Homework Assignment 4: Network Flows

5273

Introduction

For this work graph generators of the library NetworkX of Python have been used. There have been selected three of these generator for the study: Complete Graph, Wheel Graph and Star Graph. To use each one of them there have been selected a base two as logarithmic base to generate four different graph orders, so the total number of nodes for each graph generator is 4, 8, 16 and 32 nodes respectively. The total number of edges depends on the chosen graph. Edge weights have been generated following a normally distribution with mean of 12 units and a standard deviation of 0.4.
After the assigment of normally distributed weights to edges, these weights would be used as instances of the maximum flow problem. Three implementations of this problem are performed using NetworkX algorithms: Maximum Flow, Minimum Cut and Preflow Push. Each one implementations are performed for each one of the graph generators.
This work it is run on an $Intel^{®}$ Celeron CPU @ 1.10 GHz with 4 GB RAM Laptop.

Graph Generators

Complete Graph

The first graph generator chosen is the complete graph. This is a simple undirected graph which each pair of nodes is connected by a unique edge [1]. The complete graphs selected, as mentioned at Introduction, have 4, 8, 16 and 32 nodes with 6, 20, 120 and 496 edges respectively.

Wheel Graph

The second graph generator chosen is the wheel graph. This is a simple undirected graph characterized by its single hub node connected to each of the (n-1)-node cycle graph [5]. The wheel graphs selected, as well as the other graph generators, have 4, 8, 16 and 32 nodes with 6, 14, 30 and 462 edges respectively.

Star Graph

The third graph generator chosen is the star graph. This is a simple undirected graph that it can be considered as a tree with one internal node and k leaves [4]. The star graphs selected, as well as the other graph generators, have an order of 4, 8, 16 and 32 nodes with also 4, 8, 16 and 32 edges respectively.

```
for j in range(10):
    for i in range(2,6):
        i = 2 ** i
        number_of_nodes.append(i)

        #COMPLETE GRAPH
        start_C = time.time()
        C=nx.complete_graph(i)
        end_C = time.time()
        execution_time_C = end_C - start_C
        complete_graph_times.append(execution_time_C)

        assign_normally_distributed_weight(12,0.4,C)

        # WHEEL GRAPH
        start_W = time.time()
        W = nx.wheel_graph(i)
        end_W = time.time()
        execution_time_W = end_W - start_W
        wheel_graph_times.append(execution_time_W)

        assign_normally_distributed_weight(12, 0.4, W)

        # STAR GRAPH
        start_S = time.time()
        S = nx.star_graph(i)

        end_S = time.time()
        execution_time_S = end_S - start_S
        star_graph_times.append(execution_time_S)

        assign_normally_distributed_weight(12, 0.4, S)
```

graphs_t4.py

Maximum Flow Algorithms

Maximum Flow

The algorithm can be used in the three chosen graph generators since the settings of the arc capacities are not fixed and are functions of a single parameter [2], it have been defined the source and the sink in each graph and a positive capacity for every edge. The value of the flow is the net flow out of the source. The maximum flow problem is that of finding a flow of maximum value [3].

```
            #Maximum flow Complete Graph
            start_flow = time.time()
            for k in range(5):
                flow_value = nx.maximum_flow(C, ss_list[k][0], ss_list[k][1], capacity='weight
    ')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Maximum flow Wheel Graph
            start_flow = time.time()
            for k in range(5):
                flow_value = nx.maximum_flow(W, ss_list[k][0], ss_list[k][1], capacity='weight
    ')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Maximum flow Star Graph
            start_flow = time.time()
            for k in range(5):
                flow_value = nx.maximum_flow(S, ss_list[k][0], ss_list[k][1], capacity='weight
    ')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

Minimum Cut

This algorithm can performed well with the three chosen graph generators, which are undirected edge-weighted graphs. The minimum cut of an undirected graph with edge weights, consists in a set of edges with minimum sum of weights, such that its removal would cause the graph to become unconnected [6].

```
            #Minimum cut Complete Graph
            start_flow = time.time()
            for k in range(5):
                cut_value = nx.minimum_cut(C, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Minimum cut Wheel Graph
            start_flow = time.time()
            for k in range(5):
                cut_value = nx.minimum_cut(W, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Minimum cut Star Graph
            start_flow = time.time()
            for k in range(5):
                cut_value = nx.minimum_cut(S, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

Preflow Push

This implementation finds a maximum single-commodity flow using the highest-label preflow-push algorithm [5] . It returns the residual network resulting after computing the maximum flow, which also have been performed.

```
            #Preflow push Complete Graph
            start_flow = time.time()
            for k in range(5):
                R = preflow_push(C, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Preflow push Wheel Graph
            start_flow = time.time()
            for k in range(5):
                R = preflow_push(W, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
```

graphs_t4.py

```
            # Preflow push Star Graph
            start_flow = time.time()
            for k in range(5):
                R = preflow_push(S, ss_list[k][0], ss_list[k][1], capacity='weight')
            end_flow = time.time()
            execution_time_flow = end_flow - start_flow
            preflow_push_times.append(execution_time_flow)
```

graphs_t4.py

Experimental Results

Once all corresponding graphs are generated it can be evaluated throughout statistical methods and visualizations the effect of the used graph generator in the computation time, as well as the maximum flow algorithms, the graph order and the density of the graph.
For every effect a boxplot is shown.
The first boxplot shows the effect of the graph generator in the computation time.
Data for Complete Graph has a mean of 0.011743807 and a standard deviation of 0.013289318, whereas data for Wheel Graph has a mean of 0.002271925 and a standart deviation of 0.001256446, and the Star Graph Generator has a mean of 0.003683368 and a standard deviation of 0.002426251.

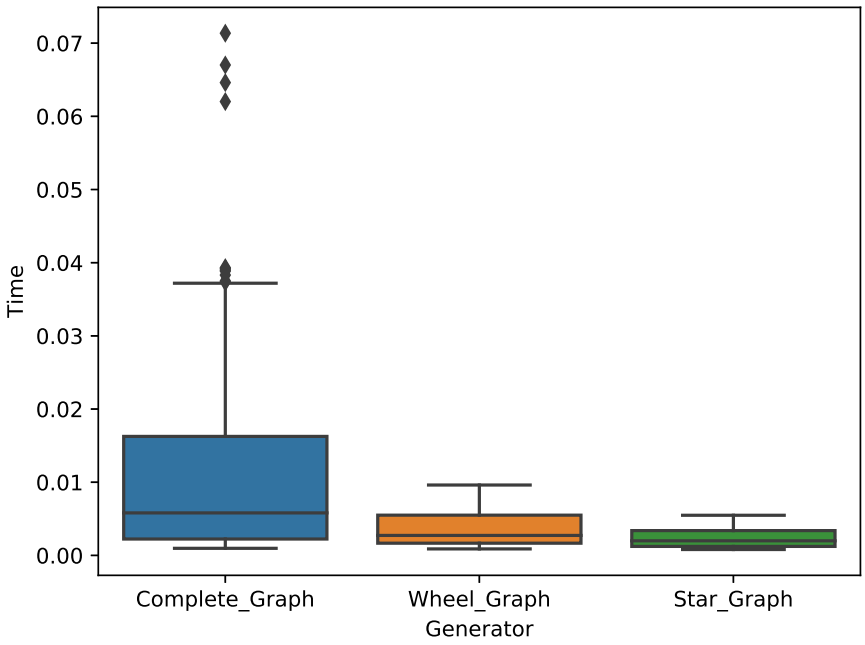

Figure 1: Boxplot of the graph generators

In this graph differences are seen between groups, especially in Complete Graph which has higher values than the other two graph generators. Also the groups have different distribution.

The second boxplot shows the effect of the maximum flow algorithms in the computation time.
Data for Maximum Flow has a mean of 0.006351 and a standard deviation of 0.009479, whereas data for Minimum Cut has a mean of 0.005868 and a standart deviation of 0.009092, and the Preflow Push has a mean of 0.005480 and a standard deviation of 0.007968.

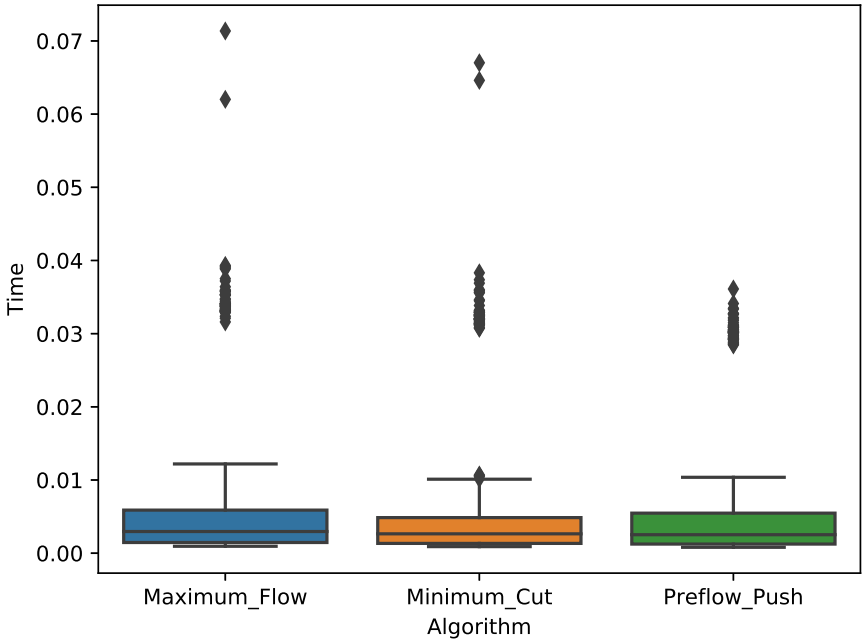

Figure 2: Boxplot of maximum flow algorithms

From the graph can be seen some similarities among the group of algorithms but further analysis must be perform to corroborate this.
The third boxplot shows the effect of the graph order in the computation time.
Data for Order 4 has a mean of 0.001124 and a standard deviation of 0.000144, whereas data for Order 8 has a mean of 0.002206 and a standart deviation of 0.000651, Order 16 has a mean of 0.005231 and a standard deviation of 0.002970 and Order 32 has a mean of 0.015039 and a standard deviation of 0.013616.

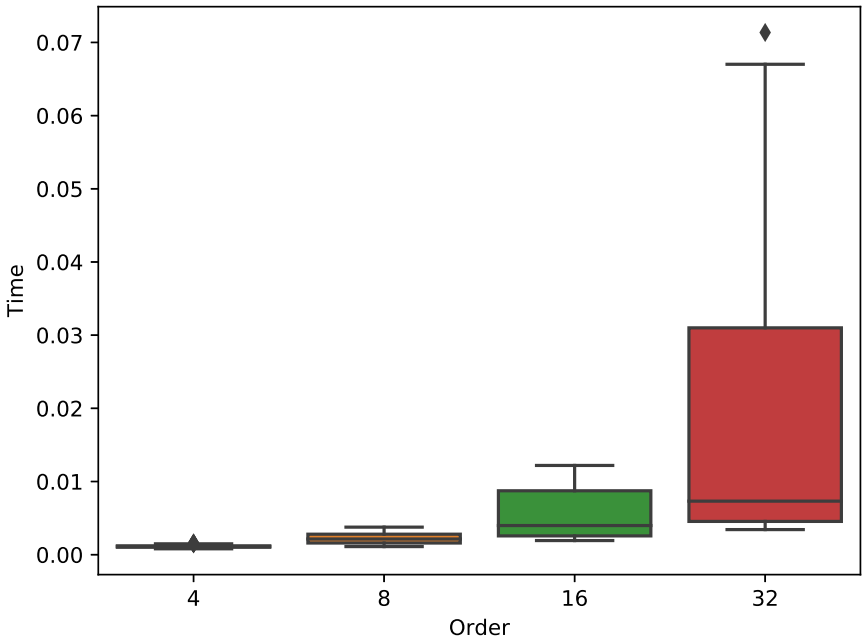

Figure 3: Boxplot of graph order

Here a difference in order is clearly seen. As the order grows the execution time grows as well, so it can be think they may be positive correlated.
The fourth boxplot shows the effect of the eight values of the density in the computation time. Data of the mean and standart deviation is shown below.

Table 1: Mean and Standart Deviation values for graph densities

	0.06	**0.12**	**0.13**	**0.25**	**0.29**	**0.50**	**0-67**	**1.00**
Mean	0.004190	0.007376	0.002408	0.004039	0.001476	0.002168	0.001014	0.009625
Standard Deviation	0.000507	0.000829	0.000300	0.000540	0.000189	0.000209	0.000112	0.012618

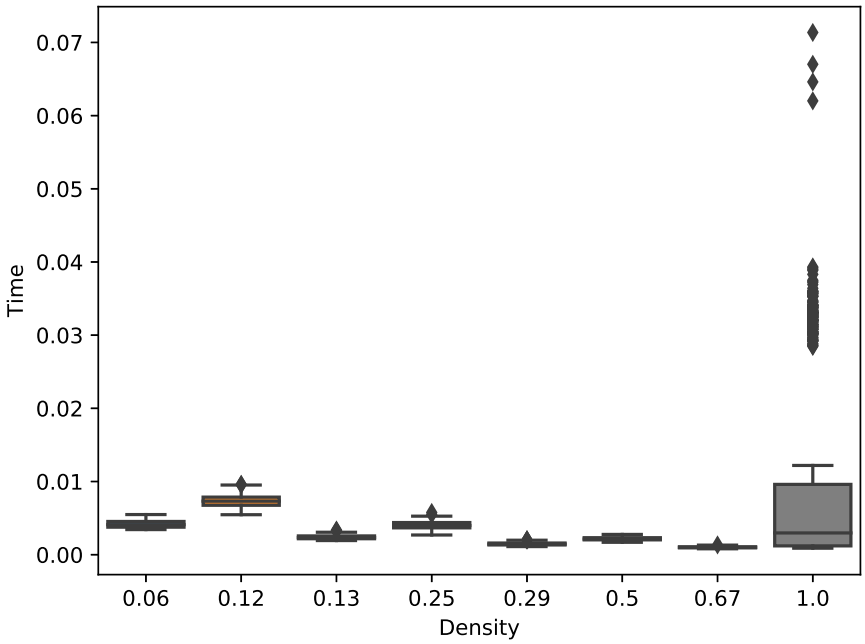

Figure 4: Boxplot of density

Here it can be seen the effect of the density in the execution time, showing differences in the distribution of each group.

In order to see if these effects have interactions an ANOVA is performed.

ANOVA.txt

	sum_sq	df	F	PR>F
Generator	5.250796e-04	2.0	56.267109	2.015921e-24
Algorithm	2.280564e-04	2.0	24.438344	3.385212e-11
Generator:Algorithm	1.227626e-05	4.0	0.657757	6.214187e-01
Order	7.006502e-02	1.0	15016.222797	0.000000e+00
Generator:Order	1.022268e-04	2.0	10.954543	1.868646e-05
Order:Algorithm	2.821883e-04	2.0	30.239076	1.217277e-13
Density	2.540693e-03	1.0	544.517337	2.639616e-105
Generator:Density	4.396328e-05	2.0	4.711070	9.107547e-03
Algorithm:Density	8.825691e-05	2.0	9.457540	8.209189e-05
Order:Density	2.257209e-07	1.0	0.048376	8.259386e-01
Residual	8.310065e-03	1781.0	NaN	NaN
c@FancyVerbLinee	8.310065e-03	1781.0	NaN	NaN

From the ANOVA it can be said p-values are quite small, so there are differences among group means, concluding that all variables are related with the execution time

For an analysis of correlation it is performed a scatter matrix, showing the relation of the quantitative variables and its corresponding scatterplots, the kernel density estimation as a way to estimate the probability density function of the variables and the coeficients of correlation between the variables, where it can be seen the highest correlation is between the variables time and order, with a value of 0.613, but still it is not a strong relation.

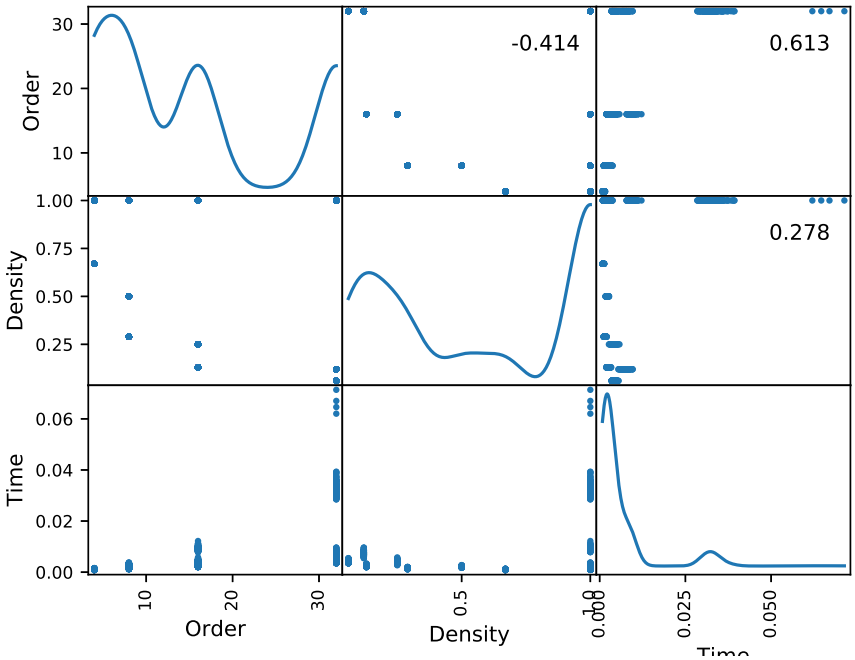

Figure 5: Scatter Matrix of the cuantitative variables

References

[1] Paul Erdös. Graph theory and probability. *Canadian Journal of Mathematics*, 11:34–38, 1959.

[2] Giorgio Gallo, Michael D Grigoriadis, and Robert E Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989.

[3] Valerie King, Satish Rao, and Rorbert Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.

[4] Victor E. Mendia and Dilip Sarkar. Optimal broadcasting on the star graph. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):389–396, 1992.

[5] NetworkX. Source code for networkx.drawing.layout, 2018. `https://networkx.github.io/documentation/latest/_modules/networkx/drawing/layout.html`, Last accessed on 2019-03-31.

[6] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4): 585–591, 1997.