AA Project Report

Alberto Pacheco, Gonçalo Paredes, Pedro Capitão and Mahima Raut

April 2020

1 Introduction

In this project we addressed two important measures in network analysis: the betweenness centrality and the clustering coefficient.

We implemented the algorithm for approximate calculation of the betweenness centrality of all vertices of a graph proposed in [1]. Given a graph, a degree of precision ε and a degree of confidence $1-\delta$, this randomized algorithm computes, through sampling, the value of betweenness centrality of each vertex with an additive error smaller than ε, with probability at least $1-\delta$.

We also implemmented the algorithm for approximate calculation of weighted clustering coefficient of a graph proposed in [2]. Given a graph, and parameters ε, ν, this random algorithm approximates the weighted clustering coefficient of the graph with an additive error smaller than ε with probability at least $\frac{\nu-1}{\nu}$.

2 Betweenness centrality

2.1 Preliminaries

Let $G = (V, E)$ be a graph, with $|V| = n$ vertices and $|E| = m$ edges. We assume that G is undirected, unweighted (all edges have unitary weight), and does not have loops or multiple edges between the same pair of vertices. A *path* p between two distinct vertices u and v is a sequence $p = (w_1, \ldots, w_k)$ of vertices such that $(w_1, w_{i+1} \in E)$ for $i = 1 \ldots, k-1$. The *size* of p, denoted $|p|$, is the number k of vertices it contains. If p is a path from u to v, then u and v are called its *end points*, and the remaining vertices of the path form the *interior* of p, denoted $\mathrm{Int}(p)$.

A *shortest path* between distinct vertices $u, v \in V$ is a path that has minimum size among all paths from u to v. The set of shortest paths from u to v is denoted $\mathcal{S}_{uv}$. When u is not connected to v, we define $\mathcal{S}_{uv} = \{p_\emptyset\}$, where $p_\emptyset$ is called the *empty path* and is such that $\mathrm{Int}(p_\emptyset) = \emptyset$. The set of all shortest paths of G is $\mathbb{S}_G = \cup_{u \neq v} \mathcal{S}_{uv}$. For each $v \in V$, we denote by $\mathcal{T}_v$ the set of shortest paths that v is internal to: $\mathcal{T}_v = \{p \in \mathbb{S}_G : v \in \mathrm{Int}(p)\}$. We can now define *betweenness centrality*: it is the function $b : V \to [0, 1]$ given by

$$b(v) = \frac{1}{n(n-1)} \sum_{p_{uw} \in \mathbb{S}_G} \frac{\mathbb{1}_{\mathcal{T}_v}(p_{uw})}{|\mathcal{S}_{uw}|},$$

where the notation p_{uw} is used to indicate that the end points of this path are u and w, and $\mathbb{1}$ represents an indicator function: $\mathbb{1}_A(x) = 1$ if $x \in A$ and $\mathbb{1}_A(x) = 0$ otherwise. The betweenness centrality $b(v)$ of a vertex v is therefore the average, over all pairs (u, w) of distinct vertices, of the fraction of shortest paths from u to w that v is internal to.

The *vertex-diameter* of G, denoted $\mathrm{VG}(G)$, is defined as the maximum size of a shortest path of G: $\mathrm{VG}(G) = \max\{|p| : p \in \mathbb{S}_G\}$. Observe that, when G is connected, $\mathrm{VG}(G) = \mathrm{diam}(G) + 1$, where $\mathrm{diam}(G)$ is the usual graph diameter.

Lemmas 2.1 and 2.2, which are used to prove the correctness of the algorithm, are related to the notion of VC-dimension. While it is not necessary for the understanding the algorithm, it is worthwhile to introduce this concept. Let D be a set and $\mathcal{R}$ a collection of subsets of D. We say that a set $B \subseteq D$ is *shattered* by $\mathcal{R}$ if any subset of B can be obtained as the intersection of B with an element of $\mathcal{R}$, that is, if $\{B \cap A : A \in \mathcal{R}\} = 2^B$. The *Vapnik-Chervonenkis dimension* (*VC-dimension*) of $\mathcal{R}$ is the largest cardinality of subsets of D shattered by $\mathcal{R}$.

2.2 The algorithm

The algorithm receives as input a graph G and $\varepsilon, \delta \in (0, 1)$. Its first step is to determine the number r of paths that must be sampled to attain the desired precision. An upper bound on this number (see Section 2.3) is $\frac{c}{\varepsilon^2}(\lfloor \log_2(\mathrm{VD}(G) - 2) \rfloor + 1 + \ln \frac{1}{\delta})$, where c is a constant. To calculate $\mathrm{VD}(G)$ exactly would require computing the shortest paths between all pairs of vertices, defeating the purpose of this approach. Therefore, we use instead an upper bound $\tilde{d}$ on the vertex-diameter of G that is simpler to compute: we choose a vertex v uniformly at random, compute the distance from v to all other vertices to which v is connected, and we take $\tilde{d}$ to be the sum of the two largest distances. When G is disconnected, this procedure is repeated on each connected component and $\tilde{d}$ is the maximum. It is easy to check that $\mathrm{VD}(G) \leq \tilde{d} \leq 2\,\mathrm{VD}(G)$. .

We now reach the fundamental part of the algorithm, which is to sample r shortest paths, where we take

$$r = \lceil \frac{c}{\varepsilon^2}(\lfloor \log_2(\tilde{d} - 2) \rfloor + 1 + \ln \frac{1}{\delta}) \rceil.$$

After having initialized $\tilde{b}(v) = 0$ for all $v \in V$, the sampling of a path, to be repeated r times, proceeds as follows. Begin by choosing, uniformly at random, a pair (u, w) of distinct vertices. Then next step is to choose one of the shortest paths from u to w. If there are no shortest paths from u to w (i.e., u is not connected to w), then this iteration is finished. Otherwise, uniformly select one path, p, and increment $\tilde{b}(v)$ by $\frac{1}{r}$ for all $v \in \mathrm{Int}(p)$.

Once the cycle ends, the algorithm outputs, for each $v \in V$, the approximation $\tilde{b}(v)$ of the betweenness centrality $b(v)$, which is the relative frequency of v being in the interior of the sampled path.

2.3 Correctness and complexity

It is intuitive that $\tilde{b}(v)$ converges to $b(v)$ as $r \to \infty$, as $\tilde{b}(v)$ is the empirical average over r samples of the random variable $\mathbb{1}_{\{v \in \mathrm{Int}(p)\}}$, (where $v \in V$ is fixed and p is a uniformly chosen shortest path between two uniformly chosen vertices) and $\mathbb{E}[\mathbb{1}_{\{v \in \mathrm{Int}(p)\}}] = b(v)$. The fact that r samples are enough to, with probability $1 - \delta$, approximate $b(v)$ within an additive error of ε for all $v \in V$ is a consequence of the following two results (Theorem 1 and Lemma 1 in [1]):

Lemma 2.1. *Let $\mathcal{R}$ be a collection of subsets of a set D with VC-dimension $\mathrm{VC}(\mathcal{R}) \leq d$ and let $\varepsilon, \delta \in (0, 1)$. Suppose π is a probability distribution on D and $S = (X_1, \ldots, X_k)$ is a sample distributed according to π, of size*

$$k = \frac{c}{\varepsilon^2}\left(d + \ln \frac{1}{\delta}\right),$$

where c is a universal constant ($c \approx 0.5$). Then

$$\sup_{A \in \mathcal{R}} |\pi(A) - \pi_S(A)| \leq \varepsilon$$

with probability at least $1 - \delta$, where $\pi_S(A) = \frac{1}{k} \sum_{i=1}^{k} \mathbb{1}_A(X_i)$ is the empirical average of $\pi(A)$ over the sample S.

Lemma 2.2. *Let $G = (V, E)$ be a graph and consider the collection $\mathcal{R}_G = \{\mathcal{T}_v : v \in V\}$ of subsets of $\mathbb{S}_G$. Then $\mathrm{VC}(\mathcal{R}_G) \leq \lfloor \log_2(\mathrm{VD}(G) - 2) \rfloor + 1$.*

The correctness of the algorithm follows by applying Lemma 2.1 to the sequence S of sampled shortest paths, the collection $\mathcal{R}_G = \{\mathcal{T}_v : v \in V\}$, and the probability distribution π on $\mathbb{S}_G$ given by

$$\pi(p) = \frac{1}{n(n-1)\sigma_{uw}}$$

for each path $p \in \mathbb{S}_G$ with end points u and w, and by observing that $b(v) = \pi(\mathcal{T}_v)$ and $\tilde{b}(v) = \pi_S(\mathcal{T}_v)$, for any $v \in V$.

The estimator $\tilde{d}$ can be calculated in time $O(n + m)$ through a breadth-first-search (BFS) starting from v, until its connected component has been found, and then by doing the same for all connected components of G. Since the vertex diameter $\mathrm{VD}(G)$ is bounded by the number of vertices of G, the sample size r is in $O(\log(n/\delta)/\varepsilon^2)$. Therefore, since the sampling of each path is performed through a modified BFS (which takes time $O((n+m))$, the overall time complexity of this algorithm is $O((n + m) \log(n/\delta)/\varepsilon^2)$. The space complexity of the algorithm is that of the BFS, which is $O(n)$.

2.4 Experimental Results

In order to see how the time cost scales with the size of the input graph, we tested the algorithm on random graphs of increasing size. We generated these graphs using the Barabási-Albert model with parameter $\mathfrak{m} = 10$ (which means that the number m of edges is approximately $10n$) and we measured the running time of the algorithm for each of them. The results, averaged over 5 computations for each graph, are shown in Figure 1. In all these computations we used $\delta = 0.1$, $\varepsilon = 0.02$ and $c = 0.5$ – which means that the calculated betweenness centrality of each vertex should be within 0.02 of the correct value at least 90% of the time. These results indicate that the dependence of the running time on the number of vertices and edges is nearly linear, which is consistent with the theoretical time complexity. The same results are detailed in Table 1.

We tested our implementation of the algorithm on a real world data graph – the `email-Enron` data set, which describes emails exchanged between employees of the Enron corporation. This graph has 36692 vertices and 183832 edges. The running time (averaged over 5 computations) on this graph is represented in Figure 2, for $\varepsilon = 0.01, 0.02, 0.04, 0.06, 0.8, 0.1$, with $\delta = 0.1$ fixed and $c = 0.5$. Here we can see that the computation time is proportional to $1/\varepsilon^2$, as indicated by the theoretical estimate of $O((n + m) \log(n/\delta)/\varepsilon^2)$. These results are detailed in Table 2.

While we were unable to obtain the exact values of betweenness centrality of the `email-Enron` graph, and so we cannot calculate the errors in our approximations, we analyzed the distribution of the values we obtained. More precisely, we ran the algorithm 5 times for each value of ε (in the same range as before, with $\delta = 0.1$ and $c = 0.5$)and calculated the standard deviation and amplitude (difference between the highest and lowest value) of the values of betweenness centrality for the vertex with highest betweenness

centrality. We can see in Figure 3 that the standard deviation sits well below the required precision ε, and so does the amplitude, which indicates that the algorithm actually has significantly greater precision than what is guaranteed in theory (note that the amplitude could go as high as 2ε with all approximations still being within ε from the correct value).

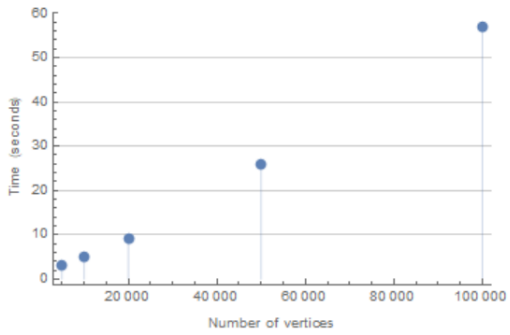

Figure 1: Running time for random graphs generated by the Barabási-Albert model.

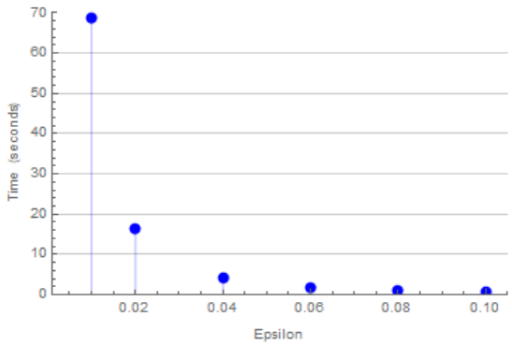

Figure 2: Running time for the `email-Enron` graph for different values of ε.

n	5×10^3	10^4	2×10^4	5×10^4	10^5
Time (s)	3.038	4.974	9.038	26.070	56.877

Table 1: Running time for the Barabási-Albert model.

ε	0.01	0.02	0.04	0.06	0.08	0.1
Time (s)	68.933	16.439	4.330	1.916	1.069	0.724

Table 2: Running time for the `email-Enron` graph.

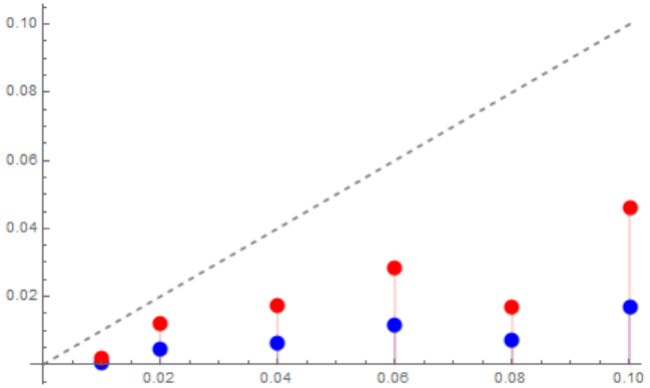

Figure 3: Standard deviation (blue) and amplitude (red) of calculations on the `email-Enron` graph, for different values of ε. The dotted line has equation $y = \varepsilon$.

2.5 Implementation details

To implement the idea of the algorithm we need to be careful with certain details. To start, we can easily calculate our estimate for the diameter using one `bfs` and storing

the two maximum distances we find. With that, we can easily calculate the number of iterations we are going to do. Now, in each iteration we start by sampling two different nodes randomly. But now we need to calculate the shortest paths between these two nodes. In order to store the information we need, we do one `bfs` starting in one of the nodes and using dynamic programming, we calculate the number of shortest paths from the starting node to each node. Furthermore, we also store the edges of the shortest paths, and their weight (That is the number of shortest paths to the node at the end of the edge, since these edges are directed now).

We do this by first storing the basic information the first time we find a node. When we reach a node that was already visited with the same distance of the first visit we update our value of number of shortest paths and our collection of edges.

Now we have what we need to sample a shortest path. We start in the end node (not the one we did the processing for) and we repeat the process: select a new node randomly from the ones that have an edge from a shortest path that start in our current node with probability equal to number of shortest paths ending in new node divided by number of shortest paths ending in our current node. Then we update the value of the centrality for each internal node. Note that the sum number of shortest paths ending in new nodes is equal to the number of shortest paths ending in our current node.

After all iterations we have our estimation of the betweenness centrality, all we have to do is print it.

3 Clustering Coefficient

3.1 Preliminaries

Let $G = (V, E)$ be a simple undirected graph with $|V| = n$ vertices and $|E| = m$ edges. The degree of a vertex v, $d(v)$, is simply the number of other vertices adjacent to it. Similarly we can define the number of triangles of a vertex v, $\delta(v)$, as the cardinality of the set $\{\{u, w\} : u, w \in V, \{v, u\}, \{v, w\}, \{u, w\} \in E\}$, that is, the number of complete subgraphs of size 3 that contain vertex v. The number of triangles of the graph is then $\delta(G) = \frac{1}{3}\sum_{v \in V} \delta(v)$, as each triangle is counted by its three vertices. A *triple* of vertex v is a path of length 2 where v is its center node. The number of triples of a vertex v is trivially given by $\tau(v) = \binom{d(v)}{2}$. Analogously to the number of triangles, it can be generalized to characterize the entire graph, with $\tau(G) = \sum_{v \in V} \tau(v)$.

With these definitions, we can finally define the clustering coefficient of a vertex v as the following: $c(v) = \frac{\delta(v)}{\tau(v)}$. From this, we define the clustering coefficient of G as the average of its vertices' clustering coefficients:

$$c(G) = \frac{1}{|V'|} \sum_{v \in V'} c(v)$$

Where $V' = \{v \in V : d(v) > 1\}$, the set of vertices with at least a triple centered in them (and that, as such, have a well defined clustering coefficient).

On the other hand, one can define the *triple based clustering coefficient* of a graph, or as we'll call it from this point onwards, *transitivity* as simply $T(G) = \frac{3\delta(G)}{\tau(G)}$.

Given a weight function $w : V' \to \mathbb{R}^+$, we define the *weighted clustering coefficient* as

$$C_w(G) = \frac{1}{\sum_{v \in V'} w(v)} \sum_{v \in V'} w(v)c(v)$$

A few weight functions seem particularly natural, but one that needs to be referred is $w = \tau$. Why is that? Because that choice proves that transitivity is a weighted clustering coefficient.

$$C_\tau(G) = \frac{1}{\sum_{v \in V'} \tau(v)} \sum_{v \in V'} (\tau(v) \frac{\delta(v)}{\tau(v)}) = \frac{3\delta(G)}{\tau(G)} = T(G)$$

Now we can finally proceed to the algorithm that intends to approximate $C_w(G)$.

3.2 The algorithm

The main idea of the algorithm is to sample triplets with probabilities according to the weights of their vertices in order to achieve the C_w as the expected value of our output. We'll go into detail on how exactly that works.

The algorithm receives as input

- a positive integer k (a "precision parameter")
- the array A of vertices in $V' = \{v \in V : d(v) \geq 2\}$
- the weights $w(A_i)$ of each node in V'
- the adjacency list for each vertex
- the adjacency matrix of the graph

The first step is creating a set that lets us sample vertices from V' with probabilities proportional to their weights. That's done by defining $W_i = \sum_{j=0}^{i} w(A_i)$ for $0 \leq i \leq |V'|$. From that point onwards, we can finally begin moving towards our result.

We then define $l = 0$. Afterwards, for k times, we do the following: we pick (at random) a vertex from V' with probability proportional to their weight. That's done by choosing, at random, an integer $1 \leq x \leq W_{|V'|}$ and then picking the vertex i such that $W_{i-1} < x \leq W_i$. That's achievable by binary search on the $i's$ and indeed fulfills our intended purpose of having a probability $w(A_i)$ of choosing A_i.

Having picked the vertex, we'll now choose two different nodes that are adjacent to it. We do that just by randomly picking one of the vertices in A_i's adjacency list, let's call it A_j, and then picking another, A_k (trying as many times as necessary to get one that isn't A_j, which shouldn't ever take too long). Doing so, we have that, after picking A_i, we pick any of its triplets with equal probability. Having done this, we increase l by one if A_j and A_k are connected, and we don't do anything otherwise.

After these k iterations, we finally, and simply, output $\frac{l}{k}$. It does seem really simple. But how does this exactly succeed at approximating C_w?

3.3 Correctness and complexity

To see if the expected value of the output is indeed C_w, we can just go ahead and compute it. Let X_i be the random variable assigned to the increment of l on the i-th iteration of the algorithm. $\mathbb{E}[l/k] = \frac{1}{k} \sum_{i=1}^{k} \mathbb{E}[X_i] = \mathbb{E}[X_1]$, as it's clear $\mathbb{E}[X_i]$ is the same for all i. So, we now only have to compute $\mathbb{E}[X_1]$. Let's note that, if the chosen vertex is v, then we have that the probability we'll increase l is precisely $c(v)$. As each of v's triples are equally likely to be chosen, by the definition of $c(v)$ we'll choose a triangle of v with probability $c(v)$. With this, we can now compute $\mathbb{E}[X_1] = \frac{1}{W_{|V'|}} \sum_{v \in V'} w(v)(c(v) \times 1 + (1 - c(v)) \times 0) = C_w(G)$.

The expected value doesn't mean much if we don't have any kind of assurance that the output will very likely be close to it. As such, we'll now prove the following lemma:

Lemma 3.1. *It is possible to choose k such that the previous algorithm outputs a value in $[C_w(G) - \varepsilon, C_w(G) + \varepsilon]$ with probability at least $\frac{\nu-1}{\nu}$ in time $\mathcal{O}(n + \frac{\ln(\nu)}{\varepsilon^2} \ln n)$.*

It's better to start off referring the k we intend to choose: $\lceil \ln(2\nu)/2\varepsilon^2 \rceil$. Now, let's head over to the time complexity. The for cycle where we compute the W_i for $0 \leq i \leq |V'|$ is done in $\mathcal{O}(n)$. Afterwards, the following cycle, with k iterations, only has one moment which isn't done in $\mathcal{O}(1)$ time: finding the i such that $W_{i-1} < x \leq W_i$. This can be done rather quickly using binary search, in $\mathcal{O}(\ln n)$ time. That gives us precisely what we wanted, with the whole algorithm taking $\mathcal{O}(n + \frac{\ln(\nu)}{\varepsilon^2} \ln n)$ time.

Now for the precision we'll take into account Hoeffding's bound. Using the same notation as previously for the random variables X_i, and knowing that they are all bounded by the interval $[0, 1]$ and independent, we get that

$$Pr\Big(\Big|\frac{1}{k}\sum_{i=1}^{k} X_i - \mathbb{E}\Big[\frac{1}{k}\sum_{i=1}^{k} X_i\Big]\Big| \geq \varepsilon\Big) \leq 2e^{-2k\varepsilon^2}$$

Now, knowing that $\mathbb{E}\Big[\frac{1}{k}\sum_{i=1}^{k} X_i\Big] = C_w(G)$ and replacing k by our chosen value, we get

$$Pr\Big(\frac{1}{k}\sum_{i=1}^{k} X_i \notin [C_w(G) - \varepsilon, C_w(G) + \varepsilon]\Big) \leq 2e^{-2(\ln(2\nu)/(\varepsilon^2))\times\varepsilon^2} = \frac{1}{\nu}$$

which is precisely what we wanted to prove.

Furthermore, it does make sense do consider ε and ν fixed parameters. Besides that, it's also natural to consider we can compute $\tau(v)$ in $\mathcal{O}(1)$ for each $v \in |V|$ (for instance, just by knowing the degree of each vertex). Under these assumptions, the previous algorithm runs in $O(n)$ time, allowing us to compute both the transitivity and the clustering coefficient in linear time.

However, one can notice that our way of creating a set which allows us to sample vertices from V' looks a little useless for the latter case. We can remove the first cycle (where we compute the W_i for $0 \leq i \leq |V'|$) and change our random choice from an integer $1 \leq x \leq W_{|V'|}$ to simply a random choice among the vertices in V'. By doing so, we remove the $\mathcal{O}(n)$ cycle from the equation and also the $\mathcal{O}(\ln n)$ binary search as well. That means we can approximate the clustering coefficient of the given graph in constant time.

3.4 Experimental results

Now it's time to test our algorithm(s). To do so, we'll have to find a few graphs whose clustering coefficients we know and use our algorithm on them. First of all, just to *start believing* in this algorithm, let's just test it on this tiny graph a few times and see what we get: both the output and how long it gets for the algorithm to end. Specifically, we're going to look at this graph that's about a dolphin social network and has only 62 nodes, taking $\nu = 100$ and $\varepsilon = 0.001$. If we run the algorithm for clustering coefficient on this graph, we get 0.2592 in $0.203s$, 0.2588 in $0.224s$ and 0.2588 in $0.231s$, which seems alright when the real value is 0.258958. For the transitivity we get 0.3086 in $0.247s$, 0.3081 in $0.231s$ and 0.3091 in $0.235s$, which is once again alright when the real value is 0.308776.

After this, it's time to do a more systematic testing of this algorithm. We want to test two things: its time complexity and its efficacy. Even though we're generally not interested in varying ε and ν, we'll do it for the sake of testing. It's very easy to believe

the time complexity will be $\mathcal{O}(n + \frac{\ln(\nu)}{\varepsilon^2} \ln n)$, but not as easy to believe the efficacy of the algorithm, one could argue, so it's still of value to effectively test this.

To test the time complexity, we'll choose a few different values for ε, ν and n and try to run our algorithm, for parameters ε, ν on graphs with n vertices. To test the efficacy, we'll just use those tests and see, for each graph, out of 1000 tests with the algorithm, how many times the output was within an error ε of the correct answer (in order to effectively see if the algorithm gets it right at least $\frac{\nu-1}{\nu}$ of the times). Let's list the different values of each parameter we intend to test. The references for the graphs can be found at the end of the document, in the correct order.

In the tables we can see, for each triplet n, ν, ε, over 100 tests (20 for the largest case), the average runtime, the longest runtime and the percentage of outputs within ε of the right answer. For ν equal to $25, 50$ and 100, one would expect a success rate of 96%, 98% and 99%, respectively. We'll start by testing the algorithm for the clustering coefficient

$n = 10670$	$\nu = 25$	$\nu = 50$	$\nu = 100$
$\varepsilon = 0.1$	0.00003 0.00012 100	0.00003 0.00008 100	0.00004 0.00009 100
$\varepsilon = 0.01$	0.0021 0.0030 100	0.0025 0.0047 100	0.0028 0.0041 100
$\varepsilon = 0.001$	0.1953 0.2843 98	0.2282 0.2599 100	0.2712 0.4198 100

$n = 18772$	$\nu = 25$	$\nu = 50$	$\nu = 100$
$\varepsilon = 0.1$	0.00009 0.0001 100	0.0001 0.0002 100	0.0002 0.0002 100
$\varepsilon = 0.01$	0.0081 0.0093 100	0.0095 0.0110 100	0.0109 0.0131 100
$\varepsilon = 0.001$	0.9604 1.0078 98	1.1376 1.1707 99	1.3125 1.4210 100

$n = 1696415$	$\nu = 25$	$\nu = 50$	$\nu = 100$
$\varepsilon = 0.1$	0.0002 0.0002 100	0.0002 0.0002 100	0.0002 0.0002 100
$\varepsilon = 0.01$	0.0132 0.0149 100	0.0157 0.0176 100	0.0179 0.0201 100
$\varepsilon = 0.001$	1.5949 1.6558 100	1.8846 1.9633 100	2.1409 2.2198 100

In theory, this algorithm should be $\mathcal{O}(\ln(\nu)/\varepsilon^2)$ in time. We do see that, for the same n, the time increases linearly when ν doubles, which is coherent with the hypothesis, and increases by about 100 times when we decrease ε by a factor of 10 which, once again, makes sense according to the theoretical time complexity. However what doesn't seem to make sense is how the time changes with n. That is probably due to some practical details. For instance, even though we ask for the adjacency matrix as the input, for the possibility of getting "are v_1 and v_2 connected?" in constant time, in general that won't be something we actually want to do, due to its space complexity. As such, we just use an unordered set to store the edges in order to be able to compute the answer to that question quickly. With that implementation choice, we end up almost doubling the time from the second case to the third case which, although it isn't perfect, still ends up being a good compromise between time and space.

What about the accuracy? There really isn't much to be said about it, as the output is successful even more frequently than the lowerbound for the probability of success would suggest. We end up concluding this algorithm for approximating the clustering coefficient is a pretty nice one, which achieves low running times even for huge graphs where exact methods would falter.

Now we can go ahead and run the algorithm to approximate the transitivity of the graphs.

$n = 10670$	$\nu = 25$	$\nu = 50$	$\nu = 100$
$\varepsilon = 0.1$	0.00013 0.00048 100	0.00013 0.00019 100	0.00014 0.00022 100
$\varepsilon = 0.01$	0.0029 0.0049 100	0.0033 0.0058 100	0.0038 0.0056 100
$\varepsilon = 0.001$	0.2573 0.3576 100	0.3004 0.4118 100	0.3570 0.5351 100

$n = 18772$	$\nu = 25$	$\nu = 50$	$\nu = 100$
$\varepsilon = 0.1$	0.0003 0.0004 100	0.0003 0.0005 100	0.0003 0.0004 100
$\varepsilon = 0.01$	0.0107 0.0127 100	0.0125 0.0161 100	0.0147 0.0189 100
$\varepsilon = 0.001$	1.2141 1.3615 97	1.4679 1.6059 99	1.5380 1.8271 100

$n = 1696415$	$\nu = 25$	$\nu = 50$	$\nu = 100$
$\varepsilon = 0.1$	0.0102 0.0116 100	0.0102 0.0119 100	0.0102 0.0116 100
$\varepsilon = 0.01$	0.0272 0.0290 100	0.0309 0.0328 100	0.0337 0.3589 100
$\varepsilon = 0.001$	1.8215 1.8782 100	2.1375 2.2244 100	2.4398 2.5179 100

Here we should expect $\mathcal{O}(n + \frac{\ln(\nu)}{\varepsilon^2} \ln n)$ time spent. For most of these values of n, ν, ε, the time spent in the first cycle (of lenght n) doesn't seem to be the most prevalent part of the algorithm, when it comes to the duration, as the increase compared to the first algorithm is rather small. When n is the largest and $\varepsilon > 0.001$, we can actually believe the first cycle is taking a rather long time when compared to the second cycle, but as we can see if we try to get decent precision with high probability the second cycle is still where we spend the most time.

Fixing n, we can make the same conclusions as before about the time complexity, unless when, taking the largest n, we go from $\varepsilon = 0.1$ to $\varepsilon = 0.01$. That's because in those cases, the $\mathcal{O}(n)$ cycle is actually relevant in the total time spent and, as such, we shouldn't expect the decrease of ε by a factor of 10 to increase the total time by a factor of 100, but only the second cycle of the algorithm.

Once again, there isn't much to be said about the accuracy of this algorithm. The relative frequency is bigger that the lower bound of the probability, as it was to be expected, and we can also once again conclude this algorithm provides a great option for the approximation of the transitivity of huge graphs.

4 Conclusions

By comparing Figures 1 and 2 of our report to Figures 5 and 4 b) of [1], respectively, we see that our computations ran in less time than what is presented in the article that introduces this algorithm, for similar graphs. This suggests that our implementation of the algorithm is indeed efficient.

For the clustering coefficient algorithm, the most we can do is admire how such a trivial algorithm can provide such a great alternative to the common deterministic algorithms. Besides that, we also think it's of value to note that trade-offs are sometimes necessary when implementing algorithms. In this precise case, it was using unordered sets instead of the proposed adjacency matrix in order to highly decrease the space complexity of the algorithm, even if slightly increasing the running time.

References

[1] M. Riondato, E. Kornaropoulos. *Fast approximation of betweenness centrality through*

sampling. In Proceedings of the 7th ACM international conference on Web search and data mining. Association for Computing Machinery, New York, NY, USA, 413–422 (2014).

[2] Thomas Schank and Dorothea Wagner, *Approximating Clustering Coefficient and Transitivity* In Journal of Graph Algorithms and Applications, Vol. 9, no. 2, pp. 265-275, 2005

[3] Network Repository, Dolphin Social Network

[4] SNAP, Autonomous systems - Oregon-1

[5] Network Repository, Astro Physics collaboration network

[6] SNAP, Autonomous systems by Skitter