Homework Assignment 3: Network Flows

5273

All Shortest Paths

Shortest Path Algorithms consist in finding a path between two nodes in a way that the sum of weights of its edges be minimum [9]. If a vertex has two or more predecessors, then there are two or more shortest paths, each of which must be followed separately if we wish to know all shortest paths from i to j [7].

```
P=nx.Graph()

P.add_nodes_from(['x1','x2','x3','x4','x5'])

P.add_edge('x1','x5',weight=1)
P.add_edge('x1','x4',weight=2)
P.add_edge('x4','x2',weight=2)
P.add_edge('x5','x2',weight=1)
P.add_edge('x2','x3',weight=8)
P.add_edge('x1','x2',weight=8)

pos=nx.kamada_kawai_layout(P)

replicas_asp=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        nx.all_shortest_paths(P,source='x1',target='x3',weight='weight')
    end=time.time()
    execution_time=end-start
    replicas_asp.append(execution_time)

normality_test=stats.shapiro(replicas_asp)
print(normality_test)
print(replicas_asp)

hist, bin_edges=np.histogram(replicas_asp,density=True)
first_edge, last_edge = np.min(replicas_asp),np.max(replicas_asp)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(asp,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.savefig("Histogram_asp.eps", format="EPS")
plt.show(1)
```

graphs.py

For the shortest path algorithm is selected an undirected graph with five nodes and six edges. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 0.0226 seconds with a standard deviation of 0.01161.
With the 30 values of the computation time a histogram is constructed.

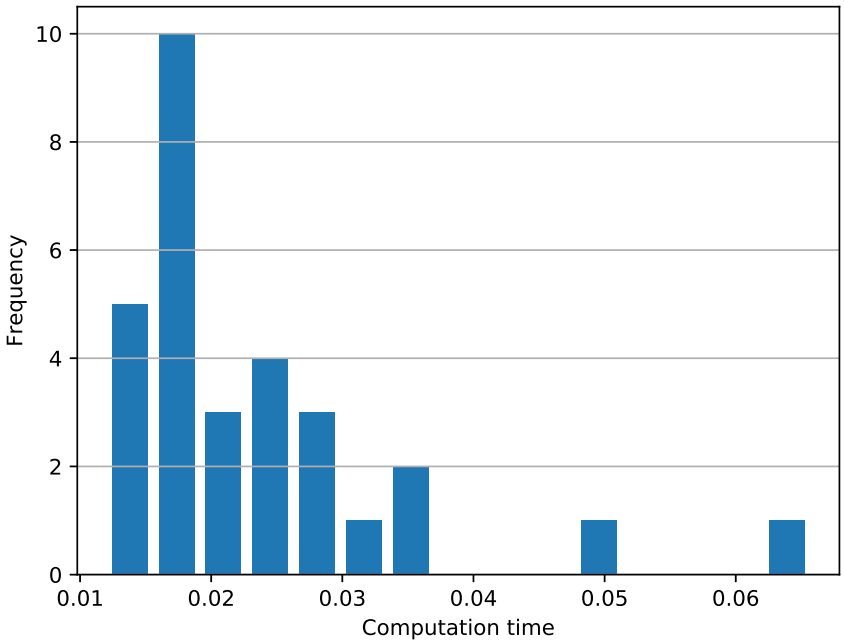


Figure 1: Histogram of All Shortest Paths Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 2.72×10^{-5} so the data is not normal distributed.

Betweenness Centrality

This algorithm measures the influence of a vertex over the flow of information in a graph based on shortest paths [10]. A node with higher betweenness centrality would have more control over the network, because more information will pass through that node [4].

```
B = nx.DiGraph()
B.add_node('DEPOT')
B.add_nodes_from(['Job1','Job2','Job3','Job6'])
B.add_nodes_from(['Job4','Job5','Job7'])
B.add_nodes_from(['Job8','Job9'])

B.add_path(['DEPOT','Job1','Job2','Job3','Job6','DEPOT'])  #Construct a path from the nodes in
    that order
B.add_path(['DEPOT','Job4','Job5','Job7','DEPOT'])
B.add_path(['DEPOT','Job8','Job9','DEPOT'])

replicas_bc=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        bw_centrality = nx.betweenness_centrality(B, normalized=False)
    end=time.time()
    execution_time=end-start
    replicas_bc.append(execution_time)

normality_test=stats.shapiro(replicas_bc)
print(normality_test)
print(replicas_bc)

hist, bin_edges=np.histogram(replicas_bc,density=True)
first_edge, last_edge = np.min(replicas_bc),np.max(replicas_bc)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_bc,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(2)
```

graphs.py

For the betweenness centrality algorithm is selected a directed graph with ten nodes and eleven edges. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 13.8352 seconds with a standard deviation of 0.4111.
With the 30 values of the computation time a histogram is constructed.

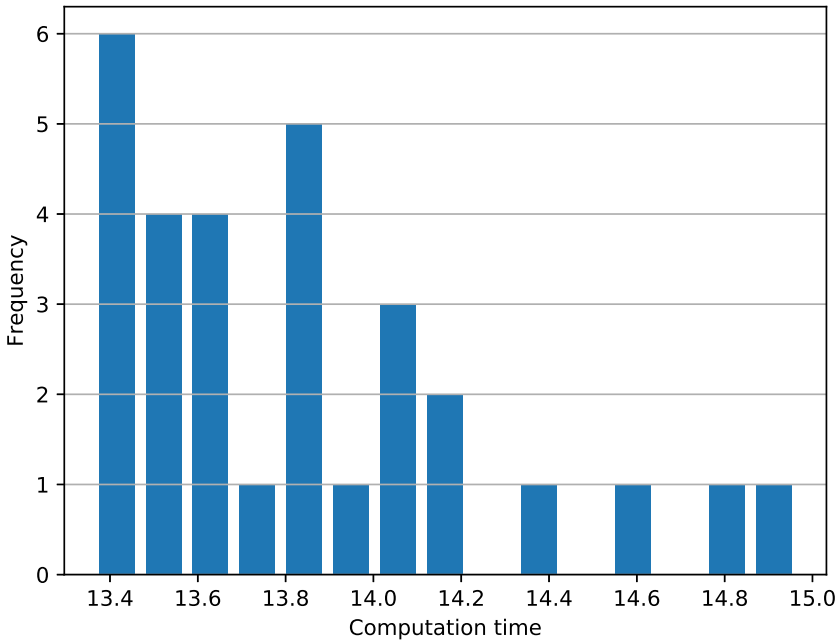


Figure 2: Histogram of Betweeness Centrality Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 0.004 so the data is not normal distributed.

Depth-First Search Tree

The Depth-First Search (DFS) has the application of traversing a graph and constructing a special structured tree, called a DFS tree [5]. A DFS tree of a graph is a spanning tree produced by performing a DFS algorithm in the graph. Starting from an arbitrary prescribed vertex , the DFS algorithm traverses the graph by repeatedly visiting an unvisited neighbor of the last visited vertex. If all the neighbors of the current vertex have already been visited, the search backtracks until it finds a vertex with an unvisited neighbor to continue [2].

```
D = nx.Graph()

D.add_node('Libro')
D.add_nodes_from(['C1','C2','C3'])
D.add_nodes_from(['s1.1','s1.2','s2.1','s2.2','s2.3'])
D.add_nodes_from(['s2.1.1','s2.1.2'])

D.add_edges_from([('Libro','C1',),('Libro','C2',),('Libro','C3')])
D.add_edges_from([('C1','s1.1',),('C1','s1.2',)])
D.add_edges_from([('C2','s2.1',),('C2','s2.2',),('C2','s2.3')])
D.add_edges_from([('s2.1','s2.1.1'),('s2.1','s2.1.2')])

replicas_dfs=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        dfs = nx.dfs_tree(D)
    end=time.time()
    execution_time=end-start
    replicas_dfs.append(execution_time)

normality_test=stats.shapiro(replicas_dfs)
print(normality_test)
print(replicas_dfs)

hist, bin_edges=np.histogram(replicas_dfs,density=True)
first_edge, last_edge = np.min(replicas_dfs),np.max(replicas_dfs)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_dfs,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(3)
```

graphs.py

For the DFS Tree algorithm is selected an undirected acyclic graph with eleven nodes and ten edges. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 2.5114 seconds with a standard deviation of 0.1115.
With the 30 values of the computation time a histogram is constructed.

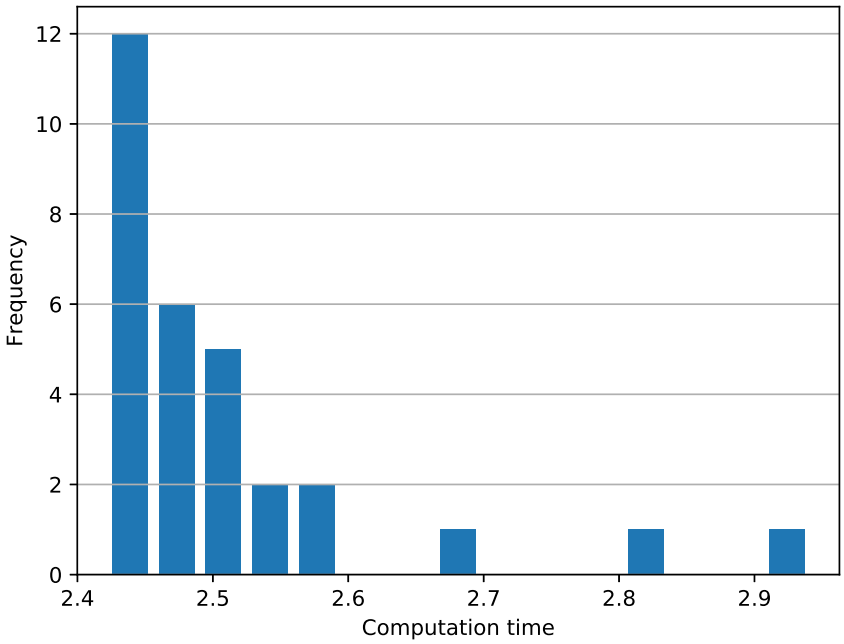


Figure 3: Histogram of DFS Tree Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 3.05×10^{-7} so the data is not normal distributed.

Strongly Connected Components

A strongly connected components of a directed graph is a maximal subset of vertices containing a directed path from each vertex to all others in the subset [3]. The traditional, algorithm for finding the strongly connected components in a graph is based on depth first search [6].

```
S=nx.DiGraph()
S.add_nodes_from(['A','B','C','D','E'])

S.add_edges_from([('A','B'),('A','C'),('B','C'),('A','D'),('A','E'),('D','D')])

color_map = []
for node in S:
    if (node == 'D'):
        color_map.append('blue')
    else:
        color_map.append('red')

pos = nx.kamada_kawai_layout(S)

replicas_scc=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        sccs = list(nx.strongly_connected_components(S))
    end=time.time()
    execution_time=end-start
    replicas_scc.append(execution_time)

normality_test=stats.shapiro(replicas_scc)
print(normality_test)
print(replicas_scc)

hist, bin_edges=np.histogram(replicas_scc,density=True)
first_edge, last_edge = np.min(replicas_scc),np.max(replicas_scc)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_scc,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(4)
```

graphs.py

For the strongly connected components algorithm is selected a directed graph with five nodes and six edges. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 1.0484 seconds with a standard deviation of 0.0459.
With the 30 values of the computation time a histogram is constructed.

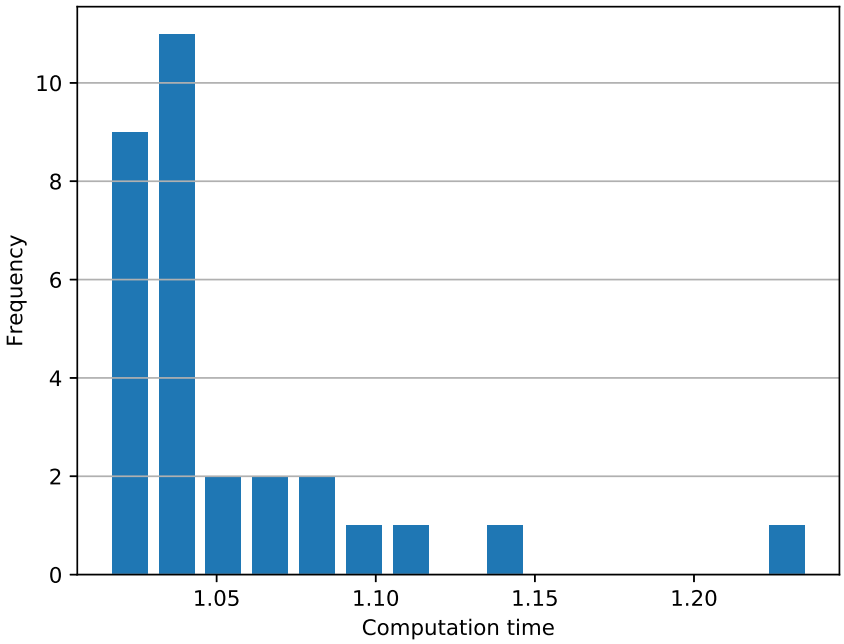


Figure 4: Histogram of Strongly Connected Components Algorithm

Shapiro-Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 1.05×10^{-6} so the data is not normal distributed.

Topological Sort

A topological sort is used to arrange the vertices of a directed acyclic graph in a linear order [8]. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks [1].

```
T = nx.DiGraph()

T.add_nodes_from(['r','t','u','v','w','x','y','z'])

T.add_edges_from([('x','y'),('r','t'),('t','v'),('t','u'),('u','w'),('x','z'),('y','z'),('y','v'),('v','w'),
                  ('v','x'),('x','z'),('z','v'),('z','w')],color='black',weight=1)

edges = T.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(T.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

pos = nx.shell_layout(T)

replicas_ts=[]
for j in range(30):
    start=time.time()
    for i in range (replicas_number):
        ts = nx.topological_sort(T)
    end=time.time()
    execution_time=end-start
    replicas_ts.append(execution_time)

normality_test=stats.shapiro(replicas_ts)
print(normality_test)
print(replicas_ts)

hist, bin_edges=np.histogram(replicas_ts,density=True)
first_edge, last_edge = np.min(replicas_ts),np.max(replicas_ts)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_ts,bins=bin_edges,rwidth=0.75)
plt.xlabel('Computation time')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show(5)
```

graphs.py

For the strongly connected components algorithm is selected a directed graph with five nodes and six edges. This algorithm is run 30 times for a number of 19 000 replicas. The data has a mean of 0.0136 seconds with a standard deviation of 0.003.
With the 30 values of the computation time a histogram is constructed.

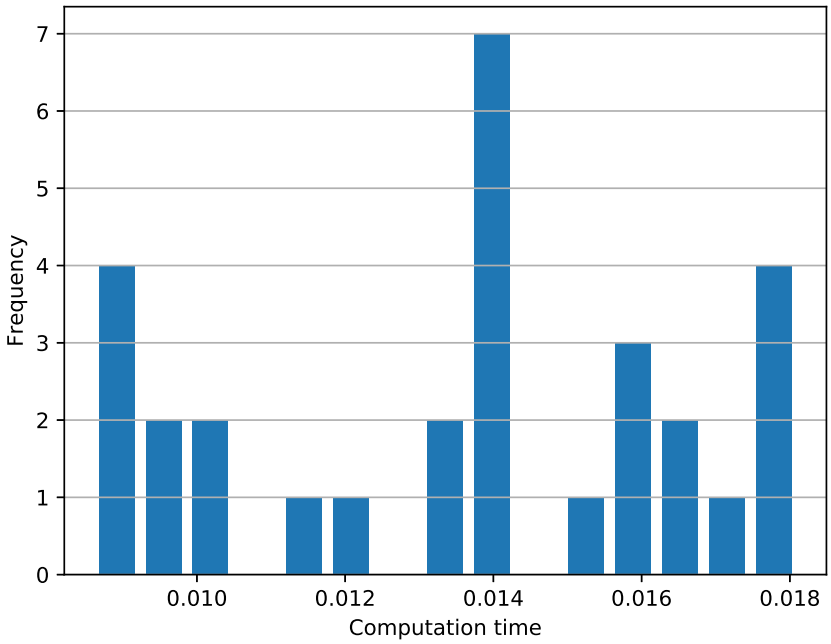
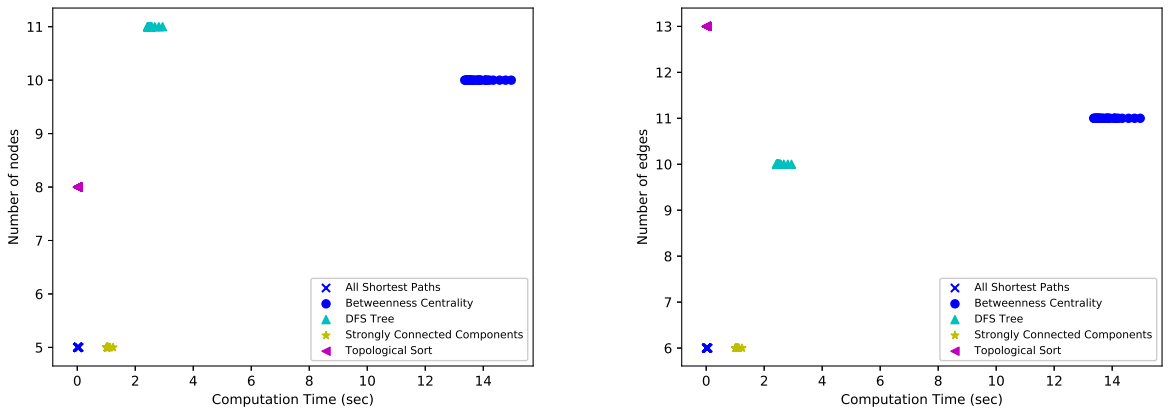


Figure 5: Histogram of Topological Sort Algorithm

Shapiro Wilk Test is performed in order to demonstrate if the data is normally distributed. The test shows a p-value of 0.0173 so the data is not normal distributed.

Conclutions

Once all algorithms are run, under the same conditions, we can conclude that none of the algorithms are normal distributed. To analyze all the data two scatterplot are also drawn. The first is of execution time vs. number of nodes, whereas the second one is of execution time vs. number of edges.



(a) Scatter plot of computation time vs. nodes

(b) Scatter plot of computation time vs. edges

Figure 6: Scatter plots.

From the analysis of scatterplots we can say that faster algorithms are: "All Shortest Path" and "Topological Sort".
A violin plot is also drawn, where the horizontal axis represents each of the algorithms and the vertical one represents the corresponding execution time. Here it is evident that the slowest algorithm is "Betweenness Centrality".

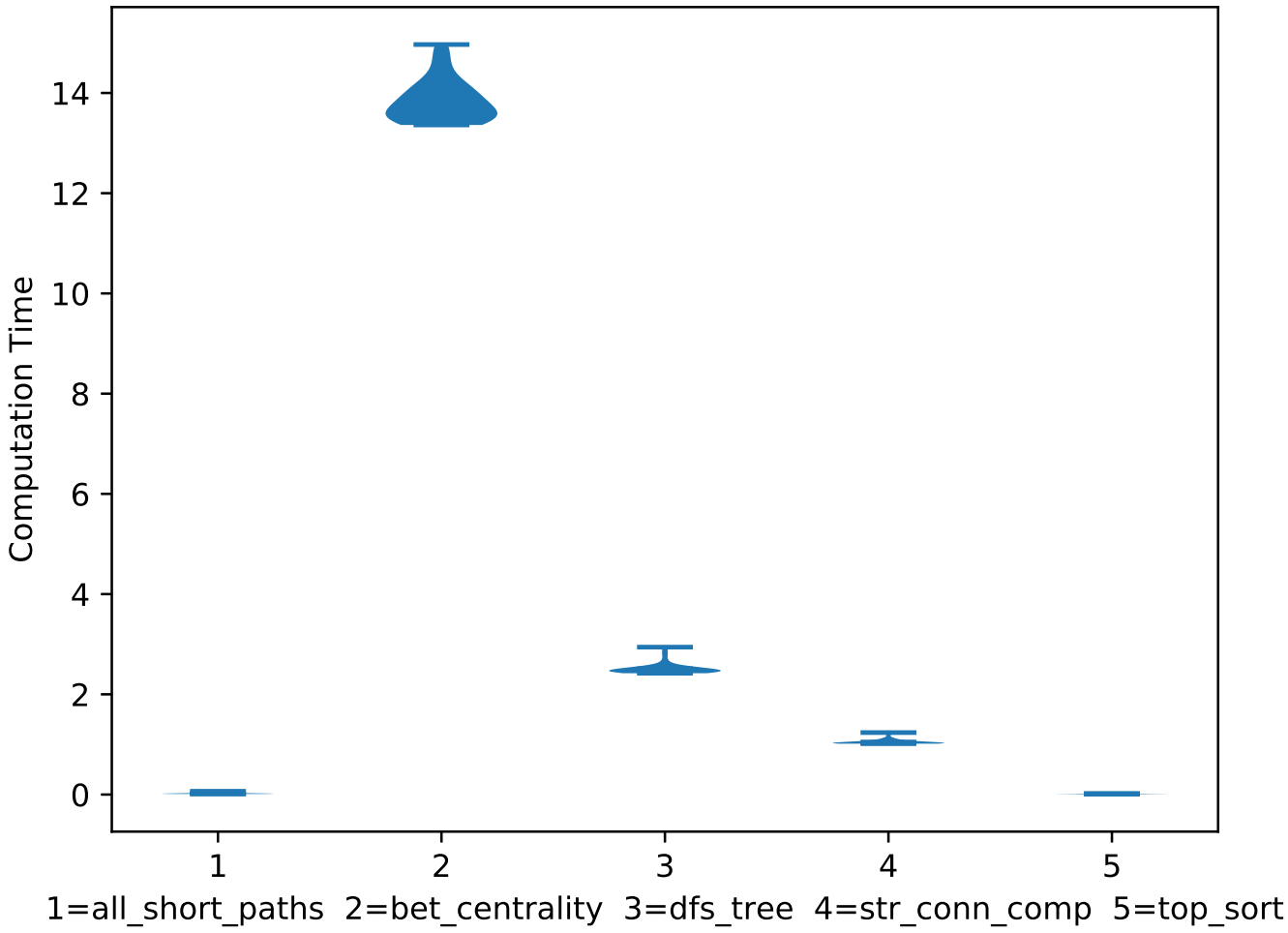


Figure 7: Violin Plot of Algorithms

References

[1] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM Journal on computing*, 10(4):657–675, 1981.

[2] Amr Elmasry, Kurt Mehlhorn, and Jens Schmidt. Every dfs tree of a 3-connected graph contains a contractible edge. *Journal of Graph Theory*, 72(1):112–121, 2013.

[3] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*, pages 505–511. Springer, 2000.

[4] Linton Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[5] Ephraim Korach and Zvi Ostfeld. Dfs tree construction: Algorithms and characterizations. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 87–106. Springer, 1988.

[6] William Mclendon, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[7] Mark Newman. Scientific collaboration networks, shortest paths, weighted networks, and centrality. *Physical Review*, 64(1):16–132, 2001.

[8] David J Pearce and Paul HJ Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics (JEA)*, 11:1–7, 2007.

[9] Jeanette Schmidt. All shortest paths in weighted grid graphs and its application to finding all approximate repeats in strings. In *Proceedings Third Israel Symposium on the Theory of Computing and Systems*, pages 67–77. IEEE, 1995.

[10] Raghavan Unnithan, Sunil Kumar, Balakrishnan Kannan, and Madambi Jathavedan. Betweenness centrality in some classes of graphs. *International Journal of Combinatorics*, pages 39–52, 2014.