Fast calculation of the average path length in large complex graphs

Master's Thesis

by

Mikuláš Poul

mikulas.poul.18@ucl.ac.uk

Supervisor

Dr Shi Zhou

s.zhou@ucl.ac.uk

This thesis is submitted as part requirement for the

Master of Science

in

Web Science and Big Data Analytics

at

University College London.

6th of September 2019

This thesis is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

An electronic version of this thesis and related source codes are available from the website `https://masters-thesis.mikulaspoul.cz/`.

Abstract

The average path length is one of the most commonly used properties of graphs, an easily interpretable measure of the efficiency of a graph. The problem with it is its high computational complexity. On graphs with tens or hundreds of thousands of vertices, which are very common these days, the calculation becomes prohibitively too slow for any real-time application or calculations on multiple graphs. This thesis introduces an improved algorithm for calculating the average path which uses the power-law distribution of the real-world scale-free graphs to prune them for purpose of the pair-wise distance calculation, which can speed up the calculation up to 8.6 times on real-world graphs. This thesis further experiments with approximation using sampling on top of the improved algorithm. In experiments with real-world graphs, this thesis achieves a maximum error of less than 1% using sample size just 6% with a further tenfold speedup.

Acknowledgements

First and foremost I would like to thank my supervisor Dr Zhou. Without his advice and thoughtful questions and comments, this thesis would not be what it is today.

None of this would be possible without the love and support from my parents, so my deep gratitude goes to them.

All science stands on top of the shoulders of giants and I am grateful to all authors who published in topics relating to my thesis, as I have drawn inspiration and guidance from their work.

I would also thank my friends who helped me with this thesis, especially in checking for typos and advising on the typography of the text. Also thanks to Miro, who supervised my bachelor's thesis and who taught me to write long academic texts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are a simple way of looking at naturally occurring phenomena in the real world, which provide many insights on how the researched objects interact. A graph consists of two sets of objects, a set of vertices, and a set of edges, which connect these vertices (also called networks, nodes, and links respectively). Many different properties can be calculated on a graph.

These days, with the emergence of social networks and improvements in data mining, many real-world graphs grow in size very rapidly. The computational complexity of calculating the properties differs by property, some can be calculated quickly even on large graphs, but some others, not even very complicated properties, have prohibitively high complexity.

One of these basic properties is the average path length (APL) which is very often used in graph analysis to show the efficiency, speed of communication or propagation within the graph. It is the average of all distances within a graph, so it shows what distance has to be traversed before reaching all other vertices on average. The computational complexity of calculating the APL goes up quickly because the distances from all vertices to all other vertices need to be calculated, and the problem gets even worse on weighted graphs.

For any single graph, one might bear this, even though the calculation could take a couple of hours on a usual workstation, but once one needs to calculate the value for multiple different graphs, any possible speedup would save time and energy. The time needed to calculate the value could be especially an issue for calculating the value on temporal graphs, special graphs which have a different set of edges at each step in time. These could be for example created by storing the structure

of a social network every day. Any improvement in the time complexity of the calculation would then be multiplied by the number of steps in the temporal graph.

The time complexity also prohibits usage of the average path length in real-time applications for large graphs. In this world, all social media sites are graphs and all technological networks running the world are graphs and the complexity prohibits the real-time calculation of the APL on the constantly evolving graphs.

In some cases, the exact value is not even strictly necessary if there is a fast and accurate approximation available. In exploratory data analysis when one needs a quick overview of some graph's properties, to get a sense of the graph, letting a computer compute the exact value is a waste of both time and the electricity to run the computer. Similarly, when for example comparing two graphs, exploring if they are similar, comparing approximations can be sufficient enough to determine the difference in values.

This thesis attempts to tackle both of these issues in the calculation of the average path length, the time complexity and the approximation. First speeding up the calculation of the exact value. I will explore speeding up the calculation by excluding some of the vertices from the graph in the pair-wise distance calculation stage and using other vertices to interpolate their distances to other vertices. There are two ways this thesis will do so, one on both unweighted and weighted graphs and the other only on weighted ones.

Vertices which are linked to only one other vertex can be excluded. Their distances to all other vertices are simply the distance from the only vertex they are connected to plus the distance to that vertex.

After excluding the vertices only linked to one other vertex, another set of vertices have a trivial distance to all other, vertices which are linked to only two other vertices, the distance to all other vertices is through one of the two. This method only works well on weighted graphs, because after the exclusion a new edge has to be created with the sum of the two excluded edges, which requires already a weighted graph or a creation of one.

I believe the effect of this exclusion methods can have a large effect on speeding up the calculation, especially in real-world scale-free graphs, where vertices with a degree just one or two form a big part of the graph. As far as I know, nobody has yet proposed this solution to speed up the average path length.

Then, the approximation of the average path length. Ye et al. proposed a random vertex sampling method to approximate the value in [1], which uses distances from a portion of the vertices to all other vertices to approximate the average. In the second part of the thesis, I will try to reproduce the results in that paper on other graphs and experiment more broadly with the sampling methods and the size of the set used for the approximation. I will also closely follow the speed of the approximation with relation to the accuracy to suggest a method to get accurate results very quickly, something the authors did not focus greatly on.

The thesis is structured as follows. The next chapter defines basic concepts used in the report and their notation. Chapter 3 reviews past research into speeding up the calculation and approximations of the average path length and some other properties. Chapter 4 focuses on the speeding up of the exact calculation and chapter 5 focuses on the approximation. The thesis closes with a conclusion in chapter 6.

Chapter 2

Basic definitions and notations

This chapter describes and defines the basic concepts used in the rest of this thesis and shows what notation will be used.

Graphs are mathematical structures used for precise descriptions of objects and their relations. A graph G is composed of a set of *vertices* V and a set of *edges* E, an explicit notation of a graph would be $G(V,E)$, but unless specific sets are referred, this can be omitted. An edge is an unordered pair with the two linked vertices, in mathematical notation $\{a,b\}$, $a,b \in V$, $a \neq b$. The unordered part means that if there exists an edge between vertices a and b there also automatically exists an edge between b and a.

This definition can be extended to add a direction to edges to create a *directed graph* compared to the *undirected graph* defined above. In a directed graph an edge would be an ordered pair where the first vertex is the *source* and the second is the *target*, however, this thesis only considers undirected graphs.

This definition does not allow *self-loops*, meaning a vertex can not be linked to itself. Similarly, *parallel edges*, where there exist multiple connections between the same two vertexes, are not allowed. The definition can be extended to allow these, but again, for simplicity, this thesis does not consider such graphs.

A weight can be associated with an edge, creating a *weighted graph*. This weight is noted by $w_{a,b} \in \mathbb{R}$, as the weight of the edge between a and b. In case of the vertices not being connected, this value is infinity, in mathematical notation $\{a,b\} \notin E \Rightarrow w_{a,b} = \inf$. Since the graph is undirected, $w_{a,b} = w_{b,a}$. This thesis only considers positive weights without zero weights.

The number of vertices is expressed by n or $|V|$, the number of edges in the graph is expressed by m or $|E|$. Graphs are also called *networks*, vertices are called *nodes* and edges *links*, this thesis will use the terms interchangeably.

The *degree* of a vertex represents the number of edges of that vertex, noted by $\deg(a)$ or $\deg_G(a)$ for a specific graph G. The set of other vertices directly linked to vertex v is called a *neighbourhood* (noted by $N(v)$) and individual vertices from that set are called *neighbours*.

A *path* is a sequence of edges in a graph which connects two vertices. *Length* of a path is defined as the number of edges in a path for an undirected graph, or as a sum of the weights of the edges in a directed graph. A *shortest path* if such a path that has the shortest length. The length of the shortest path between two vertices is called the *distance* of those two vertices. The distance is noted by $d_{a,b}$ if a path between the two vertices does not exist the distance is infinite.

The maximum distance from a vertex to any other vertex is called the *eccentricity* of that vertex, noted by e_v for vertex v. In mathematical notation it is $e_v = \max_{w \in V, a \neq w} d_{v,w}$. The maximum eccentricity in a graph is called the *diameter* of a graph, the minimum is called *radius*. The *effective diameter* is the lowest value at which a certain percentage of vertices can be reached from all vertices, e.g. 90%.

Subgraph $S(V_S, E_S)$ of a graph $G(V_G, E_G)$ is a graph where the sets of vertices and edges are subsets of the sets in the original graph, $V_S \subseteq V_G$ and $E_S \subseteq V_G$. An *induced subgraph* is a subgraph which includes all edges from E_G between vertices in V_S. A *connected component* (or sometimes just a component) is an induced subgraph such that there exists a path between any two vertices in it and any of the vertices are not connected to a vertex outside the subgraph. The *largest connected component (LCC)* is the connected component with the most vertices.

The *average path length (APL)*, also *average shortest path length* or *average distance*, within a connected component $S(V, E)$ is defined in equation 2.1. It could be re-

defined for a whole graph comprised of multiple components, but this thesis will only focus on connected components.

$$l_S = \frac{\sum_{i,j \in V, i \neq j} d_{i,j}}{|V| \cdot (|V| - 1)} \tag{2.1}$$

A *tree* is a graph where any two vertices are connected by exactly one path, or also a connected graph without any cycles. A *forrest* is a graph comprised of multiple trees.

A *temporal graph* is a graph which has a different set of edges at each point or step in time.

Chapter 3

Literature review

As far as I am aware, there is no current approach available for speeding up the exact calculation of the average path length (APL), so, unfortunately, there is nothing to review or learn from there. There have been some papers relating to the approximation of APL, which I will discuss. There has been significantly more work done in approximating some other graph properties related to the APL, which I will also discuss.

One of the main sources I am using in this thesis (mainly chapter 5) is a 2010 paper "Distance Distribution and Average Shortest Path Length Estimation in Real-World Networks" by Ye et al. [1]. This paper explores the distribution of distances within a graph and finds that it usually follows a normal distribution with a mean and standard deviation. They further explore multiple methods to estimate the APL using sampling methods. Out of the four methods they proposed, only the *random vertex sampling* produces an unbiased and stable estimation of the value, which they show on a variety of real-world graphs and even provide a mathematical basis for the estimation, based on the normal distribution of the distances. They only focus on largest connected component (LCC) on undirected unweighted graphs. The paper tested the approximation with several sample sizes on different scales, but it does not go into detail how to select the sampling size and it does not consider the speed of the approximation almost at all.

I am also only focusing on undirected graphs, but there is an interesting paper analysing the random vertex sampling method on directed graphs and specifically on the *largest weakly connected component (LWCC)* of the graph. The 2017 "Fast approximation of average shortest path length of directed BA networks" by Mao

and Zhang [2] introduces *Global Reachable Nodes (GRN)*, vertices which can reach most of the other vertices in the graph. APL on the LWCC of directed graphs is calculated similarly to APL on an LCC of undirected graphs, but only distances to vertices which can be reached from a vertex are summed and divided. Therefore the GRN have a much bigger effect on the value of the APL than other nodes, the paper shows that by using just the GRN of the network provides a value very close to the true APL. The problem, of course, is that to find all the GRN one needs to calculate reachability and distances from all vertices. They, therefore, resort to sampling vertices and they find that if the sampling is random, there are enough GRN in the sampled set to provide a good approximation. They tested on much larger graphs than this thesis, but only with a couple of sample sizes. Their maximum accuracy error was lower than 1% for all the graphs. They, unfortunately, did not go into much detail about the selection of the sample sizes.

There are a couple of approaches which use landmarks or pivots to approximate the distances on the graph itself and use those distances to calculate the average path length. For example [3] focuses on the selection of the landmarks, introducing heuristics which provide good results. Then [4] focuses on also providing the actual path as well, and it provides very quick and very accurate results at the cost of a massive pre-built index, with storage requirements exceeding the original graph.

There are properties related to the average path length which were research significantly more as it seems, namely the radius, diameter and the effective diameter. One of the papers which focused on that is a 2013 paper "Computing the eccentricity distribution of large graphs" by Takes and Kosters [5]. This paper introduced a method which can speed up the calculation of the exact value using pruning of the graph (the inspiration for a part of chapter 4) and using bounds for the values of the eccentricity of each vertex. Then they introduce an approximation as well, which uses sampling to calculate the value.

Another influential paper on this topic is the 2002 "ANF: A fast and scalable tool for data mining in massive graphs" by Palmer et al. [6]. This paper does approx-

imation on the *neighbourhood function*, another measure of the connectivity and distances in graphs, showing what percentage of vertices can be reached at a certain distance (which can also be used to calculate the effective diameter). They use multiple runs of sampling and bitwise magic on adjacency matrix to approximate the neighbourhood function – I will not pretend I understand exactly how it works. There is an extension of the approach with the 2011 paper "HyperANF: Approximating the neighbourhood function of very large graphs on a budget" by Boldi et al. [7], which uses better counters which speed it up further. The paper also provides a way to approximate the average path length using the neighbourhood function.

I would like to also mention one other paper, which is not strictly related to the average path length and that is the 2009 paper "Conservation of alternative paths as a method to simplify large networks" by Liu and Mondragón. This paper focuses on the reduction of vertices within a graph that keeps the number of alternative paths between the vertices, using vertex contraction, merging of several vertices. It reduces the number of vertices but keeps the complexity of the paths within the graph, but it inspired me for part of chapter 4 together with the eccentricity distribution paper.

Chapter 4

Faster exact calculation

This first part focuses on speeding up the calculation of the exact average path length (APL) in large graphs. The average path length is one of the most common properties used to describe a graph. It is a simple and explainable measure of efficiency and speed of communication within a graph.

The trouble is that with the growing size of a network calculating the average path length gets harder, because of the complexities of algorithms for finding the pair-wise distance between all vertices, as all the distances are needed for the exact value. This is especially the case for weighted graphs, where the algorithms are comparatively slower to unweighted graphs, as is discussed in detail in section 4.1. Large graphs are very common these days, in the age of constant data collection, and once the number of vertices reaches a certain point, the time to calculate the distances becomes prohibitive.

Many of the real-world graphs are also constantly changing, especially in the case of social networks. This means that if somebody wants to track the changes in the average path length, the value has to be calculated in all the stages of the collected temporal graph. This, depending on the size of the network and the frequency of the data collection, can make the calculation very slow. Any speedup of the calculation on a single graph then makes the speedup on temporal graphs even larger in terms of the absolute amount of time.

Any real-time analysis of these dynamic graphs is also not possible due to the complexity of the problem. For all of these reasons, any speedup to the calculation is useful.

Many of the real-world graphs are so-called *scale-free*, meaning they follow a power-law degree distribution. These scale-free graphs include social networks, web networks and many other types of real-world graphs [8]. A scale-free graph, thanks to its distribution, has a large portion of vertices with a low degree. From some of these vertices, the distances to other vertices can be much more easily calculated than through the classical algorithms, so they can be dropped from the expensive pair-wise distance calculations, leading to a speedup. This is the case for all vertices with degree one and two, and depending on the graph some vertices with a slightly higher degree can be excluded as well.

For this thesis, only undirected graphs and the largest connected components are considered. The approach could be updated to work on directed graphs, similarly also across multiple components, but it would add unnecessary complexity and edge cases for this thesis.

4.1 Analysis

4.1.1 Complexity of computing the average path length

Algorithm 1 shows the algorithm for summing of pair-wise distances. This specific version assumes a connected component as mentioned in the introduction of this chapter, for simplicity. Out of the box we see that the complexity of the sum is $\mathcal{O}(n^2)$, without even calculating the pair-wise distances between the vertices. $\mathcal{O}(n^2)$ is also the space complexity of calculating the average path length because all the distances need to be stored.

Algorithm 1 Summing of pair-wise distances

```
s = 0
for all i ∈ V do
    for all j ∈ V, j ≠ i do
        s += d_{i,j}
return s
```

This complexity could be reduced to $\mathcal{O}(n(n-1)/2)$ if a triangle sum would be used. Asymptotically it is the same complexity so for simplicity I did not include

that here or in the improvements. Not including this specific change also shows that the algorithms could be converted for directed graphs.

The complexity of getting the pair-wise distance depends on the type of graph. There are two algorithms for computing the pair-wise distance, Johnson's algorithm with complexity $\mathcal{O}(nm + n^2 \log n)$ [9] and Floyd-Warshall algorithm with complexity $\mathcal{O}(n^3)$ [10]. The Floyd-Warshall algorithm is only useful for dense graphs, and therefore is not useful here as large scale-free graphs are sparse [11]. There is also a more basic approach available for unweighted graphs, running a breadth-first search (BFS) from each vertex at complexity $\mathcal{O}(n(n+m))$ [12]. This approach also allows for easier parallelisation.

4.1.2 Properties of a power-law distribution

Scale-free graphs follow a power-law distribution, meaning that the probability of a vertex having a certain degree follows the distribution $P(d) \sim d^{-\gamma}$, where usually $2 < \gamma < 3$ [8]. An example of this degree distribution can be seen in figure 4.4 on page 34, showing the degree distributions of the four graphs used further in this chapter.

The consequence of this degree distribution is that a large portion of vertices in the graph has a very small degree. For example, for a graph with an ideal power-law distribution, depending on the γ of the specific graph, just the portion of vertices with a degree just one or two is 0.5 to 0.75. This is only the ideal case, the power-law distribution can only fit from a specific degree and the distribution until that degree can be different, but either way, the portion of such vertices is high.

In real graphs, this number can vary significantly. When considering graphs that are used further in the chapter, in graph `cond-mat-2003` 21% of vertices have degree one or two and in graph `as-22july06` it is 76%. In the randomly generated `ba-graph` the percentage is 44%. As a nice example that this high percentage is a property scale-free graphs, the `er-graph` only has 8% of vertices with degree one or two.

4.1.3 1-core simplification of a graph

The large number of vertices with degree one can be used to simplify the pair-wise distance calculation and in that speeding it up. Let's take a portion of a graph shown in figure 4.1, with dotted lines from vertices a and c indicating a connection to more vertices in the rest of the graph. The figure shows that any path from e to all other vertices passes through vertex b. Its distance to any other vertex x is $w_{e,b} + d_{b,x}$. One can then remove the vertex from the graph for the pair-wise distance calculation and then just take it in account in the summing of distances.

The situation is similar with vertices marked with an orange border, f to j, any path to all other vertices passes through vertex b, with the exception to other vertices marked with the same colour. Distance to the other vertices is therefore the sum of distance to b from both ends, for example for i the distance to a random x is $d_{i,b} + d_{b,x}$, and the distance to a vertex x with the same colour is simply $d_{i,x}$.

Note that vertices with a degree more than one and two can be included in this, take vertex g, which has degree 4.

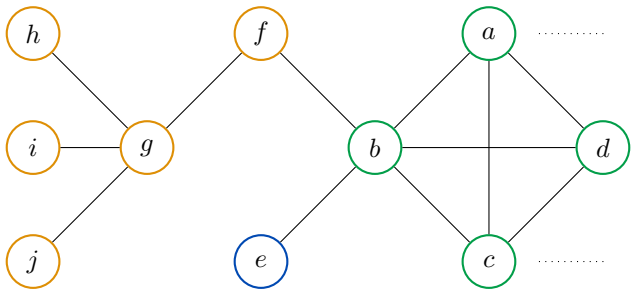


Figure 4.1: Illustration of a 1-core simplification of a graph

This simplification is inspired by the 2013 paper "Computing the eccentricity distribution of large graphs" by Takes and Kosters [5]. This paper focuses on efficiently computing the eccentricity distribution (and the radius and diameter). When calculating the eccentricity one is only interested in the longest path, not all the paths, so they reduced multiple one-degree vertices neighbouring a single vertex to one, as that retains the maximum distance. I was inspired by this to remove all the

vertices with degree one with the update to the sum of distances and realised that actually, the simplification can continue further.

4.1.3.1 Definitions

Let's describe these concepts more precisely. This reduction can be performed on the *1-core* of the graph. A 1-core is a set of vertices obtained by iteratively removing all vertices from the graph with degree 1 until all other vertices have a degree higher than 1, the 1-core being the removed vertices. This is a specific case of a *k-core*, where this can be done for any degree [13]. When an induced graph with only the vertices from the 1-core is created, it is a forest, a graph comprised of multiple trees.

Let's define some notation for vertices in the 1-core and their properties. First, let V^1 denote the set of vertices in the 1-core and n_1 the size of this set. Let p_i denote the closest vertex outside the 1-core to which there exists a path from i and d_i the distance to that vertex. Let t_i denote be the identifier of the tree on which i appears because as shown in the example figure, multiple trees can be connected to the same vertex outside the 1-core. Additionally, let V_a^t be the set of vertices with a being their tree identifier and T be set of tree identifiers.

4.1.3.2 Algorithm for the average path length calculation

Algorithm 2 shows the algorithm for summing of pair-wise distances with the 1-core simplification. For a vertex outside the 1-core, the sum of distances from that vertex is the sum of distances to other vertices outside the 1-core plus the distances to vertices in the 1-core (via their closest vertex). For a vertex in the 1-core, this can be simplified as the sum is linked to the sum of distances of its closest vertex. It is the sum of the distances from the closest vertex to all vertices, with the distances from the closest vertex to vertices on the tree being replaced with distances just within the tree, plus the distance to the closest vertex times the size of vertices outside the current tree.

Algorithm 2 Summing of pair-wise distances after 1-core reduction

$s = 0$
for all $i \in V \setminus V^1$ **do**
  $s_i = 0$ ▷ Sum of distances from vertex i
  $s_{i,t} = 0, \forall t \in T$ ▷ Sum of distances from i to tree t
  for all $j \in V \setminus V^1, j \neq i$ **do** ▷ Distances to other remaining vertices
    $s_i \mathrel{+}= d_{i,j}$
  for all $j \in V^1$ **do** ▷ Distances to 1-core vertices
    $p = p_j, q = t_j$
    $s_{i,q} \mathrel{+}= d_j + d_{i,p}$
  $s_i \mathrel{+}= \sum_{t \in T} s_{i,t}$
  $s \mathrel{+}= s_i$
for all $j \in V^1$ **do**
  $p = p_j, q = t_j$
  $s_j = s_p + d_j \cdot (|V| - |V_q^t|) - s_{p,q}$ ▷ Distances outside the tree
  for all $l \in V_q^t$ **do** ▷ Distances in the same tree
    $s_j \mathrel{+}= d_{j,l}$
  $s \mathrel{+}= s_j$
return s

4.1.3.3 Detecting the 1-core

With this updated algorithm the pair-wise distance has to be calculated only on the induced subgraph with vertices $V \setminus V^1$ and on induced subgraphs of individual trees. First, though, the 1-core has to be detected, showed in algorithm 3.

The 1-core detection is based on the algorithm for detecting the k-core value by Batagelj and Zaveršnik [14], modified to just detect the 1-core. The algorithm first works on an iterative basis, first processing vertices with degree 1, then vertices that would have a degree 1 if the previous were removed and so on while there are vertices to process. The neighbour not yet in the set V^1 is marked as the temporary closest vertex.

Once all the vertices in the 1-core are found, they are iterated over in the reverse order they were found in, and the distance from the closest vertex is calculated and the tree identifier set. The tree identifier is the vertex from a tree that is directly connected to the closest vertex. Since the order of finding vertices in the 1-core is from the furthers from the closest vertex of the tree, the distance and tree identifier

Algorithm 3 Detecting the 1-core, distances to closest vertices and tree identifiers

$V^1 =$ empty set
$Q =$ empty queue
$O =$ empty list ▷ Order of discovery
$s_v = \deg(v), p_v = \varnothing, \forall v \in V$
for all $v \in V,\ s_v = 1$ **do** ▷ Start with vertices with degree 1
    push v into Q
while Q is not empty **do**
    pop v from Q
    add v to V^1, append v to O
    for all $w \in N(v)$ **do**
        if $w \notin V^1$ **then**
            s_w −= 1 ▷ Simulate v being removed
            $p_v = w$
            if $s_w = 1$ **then** ▷ After removal of v degree of w would be 1
                push w into Q
$d_v = 0, t_v = \varnothing, \forall v \in V^1$
for all $v \in O$ in reverse order **do**
    if $p_v \notin V^1$ **then** ▷ p_v is the closest vertex
        $t_v = v$ ▷ Set tree identifier to self
        $d_v = w_{v,p_v}$
    else
        $t_v = t_{p_v}$
        $d_v = d_{p_v} + w_{v,p_v}$
for all $v \in V^1$ **do**
    $p_v = p_{t_v}$

can be passed through as a message. The final closest vertex of each edge is the temporary closest vertex of the vertex set as tree identifier.

4.1.3.4 Complexity

Let's recall that the complexity of finding the average path length with the basic approach is $\mathcal{O}(n^2 + nm + n^2 \log n)$. When using the 1-core simplification the complexity has a couple of more terms, but it is better if the 1-core is significant.

For simplicity let's define $n_r = n - n_1$ and $m_r = m - n_1$, as the number of vertices and edges in the remaining part of the graph. The number of edges removed with the 1-core equals the number of vertices in the 1-core. This is due to a tree having

exactly one less edge than the number of vertices, but each of these trees has to be connected to the rest of the graph.

Finding of the 1-core has a complexity of $\mathcal{O}(n+n_1)$, n because of the initial pass through and the rest is the loops over the vertices of the 1-core. The complexity of the sum of distances depends on how many individual trees are in the 1-core, in the worst case all of 1-core would be in one big tree and the complexity would be $\mathcal{O}(n \cdot n_r + n_1^2)$. It would be unusual, but the complexity is still better. And then pair-wise distances, for the main component $\mathcal{O}(n_r m_r + n_r^2 \log n_r)$ and then for individual trees, in the worst case $\mathcal{O}(n_1^2 + n_1^2 \log n_1)$. So the final complexity in the worst case would be $\mathcal{O}(n_r^2 \log n_r + n_1^2(2+\log n_1) + n_r m_r + n(1+n_r) + n_1)$.

That is the worst-case scenario, in real graphs the 1-core will not be one big tree, it will be many small ones, actually, most of them would be single vertices. That adds a little more improvement to the complexity. Let's say that t is the average tree size, then the complexity of the sum is $\mathcal{O}(n \cdot n_r + n_1 t)$, and the complexity of finding pair-wise distances in the trees is $\mathcal{O}(|T| \cdot t^2 \log t)$. In the four test graphs used later in the chapter, the average tree size is between 1 and 1.2, which makes the complexity better, but the most time-complex part is still the complexity of Johnson's algorithm on the remaining part of the graph. In other graphs I have examined most of the 1-core is just single vertices, just as in the four test graphs.

Let's also consider for a moment the complexity when using multiple BFSs, originally with complexity $\mathcal{O}(n^2 + n(n+m))$. The complexity of the updated algorithm is $\mathcal{O}((n+n_1) + (n \cdot n_r + n_1^2) + (n_r(n_r+m_r)) + (n_1(n_1+n_1)))$ which equals to $\mathcal{O}(n_r(n_r+m_r) + n_1^2 + n(n_r+1) + n_1)$. Still an improvement, but relatively a smaller one compared to the improvement while using Johnson's.

This algorithm also has a better memory complexity, the original having a $\mathcal{O}(n^2)$ requirement, this improved has $\mathcal{O}(n_r^2 + n_1^2 + n_1)$. The algorithm allows for parallelisation.

4.1.4 2-chains simplification of a graph

After the 1-core is removed, there are still many vertices with degree two remaining in the rest of the graph, and this can include vertices with degree originally higher thanks to a connection to the 1-core tree. These can be used to further speed up the calculation.

Figure 4.2 contains illustration of the general idea. First a graph after removing 1-core vertices in figure 4.2a. Vertices with degree 2 are marked with a green border, other with a blue border. For example vertex m, a path to any other vertex must go through either a or c, and the distance to an other vertex x is $\min(d_{a,x}+w_{a,m}, d_{c,x}+w_{c,m})$. This works even for multiple linked vertices of degree 2, e.g. vertices n and o. For them the distance to other vertices is again either through a or c, but with the distance to an other vertex x is $\min(d_{a,x}+d_{a,m}, d_{c,x}+d_{c,m})$ instead. These linked vertices of degree 2 will be further referred to as *2-chains*.

Therefore the vertices in 2-chains can be removed from the graph and instead replaced by a new edge with weight being the sum of weights of edges being removed. In case there are multiple 2-chains between the same vertices, only one edge can be created with the minimum weight. The result of this reduction is shown in figure 4.3b, new edges are marked with green with their weights listed (with the weights of the original edges being all ones).

(a) The original graph

(b) The reduced graph

Figure 4.2: Illustration of a 2-chain simplification of a graph

This idea is inspired by a 2009 paper "Conservation of alternative paths as a method to simplify large networks" by Liu and Mondragón [15], which focused on the simplification of graphs while retaining the same number of alternative paths. This

does not preserve the number alternative paths (even shown in the example, one alternative path is lost in the reduction between a and c), but it does preserve distances between the remaining vertices and connectivity.

4.1.4.1 Definitions

Let's again define things precisely, on a graph that's already been simplified by removing the 1-core. The following specification works for undirected graphs, regardless of having weights. If the graph does not have weights, unit weights are assigned to each edge.

Let *2-chain* be a term for a set of connected vertices with degree two, and let V_i^2 denote that set, i being the chain identifier and C the set of chain identifiers. Let V^2 denote the set of vertices in all 2-chains in a graph and m_2 size of that set. The 2-chain always connects two vertices, lets call them *closest vertices*, so let's say that for a vertex v, one of them is *left*, denoted by p_v^l, and other is *right*, denoted by p_v^r. There are of course no directions in graphs, but let's use that term in this context. All vertices from a single 2-chain will have the same left and right closest vertices. For a vertex v, the distance to p_v^l and p_v^r is d_v^l and d_v^r respectively. A property of all vertices in a chain having the same left and right closest vertices, the distance of two vertices v and w from the same chain is $|d_v^l - d_w^l|$ or $|d_v^r - d_w^r|$ (only works on positive weights, but others are not considered in this thesis). Let c_v be the chain identifier of vertex v, and finally, let's use d_i^2 for the distance of the chain i from one closest vertex to the other.

4.1.4.2 Algorithm for the average path length calculation

Algorithm 4 shows the algorithm for summing of pair-wise distances using the 2-chain simplification. The distance from remaining vertices to vertices in 2-chains and vice-versa is the minimum of the path through one or the other closest vertex. The distance from a vertex from a 2-chain to another vertex in a 2-chain is the minimum of distances through both closest vertices, so four options, for vertices on the same 2-chain also the path within the 2-chain must be considered.

Algorithm 4 Summing pair-wise of distances after 2-chain reduction

$s = 0$
for all $i \in V \setminus V^2$ **do** ▷ Distances from remaining vertices
  for all $j \in V \setminus V^2$, $j \neq i$ **do** ▷ Distances to other remaining vertices
    $s \mathrel{+}= d_{i,j}$
  for all $j \in V^2$ **do** ▷ Distances to 2-chain vertices
    $p^l = p^l_j$, $p^r = p^r_j$
    $s \mathrel{+}= \min(d^l_j + d_{i,p^l},\ d^r_j + d_{i,p^r})$ ▷ Through one of two closest vertices
for all $i \in V^2$ **do** ▷ Distances from 2-chain vertices
  $p^l = p^l_i$, $p^r = p^r_i$
  for all $j \in V \setminus V^2$ **do** ▷ To remaining vertices
    $s \mathrel{+}= \min(d^l_i + d_{j,p^l},\ d^r_i + d_{j,p^r})$ ▷ Again through one or other
  for all $j \in V^2$, $j \neq i$ **do** ▷ To other 2-chain vertices
    $n^l = p^l_j$, $n^r = p^r_j$
    ▷ Distance through both closest vertices of both vertices
    $l = \min(d_{p^l,n^l} + d^l_i + d^l_j,\ d_{p^l,n^r} + d^l_i + d^r_j)$
    $r = \min(d_{p^r,n^l} + d^r_i + d^l_j,\ d_{p^r,n^r} + d^r_i + d^r_j)$
    if $c_i = c_j$ **then** ▷ On the same 2-chain
      $x = \min(l,\ r,\ |d^l_i - d^l_j|)$ ▷ Consider distance within chain
    else
      $x = \min(l,\ r)$
    $s \mathrel{+}= x$
return s

4.1.4.3 Detecting the 2-chains

Algorithm 5 shows detecting 2-chains and their properties. First, all vertices with degree 2 are added to the set V^2, and then the neighbouring vertices are traversed until all vertices in V^2 are not assigned to a specific chain. In the traversal in both directions, the closest vertices are also detected. Once the closest vertices are detected, each chain is traversed in order and the distance is calculated by adding weights of the edges to each vertex from one side, then the chain is traversed back and distances are set from the other side.

4.1.4.4 Complexity

The complexity of calculating the average path length, mainly the summing of all distances, is still $\mathcal{O}(n^2)$ just as in the naive approach. The summing is two nested

Algorithm 5 Detecting 2-chains, their closest vertices, distances to them, sorting into individually identified chains

function TRAVERSE(v, c) ▷ Traverses chain of vertices with degree 2
  $R = \{v\}$
  $p = v$ ▷ Previous in path
  while $\deg(c) = 2$ **do**
    add c to R
    $n = N(p) - \{p\}$ ▷ Next is the unvisited neighbour
    $p = c$, $c = n$
  return R, c ▷ Returns traversed vertices and vertex connected to it
V^2 = empty set
H = empty set ▷ Working copy of V^2
for all $v \in V$, $\deg(v) = 2$ **do**
  add v to V^2
copy V^2 to H
C = empty set, $i = 0$ ▷ Set of chain identifiers
while H is not empty **do**
  pop v from H
  R_0, p^l = TRAVERSE(v, $N(v)[0]$) ▷ Traverse in one direction
  R_1, p^r = TRAVERSE(v, $N(v)[1]$) ▷ Traverse in other direction
  for all $w \in R_0 \cup R_1$ **do** ▷ Process all found vertices on chain
    $p^l_w = p^l$, $p^r_w = p^r$
    $c_w = i$, push w into set V^2_i
    pop w from H
  add i to C, i += 1
for all $i \in C$ **do**
  $p^l = p^l_v$, $p^r = p^r_v$, $v \in V^2_i$ ▷ Any vertex from V^2_i
  D = empty list, O = empty list
  $p = p^l$, $d = 0$
  $n = v(\in N(p_l) \cap V^2_i)$
  while $n \neq p^r$ **do** ▷ Traverse chain from one direction setting distances
    d += $w_{n,p}$, $d^l_n = d$
    add d to D, n to O
    $m = x \in (N(n) - \{p\})$
    $p = n$, $n = m$
  d += $w_{n,p}$, $d^2_i = d$
  for all $n \in O$, $e \in D$ in reverse order **do** ▷ Traverse from other direction
    $d^r_n = d - e$

for loops over all vertices, with some extra operations, but asymptotically still $\mathcal{O}(n^2)$.

The detection of the 2-chains has time complexity of $\mathcal{O}(n+n_2)$, first the pass over all vertices and then several passes over all the chains to find the closest vertices and the distances.

The pair-wise distances have to be calculated on the rest of the graph, which has $n_r = n - n_2$ vertices, and $m_r = m - n_2$ edges. Each chain has one more edge than the number of vertices on it, and one edge has to be created for each chain, in the worst-case scenario, if there is exactly one chain between any two sets of closest vertices, if there are multiple the reduction in edges is bigger.

Since this simplification always creates a weighted graph, it is unsuitable for previously unweighted graphs to use this approach, because the multiple BFS algorithm cannot be used. The complexity of finding the pair-wise distance on the rest of the graph is then $\mathcal{O}(n_r m_r + n_r^2 \log n_r)$.

Overall complexity using just this approach is then $\mathcal{O}(n_r m_r + n_r^2 \log n_r + n^2 + n + n_2)$, compared to the original $\mathcal{O}(nm + n^2 \log n + n^2)$.

4.1.5 Combined algorithm

When the two simplifications are combined, all of the vertices with degree 1 or 2 and some other vertices depending on the graph can be removed from the graph for the purposes of calculating the pair-wise distance, the most significant part of the time complexity. However, the 2-chain simplification requires the graph to be converted to a weighted graph and Johnson's algorithm is needed for the pair-wise distance calculation on them. Therefore, for originally unweighted graphs, only 1-core simplification should be applied.

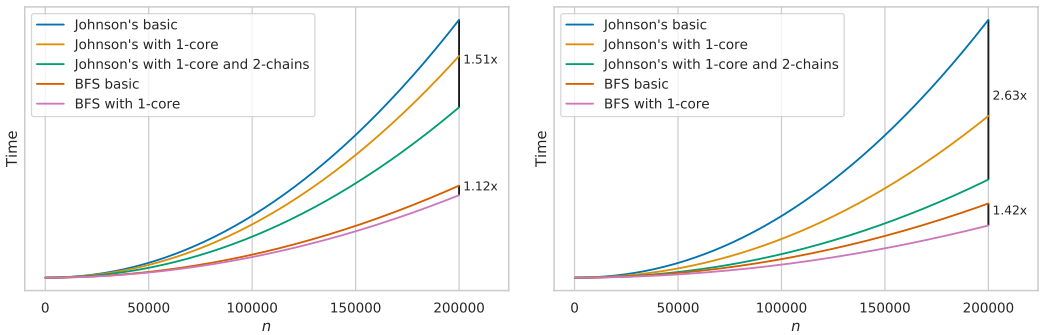
When combining the two simplifications, one must only consider the subgraph without the 1-core when detecting 2-chains, and in the sum implement a special case for 1-core trees which have their closest vertex on a 2-chain.

The time complexity for unweighted graphs is then the same as listed above, $\mathcal{O}(n_r(n_r+m_r)+n_1^2+n(n_r+1)+n_1)$.

The final complexity for weighted graphs requires further analysis. Let's recall, that the number of edges in the 1-core is n_1 and in 2-chains is n_2. Complexity of detecting the 1-core is $\mathcal{O}(n+n_1)$, complexity of detecting the 2-chains is $\mathcal{O}((n-n_1)+n_2)$. The complexity of the sum has the same complexity as the sum just using the 1-core, so $\mathcal{O}(n(n-n_1)+n_1^2)$. Let's define the sizes of remaining subgraph as $n_r = n-n_1-n_2$ and $m_r = m-n_1-n_2$. The pair-wise distances have to be calculated for the remaining vertices at complexity $\mathcal{O}(n_r m_r + n_r^2 \log n_r)$, and within the trees at $\mathcal{O}(n_1^2(1+\log n_1))$. The distance within 2-chains is calculated with the detection. So the final complexity is $\mathcal{O}(n_r m_r + n_r^2 \log n_r + n(n-n_1+2)+n_1^2(2+\log n_1)+n_2)$.

The space complexity is $\mathcal{O}(n_r^2+n_1^2+n_1+n_2)$.

The figure 4.3 shows the theoretical speedup of the algorithms on graphs with two slightly different properties. It is apparent that the average degree (and with that the number of edges), and percentage of vertices with degree one or two matters greatly. The figure also shows that the calculation for undirected graphs is quicker.



(a) For a graph with similar properties as `cond-mat-2003`, average degree 4.2, 1-core being 8% of vertices and 2-chains being 13%

(b) For a graph with similar properties as `ba-graph`, average degree 2.5, 1-core being 25% of vertices and 2-chains being 21%

Figure 4.3: Theoretical speed improvement in calculation of the APL. Vertical lines show the multiplicity of the original time complexity versus the improved.

4.2 Implementation

I implemented the improved algorithms as an extension to the `graph-tool` [16] package, an efficient package for working with graphs. It is made for Python, but most of it is implemented in C++, using the `boost` graph library [17], which makes it very fast and it allows for parallelisation of multiple algorithms. I chose it over other Python packages because of its speed and the parallelisation functionality (more about the speed in section 4.3.3). `Graph-tool` uses `OpenMP` [18] for parallelisation and I did as well in the implementation of my algorithms.

For pairwise distance `graph-tool` uses Johnson's algorithm in case of weighted graphs and the parallel BFS in case of unweighted.

I implemented the algorithms as an extension to `graph-tool`, meaning that once `graph-tool` is installed (in any way described in its documentation), one just needs to compile the extension using a provided Makefile. The extension, nicknamed `springbok`, comes in two parts, the Python interface, and the C++ implementation.

The Python interface contains two functions, one which provides the APL using the naive algorithm, described in section 4.1.1. The other function provides the APL using the improved algorithm, described in section 4.1.5.

Both of these functions use a combination of `graph-tool` functionality and the C++ implementation. The functionality used from `graph-tool` is mostly the pairwise distance calculations and creating subgraphs with vertices from the 1-core and 2-chains removed. The C++ implementation provides several functions as well, some parallelisable, for calculating the sum of distances and the average path length in the naive and improved ways, and detection of 1-core and 2-chains.

The code open-sourced under the MIT license [19] can be found in the repository of my proof of concept code for this thesis, link provided on page 2.

4.3 Experiments

To verify the correctness of the algorithm and the theoretical speedup I ran both the original algorithm and the improved algorithms on four test graphs. For each

graph I ran the variants for both unweighted and weighted graphs ten times, to get an average time both in actual time passed and CPU time. I ran the code on an Intel® Core™ i7-5500U CPU @ 2.40GHz, 4-core computer with 8GB memory. None of the graphs had weights associated with them, so I assigned unit weights to edges in the weighted experiment, to verify the correctness of the results against unweighted graphs as well.

4.3.1 Test graphs

Two of the test graphs are real-world and two are randomly generated. All four are undirected and unweighted, so for experiments on weighted graphs I assigned edges unit weights, so results of the average path length are comparable.

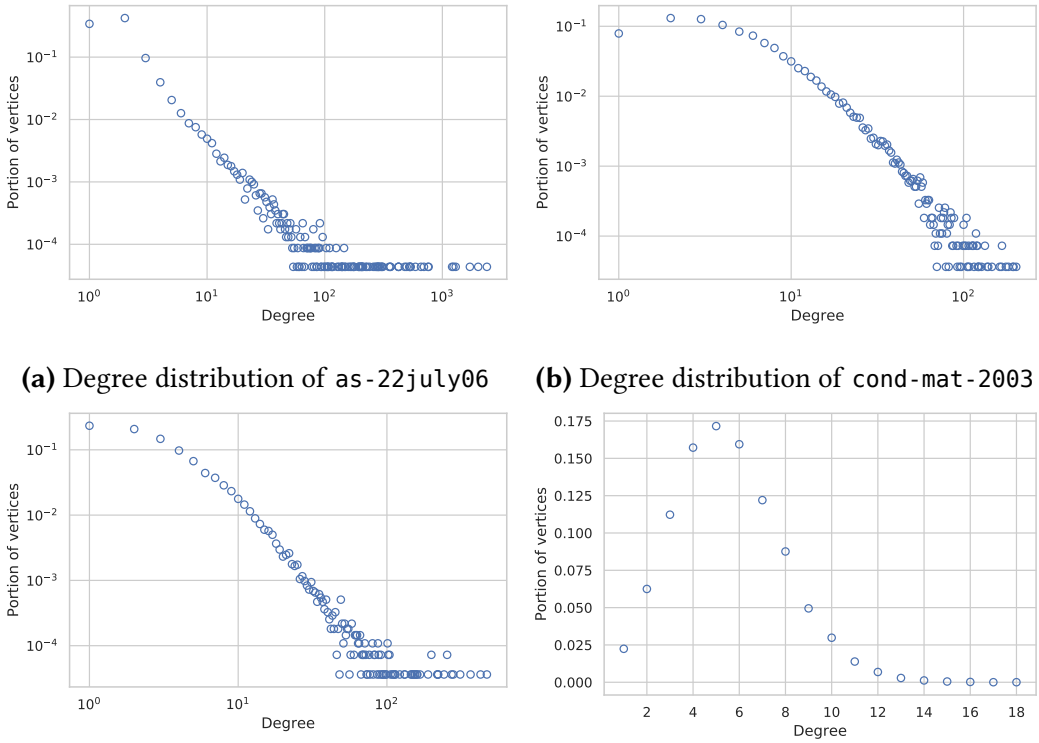
The first real-world network I used was `as-22july06`, a "*a symmetrised snapshot of the structure of the Internet at the level of Autonomous Systems (AS)*" from 2006 [20]. This is an example of a technological network. The largest connected component (LCC) contains 22963 vertices and 48436 edges.

The second real-world network I used was `cond-mat-2003`, a network of coauthorships between scientists posting preprints to a journal about condensed matter [21]. This is an example of a social network. The LCC contains 27519 vertices and 116181 edges.

The first randomly generated graph is a Barabási-Albert (BA) random graph [8], generated with m one to four in equal proportions, and then five iterations of a random rewire of all edges. The random rewire algorithm I used is such that the graph retains the degree sequence of the graph, implemented by `graph-tool` [16]. I will be referring to this graph as `ba-graph`. The LCC contains 27621 vertices and 69794 edges.

The second generated graph is a Erdős-Rényi (ER) random graph [22], with each vertex having a random degree between 1 and 10. I will be referring to this graph as `er-graph`. The LCC contains 27884 vertices and 77075 edges.

Figure 4.4 contains the degree distributions of the four graphs. All except the `er-graph` are scale-free, meaning they have a power-law degree distribution.



(a) Degree distribution of `as-22july06`

(b) Degree distribution of `cond-mat-2003`

(c) Degree distribution of `ba-graph`

(d) Degree distribution of `er-graph`

Figure 4.4: Degree distributions of the four test graphs

Table 4.1 contains the basic overview of the graphs. Only a small portion of vertices with degree higher than 2 were removed with the 1-core and 2-chains, 280 for `as-22july2006`, 169 for `cond-mat-2003`, 691 for `ba-graph` and 34 for `er-graph`.

Name	LCC n	LCC m	1-core	2-chains	Reduced n	Reduced m
`as-22july06`	22963	48436	7997	9823	5143	25759
`cond-mat-2003`	27519	116181	2290	3659	21570	109974
`ba-graph`	27621	69794	7077	5816	14728	56900
`er-graph`	27884	77075	637	1765	25482	74673

Table 4.1: Overview of sizes of the four test graphs

4.3.2 Speed improvement

Table 4.2 contains the average speedup using the improved algorithm, for unweighted and weighted variants, in both for real time and CPU time. The standard

deviation on the measured times was very small, so I did not include the ranges. For all of the graphs, both variants of the improved algorithm calculated the average path length correctly.

Name	Speedup (real)	Speedup (user)	Weighted (real)	Weighted (user)
`as-22july06`	2.18×	2.21×	8.60×	3.32×
`cond-mat-2003`	1.17×	1.18×	1.48×	1.42×
`ba-graph`	1.75×	1.78×	2.97×	2.40×
`er-graph`	1.01×	1.01×	1.20×	1.18×

Table 4.2: Overview of the speedup on the test graphs. Real time refers to actual time passed, user time refers to CPU time.

The speedup approximately matches what was expected from the theoretical analysis of complexities. The user time (the CPU time), shows how much work was done in all threads compared to the real time since I implemented the algorithms to work in parallel. This allows us to see both how much work was actually done and how well the algorithm parallelises. The user time can also be seen as how much time would be necessary if only one core was available.

On unweighted graphs, we see pretty much the same decrease in time on both CPU and real time. This is due to the pair-wise distance calculation and the sum being already parallelised in the naive algorithm, so the real time improvement is not as good. To show things in perspective, for example for the `as-22july06` graph, on average, naive average path length calculation took 13 seconds in real time, with 50 seconds on the CPU, that decreased to 6 and 23 with the improved algorithm. The `er-graph` has practically no improvement compared to the naive algorithm, this is due to it having very small 1-core with proportion to its size. The improvement in `cond-mat-2003` was smaller than in the other two graphs because it had a smaller portion of vertices in the 1-core and a high number of edges overall.

On weighted graphs, the improvement is different. The improvement of CPU time is already a bit better than on unweighted graphs, but the real time improves significantly, thanks to a big portion of the complexity being moved to parallelised code.

For perspective, for the same graph, `as-22july06`, the naive average path length calculation took on average 88 seconds of real time and 89 seconds of CPU time. This confirms that the algorithm is not parallelised and the weighted pair-wise distance calculation has much worse time complexity. With the improved algorithm, the time drops to 10 seconds of real time and 26 seconds of CPU time. A good improvement of the overall work necessary, but an excellent improvement in parallelisation. The improvement on `ba-graph` is also very good, and a reasonable improvement on `cond-mat-2003` (228 seconds of real time down to 153 on average). The `er-graph` is not scale-free and therefore the improvement there is minimal.

In conclusion, the improved algorithm does speed up the calculation of APL, slightly on unweighted graphs and substantially on weighted graphs, with a much bigger improvement when used on multiple cores. The improved algorithm works well only on scale-free graphs and the size of the speedup depends on the size of 1-core and 2-chains, also on how much edges were in the graph.

4.3.3 Comparison to other tools

The package `graph-tool` is, to the extent of my knowledge, one of the fastest Python packages for analysing graphs, depending on the task it is used for. This is illustrated by the benchmarks comparison provided by the author of `graph-tool` in [23] and by further and more recent analysis by Timothy Lin in [24]. For this thesis, the most interesting part of the first comparison is a calculation of betweenness on a slightly larger graph than the test graphs here, betweenness having a similar complexity involving the topology of the graph. Graph-tool was about five times faster than `igraph` [25] and more than 200x times faster than `NetworkX` [26]. In the second comparison, while `networkkit` [27] beats `graph-tool` in detecting the k-core, but the shortest path length, the most important measure with regards to this thesis, is several times faster with `graph-tool`.

Based on this benchmarks I only chose to compare to `igraph` and `networkkit`, `NetworkX` is a pure Python tool and as such can not beat a C++ implementation.

Only unweighted graphs are supported in the calculation of the average path length with `igraph`. The comparison is shown in table 4.3. It shows that `graph-tool` uses more of CPU time to do the pair-wise distance calculation, but in terms of real time it is up to four times faster. This might be partly caused because `igraph` uses the triangle sum while my naive implementation does not, but even so it is faster in real time.

Name	`graph-tool` real	`graph-tool` user	`igraph` real	`igraph` user
`as-22july06`	12.97s	50.30s	39.60s	39.53s
`ba-graph`	24.52s	95.77s	69.71s	69.59s
`cond-mat-2003`	29.66s	115.98s	75.54s	75.40s
`er-graph`	32.55s	127.42s	78.82s	78.68s

Table 4.3: Comparison of the `igraph`'s APL algorithm with my naive algorithm using `graph-tool`. Times are in seconds and are average of 5 runs.

I could not successfully run the benchmark with `networkkit` on the same computer as I ran the other benchmarks. It crashed my computer a couple of times due to running out of memory and swap. This is curious since the average path length calculation ran without an issue with `igraph` and `graph-tool`. The package does not have a function to calculate the distance directly, so I think this might be because when retrieving the 2D matrix of distances a copy is returned instead of a reference, so two times the amount of memory is required. Since Timothy Lin compared the shortest path length quite recently with his benchmark in [24], I did not make further effort to get my comparison.

I tried to venture outside of Python, I wanted to compare with Gephi [28], however, Gephi also runs several centrality statistics and eccentricity distribution calculation, so the comparison would be unbalanced. Furthermore, I did not find a way to measure accurately how long it takes to run these calculations.

4.4 Further usage

The algorithm I devised in this chapter is generalizable to other problems than just the average path length. With some tweaks, it is, in fact, an algorithm to speed up any pair-wise distance problem on a graph. This means it could be used to get the

pair-wise distance matrix or for other usages involving distances from all vertices to all other. The main part that would have to change is the optimisation which calculates distances from vertices in the 1-core – the sum of the closest vertex is used as a base, instead one would have to loop over all vertices. This means a slight increase in complexity (both time and space), but the main improvement is reducing the complexity of Johnson's algorithm and that would remain. Similar principles could be used for example for speeding up betweenness on a graph as well.

Chapter 5

Approximation using sampling

While the algorithm I devised to speed up the calculation of the average path length (APL) makes improvements in the time complexity, for large graphs (and especially large temporal graphs) it still might be prohibitively slow. Any real-time application might not be usable with the exact value, as it can take hours to do those calculations. For those and other cases an approximation is required.

For some cases the exact value is not even needed, an accurate and fast approximation can be enough. For exploratory data analysis, one only needs an overview of the properties, not exact values. For comparing two graphs an approximation is enough to see the difference between graphs, the second or third decimal does not matter. Some graphs are just a sample of the underlying true graphs, and the exact value in the sampled graph would be just an approximation of the true APL.

In this chapter, I will, therefore, implement an approximation method on top of the algorithm I devised in the previous chapter. The approximation is based on *random vertex sampling* on the whole graph as introduced in "Distance Distribution and Average Shortest Path Length Estimation in Real-World Networks" by Ye et al. [1]. I devised the same method myself when I ran into trouble with calculating the average path length on even much larger graphs than those examined in this thesis, but when doing a literature review for this thesis I found this paper which had introduced the approximation in 2010.

The approximation works on sampling some vertices as sources and calculating distances just from sources to all other vertices and averaging the value. Apart from trying to reproduce the results of that paper I will also try to expand on it

with a couple of modifications in the sampling and will focus more on the number of sources needed to get an accurate approximation well.

5.1 Analysis

The paper by Ye at al. compares multiple different sampling methods to approximate the APL. Two are based on sampling either vertices or edges and creating a subgraph, one is a snowball method iteratively adding neighbours to a subgraph and the final one is sampling a set of *source* vertices and calculating distances from only those source vertices.

Out of these, the last one, the method the authors called *random vertex sampling*, works best for approximating the APL. The others have a large bias in over- or underapproximating the value. The choice was clear, I chose to implement that method, with some modifications and more experiments.

Since I already implemented an algorithm to speed up the calculation, it only makes sense to speed up this approximation with it as well. The implementation is described in the next section, what most matters is the selection of the source vertices. I eventually came up with three methods with relation to the algorithm devised in the previous chapter. Let p denote the parameter for the approximation, which stands for the portion of vertices to be sampled or the *sample size* as it will be referenced in the rest of this thesis. The parameter will have slightly different meanings in the methods.

Both methods **Reduced-Proportional (RP)** and **Reduced-Uniform (RU)** sample $\lceil p \cdot n_r \rceil$ vertices from the reduced graph (graph after removing of 1-core and 2-chains). Recall that n_r is the number of vertices in that reduced graph. Let's call those selected vertices *sampled* and let S denote the set. What differs in those two methods is the probability of selecting each vertex, however, duplicates are not allowed in either.

To calculate the approximation, we have calculate distances on the reduced graph from these vertices. However thanks to the improved algorithm, we can also count

the vertices in 1-core trees connected to vertices in S and if both closest vertices are in S also the 2-chain vertices, with just a couple of extra computations. Let's call those vertices together with vertices from S *counted* and let C denote the set.

The approximation then is expressed in equation 5.1.

$$\bar{l_S} = \frac{\sum_{i \in C, j \in V, i \neq j} d_{i,j}}{|C| \cdot (|V| - 1)} \tag{5.1}$$

Method **Reduced-Proportional (RP)** samples vertices from the reduced graph based on the number of vertices that would be included in C if it were selected. The motivation behind is to count as many vertices as possible with the least amount of computation necessary.

The probability of selecting each vertex v is expressed in equation 5.2. The vertex itself is always counted, and then if a 1-core tree is connected, all vertices from it can be also counted, as expressed by $n_{1,v}$. If the graph is weighted, 2-chain vertices can be counted if the other closest vertex is also in the sampled set, so I multiplied the count $n_{2,v}$ with a half, to take in account this relation. If the graph is not weighted, $n_{2,v}$ is zero.

$$p_v \propto 1 + n_{1,v} + 0.5 n_{2,v} \tag{5.2}$$

Method **Reduced-Uniform (RU)** samples vertices from the reduced graph uniformly. As will be shown in the sections to come, method RP does not perform well, introducing a significant bias. Therefore I tried to salvage the method of sampling on the reduced graph with a uniform sampling, which could remove the bias.

Method **Original-Uniform (OU)** samples $\lceil p \cdot n \rceil$ vertices from the original graph before any reduction with 1-core and 2-chains. The sampling is uniform. In this case, the sampled set S is identical to the counted set C, but pair-wise distances on the reduced graph only have to be performed on set D. Set D is a set which contains the closest vertices instead of vertices that are in the 1-core or 2-chains.

Again, as we will see later, method RU does not perform very well on weighted graphs – this is basically the original method introduced in the paper by Ye at al. However with the improved APL algorithm, from a certain number of vertices, the set D will be smaller than S and the calculation quicker.

5.1.1 Complexity

Recall, that the complexity of the APL calculation with the improved algorithm is $\mathcal{O}(n_r(n_r+m_r)+n_1^2+n(n_r+1)+n_1)$ for unweighted graphs, and $\mathcal{O}(n_r m_r + n_r^2 \log n_r + n(n-n_1+2)+n_1^2(2+\log n_1)+n_2)$ for weighted graphs using the combined algorithm.

The complexity of the pair-wise distance calculation on the reduced graph is $\mathcal{O}(n_r(n_r+m_r))$ for unweighted graphs and $\mathcal{O}(n_r m_r + n_r^2 \log n_r)$ on weighted graphs. The complexity of the sum of the distances is the same for both types of graphs, $\mathcal{O}(n \cdot n_r + n_1^2)$, just the notation is slightly different for weighted graphs. The complexities of those two parts are different for the approximation's complexity.

On unweighted graphs, instead of running n_r breadth-first searches (BFSs), only $|S|$ are needed ($|D|$ for method Original-Uniform (OU)), putting the complexity at $\mathcal{O}(|S|(n_r+m_r))$ On weighted graphs, instead of using Johnson's algorithm, $|S|$ (or $|D|$) separate Dijkstra's algorithms are required, putting the complexity at $\mathcal{O}(|S|(m_r+n_r \log n_r))$.

The sum of distances is modified to $\mathcal{O}(n \cdot |S| + n_1^2)$ for both types of graphs, this time with the same notation. For method OU the S is replaced with D.

One must of course also consider the complexity of the sampling. Since samples without replacement are required, the easiest solution is to shuffle the options with the Fisher-Yates shuffle, which has complexity $\mathcal{O}(n)$ [29], and then take elements from the shuffled list until the required count is needed. The complexity is then $\mathcal{O}(n_r)$ and $\mathcal{O}(n)$ for method OU.

5.2 Implementation

I implemented this approximation in the `springbok` package I wrote in the previous chapter.

The modification in the Python part was basic, just adding more arguments to the function calculating the APL and if those were set, sampling the set of source vertices. If the method was RU or RP, from the reduced vertices, if the method was OU from the original vertices. I used `numpy` [30] for this, its randomisation functionality. For method OU I then calculated the D set as well.

Then I only calculated the distances to all other vertices from S in the reduced graph (D with method OU), and passed those distances instead of the full matrix to the C++ extension, together with these sets. I modified the C++ code for approximation by only iterating over the appropriate vertices and to change the calculation of the average path length from the sum with the size of the C instead of V.

Only pair-wise distances on full graph and distances from a single source are implemented in `graph-tool`, so I implemented another little tool, which can run multiple single-source searches (BFS or Dijkstra's) in parallel, as a way to further parallelise the process. This means that for approximation the process is fully parallelisable in the two more time-sensitive parts, the pair-wise distances and the sum of the distances.

5.3 Experiments

To test the performance of individual methods for sampling the vertices I ran multiple experiments on both types of graphs and multiple weight distributions.

5.3.1 Test graphs

For testing of the approximation I used the same four graphs I used in the previous section, the two real-world graphs `as-22july06` and `cond-mat-2003`, and the two generated, `ba-graph` and `er-graph`.

For each of these graphs, I generated three different edge weights to test approximation in weighted graphs. First unit weights, where all weights are 1. Second, uniformly distributed weights, where each edge has a weight uniformly selected between 0.001 (to avoid zero weights) and 6. And then, normally distributed weights, where each edge has a weight sampled from a random distribution with mean 3 and standard deviation 1. If the values from the normal distribution went beyond the range of the uniform distribution, I sampled the edge weight again to keep the range same and to prevent non-positive weights.

In the rest of the chapter when I will reference to unit, normal or uniform weights, I will be referring to these distributions of the weights.

5.3.2 Experiments design

To test how well the various methods approximate, how they compare and what sample size is required to get a reasonable approximation, I ran multiple experiments. For each test graph, I ran the approximation multiple times for different sample sizes p and compared them to the true value calculated using the exact algorithm from the previous chapter.

I ran experiments with 53 different values of p, chosen on a logarithmic scale, to see how few samples can provide a good estimate. First nine values on each of the 10^{-4}, 10^{-3} and 10^{-2} scales. Then values increment from 0.1 to 0.2 by 0.01, the remaining values increment by 0.05 from 0.2 up to 0.95. For `er-graph` I reduced the number of p to 33, because the calculation for that graph is slower, even with my improvement (as it is not scale-free).

On unweighted graphs, I ran the experiments 40 times for each graph. On weighted graphs, since the calculation is slower, I only ran the experiment 20 times, but for three distributions of weights.

The experiments produced a lot of data, which I plotted and patterns can be seen from them. With the sizes of the test graph, the lowest p produce a set of a few vertices, meaning even bellow 10. While some methods did produce a result with

a mean close to the true value at the small sample sizes, the extremes were too large and would distract from patterns of larger sample sizes, so I did not include them in these plots.

While some conclusions can be drawn from the plots, to have some kind of quantitative results I decided to test the methods based on thresholds. Basically, what p is necessary to get a mean or maximum accuracy error under some threshold t.

I measured the error in percentages, so the values are comparable across graphs. If l is the true value of the APL and $\bar{l}$ is its approximation in a experiment, let's define error e in equation 5.3.

$$e = \frac{|l - \bar{l}|}{l} \cdot 100 \tag{5.3}$$

Let's also define $\max e$ to be the maximum error at sample size p in a specific experiment, $\min e$ the minimum error, $\bar{e}$ to be the mean error and σ_e the standard deviation.

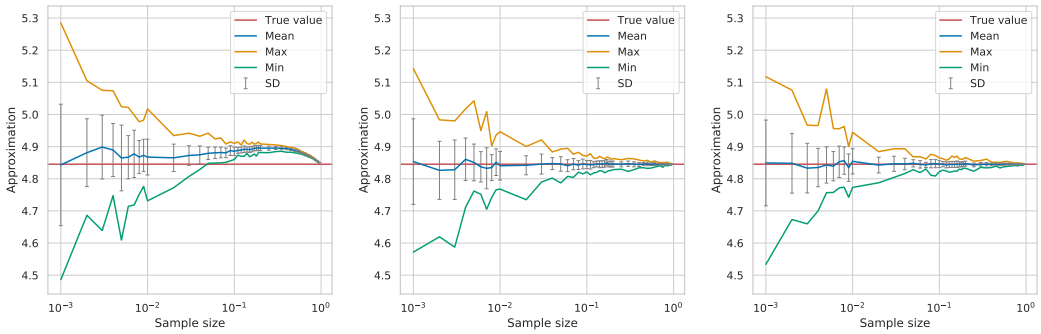
For the thresholds on the mean accuracy error, I chose t to be 5, 1, 0.5 and 0.1, to mimic the usual confidence levels in statistics. For the thresholds on the maximum accuracy error, I chose 10, 5 and 1. The unit of t is the same as the unit of e, percentages.

Together with the approximation value I also noted how many vertices were in sets C for methods RU and RP and D for method OU. I did this to see how efficiently they use the speedup of 1-core and 2-chains reduction.

5.3.3 Experiments results

The full experiment results can be seen plotted in appendix C. Since there is a lot of combinations of graphs, methods and types of weights, there is a lot of individual plots, so I am only going to include a couple of them here as examples on the '*neutral*' scale-free `ba-graph`. For the rest of the graphs, I will only include the quantitative results with thresholds.

Figure 5.1 shows the results of experiments on the unweighted `ba-graph` with the three methods. After 40 runs of the experiments, we can see that on unweighted graphs methods RU and OU are unbiased, the mean is very close to the true value even for very small p. The method RP is slightly biased, producing a value above the true value of the property. The standard deviation of the RP method is also larger than for the other two, which seem to have comparable standard deviations. The extremes of the approximation are also shown in the plots.



(a) Method RP **(b)** Method RU **(c)** Method OU

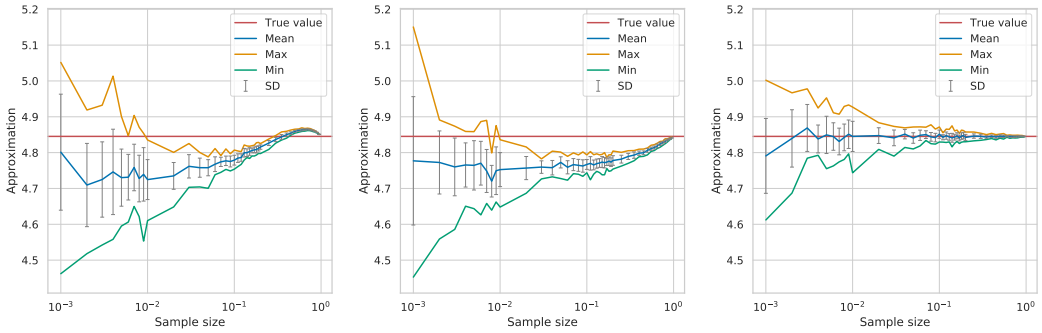
Figure 5.1: Approximation results on the unweighted `ba-graph`. Note the logarithmic scale on the X-axis.

Figure 5.2 shows the results of experiments on the `ba-graph` with unit weights. After 20 runs, the differences between the methods are more apparent here than in the unweighted variants. There is now a bias in both methods RU and RP, usually under approximating the value, but in RP for high values of p, it is also overapproximating the value. Method OU remains unbiased and its standard deviation and extremes are better compared to the other methods.

With other weight distributions for this graph and the three other graphs, the overall trend is very similar. For some graphs and some weights the direction and size of the bias are different with methods RP and RU, but the method OU is always unbiased. This can be seen in the figures in appendix C.

5.3.3.1 Quantitative threshold results

From the plots in the previous section, we can already see that method OU performs the best both on unweighted and weighted graphs. So here in this section, I



(a) Method RP **(b)** Method RU **(c)** Method OU

Figure 5.2: Approximation results on the weighted `ba-graph` with unit weights

will only include its threshold results, but the results for all methods are included in the appendix D. Furthermore, as can be seen from the results in appendix D, the thresholds for unit weights are practically identical to unweighted graphs, so I omitted those results from this section as well. That is because the distances are the same and method OU is not dependent on the effect of the reduction.

Table 5.1 shows the threshold results on mean error for method OU on unweighted graphs and weighted graphs with normal and uniform weights. On unweighted graphs, we can see that the 5% threshold can be reached with p on the 10^{-4} scale and then each further threshold is reached with an increase of an order of magnitude of p. The `er-graph` is an exception to that, about an order of magnitude smaller p is required for the thresholds.

On normally distributed weights, the performance of the approximation drops in all graphs, but not significantly for most thresholds. The only threshold where the increase is significant is 0.1%, where it raises the sample size up to 0.5, half of the whole graph has to be used as a source.

On uniformly distributed weights the performance drops again compared to normal weights. For thresholds higher than or equal to 0.5% the approximation is still a viable option that will make the calculation quicker, but for 0.1% p raises up to 0.85, practically using the whole graph.

Graph	t	UW p	UW $\bar{e}$	WN p	WN $\bar{e}$	WU p	WU $\bar{e}$
as-22july06	5%	0.0004	3.521%	0.0004	3.507%	0.002	4.351%
as-22july06	1%	0.007	0.912%	0.008	0.783%	0.04	0.863%
as-22july06	0.5%	0.02	0.438%	0.04	0.385%	0.14	0.423%
as-22july06	0.1%	0.35	0.086%	0.5	0.094%	0.85	0.061%
cond-mat-2003	5%	0.0002	3.837%	0.0002	4.275%	0.0008	4.763%
cond-mat-2003	1%	0.005	0.875%	0.008	0.873%	0.03	0.977%
cond-mat-2003	0.5%	0.02	0.452%	0.03	0.480%	0.13	0.377%
cond-mat-2003	0.1%	0.3	0.098%	0.5	0.059%	0.75	0.079%
ba-graph	5%	0.0002	4.027%	0.0003	4.132%	0.0008	4.463%
ba-graph	1%	0.005	0.860%	0.006	0.999%	0.03	0.840%
ba-graph	0.5%	0.02	0.408%	0.02	0.382%	0.08	0.474%
ba-graph	0.1%	0.3	0.092%	0.35	0.090%	0.75	0.075%
er-graph	5%	0.0001	2.526%	0.0001	4.957%	0.0003	2.330%
er-graph	1%	0.0006	0.824%	0.001	0.897%	0.006	0.506%
er-graph	0.5%	0.004	0.303%	0.004	0.468%	0.01	0.471%
er-graph	0.1%	0.07	0.093%	0.13	0.078%	0.3	0.088%

Table 5.1: Threshold results with method OU on unweighted graphs (UW), weighted graphs with normal weights (WN) and uniform weights (WU)

Table 5.1 shows the threshold results on the maximum error. We can again see a general rule that each threshold is reached with an increase in the magnitude of p and that maximum 1% error is reached with p on the 10^{-2} scale, single percentages of the graph used as sources. As in the mean error, the approximation is not as performant on the uniformly distributed weights, requiring almost an order of magnitude larger p than other weight distributions, however, $p = 0.2$ still always results in an error less than 1%. I will try to explore why the approximation performs worse on uniform weights in section 5.4.2.

5.3.3.2 Numbers of vertices sampled or counted

As mentioned in the section about the design of experiments, I was also tracking how many vertices were sampled, counted with methods RU and RP and for how many vertices the distances had to be calculated in method OU.

Figure 5.3 contains the results on unweighted graphs, averages of 40 runs. The size of C for method RU is almost identical to the size of S for method OU, so it

Graph	t	UW p	UW max e	WN p	WN max e	WU p	WU max e
as-22july06	10%	0.0009	6.580%	0.0006	8.075%	0.002	9.626%
as-22july06	5%	0.003	2.676%	0.005	3.460%	0.01	3.514%
as-22july06	1%	0.05	0.985%	0.07	0.980%	0.2	0.855%
cond-mat-2003	10%	0.0004	7.634%	0.0009	9.298%	0.003	8.227%
cond-mat-2003	5%	0.002	4.301%	0.003	3.223%	0.008	2.911%
cond-mat-2003	1%	0.07	0.834%	0.09	0.640%	0.2	0.662%
ba-graph	10%	0.0007	7.605%	0.0004	9.133%	0.002	8.539%
ba-graph	5%	0.002	4.756%	0.002	3.423%	0.005	4.825%
ba-graph	1%	0.03	0.989%	0.03	0.782%	0.14	0.777%
er-graph	10%	0.0001	8.242%	0.0003	5.363%	0.0003	7.172%
er-graph	5%	0.0003	4.921%	0.0004	3.580%	0.003	3.106%
er-graph	1%	0.006	0.985%	0.03	0.376%	0.03	0.790%

Table 5.2: Threshold results with method OU on unweighted graphs (UW), weighted graphs with normal weights (WN) and uniform weights (WU)

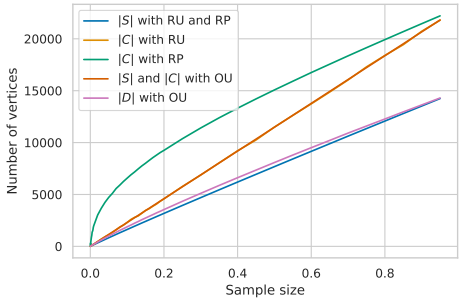
is practically invisible. The key finding is that the size of D in method OU grows similarly to the size of S in the other methods, so time-wise they should perform similarly, as the same number of distances need to be calculated for all methods.

For each graph we can also see how many of the vertices are in the 1-core or 2-chains in the differences between the size of S and C for methods RU and RP. For example, er-graph is not scale-free and therefore the difference is small.

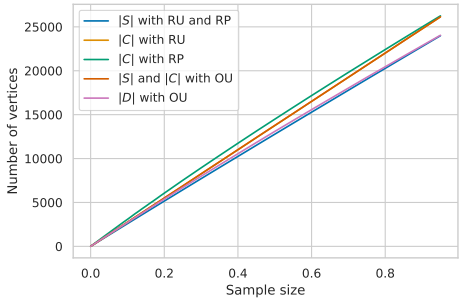
Figure 5.4 contains the results on weighted graphs, combined for all distributions of weights. Here the differences between methods are even more apparent, especially on as-22july06. The 1-core and 2-chains vertices are a big part of that graph, which results in a very quick increase of counted vertices in method RP, as a few vertices can provide results for many vertices. The difference in the size of D in OU and S in RU/RP is larger on weighted graphs, but not as significant to be worried about, especially since it provides such a stronger approximation.

5.3.4 Speed of the approximation

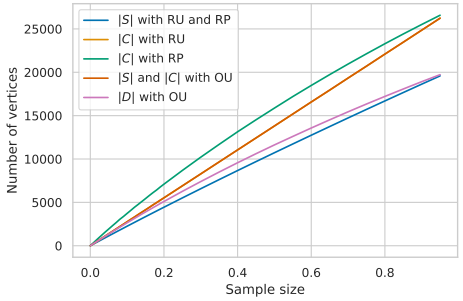
Figure 5.5 contains how much faster is the approximation to calculating the full exact value, for all test graphs and method. The comparison is comparing 10 runs of the approximation at each graph and method to the mean time from section 4.3.2.



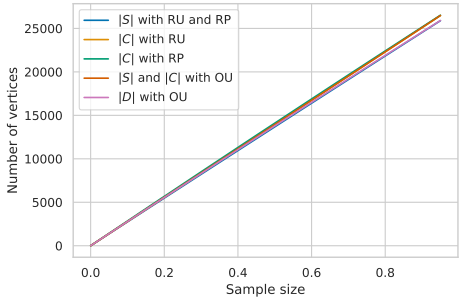
(a) For graph `as-22july06`



(b) For graph `cond-mat-2003`

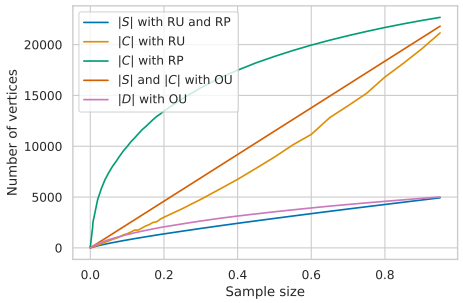


(c) For graph `ba-graph`

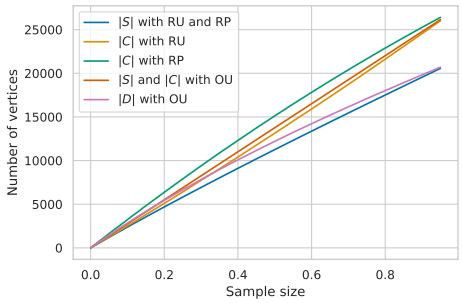


(d) For graph `er-graph`

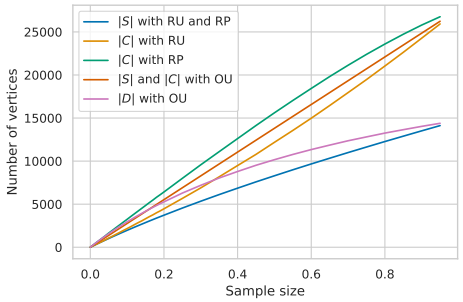
Figure 5.3: Number of vertices counted and sampled for the three methods with respect to the sample size on unweighted graphs



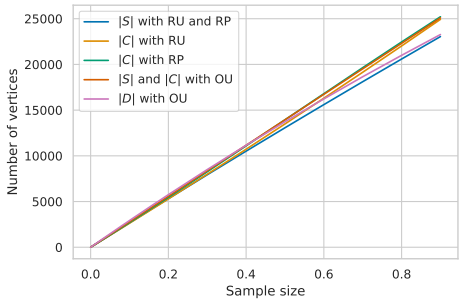
(a) For graph `as-22july06`



(b) For graph `cond-mat-2003`



(c) For graph `ba-graph`

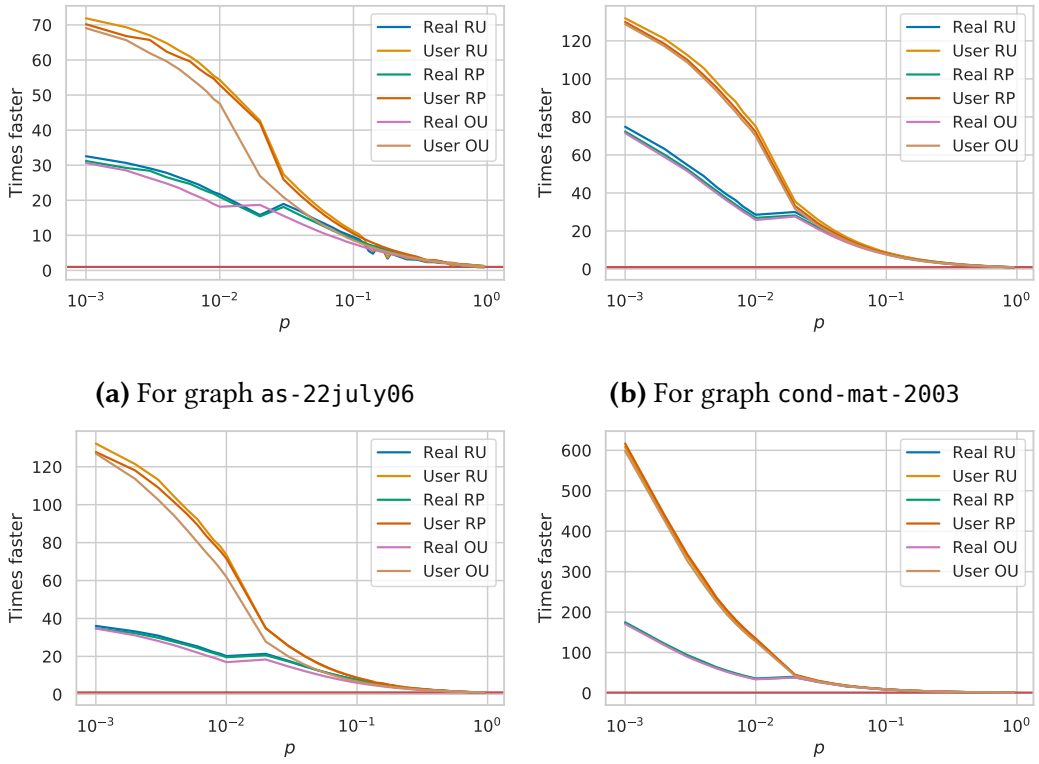


(d) For graph `er-graph`

Figure 5.4: Number of vertices counted and sampled for the three methods with respect to the sample size on weighted graphs

The main takeaway is that the speed of all the three methods is comparable. Method OU is a little bit slower than the other methods, that is due to D growing faster with p than S in the other methods.

There is a interesting uptick in the real time improvement around $p = 10^{-2}$. This is due to the implementation of parallelisation using `OpemMP`, there is some overhead involved with it, and therefore it is only used from a certain threshold in the number of vertices, that threshold being 300 (default in `graph-tool`). That makes the code run faster in the real time from that point forward.



(a) For graph `as-22july06`

(b) For graph `cond-mat-2003`

(c) For graph `ba-graph`

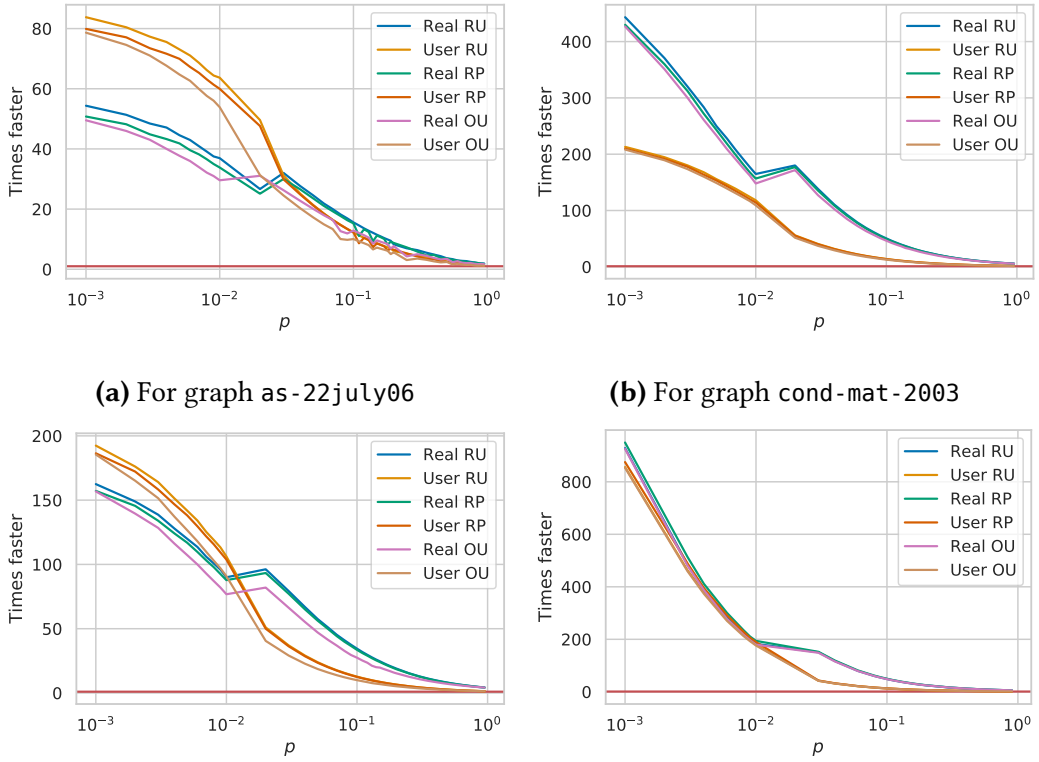
(d) For graph `er-graph`

Figure 5.5: Comparison of time needed to calculate an approximation compared to calculating the exact value on unweighted graphs

Figure 5.6 contains the results for weighted graphs. The patterns are similar to that of unweighted graphs, just on a bigger magnitude. Calculating the APL on weighted graphs is slower in general, so the differences will be bigger.

One main difference is that real time has a much bigger improvement on weighted graphs compared to unweighted. That is due to the original exact algorithm is

running Johnson's algorithm, which is not parallelisable, but this approximation is using parallel Dijkstra's algorithms, which uses the computer more efficiently.



(a) For graph `as-22july06`

(b) For graph `cond-mat-2003`

(c) For graph `ba-graph`

(d) For graph `er-graph`

Figure 5.6: Comparison of time needed to calculate an approximation compared to calculating the exact value on weighted graphs

5.4 Interpretation of results

In previous sections I already concluded that method OU is the best for approximating the APL – it is unbiased and its accuracy improves quickly with growing sample size on all graphs with all weight distributions. Method RU also shows these properties, but only on unweighted graphs.

Now since the method selection is clear, the selection of p. That selection, of course, depends on what the priorities are and there are trade-offs involved. Selecting a smaller p means a less accurate approximation but a faster calculation. Higher p provides a more accurate approximation at the cost of more time. The tables in section 5.3.3.1 and figures in section 5.3.4 can help with this trade-off.

To show more of a unified comparison, I put together some overview tables with the full accuracy statistics (mean, standard deviation, minimum and maximum) and the time improvement against the calculation of the exact value. The tables 5.3, 5.4 and 5.5 show the results for three specific p values – 0.006, 0.06 and 0.3. Each table contains results for one type of weights – no weights, normal weights and uniform weights, unit weights are omitted as the results are similar to no weights at all.

One thing of note is that the improvement against the calculation of the exact value is using the results of the improved algorithm, meaning that if one would compare to the naive algorithm, the improvement would be multiplied by the results in section 4.3.2. We can also see the improvement is not linear with respect to p, that is of course because the complexity of the calculation is not linear and there are some computations which must be done regardless of p – the reduction of the graph.

Graph	p	$\bar{e}$	σ_e	$\min e$	$\max e$	Faster (real)	Faster (user)
as-22july06	0.006	1.023%	0.897%	0.015%	3.497%	22.05×	54.92×
as-22july06	0.06	0.292%	0.230%	0.018%	0.885%	10.56×	12.78×
as-22july06	0.3	0.119%	0.091%	0.014%	0.342%	3.29×	3.43×
cond-mat-2003	0.006	0.786%	0.569%	0.004%	2.134%	36.02×	87.66×
cond-mat-2003	0.06	0.261%	0.211%	0.001%	1.102%	11.95×	12.48×
cond-mat-2003	0.3	0.098%	0.061%	0.006%	0.258%	2.71×	2.73×
ba-graph	0.006	0.738%	0.586%	0.016%	2.289%	21.98×	80.22×
ba-graph	0.06	0.184%	0.146%	0.008%	0.543%	9.12×	10.85×
ba-graph	0.3	0.092%	0.064%	0.000%	0.239%	2.38×	2.47×
er-graph	0.006	0.311%	0.228%	0.007%	0.985%	52.22×	194.35×
er-graph	0.06	0.104%	0.075%	0.013%	0.289%	14.16×	14.57×
er-graph	0.3	0.031%	0.028%	0.002%	0.129%	2.93×	2.96×

Table 5.3: Accuracy and speed of approximation for specific sample sizes for unweighted graphs. Time is compared to the calculation of the exact value.

5.4.1 Differences between methods

The differences between methods come from how the graph is reduced by removing the 1-core and 2-chains and how they are counted in the approximation. As it turns out, when sampling on the reduced graph and counting the extra vertices

Graph	p	$\bar{e}$	σ_e	$\min e$	$\max e$	Faster (real)	Faster (user)
as-22july06	0.006	1.366%	0.952%	0.168%	3.212%	35.97×	62.65×
as-22july06	0.06	0.474%	0.341%	0.090%	1.326%	17.96×	14.95×
as-22july06	0.3	0.132%	0.102%	0.003%	0.383%	5.00×	3.55×
cond-mat-2003	0.006	0.999%	0.703%	0.073%	2.667%	209.49×	139.16×
cond-mat-2003	0.06	0.450%	0.304%	0.058%	1.204%	72.57×	20.33×
cond-mat-2003	0.3	0.131%	0.112%	0.000%	0.374%	16.46×	4.45×
ba-graph	0.006	0.999%	0.571%	0.107%	2.103%	99.97×	117.96×
ba-graph	0.06	0.283%	0.193%	0.034%	0.850%	41.32×	15.91×
ba-graph	0.3	0.103%	0.102%	0.001%	0.382%	10.75×	3.61×
er-graph	0.006	0.387%	0.280%	0.075%	0.986%	277.60×	270.33×
er-graph	0.06	0.111%	0.089%	0.001%	0.321%	79.05×	21.25×
er-graph	0.3	0.043%	0.034%	0.003%	0.121%	16.35×	4.32×

Table 5.4: Accuracy and speed of approximation for specific sample sizes for graphs with normal weights. Time is compared to the calculation of the exact value.

Graph	p	$\bar{e}$	σ_e	$\min e$	$\max e$
as-22july06	0.006	2.279%	1.226%	0.231%	4.399%
as-22july06	0.06	0.869%	0.674%	0.074%	2.063%
as-22july06	0.3	0.186%	0.145%	0.022%	0.646%
cond-mat-2003	0.006	2.372%	1.451%	0.545%	6.171%
cond-mat-2003	0.06	0.510%	0.353%	0.048%	1.354%
cond-mat-2003	0.3	0.195%	0.129%	0.014%	0.430%
ba-graph	0.006	1.417%	1.223%	0.084%	3.800%
ba-graph	0.06	0.536%	0.364%	0.003%	1.287%
ba-graph	0.3	0.248%	0.176%	0.006%	0.639%
er-graph	0.006	0.506%	0.486%	0.006%	1.806%
er-graph	0.06	0.342%	0.183%	0.009%	0.711%
er-graph	0.3	0.088%	0.087%	0.010%	0.362%

Table 5.5: Accuracy and speed of approximation for specific sample sizes for graphs with uniform weights. Faster columns omitted here to prevent duplicity.

automatically, a bias is introduced. If there are many 1-core vertices in the graph, the paths from those vertices to all other vertices are slightly longer on average, resulting in an over-approximation. The paths from 2-chain vertices are on the other hand shorter than average, resulting in under-approximation.

Method RP suffers from this even further, because vertices which introduce the bias are selected with a higher probability. The original hypothesis was to count as many vertices as possible with as little computation needed, but from the results, we can see that the result is biased and the difference in speed is not that great to be concerned with.

5.4.2 Differences between weight distributions

Comparing weighted and unweighted graphs, it is not surprising that the results for unweighted graphs and unit weights are practically the same, they do have the same distance distribution, and the sampling on the original graph removes any differences in how the value is calculated. The distribution of the weights seems to differ, the required p is bigger for uniform weights than for normal weights. In figure 5.7 I plotted the distribution of all the paths within each graph with the weights I generated.

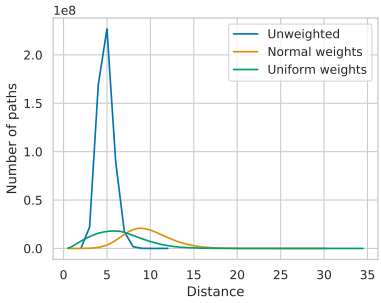
The obvious difference between the uniform and normal weights is that the distances are shifted slightly, which makes the resulting APL different. But I do not think that is the main reason behind the worse performance of the approximation on uniform weights. I took a look at the standard deviation and the excess kurtosis of the distributions, shown in tables 5.6 and 5.7.

The standard deviation for unweighted graphs is very small, which is maybe one of the contributing factors why the approximation of unweighted graphs is easier – a smaller p is required. The differences in the standard deviation between the two weight distributions go both ways, there are no consistent differences.

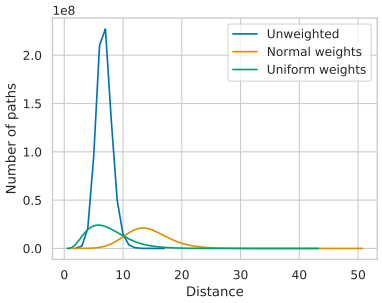
But in excess kurtosis, the difference is consistent across the scale-free graphs – normal weights have a higher kurtosis compared to unweighted graphs and uniform weights have a higher kurtosis compared to normal. The absolute values do

not matter, only the comparison between types of weights. Higher excess kurtosis means more outlier values, meaning that from the data we see for uniform weights have more outliers and it is, therefore, harder to approximate the mean, because the outliers affect it. Again, `er-graph` is an exception, the p does not increase dramatically for uniform weights and here we see that the kurtosis is not too different.

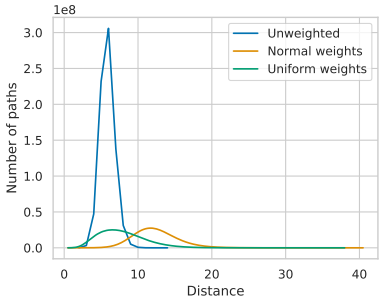
I am not saying that is the definite reason, but it is sort of an educated guess. To actually confirm that more experiments would be required, first of all, generating multiple edge weights in the same distribution, to see how much the result differs if it is not just the result of these specific weights. Then I think it would be wise to also test with different parameters of the distributions, to see how much the select ones matter.



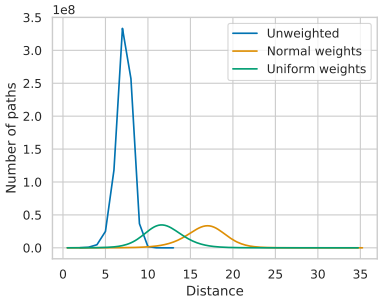
(a) For graph `as-22july06`



(b) For graph `cond-mat-2003`



(c) For graph `ba-graph`



(d) For graph `er-graph`

Figure 5.7: Distance distributions of the four test graphs

Graph	Unweighted	Normal weights	Uniform weights
as-22july06	0.802	7.014	8.430
cond-mat-2003	1.757	14.369	11.696
ba-graph	0.989	8.747	9.925
er-graph	0.860	6.594	5.895

Table 5.6: Standard deviation of the distance distributions

Graph	Unweighted	Normal weights	Uniform weights
as-22july06	0.248	0.422	0.533
cond-mat-2003	0.472	0.690	1.322
ba-graph	0.482	0.788	0.851
er-graph	0.993	1.018	0.834

Table 5.7: Excess kurtosis of the distance distributions

Chapter 6

Conclusion

This thesis set out two goals – to speed up the exact calculation of the average path length (APL) and then use that algorithm to provide an even quicker and accurate approximation. I think I succeeded in both.

I used two tricks to speed up the calculation, both based on reducing the number of vertices and edges in the most expensive part of the calculation, the pair-wise distance calculation within a graph, and calculating the distances to those vertices in alternative ways. The first trick which works on all graphs is removing the 1-core from the graph – the path from a vertex in a 1-core always goes through one specific vertex and the distance from that vertex can be used. The second trick only works on weighted graphs is removing vertices which have a degree two – the paths from those vertices always go through either one or the other connected vertex.

Both of these tricks reduce the size of the graph for the expensive pair-wise distance calculation significantly mainly on scale-free graphs, which is the class of most real-world graphs. These graphs follow a power-law degree distribution, meaning there is a high portion of vertices in the graph which have a very low degree. The improved algorithm has both a better computational complexity and better space complexity.

After implementing the algorithm in C++ with Python bindings I tested the algorithm on four large graphs, a graph of Autonomous Systems (AS), a citation network, a Barabási-Albert (BA) and Erdős-Rényi (ER) random graphs, ranging in size from 22k to 30k vertices. When comparing the time needed to calculate the APL I found that the improved algorithm sped up the calculation up to 2.2 times,

with the specific time depending on the size of the 1-core and the overall number of vertices and edges in the graph.

When considering weighted graphs, the improved algorithm sped up the calculation up to 8.6 times when real time was measured and 3.3 times when CPU time was measured. Apart from reducing the size of a graph for pair-wise distance calculation my improved algorithm also enables more of parallelisation.

The algorithm I devised can be generalised to be applied to any pair-wise distance problem, but in this thesis, it does focus on the average path length. It can be used to speed up the creation of the 2D distance matrix for graphs and therefore all problems that depend on it. It could even be used for example for calculation of betweenness, where the number of paths is important as well, with some modifications. The core principle is that the paths from some vertices are trivial with respect to some other vertex and the whole graph does not have to be explored to have the same information.

In the second part of the thesis, I implemented an approximation on top of the improved algorithm. The approximation selects several vertices and only counts distances from those vertices instead from all. This method was already published in 2010 and I mostly experimented with new methods of selecting the vertices, with relation to the improved algorithm. I was hoping that one of the methods I would propose would do better, but they introduced a bias to the approximation instead. So while the idea behind the method remains unchanged in this thesis I further sped it up with my improved APL algorithm, which improves both the computational complexity and space complexity.

I used the same graphs and generated multiple weights distributions and did more analysis on the sample sizes and resulting accuracy and I measured how much quicker it is compared to the calculation of the exact value. The speedup and accuracy of the method are satisfactory, even with very small sample sizes the maximum error rarely exceeds 5% and it is multiple times to hundreds of times faster. The speed and the resulting accuracy, of course, form a trade-off, the big-

ger the sample size the more accurate the approximation is while taking more time to calculate.

To summarise my contributions, I devised and implemented a faster exact calculation of the average path length on scale-free graphs with moderate to good improvements in speed. Then I reproduced results of a paper on approximations of the average path length using sampling, improved its speed, experimented with more methods of the sampling, and mainly experimented with more different sampling sizes and measured both the accuracy and time improvement, to provide a trade-off based on priorities placed on the approximation.

Appendix A

Bibliography

[1] Qi Ye, Bin Wu, and Bai Wang. "Distance distribution and average shortest path length estimation in real-world networks". In: *International Conference on Advanced Data Mining and Applications*. Springer. 2010, pp. 322–333.

[2] Guoyong Mao and Ning Zhang. "Fast approximation of average shortest path length of directed BA networks". In: *Physica A: Statistical Mechanics and its Applications* 466 (2017), pp. 243–248.

[3] Michalis Potamias et al. "Fast shortest path distance estimation in large networks". In: *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM. 2009, pp. 867–876.

[4] Andrey Gubichev et al. "Fast and accurate estimation of shortest paths in large graphs". In: *Proceedings of the 19th ACM international conference on Information and knowledge management*. ACM. 2010, pp. 499–508.

[5] Frank Takes and Walter Kosters. "Computing the eccentricity distribution of large graphs". In: *Algorithms* 6.1 (2013), pp. 100–118.

[6] Christopher R Palmer, Phillip B Gibbons, and Christos Faloutsos. "ANF: A fast and scalable tool for data mining in massive graphs". In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2002, pp. 81–90.

[7] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. "HyperANF: Approximating the neighbourhood function of very large graphs on a budget". In: *Proceedings of the 20th international conference on World wide web*. ACM. 2011, pp. 625–634.

[8] Albert-László Barabási and Réka Albert. "Emergence of scaling in random networks". In: *science* 286.5439 (1999), pp. 509–512.

[9] Donald B Johnson. "Efficient algorithms for shortest paths in sparse networks". In: *Journal of the ACM (JACM)* 24.1 (1977), pp. 1–13.

[10] Robert W Floyd. "Algorithm 97: shortest path". In: *Communications of the ACM* 5.6 (1962), p. 345.

[11] Matthias Scholz. "Node similarity as a basic principle behind connectivity in complex networks". In: *arXiv preprint arXiv:1010.0803* (2010).

[12] Edward F Moore. "The shortest path through a maze". In: *Proc. Int. Symp. Switching Theory, 1959*. 1959, pp. 285–292.

[13] Stephen B Seidman. "Network structure and minimum degree". In: *Social networks* 5.3 (1983), pp. 269–287.

[14] Vladimir Batagelj and Matjaž Zaveršnik. "Fast algorithms for determining (generalized) core groups in social networks". In: *Advances in Data Analysis and Classification* 5.2 (2011), pp. 129–145.

[15] Ling Liu and Raúl J Mondragón. "Conservation of alternative paths as a method to simplify large networks". In: *Proceedings of the 1st Annual Workshop on Simplifying Complex Network for Practitioners*. ACM. 2009, p. 1.

[16] Tiago P. Peixoto. "The graph-tool python library". In: *figshare* (2014). DOI: 10.6084/m9.figshare.1164194. URL: http://figshare.com/articles/graph_tool/1164194 (visited on 09/10/2014).

[17] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *The boost graph library: user guide and reference manual*. Addison-Wesley, 2002.

[18] Leonardo Dagum and Ramesh Menon. "OpenMP: An industry-standard API for shared-memory programming". In: *Computing in Science & Engineering* 1 (1998), pp. 46–55.

[19] Open Source Initiative. *The MIT License (MIT)*. URL: https://opensource.org/licenses/MIT (visited on 08/31/2019).

[20] Mark EJ Newman. *Network data*. URL: http://www-personal.umich.edu/~mejn/netdata/ (visited on 07/30/2019).

[21] Mark EJ Newman. "The structure of scientific collaboration networks". In: *Proceedings of the national academy of sciences* 98.2 (2001), pp. 404–409.

[22] Paul Erdős and Alfréd Rényi. "On Random Graphs I". In: *Publicationes Mathematicae Debrecen* 6 (1959), pp. 290–297.

[23] Tiago de Paula Peixoto. *Performance Comparison - graph-tool: Efficent network analysis with python*. URL: https://graph-tool.skewed.de/performance (visited on 07/31/2019).

[24] Timothy Lin. *Benchmark of popular graph/network packages*. URL: https://www.timlrx.com/2019/05/05/benchmark-of-popular-graph-network-packages/ (visited on 08/20/2019).

[25] Gabor Csardi and Tamas Nepusz. "The igraph software package for complex network research". In: *InterJournal* Complex Systems (2006), p. 1695. URL: http://igraph.org (visited on 07/31/2019).

[26] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[27] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. "NetworKit: A tool suite for large-scale complex network analysis". In: *Network Science* 4.4 (2016), pp. 508–530.

[28] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. *Gephi: An Open Source Software for Exploring and Manipulating Networks*. 2009. URL: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154.

[29] Richard Durstenfeld. "Algorithm 235: random permutation". In: *Communications of the ACM* 7.7 (1964), p. 420.

[30] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (2011), p. 22.

Appendix B

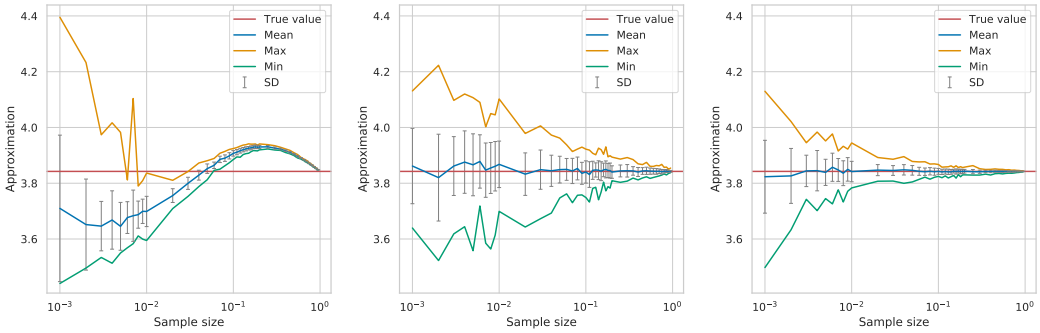
Acronyms

APL	Average Path Length
AS	Autonomous System
BA	Barabási-Albert
BFS	Breadth-First Search
ER	Erdős-Rényi
GRN	Global Reachable Nodes
LCC	Largest Connected Component
LWCC	Largest Weakly Connected Component
OU	Original-Uniform
RP	Reduced-Proportional
RU	Reduced-Uniform

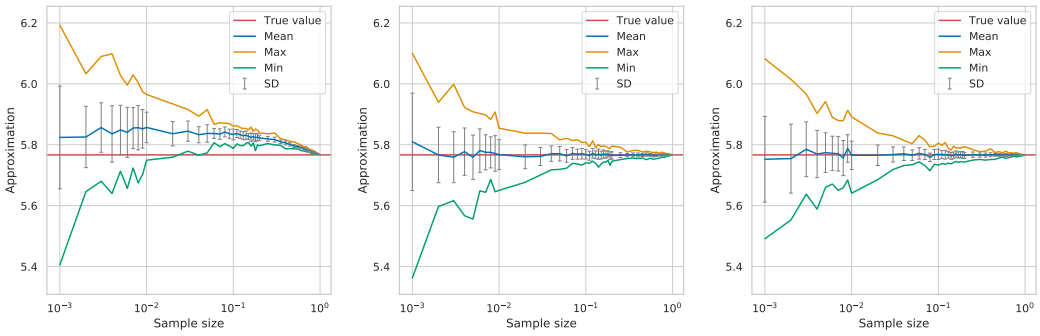
Appendix C

Approximation experiments results

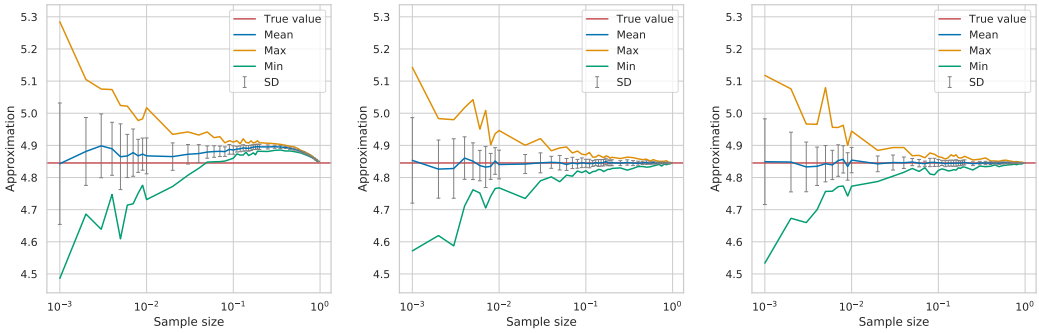
See next pages.



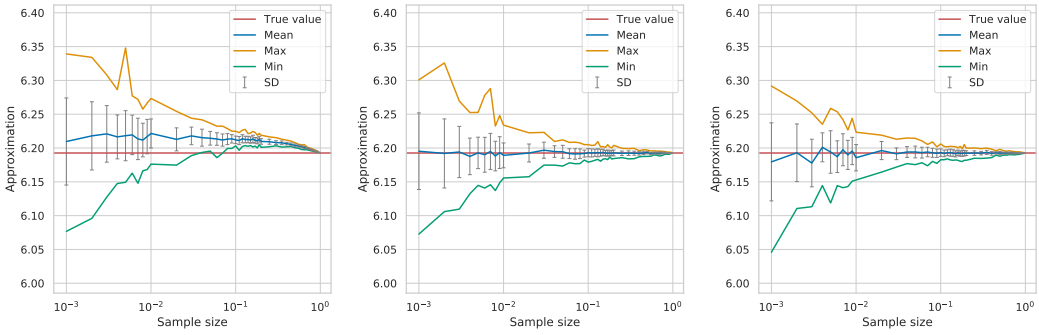
(a) `as-22july06`: Method RP **(b)** `as-22july06`: Method RU **(c)** `as-22july06`: Method OU



(d) `cond-mat-2003`: Method RP **(e)** `cond-mat-2003`: Method RU **(f)** `cond-mat-2003`: Method OU

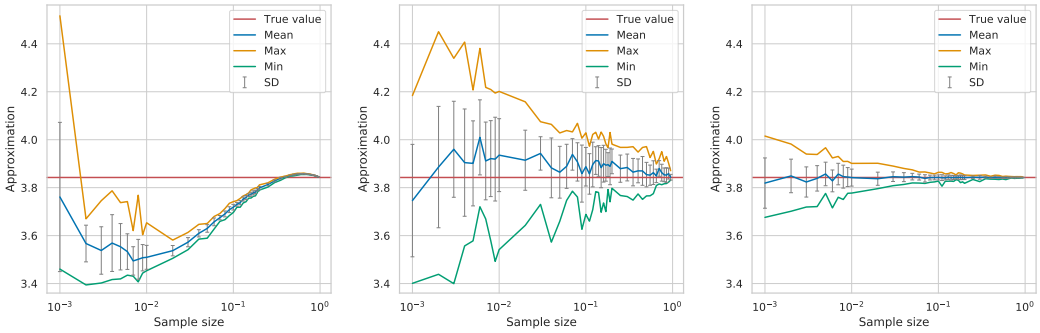


(g) `ba-graph`: Method RP **(h)** `ba-graph`: Method RU **(i)** `ba-graph`: Method OU

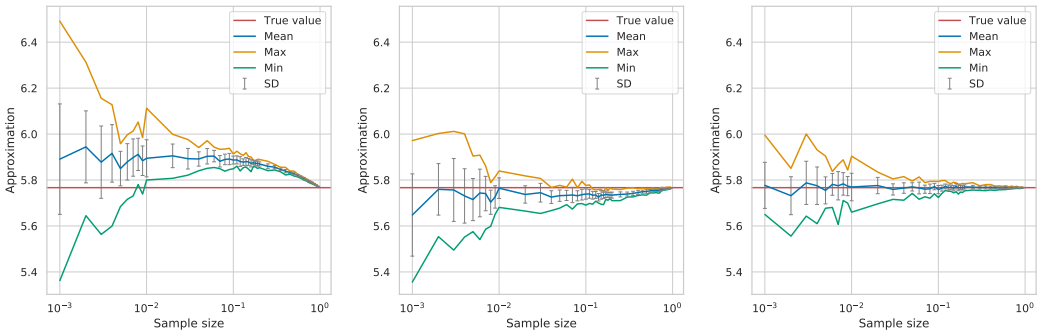


(j) `er-graph`: Method RP **(k)** `er-graph`: Method RU **(l)** `er-graph`: Method OU

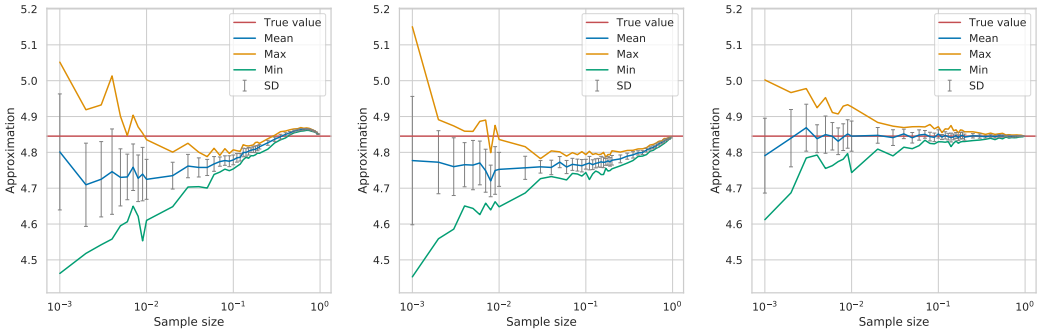
Figure C.1: Results of approximation experiments on the test graphs without weights using the three described methods. Average of 40 runs. Note the logarithmic scale on the X-axis.



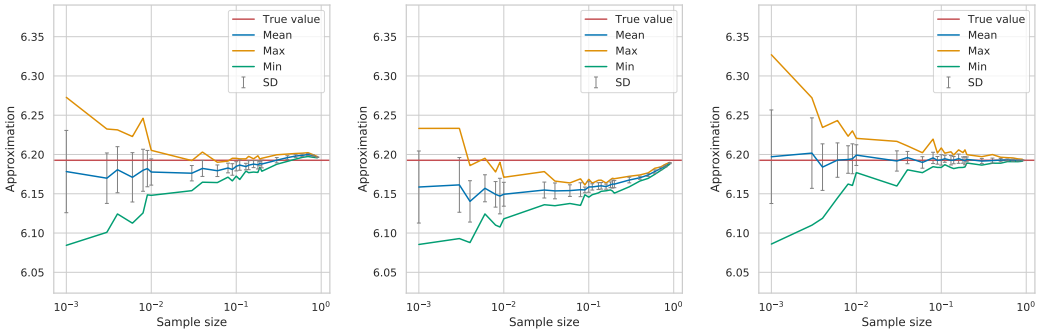
(a) `as-22july06`: Method RP **(b)** `as-22july06`: Method RU **(c)** `as-22july06`: Method OU



(d) `cond-mat-2003`: Method RP **(e)** `cond-mat-2003`: Method RU **(f)** `cond-mat-2003`: Method OU

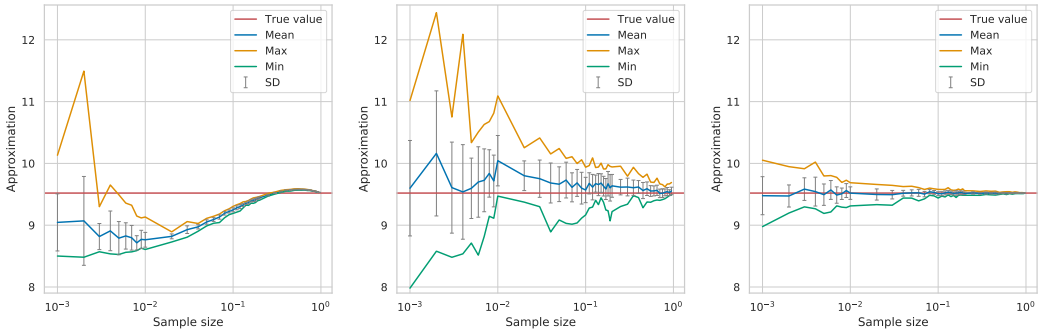


(g) `ba-graph`: Method RP **(h)** `ba-graph`: Method RU **(i)** `ba-graph`: Method OU

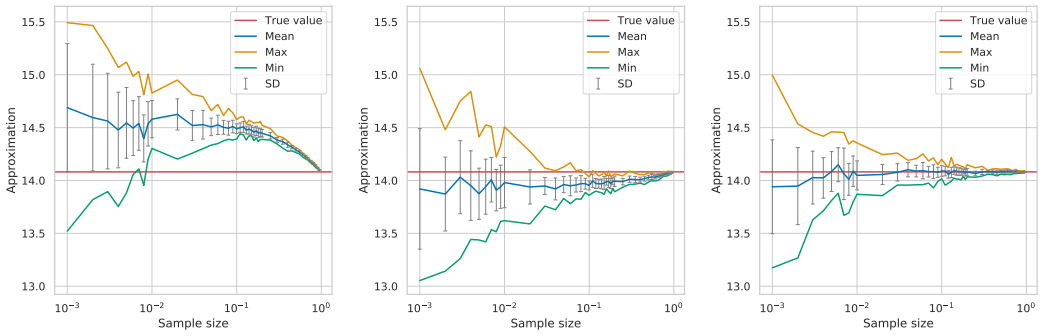


(j) `er-graph`: Method RP **(k)** `er-graph`: Method RU **(l)** `er-graph`: Method OU

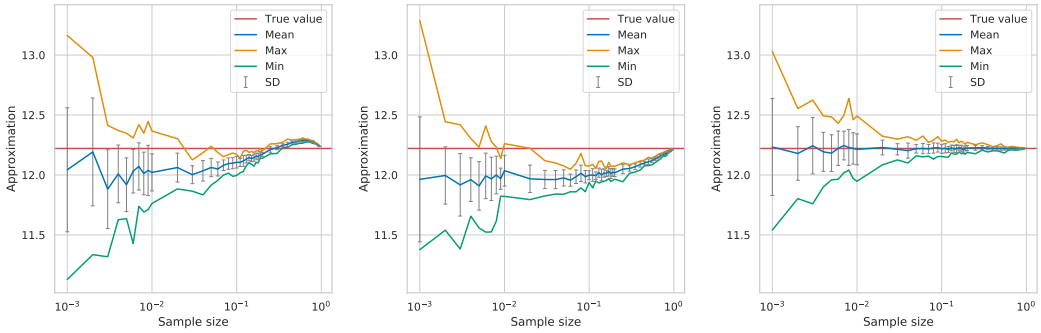
Figure C.2: Results of approximation experiments on the test graphs with unit weights using the three described methods. Average of 40 runs. Note the logarithmic scale on the X-axis.



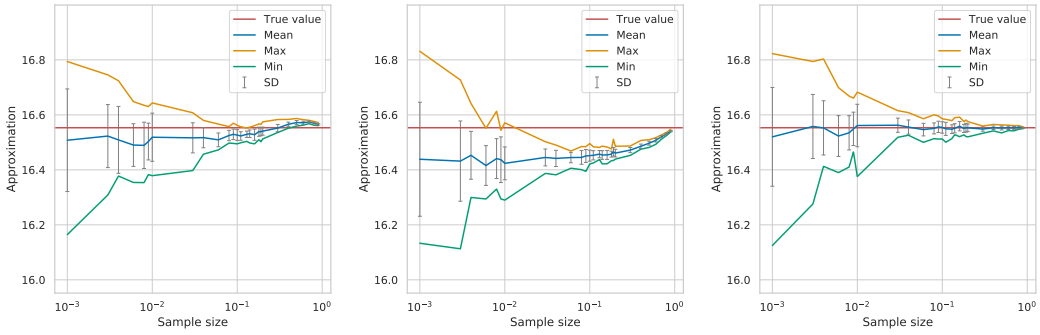
(a) `as-22july06`: Method RP **(b)** `as-22july06`: Method RU **(c)** `as-22july06`: Method OU



(d) `cond-mat-2003`: Method RP **(e)** `cond-mat-2003`: Method RU **(f)** `cond-mat-2003`: Method OU

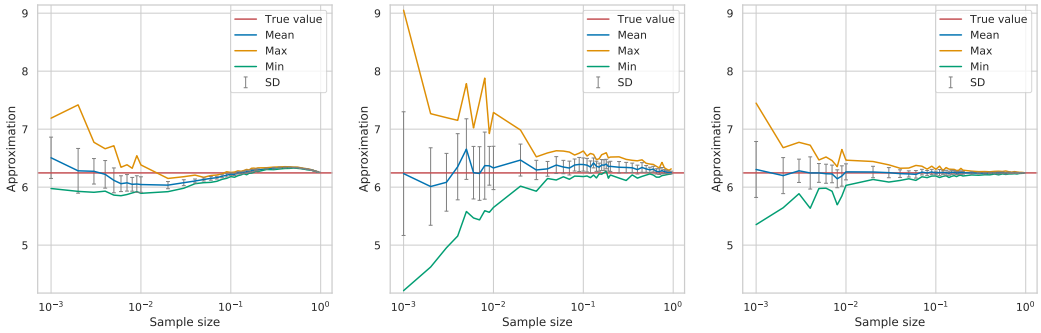


(g) `ba-graph`: Method RP **(h)** `ba-graph`: Method RU **(i)** `ba-graph`: Method OU

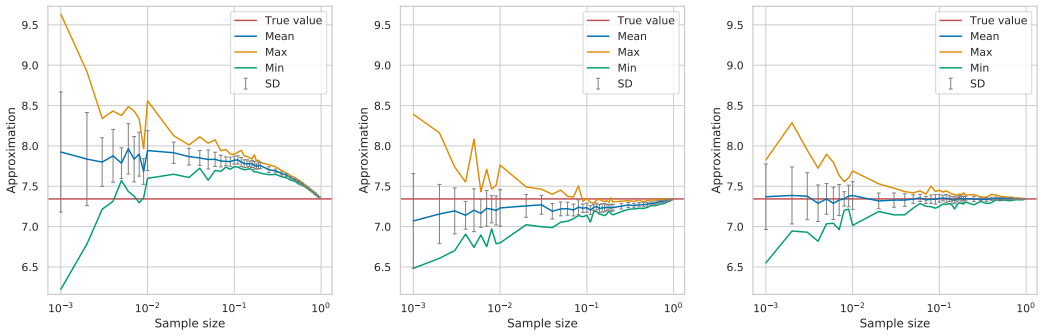


(j) `er-graph`: Method RP **(k)** `er-graph`: Method RU **(l)** `er-graph`: Method OU

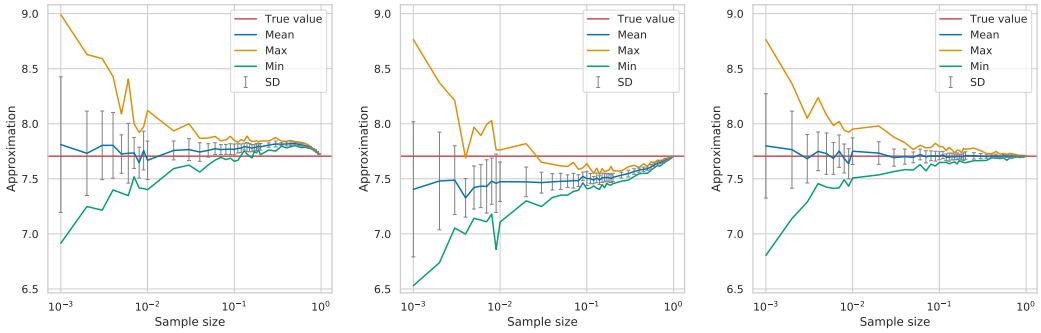
Figure C.3: Results of approximation experiments on the test graphs with normal distributed weights using the three described methods. Average of 40 runs. Note the logarithmic scale on the X-axis.



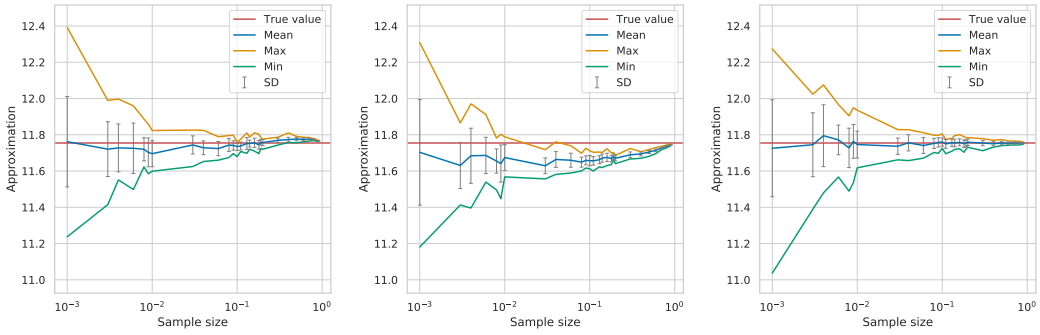
(a) `as-22july06`: Method RP **(b)** `as-22july06`: Method RU **(c)** `as-22july06`: Method OU



(d) `cond-mat-2003`: Method RP **(e)** `cond-mat-2003`: Method RU **(f)** `cond-mat-2003`: Method OU



(g) `ba-graph`: Method RP **(h)** `ba-graph`: Method RU **(i)** `ba-graph`: Method OU



(j) `er-graph`: Method RP **(k)** `er-graph`: Method RU **(l)** `er-graph`: Method OU

Figure C.4: Results of approximation experiments on the test graphs with uniformly distributed weights using the three described methods. Average of 40 runs. Note the logarithmic scale on the X-axis.

Appendix D

Approximation experiments thresholds

See next pages.

Graph	t	RU p	RU $\bar{e}$	RP p	RP $\bar{e}$	OU p	OU $\bar{e}$
as-22july06	5%	0.0007	4.344%	0.006	4.310%	0.0004	3.521%
as-22july06	1%	0.05	0.941%	0.7	0.920%	0.007	0.912%
as-22july06	0.5%	0.2	0.465%	0.85	0.437%	0.02	0.438%
as-22july06	0.1%	0.9	0.079%	∅	∅	0.35	0.086%
cond-mat-2003	5%	0.0002	4.454%	0.0004	4.189%	0.0002	3.837%
cond-mat-2003	1%	0.007	0.819%	0.2	0.965%	0.005	0.875%
cond-mat-2003	0.5%	0.03	0.402%	0.55	0.475%	0.02	0.452%
cond-mat-2003	0.1%	0.3	0.079%	0.9	0.094%	0.3	0.098%
ba-graph	5%	0.0001	4.745%	0.0001	4.944%	0.0002	4.027%
ba-graph	1%	0.008	0.728%	0.45	0.944%	0.005	0.860%
ba-graph	0.5%	0.04	0.300%	0.75	0.493%	0.02	0.408%
ba-graph	0.1%	0.45	0.081%	∅	∅	0.3	0.092%
er-graph	5%	0.0001	2.433%	0.0001	2.518%	0.0001	2.526%
er-graph	1%	0.0007	0.948%	0.001	0.875%	0.0006	0.824%
er-graph	0.5%	0.003	0.475%	0.007	0.498%	0.004	0.303%
er-graph	0.1%	0.08	0.091%	0.75	0.093%	0.07	0.093%

Table D.1: Threshold results on mean error on unweighted graphs

Graph	t	RU p	RU max e	RP p	RP max e	OU p	OU max e
as-22july06	10%	0.001	7.512%	0.003	8.031%	0.0009	6.580%
as-22july06	5%	0.03	4.415%	0.02	3.468%	0.003	2.676%
as-22july06	1%	0.4	0.794%	0.75	0.818%	0.05	0.985%
cond-mat-2003	10%	0.001	7.013%	0.0007	7.928%	0.0004	7.634%
cond-mat-2003	5%	0.002	2.996%	0.005	4.526%	0.002	4.301%
cond-mat-2003	1%	0.05	0.851%	0.35	0.925%	0.07	0.834%
ba-graph	10%	0.0005	7.870%	0.0008	6.706%	0.0007	7.605%
ba-graph	5%	0.004	3.551%	0.003	4.745%	0.002	4.756%
ba-graph	1%	0.07	0.850%	0.55	0.939%	0.03	0.989%
er-graph	10%	0.0001	7.241%	0.0001	8.450%	0.0001	8.242%
er-graph	5%	0.0003	3.972%	0.0004	4.570%	0.0003	4.921%
er-graph	1%	0.008	0.896%	0.02	0.995%	0.006	0.985%

Table D.2: Threshold results on maximum error on unweighted graphs

Graph	t	RU p	RU $\bar{e}$	RP p	RP $\bar{e}$	OU p	OU $\bar{e}$
as-22july06	5%	0.003	4.583%	0.06	4.818%	0.0003	4.104%
as-22july06	1%	0.75	0.806%	0.25	0.768%	0.006	0.917%
as-22july06	0.5%	0.95	0.240%	0.3	0.371%	0.03	0.392%
as-22july06	0.1%	∅	∅	0.95	0.091%	0.35	0.089%
cond-mat-2003	5%	0.0003	3.603%	0.0007	2.685%	0.0002	4.750%
cond-mat-2003	1%	0.009	0.932%	0.55	0.934%	0.005	0.836%
cond-mat-2003	0.5%	0.3	0.483%	0.75	0.472%	0.02	0.487%
cond-mat-2003	0.1%	0.85	0.092%	0.95	0.092%	0.3	0.084%
ba-graph	5%	0.0003	4.659%	0.0003	4.639%	0.0002	3.647%
ba-graph	1%	0.4	0.966%	0.13	0.941%	0.004	0.690%
ba-graph	0.5%	0.7	0.434%	0.25	0.383%	0.02	0.364%
ba-graph	0.1%	0.95	0.077%	∅	∅	0.35	0.063%
er-graph	5%	0.0001	2.105%	0.0001	1.813%	0.0001	3.286%
er-graph	1%	0.001	0.755%	0.0009	0.871%	0.0006	0.721%
er-graph	0.5%	0.19	0.488%	0.003	0.485%	0.004	0.406%
er-graph	0.1%	0.9	0.058%	0.8	0.096%	0.08	0.082%

Table D.3: Threshold results on mean error on weighted graphs with unit weights

Graph	t	RU p	RU max e	RP p	RP max e	OU p	OU max e
as-22july06	10%	0.007	9.780%	0.02	8.777%	0.0004	8.890%
as-22july06	5%	0.1	4.851%	0.07	4.765%	0.001	4.497%
as-22july06	1%	0.95	0.736%	0.3	0.550%	0.04	0.980%
cond-mat-2003	10%	0.0006	7.214%	0.002	9.470%	0.0003	6.965%
cond-mat-2003	5%	0.002	4.090%	0.02	4.028%	0.001	3.962%
cond-mat-2003	1%	0.17	0.880%	0.6	0.903%	0.03	0.883%
ba-graph	10%	0.0005	6.319%	0.0007	5.144%	0.0004	8.333%
ba-graph	5%	0.004	4.020%	0.01	4.852%	0.001	4.812%
ba-graph	1%	0.55	0.870%	0.19	0.983%	0.04	0.637%
er-graph	10%	0.0001	3.681%	0.0001	5.831%	0.0001	9.154%
er-graph	5%	0.0001	3.681%	0.0003	4.272%	0.0004	3.547%
er-graph	1%	0.03	0.914%	0.009	0.711%	0.006	0.817%

Table D.4: Threshold results on maximum error on weighted graphs with unit weights

Graph	t	RU p	RU $\bar{e}$	RP p	RP $\bar{e}$	OU p	OU $\bar{e}$
as-22july06	5%	0.02	3.171%	0.05	4.917%	0.0004	3.507%
as-22july06	1%	0.55	0.788%	0.19	0.942%	0.008	0.783%
as-22july06	0.5%	0.9	0.335%	0.75	0.491%	0.04	0.385%
as-22july06	0.1%	∅	∅	∅	∅	0.5	0.094%
cond-mat-2003	5%	0.0008	3.871%	0.002	4.192%	0.0002	4.275%
cond-mat-2003	1%	0.05	0.900%	0.7	0.878%	0.008	0.873%
cond-mat-2003	0.5%	0.45	0.457%	0.85	0.426%	0.03	0.480%
cond-mat-2003	0.1%	0.9	0.084%	∅	∅	0.5	0.059%
ba-graph	5%	0.0008	3.651%	0.0005	4.500%	0.0003	4.132%
ba-graph	1%	0.45	0.997%	0.1	0.940%	0.006	0.999%
ba-graph	0.5%	0.75	0.406%	0.75	0.485%	0.02	0.382%
ba-graph	0.1%	0.95	0.080%	∅	∅	0.35	0.090%
er-graph	5%	0.0001	2.476%	0.0001	2.716%	0.0001	4.957%
er-graph	1%	0.003	0.919%	0.003	0.551%	0.001	0.897%
er-graph	0.5%	0.3	0.491%	0.009	0.412%	0.004	0.468%
er-graph	0.1%	0.9	0.073%	0.9	0.074%	0.13	0.078%

Table D.5: Threshold results on mean error on weighted graphs with normally distributed weights

Graph	t	RU p	RU max e	RP p	RP max e	OU p	OU max e
as-22july06	10%	0.02	7.701%	0.007	9.993%	0.0006	8.075%
as-22july06	5%	0.16	4.088%	0.08	4.191%	0.005	3.460%
as-22july06	1%	∅	∅	0.25	0.520%	0.07	0.980%
cond-mat-2003	10%	0.001	7.299%	0.002	9.830%	0.0009	9.298%
cond-mat-2003	5%	0.005	4.573%	0.05	4.120%	0.003	3.223%
cond-mat-2003	1%	0.3	0.841%	0.7	0.947%	0.09	0.640%
ba-graph	10%	0.0008	9.107%	0.0007	9.894%	0.0004	9.133%
ba-graph	5%	0.008	4.961%	0.007	3.948%	0.002	3.423%
ba-graph	1%	0.6	0.978%	0.17	0.949%	0.03	0.782%
er-graph	10%	0.0001	6.197%	0.0001	5.881%	0.0003	5.363%
er-graph	5%	0.0003	4.493%	0.0006	3.430%	0.0004	3.580%
er-graph	1%	0.06	0.889%	0.03	0.940%	0.03	0.376%

Table D.6: Threshold results on maximum error on weighted graphs with normally distributed weights

Graph	t	RU p	RU $\bar{e}$	RP p	RP $\bar{e}$	OU p	OU $\bar{e}$
as-22july06	5%	0.009	4.342%	0.002	3.999%	0.002	4.351%
as-22july06	1%	0.65	0.725%	0.75	0.921%	0.04	0.863%
as-22july06	0.5%	0.85	0.386%	0.9	0.388%	0.14	0.423%
as-22july06	0.1%	∅	∅	∅	∅	0.85	0.061%
cond-mat-2003	5%	0.002	4.716%	0.25	4.936%	0.0008	4.763%
cond-mat-2003	1%	0.4	0.968%	0.85	0.929%	0.03	0.977%
cond-mat-2003	0.5%	0.7	0.471%	0.95	0.318%	0.13	0.377%
cond-mat-2003	0.1%	∅	∅	∅	∅	0.75	0.079%
ba-graph	5%	0.003	4.390%	0.002	3.900%	0.0008	4.463%
ba-graph	1%	0.65	0.848%	0.8	0.860%	0.03	0.840%
ba-graph	0.5%	0.8	0.455%	0.9	0.498%	0.08	0.474%
ba-graph	0.1%	∅	∅	∅	∅	0.75	0.075%
er-graph	5%	0.0003	3.061%	0.0001	4.911%	0.0003	2.330%
er-graph	1%	0.04	0.778%	0.008	0.492%	0.006	0.506%
er-graph	0.5%	0.5	0.421%	0.03	0.339%	0.01	0.471%
er-graph	0.1%	0.9	0.076%	0.9	0.095%	0.3	0.088%

Table D.7: Threshold results on mean error on weighted graphs with uniformly distributed weights

Graph	t	RU p	RU max e	RP p	RP max e	OU p	OU max e
as-22july06	10%	0.03	5.061%	0.003	8.450%	0.002	9.626%
as-22july06	5%	0.19	4.144%	0.03	4.229%	0.01	3.514%
as-22july06	1%	∅	∅	0.8	0.809%	0.2	0.855%
cond-mat-2003	10%	0.006	6.099%	0.05	9.414%	0.003	8.227%
cond-mat-2003	5%	0.02	4.354%	0.35	4.820%	0.008	2.911%
cond-mat-2003	1%	0.7	0.873%	0.9	0.703%	0.2	0.662%
ba-graph	10%	0.01	7.766%	0.004	9.382%	0.002	8.539%
ba-graph	5%	0.04	4.879%	0.02	2.975%	0.005	4.825%
ba-graph	1%	0.75	0.923%	0.8	0.944%	0.14	0.777%
er-graph	10%	0.0003	7.144%	0.0006	9.603%	0.0003	7.172%
er-graph	5%	0.001	4.881%	0.003	2.892%	0.003	3.106%
er-graph	1%	0.19	0.782%	0.04	0.868%	0.03	0.790%

Table D.8: Threshold results on maximum error on weighted graphs with uniformly distributed weights