Advanced Algorithms Report

Diogo Ramalho 86407 Francisco Sena 86420

Joana Alves 86439 Tomás Vieira 86523

May 2020

1 Introduction

The aim of this project is to study efficient and scalable approaches to compute three common used metrics in complex networks analysis: the average path length, the clustering coefficient, and the betweenness centrality of the nodes of the network. The target will be large graphs for which even polynomial time algorithms may not work in practice and so we will focus on approximation algorithms. All the experiments with respect to the APL were performed in a MacBook Pro 2.4GHz Quad-Core Intel Core i5 with 16GB of RAM. The experiments regarding the two other metrics were performed in a MacBook Air 1.6GHz Intel Core i5 with 4GB of RAM. To assess the running time of the algorithms, we used the C++ chrono library. The scale-free networks used for the experiments were generated with the DMS minimal model.

2 Graph Representation

For all the algorithms, the main operation is to access the neighbors of a given vertex. We stored the graph using the CSR (Compressed Sparse Row) format. In theory, the space required is $O(V + E)$ since we only need to store the adjacencies of all the vertices hence the O(E), plus the offset vector which is $O(V)$. Asymptotically, it is as expensive as adjacency lists. In fact, CSR is cheaper in memory because it just needs to store two contiguous arrays, whereas the adjacency list has all the pointers overhead which not only requires more effective memory but also may slow down the real execution time of the queries on the graph. The implemented clustering coefficient algorithm is heavy on the operation of checking if two nodes are connected. However, we have decided to stick with the CSR format although it requires $O(V)$ time for that operation when it is possible to perform it in $O(1)$ time using other data structures such as an adjacency matrix.

3 Average Path Length

3.1 Implementation details

To compute the APL we decided to implement HyperANF that uses HyperLogLogCounters to approximate the size of the Neighborhood functions. We used a Jenkins Hash function to hash the elements in the Add operation, that provides us a 64-bit hashed word. We initialize the counters at 0 instead of - because if one of the registers is left untouched the approximated size of the set would be 0. As we need good approximations since the beginning, we decided to set the counter to 0 knowing that it will not affect the unbiased character of the estimate, as explained in [1]. We iterate until the counters converge. In HyperANF [2] they speed up the computation of Union via broadword programming techniques. It is also suggested a slower but simpler solution, which is to store the values in a temporary counter. However, we opted for a vector of vectors in which every time a counter needs to be updated, we add a vector with 3 entries: the node, the register and the value to be updated. After doing the Union for each edge, we run through the vector to update all the new entries in the counter. In Figure 1, the solid lines represent the memory allocated for each distance to update the counters using our implementation. The dashed lines represent the memory used if we had stored the values in a temporary counter with a fixed size equal to the size of the original counter. We can see that when there are many values to update, the memory can go up to 5x higher. However, as the distance increases the allocated memory goes to 0. At each distance d,

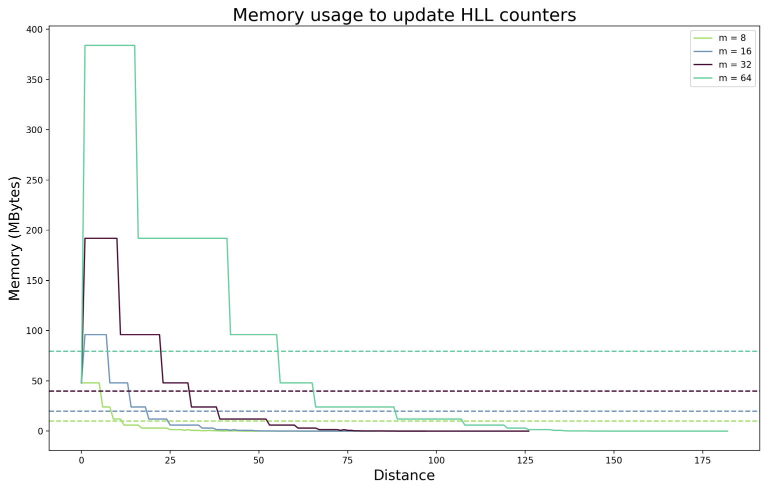

Figure 1: Variation of memory usage by update vector given number of registers with fixed number of nodes (cnr-2000)

we use the Size operation to approximate the neighborhood N(v,d). Then we get $N(d) = \sum_{v \in V} N(v,d)$, that gives us the approximate number of distinct nodes at distance at most d starting from any node v. We then use these values to compute the APL by the following equation $\sum_d \frac{N(d+1)-N(d)}{max_d N(d)} * d$

3.2 Theoretical results

3.2.1 Notation

We consider V the number of nodes, E the number of edges, m the number of registers and D the diameter of the graph.

3.2.2 Time

The algorithm runs through all the edges until when the counters stop changing. In the worst case, this happens when we reach the diameter of the graph, which gives $O(ED)$. In real graphs the diameter is much smaller than the number of nodes. So, as $D << V$, we can approximate $D \approx \log V$, and the time complexity will then be $O(E \log V)$.

3.2.3 Space

Firstly, in [1] each HyperLogLog counter is made by a number of registers m, and the size of each register is doubly logarithmic in the number of nodes of the graph, so HyperANF, for a fixed precision scales almost linearly in memory, O(V m loglogV). However, we fixed the number of nodes to at most 2^{64}, so the log $\log N$ is at most 8 bits (1 byte), so storing the counters is $O(Vm)$. Secondly, the worst case is if all the registers of a node have to be updated for all the edges we perform the Union operation. As the update vectors have 3 entries, as explained in the implementation details, then the complexity is O(3Em). However, as will be shown in section 3.3.2, the memory usage is usually much smaller than this bound. Thirdly, we have to store the approximate size of the neighborhood function for each distance $N(d)$. Considering the maximum distance to be the number of nodes V, we get O(V) At last, this gives a total spatial complexity of $O(Vm + 3Em + V) = O(Em)$, which is worse in the worst case in comparison to $O(Vmloglog V)$, but hopefully not in the average case.

3.2.4 Error

The values of approximate counters for each node are highly dependent, and adding them in a large amount to approximate N(d), can lead to an arbitrarily large variance. For n $\to \infty$, the relative standard deviation (that is, the ratio between the standard deviation of E and n) is at most $\frac{1.06}{\sqrt{m}}$. For example, if M=16 the relative std dev is 26.5%. Assume error ϵ with confidence $1 - \delta$. Assume a relative error of kηm with confidence 1-$\frac{4}{9k^2}$ (e.g., 2ηm with 90% confidence, or 3ηm with 95% confidence).

3.3 Experimental Evaluation

3.3.1 Brief description

We used 2 different types of graphs with small and medium ranges:

- small: 50 scale-free networks with nodes between 203 and 10003, with increasing intervals of 200 nodes.
- medium: the italian .CNR domain in 2000 from the dataset page of LAW of University of Milan. It is a directed graph with 325 557 nodes and 3 216 152 edges.

3.3.2 Space

The theoretical spatial complexity was $O(Vm + 3Em + V)$. Each register has 4 bytes (int), the entries of the update vector also have 4 bytes (int) and the approximate size of the neighborhood function is stored with 8 bytes (unsigned long int). So, in order to evaluate the asymptotic behavior of the memory, we selected the function f(V,m,E) = 2(Vm*4bytes + 3Em*4bytes + V*8bytes). We multiplied by two because sometimes the program has to duplicate the size of the data structures, even if almost half of it is empty. As can be seen in Fig.2, the values of the used memory never surpass the theoretical bound as expected. The difference between the bound and the memory increases as the number of nodes V increases, because the number of edges E also increases and that has a strong impact in the theoretical spatial complexity.

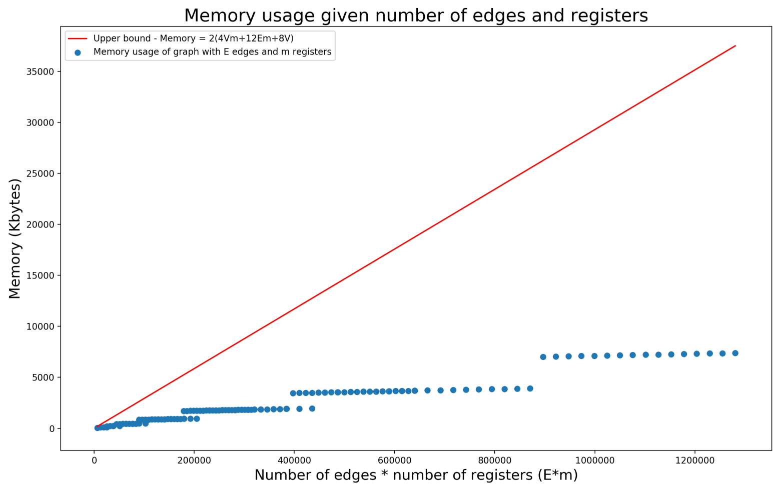

Figure 2: Asymptotic behavior of total memory with different values for nodes and registers.

3.3.3 Time

In Fig.3 we can see the running time of the 50 SFN for number of registers 8, 16 and 32. In the horizontal axis we have the time complexity $O(E)$ and we can see that for each register, the time values of the SFN grow proportionally to this bound as we expected.

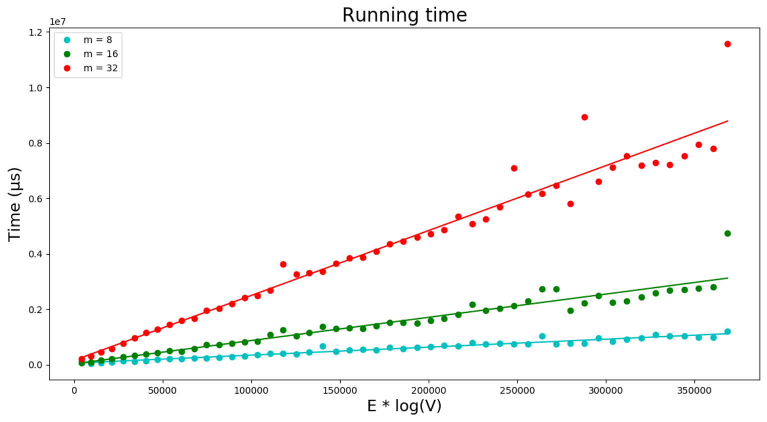

Figure 3: Running time of 50 SFN with different number of registers.

3.3.4 Error

The APL of .CRN dataset is 17.35 ± 0.313 [1]. The values for our APL were:

- m = 8: 25.11
- m = 16: 37.25
- m = 32: 61.33
- m = 64: 116.23

In Fig.4 we have the cumulative distribution of the Neighborhood function. We can see that as we increase the number of registers, the curves shift right, which means that the average distance increases. In terms of the algorithm this may happen because when we do the Add operation, the number of buckets the element can fall in duplicates if we duplicate the number of registers. And so when we are performing the Union operation in the beginning, the probability of updating the counters will decrease proportionally. In Fig.5 we can observe the same behavior. The number of registers that give the best approximation to the real value are 8 and 16. However, we also know that the relative standard deviation is at most $\frac{1.06}{\sqrt{m}}$. This gives a relative std dev of 53% (m=8) and 26.5% (m=16), which is very high.

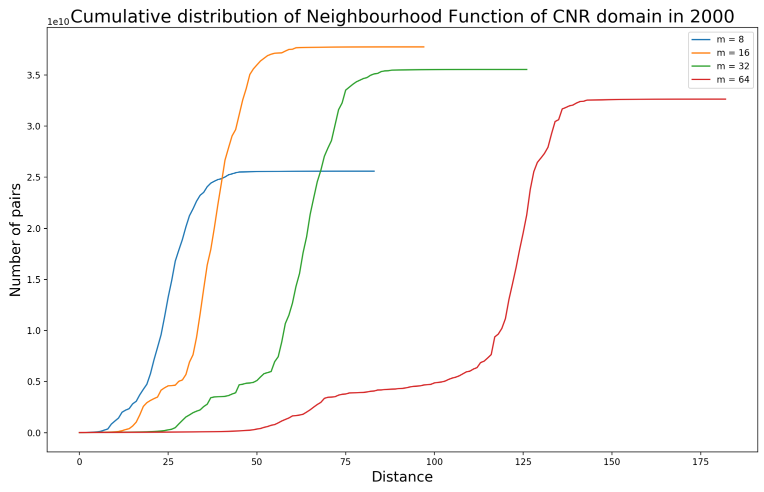

Figure 4: Cumulative distribution of Neighborhood Function in .CNR dataset for different number of registers

3.4 Brief conclusions

In order to reduce the error we could have used Jackknife to compute the values, for instance 10 times (even though it would take a lot of time). We would then use 9 of these values to compute the "real" average distance and the other to compute the error. Repeating 10 times this process would give us stronger estimates. If we had had more time, we would have liked to implement the replicated vector to update the counters and compare it with our solution. In order to decrease the running time it is possible to implement a distributed approach and some other techniques expressed in [2]. Also, in their experimental testing they used a computer with 32 cores and 128GB of RAM which gives them a lot more computational power. If we had more of this, we would have tried to experiment on larger networks.

4 Betweenness Centrality

4.1 Theoretical results - time and space analysis

The Brandes algorithm [3] gives us the betweenness of all the vertices in a network in polynomial time $O(VE)$ for unweighted graphs. The algorithm computes, for each vertex v, the shortest path to every other vertex and then traverses these paths backwards to efficiently compute the contribution of the shortest paths starting at v to the betweenness of the other vertices. This approach is not scalable for large graphs. We have decided to implement an

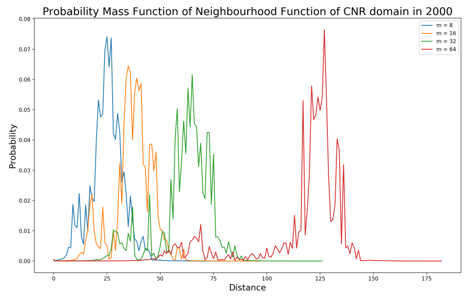

Figure 5: Probability mass function of Neighborhood sizes for each distance in .CNR dataset for different number of registers

approximation algorithm through the sampling of shortest paths of [4] and also the Brandes algorithm for comparison purposes. The approximation algorithm first computes the sample size which is denoted by r. Since the value of r is directly affected by the value of the graph's diameter (size of the shortest path with maximum size), we first need to devise a way of calculating the diameter. To solve that, we approximate the graph's diameter. To randomly select the shortest paths, first we sample two vertices, u and v. Then, we compute all the shortest paths from u to v. Finally, we randomly select one shortest path and increment the betweenness of all the vertices on that path by $1/r$. To select a shortest path we start from v and build the path backwards where the predecessor of v is randomly selected using weighted sampling with respect to the number of shortest paths from u to the predecessor of v. We repeat this process until we reach node u. Regarding the theoretical analysis of the time of the algorithm, its asymptotic bound is given by $O(r(V+E+E))$ since we need to compute a BFS r times and, for each of those, update the betweenness of all the vertices that lie on the sampled shortest path (which is E in the worst case). The time is then $O(r(V+E))$ which is essentially $O(rE)$. Regarding the space analysis, it is only required to keep the graph in memory during the execution of the algorithm and some auxiliary data structures for the sampling of the shortest paths and the computation of the BFS. Those auxiliary data structures do not have an impact on the asymptotic analysis of the space since they all are $O(V)$ or $O(E)$. For some ϵ and δ, the probability that the computed approximations over all the vertices does not exceed ϵ is at least $1-\delta$, equivalently: $P[\exists v \in V s.t. \mid b[v] - b'[v] \mid > \epsilon\ ] < \ \delta$, so for smaller values of ϵ the quality of the estimation increases and for smaller values of δ the probability that the expected estimation fails decreases. Summing up, smaller values of ϵ and δ should provide a better estimate, but, consequently, increase the sample size and the effective running time of the algorithm.

4.2 Implementation details

A critical step of the algorithm is the computation of all the shortest paths from a node u to a node v. For that, we implemented an adaption of a BFS (for weighted graphs the same trick can be used for Dijkstra's algorithm). Although there are exponentially many shortest paths between any two nodes, our BFS computes all of them in time O(V+E). Let u, v and x denote nodes in a graph. We defined two auxiliary arrays for the problem of computing all the shortest paths from u to v:

- predecessors: predecessors[x] is an array with the predecessors of x in all the shortest paths from u to x.
- sigma_u: sigma_u[x] is an integer value that gives us the number of different shortest paths from u to x.

Along the execution of the BFS, whenever we are expanding a node k and visit a node x at a distance that corresponds to the smallest distance from u to x, we append k to the array of predecessors at index x and then add to the value of sigma_u[x] the number of shortest paths from u to k, which corresponds to sigma_u[k]. This two arrays allows us to efficiently sample a shortest path since we have all the information required to recover the shortest paths from u to v and also to sample the predecessors of a node along the set of shortest paths. Another small detail is that the algorithm requires the value of the graph's diameter. That could be done by solving the APSP (All Pairs Shortest Paths) problem. However, algorithms for that problem are too slow for big networks. To solve that problem, we approximated the diameter of the graph with a 2-approximation algorithm by simply performing a BFS starting on a vertex chosen uniformly at random and then returning the sum of the two largest paths with that node as the source (as suggested by the authors).

4.3 Experimental Evaluation

In real networks the diameter tends to be small [5], in particular in scale-free networks. This is a key factor of why this approximation algorithm provides good results with low running times (the sample size depends directly on the size of graph's diameter). We evaluated the quality of the approximations by computing the average estimation of the absolute errors over all the vertices, which is given by: $(1/V) \sum v \in V \mid b[v] - b'[v] \mid$. In Figure 6 we can see that increasing the value of ϵ increases the average absolute error, as expected. We can not see, however, if the algorithm is providing us estimates accordingly to the theoretical bound. For that, we built Table 1, where for each value of ϵ we computed the maximum absolute error over all the vertices. We can see that the empirical results follow the theoretical bounds according to δ and ϵ.

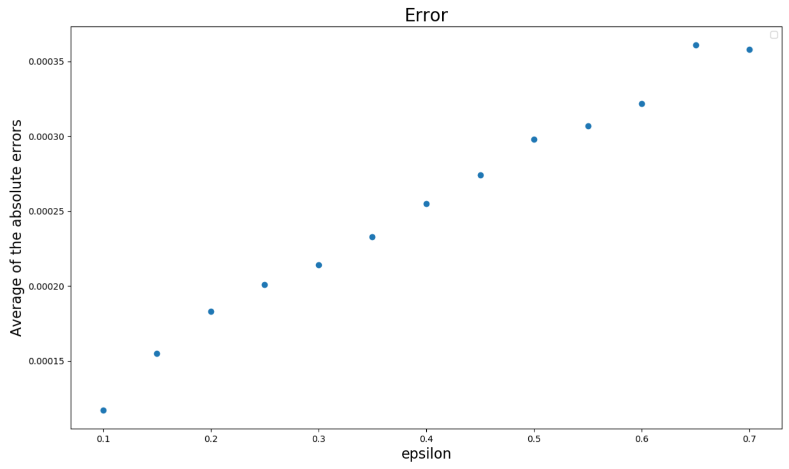

Figure 6: Scale-free network with 10.000 nodes, c=0.5, delta=0.1. The real value of the betweennesses were computed using Brandes' algorithm.

Epsilon	Maximum absolute error
0.100000	0.069396
0.150000	0.066387
0.200000	0.089599
0.250000	0.167863
0.300000	0.175023
0.350000	0.166537
0.400000	0.145046
0.500000	0.266271
0.600000	0.368108
0.700000	0.526522

To assess the running time of the algorithm, we fixed some values of δ, ϵ and c, and generated scale-free networks with some value of V and computed the execution time of the algorithm. In fact, the practical results follow the theoretical bound discussed previously. Regarding scalability, we can clearly see from Figure 8 that the approximation algorithm is much more scalable than Brandes' algorithm.

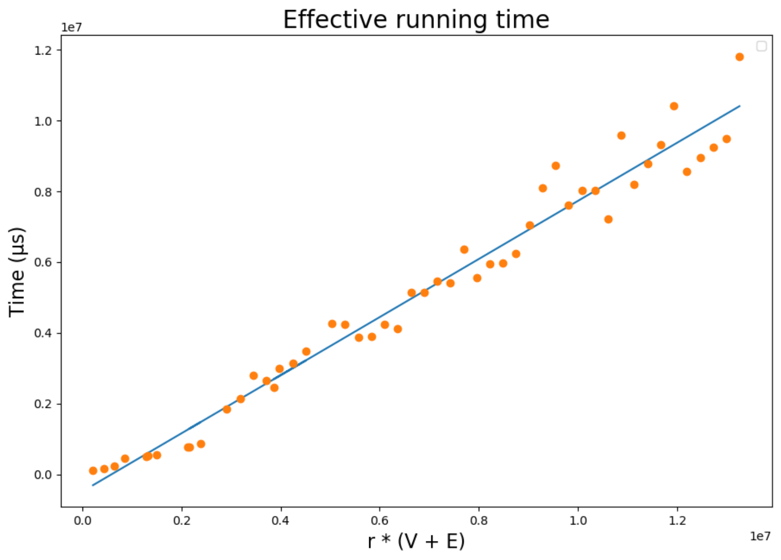

Figure 7: Scale-free networks, c=0.5, epsilon=0.1, delta=0.1

5 Clustering Coefficient

5.1 Theoretical results - time and space analysis

For the clustering coefficient we implemented the first algorithm presented in [6], called "Uniform Wedge". Essentially, it samples wedges uniformly at random and then checks if the two end points of that wedge are connected (a wedge is a set of three vertices such that one of them, the center, is connected to the other two). The author does not provide any theoretical analysis on the complexity of the algorithm. In spite of that, we claim that the time is given by $O(S(\log V + V))$ which is $O(SV)$, where S denotes the sample size (the number of wedges to be sampled). This is because the algorithm runs S times, and for each of those performs a complete binary search and finally checks if two nodes are connected or not. Regarding the space, we only need an additional vector for the accumulative wedge counting, which takes $O(V)$ space. Therefore, the space remains $O(V + E)$.

5.2 Implementation details

The algorithm was straightforward to implement. The only relevant detail is in the binary search. Because it may happen that we're looking for a wedge i that is not present in the accumulative wedge count array, we had to do some adaptations in the binary search algorithm, which results in an execution time

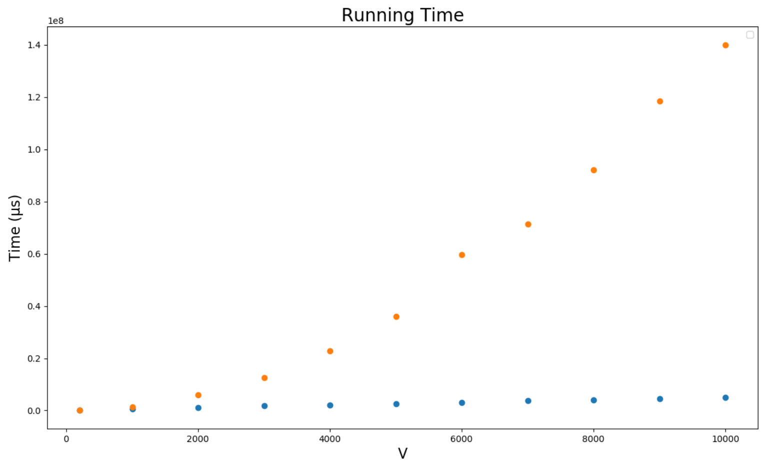

Figure 8: Running time on scale-free networks. Orange-Brandes algorithm. Blue-Approximation algorithm with c=0.5, epsilon=0.1, delta=0.1.

of $\Theta(\log V)$. To illustrate the problem, if we're looking for the wedge number 3 and the accumulative wedge count array is [0, 2, 5], the binary search should return 1, since it is the identifier of the vertex corresponding to that wedge. If the wedge number were 6, then the search should return 2.

5.3 Experimental Evaluation

It is expected that increasing the sample size should result in a better estimate of the clustering coefficient. In fact, Figure 9 supports that observation. An interesting fact that we can draw from the plot is that it seems that considering samples with dimensions higher than 400000, the relative error is not that much further reduced, and so we could say that in principle a size of 40000 wedges to be sampled is a good trade of between running time and quality of the approximation. To build the plot, we fixed a scale-free network with 5000 nodes, calculated the number of wedges of that specific graph, which is given by $\sum v \in V(d[v] * (d[v]-1))/2$, where $d[v]$ denotes the degree of vertex v, and then progressively decreased the sample size and calculated the relative error for each of those values of the samples size. To assess the running time of the algorithm, we generated scale-free networks with some values of V, fixed the sample size and computed the execution time of the algorithm. In fact, the practical results follow the theoretical bound discussed previously.

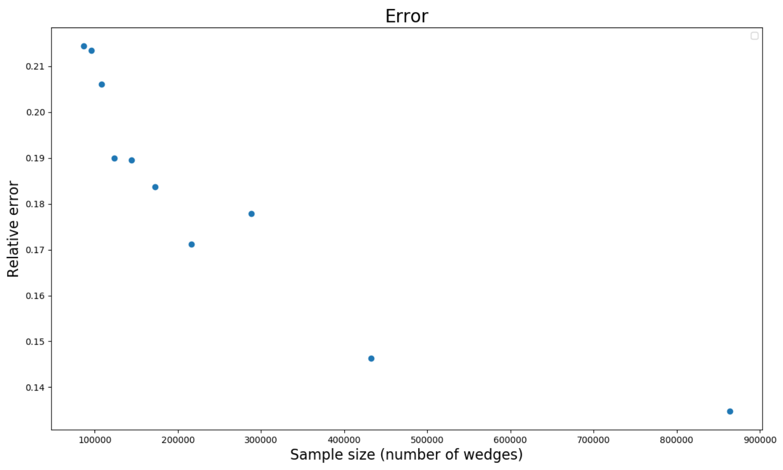

Figure 9: Impact of the sample size in the relative error of the estimate. Tests ran in a scale-free network with 5000 nodes.

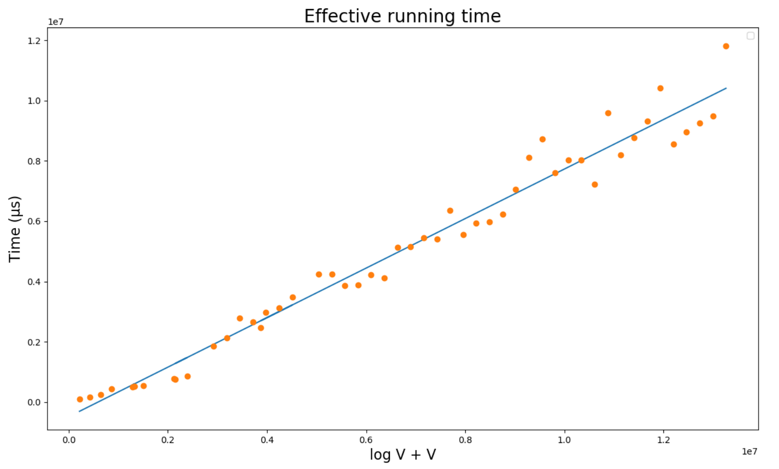

Figure 10: Running time in scale-free networks with a fixed value of the number of wedges to be sampled (S=100000).

References

[1] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. 03 2012.

[2] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. Hyperanf: Approximating the neighbourhood function of very large graphs on a budget. *Computing Research Repository - CORR*, 11 2010.

[3] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[4] Matteo Riondato and Evgenios Kornaropoulos. Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery*, 30:413–422, 02 2014.

[5] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[6] Siddharth Bhatia. Approximate triangle count and clustering coefficient. page 1809–1811, 2018.