CS301

2022-2023 Spring

Project Report

Group 152

::Group Members::

Safa Abdullah Söğütlügil, 29214

Pelinsu Saraç, 28820

1. Problem Description

What Longest Path Problem is about actually lies in its name: it is the problem that is about finding the longest simple path possible (or simple path with the maximum possible length) of a given graph. One can also specify the beginning and ending vertices of the path. This intuitive definition can be considered as an optimization problem. However, we can describe it as a decision problem, too. Decision version of the problem and a more formal description would be as the following:

"Given an n-node undirected graph G(V,E); vertices s and t from V and a positive integer k; is there a simple path between vertices s and t in G; which contains at least k edges?"

In terms of usage in practice, finding the longest path in undirected graphs seems to have fewer application areas compared to other problems. An example is, if the graph is directed and acyclic, finding the longest path helps with analysis of electronic circuits (Ando et al., 2002). However, as stated, no clear, practical application of the longest path in an undirected graph was found as a result of our research.

The decision version of the longest path problem is NP-complete. We can show this by reducing another NP-complete problem, namely Hamiltonian Path Problem, to Longest Path Problem. The process of reducing is as follows: for a given instance of Hamiltonian Path Problem with graph G(V, E), we can create a Longest Path Problem out of it by assigning graph of Longest Path as G and threshold value k as $|V| - 1$. Finally, we can conclude that G contains a Hamiltonian Path if and only if there is a simple path of length k in G.

For further discussion, Combinatorial Optimization: Polyhedra and Efficiency (Schrijver, 2003, pp. 114-115) and Introduction to Algorithm (Cormen et al., 2022, p. 1043) can be checked.

2. Algorithm Description

a. Brute Force Algorithm

A basic brute force algorithm for finding the longest path would be no different than what can be done for the shortest path problem: generating all possible paths and finding the longest among them. However, for this report, the algorithm that will be mentioned below is to solve the decision version of the problem, i.e. the formal way.

Despite the difference in the formulation of the problem, what the algorithm does fundamentally isn't changing; it's still generating all possible paths starting from vertex s. What is different is that, at each generated path, it checks whether this path terminates at vertex t and has a length more than or equal to the given integer k. Thus, it can technically terminate before generating all paths, but this doesn't change the fact that for the worst-case scenario, all paths must be generated. This leads to a possible upper bound of $O(2^n*n^2)$ in terms of time complexity. Details of the worst case scenario and time complexity analysis will be given in the Algorithm Analysis section.

The way that the algorithm generates all possible paths starting from s is based on the Depth-First search (DFS) algorithm. The algorithm starts by initializing a stack with the tuple consisting of vertex s and a "path" only consisting of vertex s. This tuple will be representing the terminating vertex of a generated path and that path itself. Then, algorithm pops this tuple from the stack and checks whether the path is terminating at t and length of the path is at least k. For the first iteration, it is not true thus, it continues with the "else" part. Here, new paths are added to the stack with neighbors of s, following the vertex s. After that, iterations continue.

In a more general way, the algorithm pops the lastly added path from the stack, checks whether it is terminating with t, and has length of at least k. If not, it adds new paths to the stack. These new paths are constructed as follows: on top of the existing path, neighbors of the last vertex that are not already in the path (because otherwise, paths wouldn't be simple, and there would be cycles), are added separately to construct separate paths. Then, these new paths are added to the stack. On the other hand, if the first check is successful, algorithm terminates and returns that path.

This technique allows all possible paths to be added to the stack and evaluated throughout the execution.

Here is the pseudo-code of the algorithm for better understanding:

```
Function longest_path_brute_force(Graph, StartingVertex, EndVertex, Minimum Number of Edges):
    // Create a stack with the starting vertex and its path
    stack ← [(s, [s])]
    // While there are vertices on the stack
    while the stack is not empty:
        // Pop the last vertex and its path from the stack
        (vertex, path) ← stack.pop()
        // If the current vertex is the target and the path length is at least k
        if vertex == EndVertex and length of path - 1 >= Minimum Number of Edges:
            print the path
            // Return true to indicate that a path was found
            return true
        // Otherwise, iterate over the neighbors of the current vertex
        else:
            for each neighbor of vertex:
                // If the neighbor is not already in the path
                if neighbor not in path:
                    // Add the neighbor and its path to the stack
                    stack.append((neighbor, path + [neighbor]))
    // Return false to indicate that no path was found
    return false
```

b. Heuristic Algorithm

Before continuing with the algorithm, it must be noted that this algorithm that we have found is trying to answer the optimization version of the main problem. Thus, what it is returning is a candidate for the longest path.

The algorithm found for this section was proposed by Phuong et al. (2019) and it is actually based on a very simple idea (pp. 63-64). Phuong et al. (2019) state that the algorithm is basically DFS, like brute force, but without backtracking (p. 63). In other words, it is just going forward,

as long as there is some neighbor to continue with, and when search cannot continue anymore, the algorithm terminates. The steps of the algorithm are as follows:

1- Start with an arbitrary node from V of G and mark it as visited.
2- Repeat this step as long as it is possible: choose an arbitrary (or according to some rule) node from unvisited neighbors of the lastly visited node and mark it as visited.
3- Return the path that is constructed by the visited nodes, in order (Phuong et al., 2019, pp. 63-64).

One note that Phuong et al. (2019) adds is that if the nodes are named with some integers, then at each iteration of Step 2, one can basically select the neighbor with the smallest number (p. 64). In the implementation of this algorithm, nodes were chosen according to this note. In the implementation, an arbitrary node was also given to the function that is implementing the algorithm as a parameter. Here is the pseudo-code for the implementation of the algorithm:

```
function marked_path(graph(V,E), int start_node) {
        Path = [] //empty list
        int current_vertex = start_node
        While current_vertex is not NULL {
                Add current_vertex to path
                For each neighbor of current_vertex {
                        If neighbor is not in path {
                                Add neighbor to unmarked_neighbors
                        }
                }
                If unmarked_neighbors is not empty {
                        Current_vertex = min(unmarked_neighbors)
                } else {
                        Current_vertex = NULL
                }
        }
        Return path
}
```

The algorithm starts by getting the input graph and a starting node. The algorithm by Phuong et al. (2019) suggests starting with a random node, but it was modified to accept a starting node so that it is easier to compare how close the approximation algorithm is to the brute force solution, which also accepts a starting node (pp. 63-64).

After that, there is an initialization of a list path. As it can be understood from its name, the path is to track the visited nodes in order, and it also works a list to track the marked nodes technically. Then a final initialization comes, current_node, which acts as a cursor to show at which node we are, and because the algorithm is just about to begin, it is initialized to the starting node.

Condition of the while loop can be translated to "as long as there is a neighbor to continue with, continue", which is the main idea behind the algorithm. At each iteration, the current node we are at is added to the path and thus technically becomes marked.

Then, the algorithm needs to check whether there are neighboring nodes that path can continue with. To do that, it iterates over all neighbors of the current node and stores the ones that are not visited yet in another list called unmarked_neighbors. Right after this part, whether there are such nodes or not is checked within an if-else block. If the unmarked_neighbors is empty, it means that it is the end of the algorithm, thus, current_node is updated with NULL value so that the loop can stop and return the path. If not, there are still neighbors to continue with. Among them, current_node is updated with the one acquiring the smallest number as label, like suggested by Phuong et al. (2019).

The ratio bound of this algorithm stated by Phuong et al. (2019), is basically as following:

$$R_{ODFS_{\mathrm{LPath}}}(G) = \frac{OPT_{\mathrm{LPath}}(G)}{ODFS_{\mathrm{LPath}}(G)} < 1 + 4\sqrt{\frac{\log n}{n}}$$

Where OPT represents the true length of the longest path and the ODFS represents the length of path found by the algorithm (p. 66). Hence, R represents the ratio. Of course, n is the number of nodes in a graph, i.e., the problem size.

The full expression for the bound is stated in Theorem 5 in paper (Phuong et al., 2019, p. 66). The original expression stated by theorem is actually probabilistic:

$$\Pr\left[G \in \mathcal{G}_n : R_{ODFS_{\text{LPath}}}(G) < 1 + 4\sqrt{\frac{\log n}{n}}\right] > 1 - \frac{1}{\log n},$$

and told to be true for sufficiently large n. In other words, as stated in Theorem 5 this is the bound for *almost every graph* (Phuong et al., 2019, p. 66).

Proof of the bound starts with bounds of two expressions that form the ratio: OPT and ODFS. OPT is smaller than or equal to n -1, as the path should be simple, and n -1 is the longest possible simple path using all nodes. Another theorem in Phuong et al. (2019), namely Theorem 4, states that ODFS is strictly greater than n-3√(n log n) (pp. 64-66). Theorem 4 is not given here due to its length and also because it uses another theorem, Theorem 3 from the paper (Phuong et al., 2019, p. 63).

Using these two bounds, the ratio bound is constructed as the following in the proof (Phuong et al., 2019, p. 66):

$$R_{ODFS_{\text{LPath}}}(G) = \frac{OPT_{\text{LPath}}(G)}{ODFS_{\text{LPath}}(G)} < \frac{n-1}{n - 3\sqrt{n \log n}} < 1 + 4\sqrt{\frac{\log n}{n}}$$

again, for sufficiently large n.

Before continuing, remarks by Phuong et al. (2019) can be discussed to show the hardness of the problem (p. 66). They say that the longest path problem is NP-hard even in approximate solutions (Phuong et al. 2019, p. 66). In the worst case, they state, unless P = NP, the longest path problem cannot be solved by a polynomial time approximation algorithm, which also has a ratio less than infinity (Phuong et al., 2019, p. 66). However, as their theorem states, the simple approximation algorithm they suggested finds a solution extremely close to optimal in almost every case (Phuong et al., 2019, p. 66).

3. Algorithm Analysis

a. Brute Force Algorithm

Correctness of the algorithm can be seen by a basic proof by contradiction. There are two cases that algorithm terminates: returning true, which means that it found a path from s to t with length at least k, or returning false, which means that it checked all possible paths and decided there is no such path.

For the first case, let's assume that the algorithm made a mistake, there is actually no such path. However, for the algorithm to return true, it checks whether the path is terminated at vertex t and length is at least k. On top of that, paths it checks are all simple paths starting from vertex s. Thus, it cannot happen.

For the second case, let's assume that there is actually such a path, but algorithm couldn't find it. It is basically not possible since algorithm must go through every possible path to reach the "return false" statement, thus, if there had been such path, the algorithm would have found it.

As stated in the algorithm description section, algorithm can terminate before generating and checking all possible paths. However, the worst case for a given graph would be to generate all paths (or a number close to complete number of paths) and check them. We can also go further and think of a graph that makes things worse. As a path consists of edges, if there are maximum possible number of edges present in a graph, it would have induced the worst case. Such a graph is a complete graph, in other words, all possible edges are present in the graph. In short, worst case would be generating all paths with a given complete graph.

Before continuing, we can note how a number of vertices and the number of paths are related in such case. Since all vertices are connected to each other, we can basically think of paths as subsets of the nodes. We know that paths must start from s. Thus, number of possible paths to check would be $2^{(n-1)}$ (we can subtract 1 to remove empty set).

It has been said that, throughout the execution of the algorithm, if not terminated at an early time, all possible paths exist at the stack at some point, and they are then removed. Addition to the stack is done by the single line in the for-loop body (the append method), and removal of paths one by one is handled by first line of the while loop body (pop method). We

can say that lines that handle addition and deletion of paths must be executed for the number of paths, i.e., 2^{n-1} times. We can also understand that while loop will iterate that many times too as removals happen only once at each iteration of while. This helps us to see how many times the if check, which controls the termination conditions will be executed, which is the same with number of iterations of the while loop. Up until this point we have:

- 2^{n-1} (append)
- 2^{n-1} (delete)
- 2^{n-1} (while loop condition check)
- 2^{n-1} (termination if check)

The remaining parts are for loop inside the while loop and the if check inside of it. According to the condition of for loop, it iterates for all neighbors of a vertex, which means n-1 times in our case. If checked inside will also be executed this many times, but we must also note how much time does it take itself. Even if it looks like a single statement, it causes a linear search to happen over the current path to check whether the neighbor is already in or not. Number of vertices at each iteration is changing technically, but since we are looking for an upper bound, we can take it as n (maximum number of vertices in a path). So, we can say that for loop and if check inside takes n(n-1) steps of execution for each path. Then we can combine it with the number of paths: $(2^{n-1})*n(n-1)$.

When all of these added together, dominating term is the $(2^{n-1})*n(n-1)$. -1 in the exponent basically means dividing the expression by 2, thus we don't have to consider it in the big-O notation. What remains is $(2^n)*n(n-1)$, or we can say, $(2^n)*(n^2 - n)$ and the dominating term here would be $(2^n)*n^2$. In other words, worst case time complexity of the algorithm is $O((2^n)*(n^2))$, which grows even faster than exponential time algorithms, hence it is impractical to use for large number of n.

b. Heuristic Algorithm

As the algorithm is an approximation algorithm, we cannot show that whether it finds the exact solution for sure or not. Instead, whether the algorithm terminates and whether it truly produces a simple path is discussed.

Issue whether the algorithm will eventually terminate or not is depending on the while loop, as rest of the algorithm is just initializations or return. By looking at beginning of while loop, it can be said that each iteration of while loop adds a node to path. If this process is somehow limited, then the algorithm will terminate. Technically, it is limited as by definition, a simple path on a graph with limited number of nodes cannot be infinitely long. However, for this argument to be enough, we must also be sure that the algorithm produces simple paths.

Inside the body of while loop, there is another loop that traverses over neighbors of the current node to check whether there are neighbors that are not visited/marked yet. Number of neighbors is limited for all nodes, as the number of nodes in total is limited in a graph. Thus, this inner loop will terminate eventually. Inside this inner loop, whether the neighbor is marked or not is checked. This is done by traversing over current path, which also have a limited number of nodes thus this check will also terminate for each node.

After this check, inside the newly formed unmarked neighbors list, what remains is the neighbors that are not yet visited, i.e. not yet in the path. At the next stage, next node to continue with is only chosen from this list, meaning that path produced never includes a node twice hence what it produces is a simple path.

Worst case scenario for the approximation algorithm in terms of time complexity would be similar to the worst case for brute force, a complete graph. It is because a complete graph is maximizing the number of neighbors to check at each iteration of the while loop. It also allows the maximum possible length of a simple path on a graph with n nodes to occur, which is n-1.

Initialization stages of the algorithm are constant time, i.e. O(1) as they are independent of input size. Because a single node is added to the path at each iteration of the while loop, we can say that the loop will iterate for n-1 times.

Inside the loop body, adding the current node to the path list is again constant time. To find unmarked neighbors, there is an inner loop that traverses over all neighbors of the current node and performs a check. Number of neighbors for all nodes in the worst case is n-1, thus we can say that the inner loop will iterate for n-1 times. The check for marked or not needs a traversal over path list. Length of the path is different in each iteration of the outer while loop, but we can say that it will be n-1 towards the end of execution. Thus, the total complexity of

this inner loop is $O(n^2)$. Finally, the worst-case complexity of the if-else block would be O(n), as in case of there are unmarked neighbors, one with the minimum label is found among them which can be considered to take O(n) time, especially at the beginning of the execution. Otherwise, only the current node is updated to NULL, thus constant time. In the end, the body of the while loop takes $O(n^2)$ time.

Overall complexity of the algorithm would be in this case, $O(n^3)$, which is indeed polynomial time.

4. Sample Generation (Random Instance Generator)

```
import random
import networkx as nx

def generate_random_graph(n, p_edge):
    G = nx.Graph()
    for i in range(n):
        G.add_node(i)
    for i in range(n):
        for j in range(i+1, n):
            p = random.random()
            if p <= p_edge:
                G.add_edge(i, j)
    return G

edge_probability = random.uniform(0, 1)
node_count = random.randint(1, 50)
G = generate_random_graph(node_count, edge_probability)

```

This is a Python function that generates a random undirected graph using the NetworkX library. There are two input parameters:

- n is the number of nodes in the graph
- p_edge is the probability of having an edge between two nodes

First, an empty graph G is generated using the Graph() function from the NetworkX library. Afterwards, n nodes are added to the graph using a for loop and the add_node() function.

Edges are inserted between nodes in the graph with a nested for loop. The outer loop iterates over all nodes in the graph and the inner loop iterates over nodes with an index greater than the current node. This is to avoid having duplicate edges in the graph.
For each pair of nodes, the function generates a random number p between 0 and 1 using the random.random() function. If p is less than or equal to the input parameter p_edge, the function adds an edge between the two nodes using the add_edge() function.
Finally, graph G is returned by the function to be used later in the program.

The function is used with a randomly generated edge_probability variable between 0 and 1 and a random node count of up to 50. Even 50 nodes caused some very long function runs and manual terminations. This is caused by the inefficiency of our brute force algorithm. Apart from taking the problem size (number of nodes = n) as a parameter, our function also takes the probability of there being an edge between any two nodes. This was to observe the effect of sparseness. It demonstrated that a high value of p_edge was extremely significant for the existence or lack of routes with the length of k or more.

5. Algorithm Implementations

a. Brute Force Algorithm

Fixes: In the beginning, we gave a higher upper bound to the number of nodes, but when the randoms worked against us there were multiple occasions where things took way too long, and we could not wait for it. There were a few potential solutions we came up with:
Reducing the potential maximum number of nodes
Reducing the probability of edge probability (increasing sparseness)
Increasing the minimum number of edges (k)

The other thing we had to fix was about the measurement of the length of the path. We started counting the number of nodes in the path, but then we realized that it is supposed to be the number of edges. This was easily fixed by simply editing our algorithm with decreasing the length of the path by 1 during the if check. Trying a smaller number of nodes as input helped with this debugging as it was easier to manually point out the mistakes that we did not think of checking in the first place.

Sample 1

Path: [93, 113, 138, 139, 131, 125, 128, 123, 135, 100, 96, 118, 114, 111, 115, 121, 127, 130, 108, 136, 102, 117, 133, 122, 106, 126, 137, 109, 132, 105, 129, 107, 134, 124, 120, 91, 119, 90]

Result: True

Edge Probability: 0.09784307258820535

Node Count: 140

Minimum number of edges (k): 32

Start Node: 93

End Node: 90

Sample 2

Path: [38, 199, 196, 198, 182, 197, 185, 195, 194, 190, 183, 189, 193, 192, 187, 181, 191, 169, 178, 179, 186, 184, 177, 188, 170, 176, 161, 180, 173, 159, 174, 158, 175, 172, 167, 171, 163, 157, 162, 155, 166, 153, 164, 143, 151, 149, 147, 160, 168, 154, 146, 150, 142, 141, 165, 152, 148, 140, 156, 136, 123, 139, 145, 127, 138, 130, 134, 135, 137, 144, 133, 132, 131, 113, 126, 128, 122, 129, 117, 125, 121, 119, 120, 124, 116, 110, 108, 114, 112, 105, 106, 111, 115, 107, 100, 101, 99, 102, 109, 95, 104, 93, 96, 98, 103, 97, 118, 86, 80, 92, 91, 94, 87, 88, 89, 76, 85, 84, 81, 83, 75, 82, 74, 90, 78, 77, 70, 65]

Result: True

Edge Probability: 0.26441200471813053

Node Count: 200

Minimum number of edges (k): 5

Start Node: 38

End Node: 65

Sample 3

Path: [82, 118, 117, 116, 115, 114, 113, 112, 111, 110, 109, 108, 107, 106, 105, 104, 103, 102, 101, 100, 99, 98, 96, 97, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 81, 80, 79, 78, 77, 76, 74, 75, 73, 72, 71, 70, 69, 68, 67, 66, 65, 63, 64, 62, 61, 59, 60, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41]

Result: True

Edge Probability: 0.9027397128779389

Node Count: 119

Minimum number of edges (k): 33

Start Node: 82

End Node: 41

Sample 4

Path: [5, 125, 130, 129, 124, 126, 128, 122, 127, 121, 120, 123, 119, 118, 117, 116, 115, 112, 113, 114, 110, 111, 108, 106, 109, 105, 104, 107, 103, 97, 99, 101, 100, 96, 102, 98]

Result: True

Edge Probability: 0.5511428974764291

Node Count: 131

Minimum number of edges (k): 29

Start Node: 5

End Node: 98

Sample 5

Result: False

Edge Probability: 0.1686992144229087

Node Count: 4

Minimum number of edges (k): 2

Start Node: 2

End Node: 1

Sample 6

Path: [10, 19, 18, 17, 20, 16, 14, 15, 13]

Result: True

Edge Probability: 0.6162189758455204

Node Count: 21

Minimum number of edges (k): 2

Start Node: 10

End Node: 13

Sample 7

Path: [133, 141, 140, 137, 135, 139, 132, 138, 136, 131, 130, 129, 118, 134, 123, 128, 120, 126, 125, 127, 124, 122, 119, 121, 116, 114, 112, 110, 115, 117, 109, 104, 111, 108, 113, 107, 106, 101, 105, 103, 102, 100, 97, 96, 99, 98, 95, 92, 93, 94, 91, 90, 89, 85, 86, 88, 87, 84, 79, 82, 83, 75, 77, 80, 78, 74, 81, 71, 76, 69, 73, 65, 70, 72, 68, 67, 64, 66, 61, 59, 57, 52, 62, 63, 58, 56, 55, 51, 53, 60, 48, 49, 45, 50, 47, 54, 42, 41, 46, 44, 38, 43, 37, 40, 39, 32, 34, 36, 31, 33, 30, 35, 29, 27, 28, 26, 24, 25, 22, 17, 21, 13, 23, 20, 6, 19, 14, 18]

Result: True

Edge Probability: 0.4363939395478299

Node Count: 142

Minimum number of edges (k): 62

Start Node: 133

End Node: 18

Sample 8

Result: False

Edge Probability: 0.00237746101

Node Count: 54

Minimum number of edges (k): 53

Start Node: 43

End Node: 2

Sample 9

Result: False

Edge Probability: 0.220193143026

Node Count: 13

Minimum number of edges (k): 3

Start Node: 3

End Node: 11

Sample 10

Path: [7, 12, 10, 11, 9, 8, 5, 6, 2, 4, 3]

Result: True

Edge Probability: 0.7791858455293355

Node Count: 13

Minimum number of edges (k): 10

Start Node: 7

End Node: 3

Sample 11

Result: False

Edge Probability: 0.18300716886993507

Node Count: 9

Minimum number of edges (k): 5

Start Node: 8

End Node: 6

Sample 12

Path: [99, 171, 170, 169, 168, 167, 166, 165, 164, 163, 162, 161, 160, 159, 158, 157, 156, 155, 154, 153, 152, 151, 150, 149, 148, 147, 146, 145, 144, 143, 142, 141, 140, 139, 138, 137, 136, 135, 134, 133, 132, 131, 130, 129, 128, 127, 126, 125, 124, 123, 122, 121, 120, 119, 118, 117, 116, 115, 114, 113, 112, 111, 110, 109, 108, 107, 106, 105, 104, 103, 102, 101, 100, 98, 97, 96, 95, 94]

Result: True

Edge Probability: 0.9808047725855611

Node Count: 172

Minimum number of edges (k): 22

Start Node: 99

End Node: 94

Sample 13

Path: [25, 47, 45, 43, 32, 50, 16, 40, 49, 14, 48, 51, 31, 8, 42, 29, 37]

Result: True

Edge Probability: 0.06083832505200615

Node Count: 53

Minimum number of edges (k): 8

Start Node: 25

End Node: 37

Sample 14

Path: [69, 123, 38, 83, 67, 116, 68, 122, 110, 103, 126, 127, 118, 124, 89, 113, 117, 98, 119, 66, 10, 114, 112, 37, 40, 76, 82, 88, 62, 121, 99, 92, 128, 102, 95, 108, 61, 109, 57, 115, 79, 78, 65, 48, 16, 71, 129, 45, 32, 77, 51, 105, 31, 56, 43, 53, 1, 63, 104, 50, 54, 2, 100, 52, 5, 75, 94, 64, 18, 90]

Result: True

Edge Probability: 0.02835215553566295

Node Count: 130

Minimum number of edges (k): 40

Start Node: 69

End Node: 90

Sample 15

Result: False

Edge Probability: 0.16534605447673922

Node Count: 9

Minimum number of edges (k): 6

Start Node: 1

End Node: 5

Sample 16

Path: [1, 6, 4, 3, 8]

Result: True

Edge Probability: 0.34989189840577406

Node Count: 10

Minimum number of edges (k): 3

Start Node: 1

End Node: 8

Sample 17

Path: [5, 16, 15, 13, 14, 12, 11, 9, 10, 8, 6, 7, 3]

Result: True

Edge Probability: 0.8029145091018363

Node Count: 17

Minimum number of edges (k): 7

Start Node: 5

End Node: 3

Sample 18

Result: False

Edge Probability: 0.016918912475792913

Node Count: 65

Minimum number of edges (k): 56

Start Node: 6

End Node: 15

Sample 19

Path: [18, 21, 20, 19, 16, 17, 15, 11, 8]

Result: True

Edge Probability: 0.5767671966891935

Node Count: 22

Minimum number of edges (k): 5

Start Node: 18

End Node: 8

Sample 20

Path: [17, 39, 42, 41, 37, 38, 36, 40, 34, 31, 27, 30, 35, 33, 23, 32, 29, 26, 24, 28]

Result: True

Edge Probability: 0.5127149327460272

Node Count: 43

Minimum number of edges (k): 13

Start Node: 17

End Node: 28

b. Heuristic Algorithm

Sample 1

Edge Probability: 0.26651063310167367

Node Count: 14

Approximation Path Length: 5

Approximation Path: [6, 0, 2, 8, 1, 5]

Brute Force Path Length: 11

Same Path Length: No

Sample 2

Edge Probability: 0.672528242196623

Node Count: 4

Approximation Path Length: 3

Approximation Path: [0, 1, 3, 2]

Brute Force Path Length: 3

Same Path Length: Yes

Sample 3

Edge Probability: 0.2771873644806061

Node Count: 10

Approximation Path Length: 0

Approximation Path: [8]

Brute Force Path Length: 0

Same Path Length: Yes

Sample 4

Edge Probability: 0.7903938568720252

Node Count: 11

Approximation Path Length: 10
Approximation Path: [4, 0, 2, 1, 3, 5, 6, 9, 7, 8, 10]

Brute Force Path Length: 10
Same Path Length: Yes

Sample 5
Edge Probability: 0.34426831844735706
Node Count: 9

Approximation Path Length: 6
Approximation Path: [0, 1, 2, 7, 3, 6, 4]

Brute Force Path Length: 8
Same Path Length: No

Sample 6
Edge Probability: 0.2615623068526348
Node Count: 15

Approximation Path Length: 11
Approximation Path: [9, 1, 8, 3, 6, 4, 7, 2, 13, 5, 0, 12]

Brute Force Path Length: 14
Same Path Length: No

Sample 7
Edge Probability: 0.6505331339467348
Node Count: 4

Approximation Path Length: 3
Approximation Path: [3, 1, 2, 0]

Brute Force Path Length: 3
Same Path Length: Yes

Sample 8
Edge Probability: 0.792260084432634
Node Count: 4

Approximation Path Length: 1
Approximation Path: [0, 1]

Brute Force Path Length: 1
Same Path Length: Yes

Sample 9
Edge Probability: 0.47192780795774436
Node Count: 12

Approximation Path Length: 11
Approximation Path: [1, 0, 3, 2, 4, 6, 7, 5, 10, 8, 11, 9]

Brute Force Path Length: 11
Same Path Length: Yes

Sample 10
Edge Probability: 0.8880249943249453
Node Count: 7

Approximation Path Length: 5
Approximation Path: [0, 1, 2, 5, 3, 4]

Brute Force Path Length: 6
Same Path Length: No

Sample 11

Edge Probability: 0.3411915304531381

Node Count: 13

Approximation Path Length: 9

Approximation Path: [5, 0, 2, 1, 3, 8, 10, 7, 4, 9]

Brute Force Path Length: 11

Same Path Length: No

Sample 12

Edge Probability: 0.7201350704480369

Node Count: 11

Approximation Path Length: 10

Approximation Path: [9, 0, 1, 2, 3, 4, 7, 5, 6, 8, 10]

Brute Force Path Length: 10

Same Path Length: Yes

Sample 13

Edge Probability: 0.9408497973382316

Node Count: 9

Approximation Path Length: 8

Approximation Path: [3, 0, 1, 2, 4, 5, 6, 7, 8]

Brute Force Path Length: 8

Same Path Length: Yes

Sample 14

Edge Probability: 0.4527090342259005

Node Count: 8

Approximation Path Length: 6
Approximation Path: [5, 1, 0, 2, 3, 6, 7]

Brute Force Path Length: 7
Same Path Length: No

Sample 15
Edge Probability: 0.27048385016610066
Node Count: 15

Approximation Path Length: 10
Approximation Path: [14, 11, 0, 1, 2, 3, 7, 4, 8, 5, 6]

Brute Force Path Length: 14
Same Path Length: No

Sample 16
Edge Probability: 0.9577933377575165
Node Count: 11

Approximation Path Length: 10
Approximation Path: [7, 0, 1, 2, 3, 4, 5, 6, 8, 9, 10]

Brute Force Path Length: 10
Same Path Length: Yes

Sample 17
Edge Probability: 0.9758801676592568
Node Count: 7

Approximation Path Length: 6
Approximation Path: [4, 0, 1, 2, 3, 5, 6]

Brute Force Path Length: 6

Same Path Length: Yes

Sample 18

Edge Probability: 0.3546966849353852

Node Count: 11

Approximation Path Length: 3

Approximation Path: [0, 1, 8, 7]

Brute Force Path Length: 3

Same Path Length: Yes

Sample 19

Edge Probability: 0.7402973845314536

Node Count: 8

Approximation Path Length: 7

Approximation Path: [5, 0, 2, 1, 4, 3, 6, 7]

Brute Force Path Length: 7

Same Path Length: Yes

Sample 20

Edge Probability: 0.008132026591724628

Node Count: 8

Approximation Path Length: 0

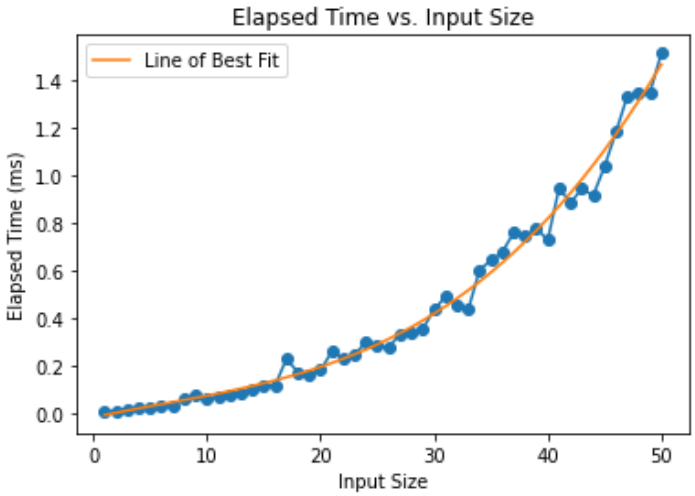
Approximation Path: [6]

Brute Force Path Length: 0

Same Path Length: Yes

6. Experimental Analysis of The Performance (Performance Testing)

To measure the practical time complexity of the approximation algorithm, we decided to try a range of input sizes (n: number of vertices in the graph) between 1 to 50. For each size, time measurements repeated for 100 times and average of the measurements for each size was taken into account. Here is the plot of the averages:



Fitted line on the graph is a polynomial fit with degree 3 (can be checked from codes). Log-log graph was also drawn, however it seemed worse. Considering the fact that the theoretical time complexity of the algorithm is $O(n^3)$, it can be said that the practical and theoretical results are coinciding. Here is the fitted line equation:

$$1.213\text{e-}05\ x^3 - 0.0002138\ x^2 + 0.01002\ x - 0.01363$$

However, to be sure of averages/means representing the true time complexity, confidence intervals must also be considered. For 95% confidence, here are the intervals calculated for some input sizes:

Input Value	Confidence Interval
n = 1	0.01 ±0.0002
n = 10	0.06 ±0.0018
n = 20	0.18 ±0.004
n = 30	0.44 ±0.012
n = 40	0.73 ±0.02
n = 50	1.52 ±0.03

First thing to note is how the range of the interval is increasing with the input size. This can raise a question for large input sizes. However, another thing to consider is the percentage of the plus-minus value. In none of the intervals, plus-minus value/mean ratio is greater than 3%. Another thing is, plus-minus values are already quite small. In the end, averages of tests can be used to represent time needed for a specific input size.

7. Experimental Analysis of the Quality

In addition to time complexity analysis, it was also checked whether the error ratio of the trials is truly bounded by the bound suggested by Phuong et al. (2019). Averages of error ratios from 100 trials were taken for each input size and are shown on the left-hand size of inequality. Bound for each input size is calculated and shown on the right-hand side of the inequality. These tests were only conducted for input sizes from 1 to 12, due to the extreme time complexity of the brute force algorithm. Another thing to note is that, because the original brute force algorithm introduced is for the decision version of the longest path problem, we technically used a different algorithm to be able to compare length of found paths. This new algorithm is not incredibly different from the first one, as this one is basically a modified version of DFS that returns all possible paths starting from a specific node. One extra step to DFS is to find the longest among the returned paths. Here is the Python implementation of the new brute force algorithm:

```
def DFS(G,v,seen=None,path=None):

    paths = []
    if seen is None: seen = []
    if path is None:
        path = [v]
        paths.append(path)

    seen.append(v)


    for t in G[v]:
        if t not in seen:
            t_path = path + [t]
            paths.append(tuple(t_path))
            paths.extend(DFS(G, t, seen[:], t_path))
    return paths

def longest_path_dfs(G, start_node):

    all_paths = DFS(G, start_node)
    max_len   = max(len(p) for p in all_paths)
    max_paths = [p for p in all_paths if len(p) == max_len]

    return (max_len-1, max_paths)
```

Now, here is the ratio bound results:

Input Size	Inequality Suggested	Does it hold?
1	$1 < 1$	Yes
2	$1 < 3.83$	Yes
3	$1 < 3.9$	Yes
4	$1.1 < 3.83$	Yes
5	$1.13 < 3.73$	Yes
6	$1.04 < 3.63$	Yes
7	$1.03 < 3.53$	Yes
8	$1.11 < 3.45$	Yes
9	$1.22 < 3.37$	Yes
10	$1.31 < 3.3$	Yes
11	$1.21 < 3.24$	Yes
12	$1.77 < 3.2$	Yes

As it can be seen in the table, for all input sizes tried, suggested bound holds on average. Looking at the ratios, one can say that the simple approximation algorithm (Phuong et al., 2019, pp. 63-64) does a fairly good job with comparatively small input sizes. Of course, there can and should be tests made with larger input sizes. This can be done within a further study. Also, further statistical analysis can be done with more resources for computation.

8. Experimental Analysis of the Correctness (Functional Testing)

Black Box Testing

Black box testing is done without the knowledge of the source code, documentation or the data. Non-systematic tests were done that are further explained below. They are edge cases and the regular use cases are already used and shown in Part 5 above.

Input:

```
# TEST CASE 1
G = nx.Graph() # Empty Graph
start_node = 0 # Otherwise Valid Start Node
approximation_path = mark_path(G, start_node)
```

Output:

```
Empty Graph
```

Empty graphs can create problems and are checked here. There is no problem and the user gets a proper message.

Input:

```
# TEST CASE 2
edge_probability = random.uniform(0, 1)
node_count = random.randint(1, 15)
start_node = 'Test' # String Start Node
G = generate_random_graph(node_count, edge_probability)
approximation_path = mark_path(G, start_node)
```

Output:

```
Non-integer Start Node
```

A string can be given as the start_node to the function as well. Here this is tested, and it gives the non-integer start node error.

Input:

```
# TEST CASE 3
edge_probability = random.uniform(0, 1)
node_count = random.randint(1, 15)
start_node = 1.537 # Float Start Node
G = generate_random_graph(node_count, edge_probability)
approximation_path = mark_path(G, start_node)
```

Output:

```
Non-integer Start Node
```

A float can be given as the start_node to the function as well. Here this is tested, and it gives the non-integer start node error.

Input:

```
# TEST CASE 4
edge_probability = random.uniform(0, 1)
node_count = random.randint(1, 15)
start_node = -2 # Negative Start Node
G = generate_random_graph(node_count, edge_probability)
approximation_path = mark_path(G, start_node)
```

Output:

```
Negative Start Node
```

The minimum value the start_node can take is 0, which is the case even if there is only 1 node in the graph. When tested, it gives a negative start node error.

Input:

```
# TEST CASE 5
edge_probability = random.uniform(0, 1)
node_count = random.randint(1, 15)
start_node = node_count+1 # Start Node Larger Than Node Count
G = generate_random_graph(node_count, edge_probability)
approximation_path = mark_path(G, start_node)
```

Output:

```
Start Node Too Large
```

The largest value the start node can take is node_count-1, here a value that is larger than that is tested and an appropriate error is displayed.

Input:

```
# TEST CASE 6
edge_probability = random.uniform(0, 1)
node_count = 1 # Graph With 1 Node
start_node = random.randint(0, node_count-1)
G = generate_random_graph(node_count, edge_probability)
approximation_path = mark_path(G, start_node)
```

Output:

```
Initialized variables and set current_vertex to start_node
Marked current_vertex and appended it to the path
There are no unmarked neighbors
Returning the path
```

Another case is when the graph has 1 node only. The output shows that the function runs and returns the path in the end. There are no errors, so it works.

Input:

```
# TEST CASE 7
edge_probability = random.uniform(0, 1)
node_count = random.randint(1, 15)
start_node = random.randint(0, node_count-1)
G = 3 # Non-graph Input
approximation_path = mark_path(G, start_node)
```

Output:

```
Non-graph Input
```

Finally, it is checked whether an input of non-graph parameter is handled. It can be seen from the output that it is covered, and the function prints an appropriate response instead of crashing.

White Box Testing

White box testing is done by someone who has access to the source code and knows how the code works. Here, every line below Step 1 is printed in the test case below. Rest of the checks, i.e. the ones in Step 0, are checked during the black box testing above already. They will not be repeated here again.

It can be seen that the print statements below provide complete coverage of the function.

Source Code

```
def mark_path(graph, start_node):

    start = timer()
    # Step 0: Input checks
    if type(graph) != type(nx.Graph()):
        print('Non-graph Input')
        return []
    if graph.number_of_nodes() == 0:
        print('Empty Graph')
        return []
    if type(start_node) != int:
        print('Non-integer Start Node')
        return []
    if start_node < 0:
        print('Negative Start Node')
        return []
    if start_node > graph.number_of_nodes()-1 :
        print('Start Node Too Large')
        return []

    # Step 1: Initialize variables
    path = []

    # Step 2: Mark vertices until no further marking is possible
    current_vertex = start_node
    print('Initialized variables and set current_vertex to start_node')
    while current_vertex is not None:
        # Mark the current vertex
        path.append(current_vertex)
        print('Marked current_vertex and appended it to the path')

        # Find unmarked neighbors of the current vertex
        unmarked_neighbors = [neighbor for neighbor in graph[current_vertex] if neighbor not in path]

        if unmarked_neighbors:
            print('There are unmarked neighbors')
            # Step 2: Pick the vertex with the smallest integer value among unmarked neighbors
            current_vertex = min(unmarked_neighbors, key=int)
        else:
            print('There are no unmarked neighbors')
            # No unmarked neighbors, terminate the loop
            current_vertex = None

    end = timer()
    time_elapsed = round(1000*(end - start), 6)
    print('Returning the path')
    return (path, time_elapsed)
```

Output

```
Initialized variables and set current_vertex to start_node
Marked current_vertex and appended it to the path
There are unmarked neighbors
Marked current_vertex and appended it to the path
There are unmarked neighbors
Marked current_vertex and appended it to the path
There are no unmarked neighbors
Returning the path
```

9. Discussion

The most important problem we had with the algorithm is not caused by its own shortcoming but is due to the nature of the problem at hand. Unlike the shortest path problem, the variety and number of algorithms here are not as abundant. The reason for this could be that unless P = NP, there is no polynomial approximation algorithm that has an absolute performance ratio less than infinity for the longest path problem. Still, our algorithm works for most cases and is a suitable one for our problem (Phuong et al., 2019, p. 66).

Another issue we had was that the algorithm did not consider any edge cases and assumed correct input from the user. There was an inconsistency in terms of correctness and functionality of the code. This is because the paper we found the article in was mainly interested in the mathematical aspect and the performance in terms of time complexity. We had to keep that in mind and add necessary input checks to our function. This way, even if the user enters a problematic input parameter the program will not crash. Instead, an appropriate error message is printed to the user, and they will know what the issue was. This was further demonstrated in the black box testing part above. Additionally, white box testing has shown that every line of the code works and there is complete coverage.

A potential improvement could be increasing the number of maximum nodes while measuring the error ratio. However, as the brute force DFS algorithm that we had to compare with our approximation algorithm was slowing things down way too much, it was not possible to test higher node counts with our current computational power. This, however, was further proof of the efficiency of our algorithm compared to the brute force approach as it easily handled node counts and iterations 4-5 times larger while measuring the time.

This brings us to the topic of performance. The reason we use approximation algorithms in the first place is their performance. Here, we were aiming for a polynomial solution, and we got it as can be seen in Part 6. The algorithm was tested with up to 50 nodes and with 100 iterations each. A polynomial fit of degree 3 fitted the experimental timing results we got quite well. This shows that the performance of our algorithm actually is polynomial as was our aim and theoretical and experimental analyses are supporting each other. The confidence interval is not too wide, and our mean is a good representation of the data. Error bounds between the input sizes (node count) of 1 and 12 always hold.

References

Ando, E., Yamashita, M., Nakata, T., & Matsunaga, Y. (2002). The statistical longest path problem and its application to delay analysis of logical circuits. *Proceedings of the 8th ACM/IEEE international workshop on Timing issues in the specification and synthesis of digital systems - TAU '02*. https://doi.org/10.1145/589411.589440

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms, fourth edition*. MIT Press.

Phuong, N. T., Duc, T. V., & Thanh, L. C. (2019). On the performance of a simple approximation algorithm for the longest path problem. *Journal of Computer Science and Cybernetics*, *35*(1), 57–68. https://doi.org/10.15625/1813-9663/35/1/12935

Schrijver, A. (2003). *Combinatorial optimization: Polyhedra and efficiency*. Berlin: Springer.