# Sabancı University

# Fall 2019

# PROJECT REPORT

# Longest Path

Artun Sarıoğlu, Gülce Lale, Deniz Ulaş Tura, Ekin Oskay, Ezgi Gözen

## 1. Problem Description

The longest path problem can be defined formally as follows; given an undirected graph *G(V, E)* with V vertices, the vertices s and t which are in V and a positive integer k, the longest path problem asks to compute a simple longest path that exists between s and t in G which contains at least k edges.

Longest path problem is a NP - Hard problem for a general graph. However, the question "Is there a simple path between s and t in G that contains at least k edges?" is a part of NP - Complete class of problems.

**Proof:**

In order to prove a problem is NP-complete problem, the problem needs to be shown as a part of NP and it should be reducible to a NP-complete problem in polynomial time.

According to the study, A Simple Polynomial Algorithm for the Longest Path Problem on Cocomparability Graphs (Mertzios et al., 2010), the proof of the Longest Path problem is a NP – Complete problem can be done by reducing the Hamiltonian path problem since the most natural optimization version of Hamiltonian path problem is the Longest Path problem.

Given a graph G(V, E), has a Hamiltonian path if and only if its longest path has a length of $n - 1$, where n is the number of vertices in G.

Given a solution path P, the statements below should be verified:

- P consists of at least k edges
- These edges form a path.
- For every edge an augmented graph can be constructed which consists of G and two additional vertices that are connected to the edges stated before.

These verifications hold for every edge in E. The reduction follows directly from Hamiltonian Path and the verifications above can be done in polynomial time.

Given an instance of Hamiltonian Path on a graph G = (V, E). An instance of the Longest Path problem can be generated G', k as stated below:

- The same graph is used such that G' = G
- Set k for Vertex Cover = $|V| - 1$.

By adding two edges in G' for every edge in G. We claim that V' is a vertex cover of G if and only if V' is a feedback vertex set of G'. Given a vertex cover V' of G, every edge $\{u, v\} \in E$ must be covered by a vertex in V'. That is, every edge in E has at least one of its endpoints in V'. Thus, G' does not contain a cycle. In a feedback vertex set V' of G' every cycle in G' must contain at least one vertex from V'. Every cycle of length 2 in G' must contain at least one vertex from V'. Every cycle of length 2 corresponds to an undirected edge in G. Thus, every undirected edge in G has at least one of its endpoints in V' and thus V' is a vertex cover for G.

Therefore, there exists a simple path of length k in G' if and only if G' contains a Hamiltonian path. Because the Hamiltonian path problem is NP-complete, the reduction shows that the decision version, the question version of the Longest Path problem is also NP-complete.

The longest path problem has several applications used in practice such as, designing circuit boards, planning robot's patrolling trajectories, information retrieving for robots which requires calculating a critical path.

## 2. Algorithm Description

There is not an exact algorithm that solves Longest Path problem in polynomial time. Our project aims to find a heuristic approach that can approximate and analyze the results.

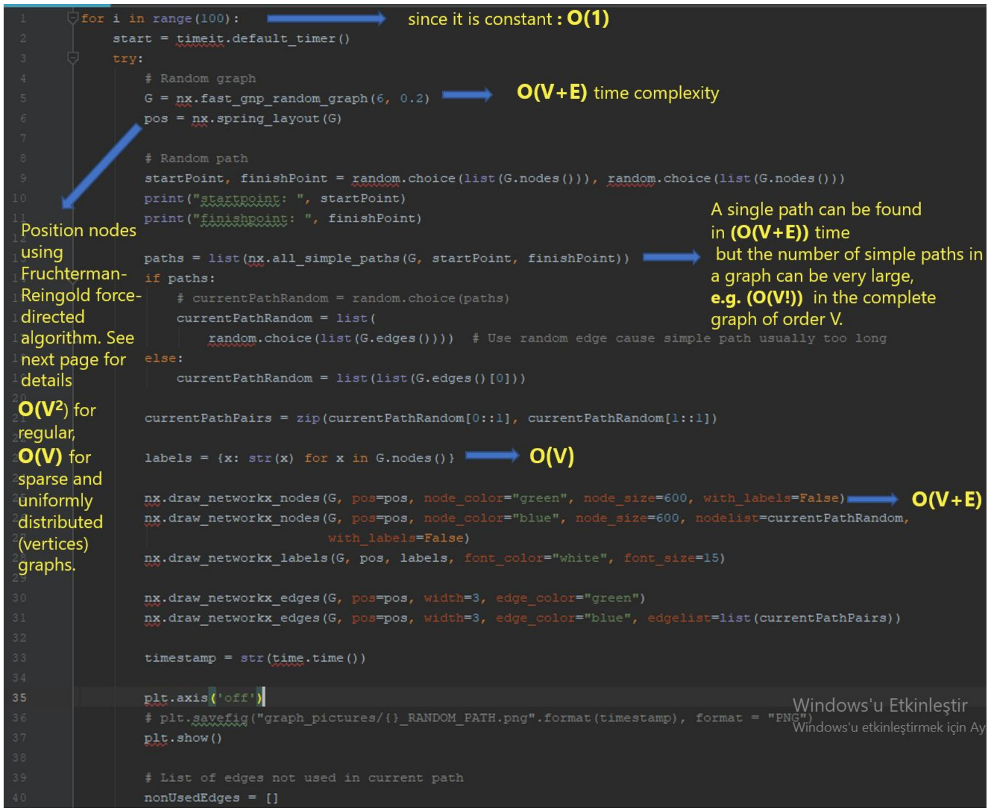
The algorithm starts with a random graph generator algorithm which indeed generates a random graph. We use Erdos Renyl Model for generating random graphs which is the default algorithm in python. In order to evaluate the performance of the algorithm, many runs are needed to be performed. To ease the problem of having different graphs , source and target vertices are selected randomly at each run. This algorithm generates a random graph and then picks two random source and target vertices.

This algorithm takes the number of iterations as input. Theoretically p value is picked randomly between 0 and 1. But for large p values we observed that the algorithm runs for a very long time. Due to time constraints we picked a relatively smaller p for every run such as 0.2 . At every iteration another random value is picked. If this random value is bigger than p, algorithm connects the related edges, if not algorithm does not connect.Strongly connected graph is generated within this way.

After that, by using a brute force approach, the algorithm checks whether there exists a longest path that contains at least k edges.Algorithm determines the longest path as the following; calculate all possible paths and check for each path that whether current path is longer than the current longest path obtained.The algorithm checks the value of k to find a longest path that has at least k edges by trying all possible k values such that k=0,1,....x where x is defined as the number of strongly connected edges.

Thus, the success rate is determined as whether if the algorithm finds such path or not. If it finds, we are successful, if not, the algorithm failed to obtain the longest simple path.

# 3. Algorithm Analysis:



**Algorithm 2:** Fruchterman-Reingold

```
area ← W ∗ L ;                                        /* frame: width W and length L */
initialize G = (V, E) ;                               /* place vertices at random */
k ← √(area/|V|) ;                                     /* compute optimal pairwise distance */
function f_r(x) = k^2/x ;                             /* compute repulsive force */
for i = 1 to iterations do
    foreach v ∈ V do
        v.disp := 0; ;                                /* initialize displacement vector */
        for u ∈ V do
            if (u ≠ v) then
                Δ ← v.pos − u.pos ;                   /* distance between u and v */
                v.disp ← v.disp + (Δ/|Δ|) ∗ f_r(|Δ|) ;    /* displacement */
    function f_a(x) = x^2/k ;                         /* compute attractive force */
    foreach e ∈ E do
        Δ ← e.v.pos − e.u.pos ;                       /* e is ordered vertex pair .v and .u */
        e.v.disp ← e.v.disp − (Δ/|Δ|) ∗ f_a(|Δ|);
        e.u.disp ← e.u.disp + (Δ/|Δ|) ∗ f_a(|Δ|);
    foreach v ∈ V do
                        /* limit max displacement to frame; use temp. t to scale */
        v.pos ← v.pos + (v.disp/|v.disp|) ∗ min(v.disp, t);
        v.pos.x ← min(W/2, max(−W/2, v.pos.x));
        v.pos.y ← min(L/2, max(−L/2, v.pos.y));
    t ← cool(t) ;                                     /* reduce temperature for next iteration */
```

## Fruchterman - Reingold Algorithm:

Each iteration of the basic algorithm computes:

•O(|E|) attractive forces and

• $O(|V|^2)$ repulsive forces.

In order to reduce the quadratic complexity of the repulsive forces, Fruchterman and Reingold suggests that:

• Using a grid variant of their basic algorithm, where the repulsive forces between distant vertices are ignored.

• For sparse graphs, and with uniform distribution of the vertices, this method allows *O(|V|)* time approximation to the repulsive forces calculation.

## 2nd Part:

```
for edge in G.edges():                                  O(E)
    if edge not in currentPathPairs and edge[::-1] not in currentPathPairs:
        nonUsedEdges.append(edge)
        nonUsedEdges.append(edge[::-1])

print(G.edges())

currentPath = currentPathRandom

currentEnergy = calculateEnergy(currentPath)            O(V)
initialTemperature = 100000
endTemperature = 1
T = initialTemperature

for i in range(1, 10000):                               O(1)
    stateCandidate, newNonUsedEdges = generateStateCandidate(currentPath, nonUsedEdges, G)    O(E³V)
    candidateEnergy = calculateEnergy(stateCandidate)   O(V)

    if candidateEnergy >= currentEnergy:
        currentPath = stateCandidate
        currentEnergy = candidateEnergy                 Loop =
        nonUsedEdges = newNonUsedEdges                  O(E³V)
    else:
        p = getTransitionProbability(currentEnergy - candidateEnergy, T)    O(1)
        if isTransition(p):
            currentPath = stateCandidate
            currentEnergy = candidateEnergy
            nonUsedEdges = newNonUsedEdges
    T = decreaseTemperature(initialTemperature, i)
    if T <= endTemperature:
        break

nx.draw_networkx_nodes(G, pos=pos, node_color="green", node_size=600, with_labels=False)
```

# <u>GenerateStateCandidate Function:</u>


```python
def generateStateCandidate(path, availableEdges, G):
    if len(path) == 1:  # Path is equal to node
        path = list(random.choice(availableEdges))  # Take any edge as a new path
        availableEdges = []
        for edge in G.edges():
            availableEdges.append(edge)
            availableEdges.append(edge[::-1])
        availableEdges.remove(tuple(path))
        availableEdges.remove(tuple(path[::-1]))

    if random.choice([True, True, False]):  # Add new edge

        if not availableEdges:  # All edges are in path
            #             print "No available edges"
            return path, availableEdges

        # Path consists from one edge or more
        if random.choice([True, False]):  # Add edge to the start
            #             print "Add to start"
            nodeCandidates = list(G.nodes())
            random.shuffle(nodeCandidates)
            for node in nodeCandidates:
                if (node, path[0]) in availableEdges and (path[0], node) in availableEdges and node != path[-1] and node not in path:
                    #                     print "Adding node - ", node
                    path.insert(0, node)
                    #                     print "New path - ", path
                    availableEdges.remove((path[0], path[1]))
                    availableEdges.remove((path[1], path[0]))
                    break
        else:  # Add edge to the end
            #             print "Add to end"
            nodeCandidates = list(G.nodes())
            random.shuffle(nodeCandidates)
            for node in nodeCandidates:
                if (node, path[-1]) in availableEdges and (path[-1], node) in availableEdges and node != path[0] and node not in path:
                    #                     print "Adding node - ", node
                    path.append(node)
                    #                     print "New path - ", path
                    availableEdges.remove((path[-1], path[-2]))
                    availableEdges.remove((path[-2], path[-1]))
                    break

    else:  # Remove edge
        if random.choice([True, False]):  # Remove start edge
            #             print "Remove start node - ", path[0]
            availableEdges.append((path[0], path[1]))
            availableEdges.append((path[1], path[0]))
            path.pop(0)
        else:  # Remove end edge
            #             print "Remove end node - ", path[-1]
            availableEdges.append((path[-1], path[-2]))
            availableEdges.append((path[-2], path[-1]))
            path.pop(-1)

    return path, availableEdges
```

O((E^3)V) is total complexity of the function

## 3rd Part:

In general, for sparse graphs, and with uniform distribution of the vertices total time complexity is: O(V(E^3))

For other graphs, it is O((V^2)+V(E^3))

```python
        plt.axis('off')
        # plt.savefig("graph_pictures/{}_METROPOLIS_WITH_ANNEALING.png".format(timestamp), format = "PNG")
        plt.show()

        currentPath = currentPathRandom

        currentEnergy = calculateEnergy(currentPath)
        initialTemperature = 100
        endTemperature = 1
        T = initialTemperature

        for i in range(1, 10000):
            stateCandidate, newNonUsedEdges = generateStateCandidate(currentPath, nonUsedEdges, G)
            candidateEnergy = calculateEnergy(stateCandidate)

            if candidateEnergy >= currentEnergy:
                currentPath = stateCandidate
                currentEnergy = candidateEnergy
                nonUsedEdges = newNonUsedEdges
            else:
                p = getTransitionProbability(currentEnergy - candidateEnergy, T)
                if isTransition(p):
                    currentPath = stateCandidate
                    currentEnergy = candidateEnergy
                    nonUsedEdges = newNonUsedEdges
            #     T = decreaseTemperature(initialTemperature, i)
            if T <= endTemperature:
                break

        nx.draw_networkx_nodes(G, pos=pos, node_color="green", node_size=600, with_labels=False)
        nx.draw_networkx_nodes(G, pos=pos, node_color="blue", node_size=600, nodelist=currentPath, with_labels=False)
        nx.draw_networkx_labels(G, pos, labels, font_color="white", font_size=15)

        currentPathPairs = zip(currentPath[0::1], currentPath[1::1])
        nx.draw_networkx_edges(G, pos=pos, width=3, edge_color="green")
        nx.draw_networkx_edges(G, pos=pos, width=3, edge_color="blue", edgelist=list(currentPathPairs))

        plt.axis('off')
        # plt.savefig("graph_pictures/{}_METROPOLIS_WITHOUT_ANNEALING.png".format(timestamp), format = "PNG")
        plt.show()
    except:
        print("can not generate a longest simple path, exiting...")
    end = timeit.default_timer()
    running_time_vector_6.append(end - start)
```

## 4. Experimental analysis

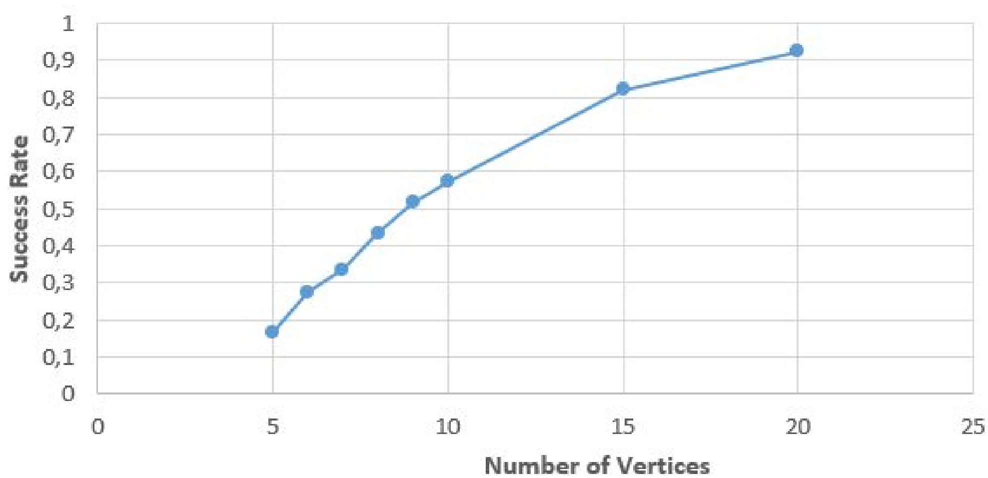
For experimental analysis, we randomly generated several graphs that has 5,6,7,8,9,10,15,20 vertices as follows. For experimental purposes, the p value, defined in section 2, is picked as 0.2 due to the time constraints.

### Success rates

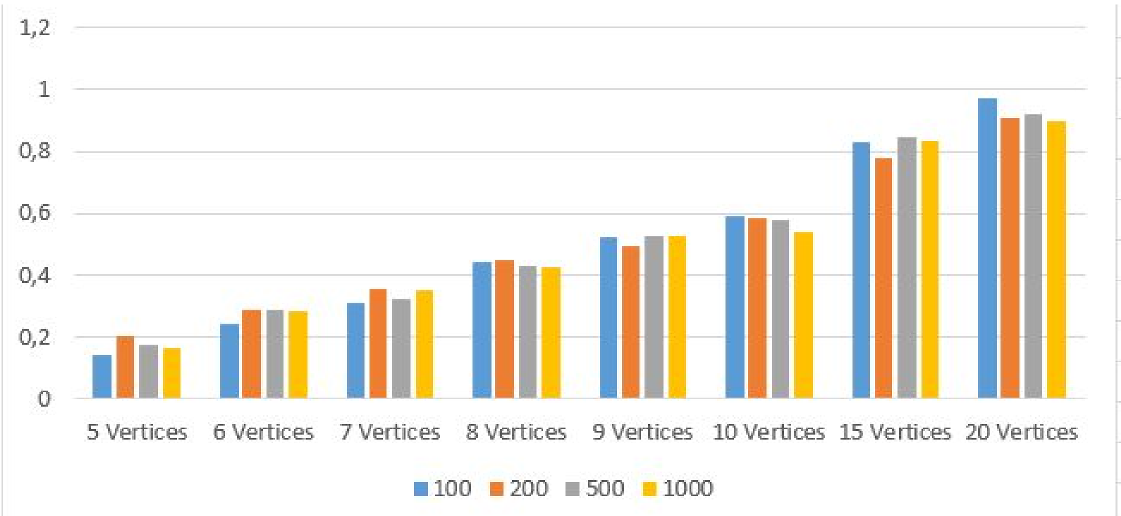
Probability of finding a longest path that includes at least k edges is below. We expect success rate to increase as number of vertices increase but since graphs are randomly generated we cannot say that certainly.

For instance, if we assume a graph with a 20 vertices and it has not many edges, (i.e. a lot of weakly connected edges) algorithm may not find a longest path.

| | 5 Vertices | 6 Vertices | 7 Vertices | 8 Vertices | 9 Vertices | 10 Vertice | 15 Vertice | 20 Vertices |
|---|---|---|---|---|---|---|---|---|
| **100** | 0,14 | 0,24 | 0,31 | 0,44 | 0,52 | 0,59 | 0,83 | 0,97 |
| **200** | 0,2 | 0,29 | 0,355 | 0,445 | 0,495 | 0,585 | 0,775 | 0,91 |
| **500** | 0,172 | 0,29 | 0,324 | 0,432 | 0,524 | 0,578 | 0,844 | 0,918 |
| **1000** | 0,16 | 0,279 | 0,352 | 0,423 | 0,526 | 0,54 | 0,832 | 0,895 |

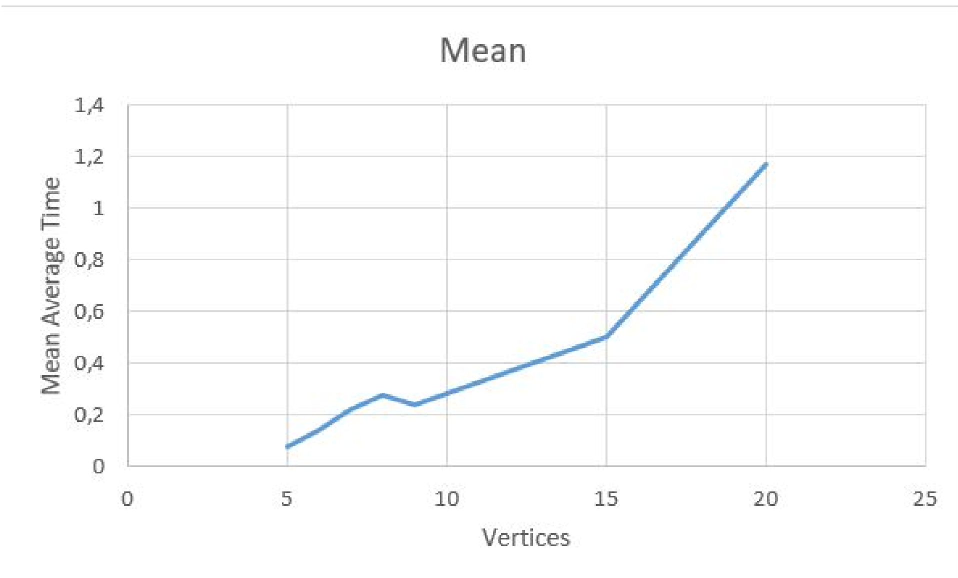


| Vertices | Success Rates |
|---|---|
| 5 | 0,168 |
| 6 | 0,27475 |
| 7 | 0,33525 |
| 8 | 0,435 |
| 9 | 0,51625 |
| 10 | 0,57325 |
| 15 | 0,82025 |
| 20 | 0,92325 |

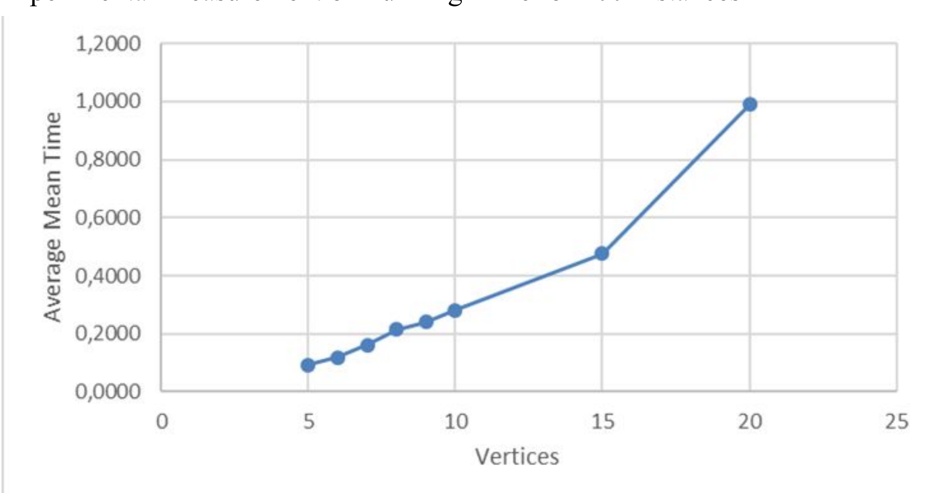


Experimental Measurement of Running Time for 100 Instances with data

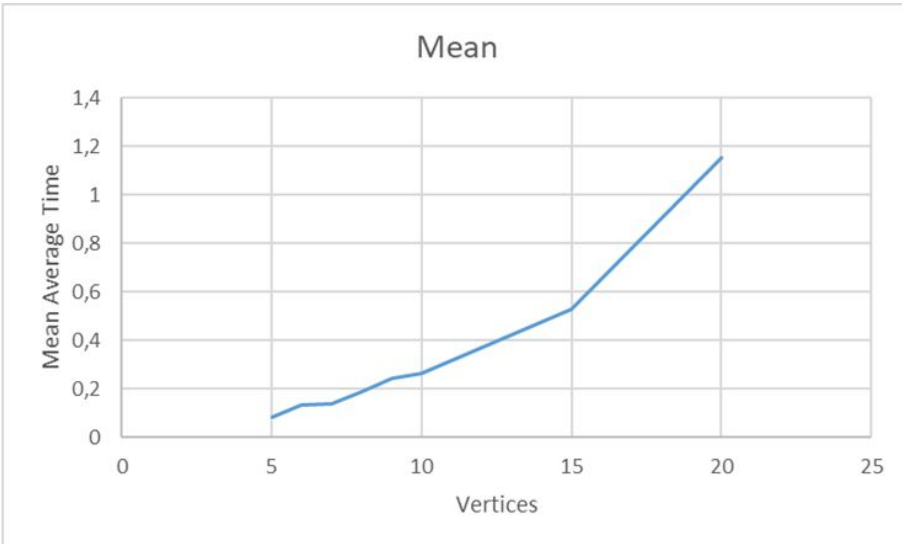
| Size | Mean | Standart Deviation | Standart Error | %90 CL | %95 CL |
|---|---|---|---|---|---|
| 5 | 0,076556 | 0,143247836 | 0,000205199 | 0,0528 - 0,1003 | 0,0481 - 0,1050 |
| 6 | 0,137534 | 0,215719239 | 0,000465348 | 0,1017 - 0,1734 | 0,0947 - 0,1803 |
| 7 | 0,220853 | 0,297493603 | 0,000885024 | 0,1715 - 0,2702 | 0,1618 - 0,2799 |
| 8 | 0,280257 | 0,316427263 | 0,001001262 | 0,2277 - 0,3328 | 0,2175 - 0,3430 |
| 9 | 0,237948 | 0,227361408 | 0,000516932 | 0,2002 - 0,2757 | 0,1928 - 0,2831 |
| 10 | 0,28329 | 0,235885264 | 0,000556419 | 0,2441 - 0,3225 | 0,2365 - 0,3301 |
| 15 | 0,501625 | 0,231589266 | 0,000536336 | 0,4632 - 0,5401 | 0,4557 - 0,5476 |
| 20 | 1,171675 | 1,110230561 | 0,012326119 | 0,9870 - 1,356 | 0,951 - 0,1392 |



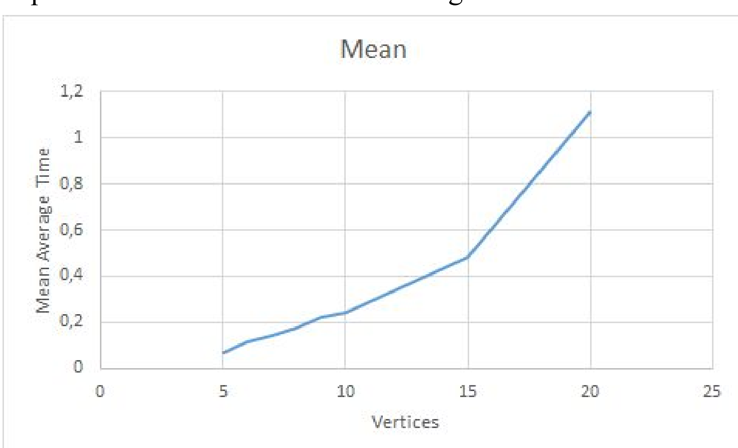

Experimental Measurement of Running Time for 200 Instances



Experimental Measurement of Running Time for 500 Instances




Experimental Measurement of Running Time for 1000 Instances

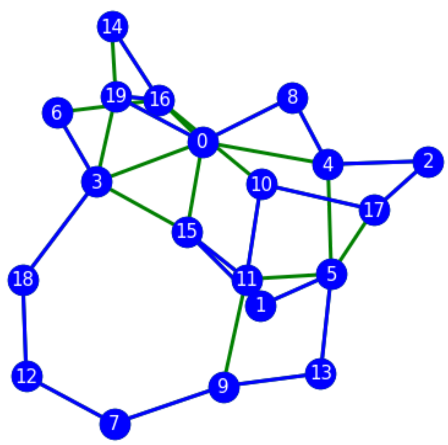


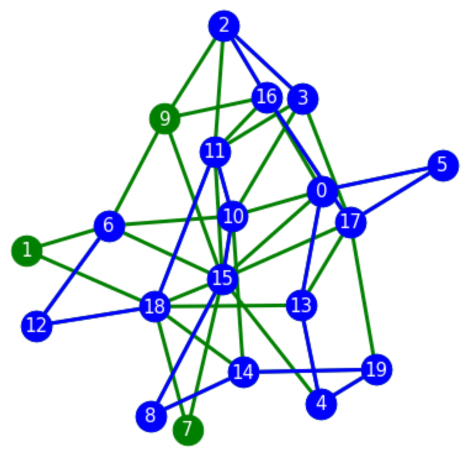

## 5. Testing

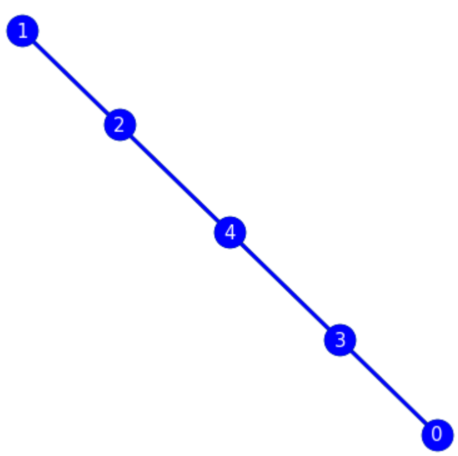
For testing the algorithm black box testing is used. For testing purposes the graphs are generated randomly with Erdos Renyl Model. Also the source and target vertices are picked randomly. Some of the test results are stated below.

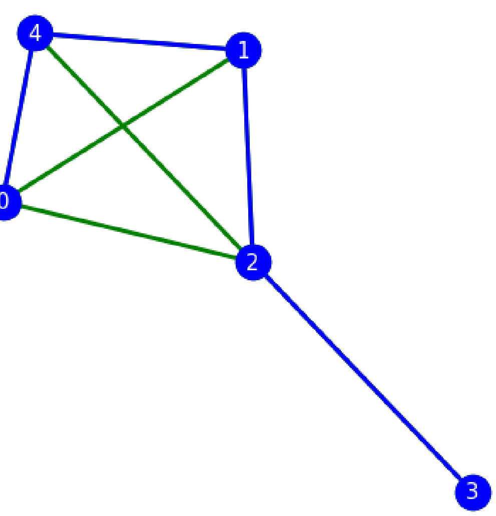
Green lines: Represents the whole graph

Blue lines: The longest path in the graph with given vertices.

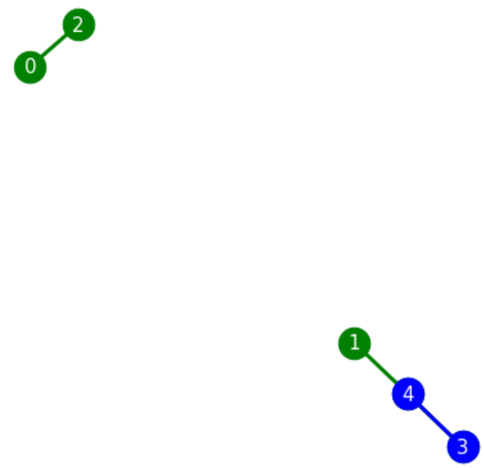








In contrast to the graphs above, above one is not an accurate example, since there are weakly connected components




## 6. Conclusion

As a conclusion, the longest path problem for general graphs are NP-hard but our specification of the problem is NP-Complete. There is no polynomial time algorithm that solves the longest path problem efficiently. Our algorithm is a heuristic approach to the solution based on brute force. Considering the results, success rate depends on the randomly generated graph. Because of the reason that we apply a heuristic approach, we do not have the success rate exactly 1. Therefore the success rate is between 0 and 1.

In general, for sparse graphs, and with uniform distribution of the vertices total time complexity is: O(V(E^3)). For other graphs, it is O((V^2)+V(E^3))

REFERENCES

https://arxiv.org/pdf/1201.3011.pdf

https://community.dur.ac.uk/george.mertzios/papers/Jour/Jour_Longest-Path-Cocomparability.pdf