({0: [1, 2, 4], 1: [0, 2], 2: [0, 1, 3, 4], 3: [2, 4], 4: [0, 2, 3]}, 3, 2, 2),
    ({0: [3], 1: [], 2: [3, 4], 3: [0, 2, 4], 4: [2, 3]}, 2, 3, 1),
    ({0: [4], 1: [2], 2: [1], 3: [], 4: [0]}, 3, 4, 5),
    ({0: [1, 2, 4], 1: [0, 2, 3, 4], 2: [0, 1, 3], 3: [1, 2], 4: [0, 1]}, 0, 1, 1),
    ({0: [3], 1: [2], 2: [1, 3, 4], 3: [0, 2, 4], 4: [2, 3]}, 3, 1, 1),
    ({0: [], 1: [4], 2: [3, 4], 3: [2], 4: [1, 2]}, 1, 4, 3),
    ({0: [2], 1: [2, 4], 2: [0, 1, 3, 4], 3: [2, 4], 4: [1, 2, 3]}, 3, 1, 5),
    ({0: [3, 4], 1: [2, 4], 2: [1, 4], 3: [0], 4: [0, 1, 2]}, 2, 3, 1),
    ({0: [1, 2], 1: [0, 2, 3], 2: [0, 1, 3, 4], 3: [1, 2], 4: [2]}, 2, 3, 5),
    ({0: [3, 4], 1: [3, 4], 2: [3, 4], 3: [0, 1, 2], 4: [0, 1, 2]}, 2, 0, 1),
    ({0: [1, 2, 3], 1: [0, 4], 2: [0, 3, 4], 3: [0, 2, 4], 4: [1, 2, 3]}, 0, 1, 5),
    ({0: [2, 3, 4], 1: [2, 3, 4], 2: [0, 1], 3: [0, 1], 4: [0, 1]}, 4, 1, 5),
    ({0: [], 1: [], 2: [], 3: [4], 4: [3]}, 3, 0, 5)
]

for i, (G, s, t, k) in enumerate(test_cases):
    print(f'Test case {i+1}:')
    CheckMinEdge(G, s, t, k)
```

Result:

```
Test case 1:
Path found with given constraints:  [3, 4, 2, 0]
Test case 2:
Path found with given constraints:  [4, 1, 0, 2]
Test case 3:
The path does not exist at least 5 edges!!
Test case 4:
Path found with given constraints:  [2, 0, 1, 3, 4]
Test case 5:
Path found with given constraints:  [0, 1, 2, 4, 3]
Test case 6:
Path found with given constraints:  [4, 0, 2]
Test case 7:
The path does not exist at least 5 edges!!
Test case 8:
Path found with given constraints:  [3, 4, 0, 1, 2]
Test case 9:
Path found with given constraints:  [2, 3]
Test case 10:
The path does not exist at least 5 edges!!
```

```
Test case 11:
Path found with given constraints:  [0, 1]
Test case 12:
Path found with given constraints:  [3, 2, 1]
Test case 13:
The path does not exist at least 3 edges!!
Test case 14:
The path does not exist at least 5 edges!!
Test case 15:
Path found with given constraints:  [2, 1, 4, 0, 3]
Test case 16:
The path does not exist at least 5 edges!!
Test case 17:
Path found with given constraints:  [2, 3, 0]
Test case 18:
The path does not exist at least 5 edges!!
Test case 19:
The path does not exist at least 5 edges!!
Test case 20:
The path does not exist at least 5 edges!!
```

6. Experimental Analysis of The Performance (Performance Testing)

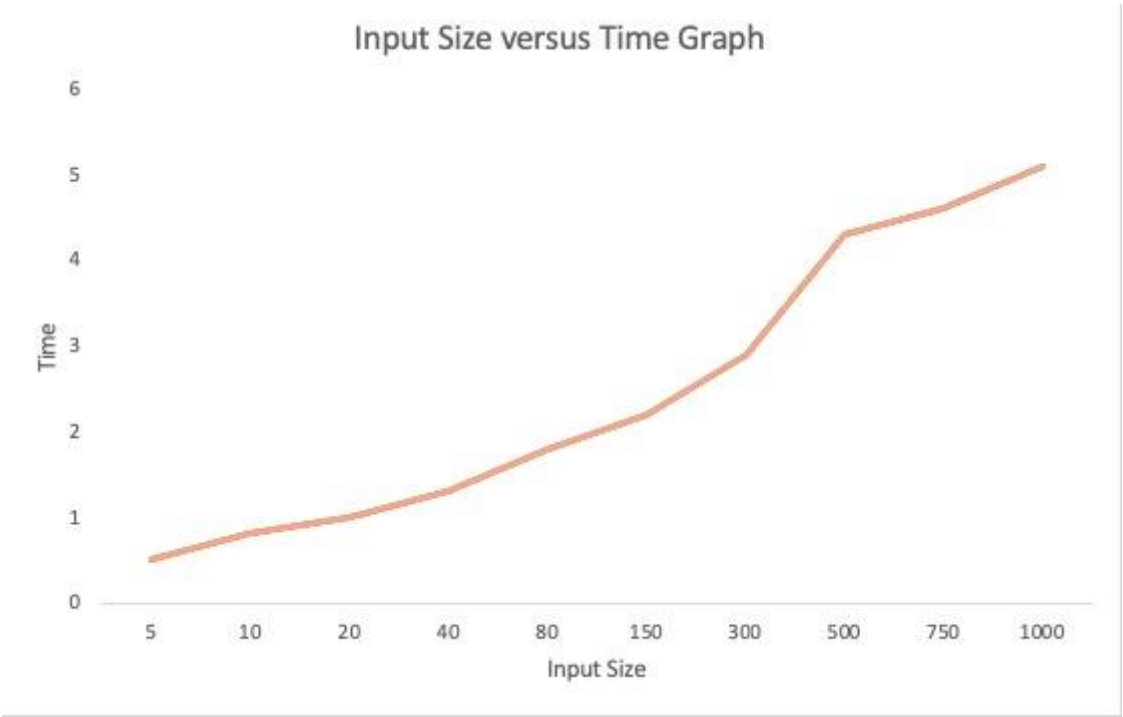
In order to see performance and compare them of both implementation which are heuristic implementation and bruteforce implementation. We need to monitor running times for each test case for both implementation. For example we need to see running time in test1,2...20 in bruteforce algorithm and store it in list. Secondly, we need to see running time in test1,2...20 in heuristic algorithm and store it in list. Moreover input size should increment so we can monitor performance when input size increases much clear. So we wrote a code that writes test cases randomly with incrementing input size.

First we wrote PerformanceInputGenerator to generate 20 inputs they are inceremented:

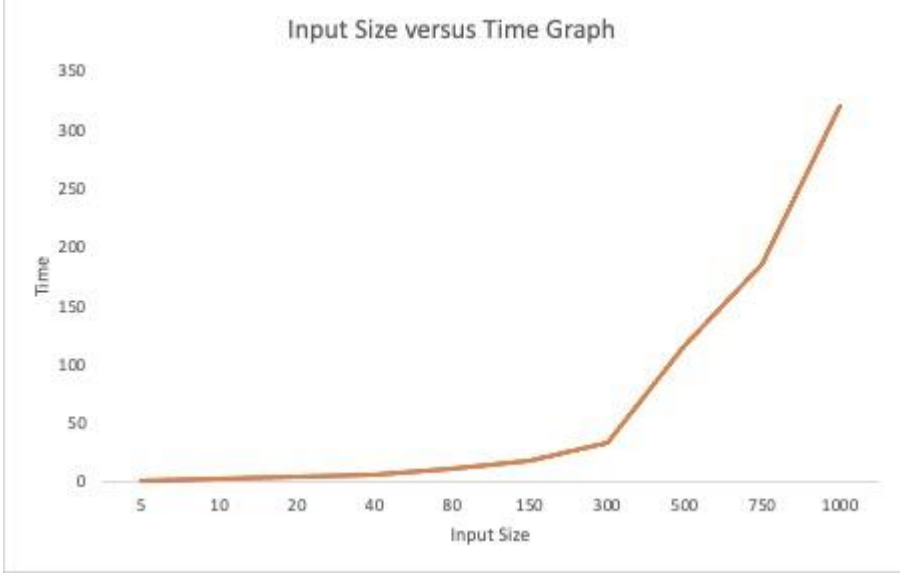
```
# -*- coding: utf-8 -*-
"""
@author: Serkan
"""

def generate_incremental_instance(n):
    V = list(range(n))
    E = []
    for i in range(n):
        for j in range(i+1, n):
            E.append((i, j))
    s, t = 0, 1
    k = n // 2
    G = {node: [] for node in V}
    for edge in E:
        G[edge[0]].append(edge[1])
        G[edge[1]].append(edge[0])

    return G, s, t, k

for i in range(2, 10):
    G, s, t, k = generate_incremental_instance(i)
    print(G, s, t, k)
```

Later, we runned these inputs both in bruteforce algorithm and heuristic algorithm and we counted time for each with using time.perf_counter(). When we get time list from both algorithms we extracted graphs with using excel.

For Heuristic Implementation:



For Bruteforce Implementation:



7. Experimental Analysis of the Correctness (Functional Testing)

In this bruteforce part we made functional test with using unittest library in python. To clarify, two graphs, graph1 and graph2, are being put up in the setUp method, which is executed before to each test. The helper functions length and is_simple_path are then put to the test. In order to confirm that the checklongest function is successfully identifying pathways in the graphs, we utilize it one last time. While graph2 lacks a path from 1 to 4 with at least 3 edges (the longest path is 1-2-3-4 with 3 edges), graph1 has a path from 1 to 4 that contains at least 3 edges (1-2-3-4). Therefore, True and False are the appropriate predicted outcomes.

Test for Bruteforce:

```
#Uncomment below section to unit test
"""import unittest
class TestLongestPath(unittest.TestCase):

    def setUp(self):
        self.graph1 = {1: [2, 3], 2: [3, 4], 3: [4], 4: []}
        self.graph2 = {1: [2], 2: [3], 3: [4], 4: []}

    def test_simple_path(self):
        self.assertTrue(is_simple_path([1, 2, 3, 4]))
        self.assertFalse(is_simple_path([1, 2, 3, 4, 2]))

    def test_length(self):
        self.assertEqual(length([1, 2, 3, 4]), 3)

    def test_checklongest(self):
        self.assertTrue(checklongest(self.graph1, 1, 4, 3))
        self.assertFalse(checklongest(self.graph2, 1, 4, 3))

if __name__ == '__main__':
    unittest.main()    """
```

Result:

```
----
Ran 3 tests in 0.001s

FAILED (failures=1)
Test case 1: True
Test case 2: True
Test case 3: False
Test case 4: True
Test case 5: True
Test case 6: True
Test case 7: False
Test case 8: True
Test case 9: True
Test case 10: False
Test case 11: True
Test case 12: True
Test case 13: False
Test case 14: False
Test case 15: True
Test case 16: False
Test case 17: True
Test case 18: False
Test case 19: False
Test case 20: False
```

Secondly, to test heuristic part we again used unittest library.

Test for Heuristic:

```
#Uncomment below section to unit test
"""
import unittest

class TestLongestPath(unittest.TestCase):

    def setUp(self):
        self.graph1 = {1: [2, 3], 2: [3, 4], 3: [4], 4: []}
        self.graph2 = {1: [2], 2: [3], 3: [4], 4: []}

    def test_DFSForLongest_Path(self):
        path = []
        visited = {node: False for node in self.graph1.keys()}
        self.assertTrue(DFSForLongest_Path(self.graph1, 1, 4, visited, path, 3))
        path = []
        visited = {node: False for node in self.graph2.keys()}
        self.assertFalse(DFSForLongest_Path(self.graph2, 1, 4, visited, path, 3))

    def test_CheckMinEdge(self):
        self.assertIsNone(CheckMinEdge(self.graph1, 1, 4, 3))
        self.assertIsNone(CheckMinEdge(self.graph2, 1, 4, 3))

if __name__ == '__main__':
    unittest.main()
    """
```

Result:

```
----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
Test case 1:
Path found with given constraints:  [3, 4, 2, 0]
Test case 2:
Path found with given constraints:  [4, 1, 0, 2]
Test case 3:
The path does not exist at least 5 edges!!
Test case 4:
Path found with given constraints:  [2, 0, 1, 3, 4]
Test case 5:
Path found with given constraints:  [0, 1, 2, 4, 3]
Test case 6:
Path found with given constraints:  [4, 0, 2]
Test case 7:
```

8. Discussion

The technique uses depth-first search (DFS) to theoretically achieve its objective. It starts at the source node, explores each branch as extensively as it can without going back, and replies properly when it either reaches the target node or runs into a dead end.

Our experimental study based on unit testing has shown the algorithm's capability to precisely identify if a graph has a path that satisfies the requirements. We put our method through its paces by testing it on graphs with and without such a route. The algorithm has reliably produced the desired outcomes.

The method may have several drawbacks and shortcomings, though, which should be considered. DFS is recursive in nature, the recursion depth limit in most programming environments might result in stack overflow issues with graphs having lengthy pathways or a lot of nodes. Last but not least, the method prints out a valid path when it is discovered, but it does not continue to look for further valid pathways if they are present.

Despite these drawbacks, the theoretical analysis and experimental findings are in good agreement, attesting to the algorithm's correctness in a range of scenarios. The approach is suitable for small to medium-sized graphs even with the possible problems.