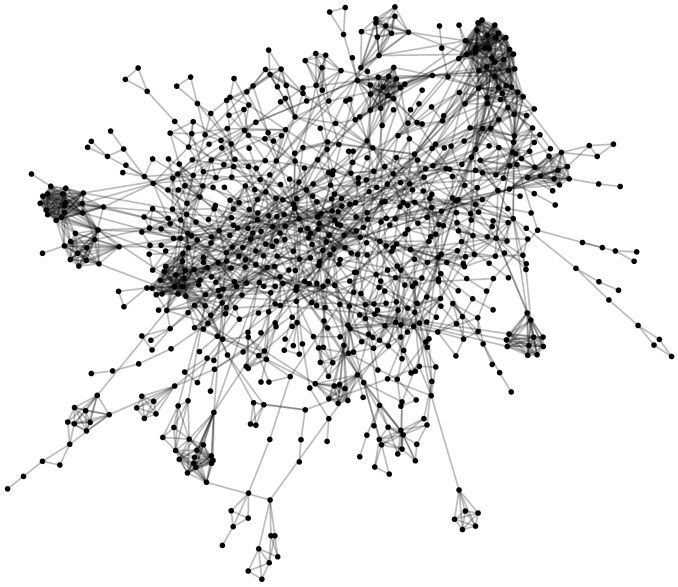
UNIVERSITY OF CAMBRIDGE

Juraj Mičko
jm2186

Framework for Empirical Analysis of Graph Metric Robustness



Jesus College
University of Cambridge

Computer Science Tripos – Part II

May 2020

Declaration

I, Juraj Mičko of Jesus College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Juraj Mičko of Jesus College, am content for my dissertation to be made available to the students and staff of the University.

Signed: 

Juraj Mičko

Date: May 22, 2020

Proforma

Candidate number:	**2343F**
Project Title:	**Framework for Empirical Analysis of Graph Metric Robustness**
Examination:	**Computer Science Tripos – Part II, 2020**
Word Count:	**11892** [1]
Line Count:	**6751** [2]
Project Originator:	**Dr Timothy Griffin**
Supervisor:	**Dr Timothy Griffin**

Original aims of the project

Networks are commonly analysed using graph metrics to identify their key properties. However, real-world data can be inherently imprecise, leading to inaccuracies in the measured metric values. A paper by Bozhilova et al. [1] tackles a similar problem, quantifying robustness of metrics in scored protein interaction networks.

The project will first follow their experimental approach to produce similar results. Second, it will extend the concept of rank robustness of metrics to unscored networks, studying how susceptible they are to small perturbations in graphs. The result will be a tool that facilitates experiments of measuring robustness of metrics on many network datasets.

Work completed

The project met all success criteria. I designed a way to analyse robustness of graph metrics on unscored networks, advancing the concept of rank robustness from the paper by Bozhilova et al. [1]; built a command-line tool `graffs` from scratch in Kotlin, for carrying out large-scale experiments of measuring robustness of metrics, in an efficient, flexible, and reproducible way; set up a high-performance computing environment; reproduced expected results and derived new interesting results about rank robustness of metrics of unscored datasets. Finally, I published `graffs` open-source, ready to be reused and extended, and proposed further research within this novel field.

[1] This word count was computed using: `texcount -1 -merge -q -utf8 -sum main.tex`

[2] This total lines of code count was computed using:

```
cloc --include-lang="Kotlin,Gradle,Python,SQL,CSS" --exclude-d=.idea \
  --quiet --hide-rate --csv graffs partii | tail -n 1 | awk -F, '{print $3+$4+$5}'
```

from `https://github.com/AlDanial/cloc`

Special difficulties

Due to world-wide measures related to the COVID-19 pandemic, I was held in Azerbaijan for 6 weeks, uncertain about possible repatriation travels, then spent two weeks in a compulsory government quarantine facility and in self-isolation in Slovakia, before returning home to a working environment in Easter Term.

I was approved a two-week extension which accounted for these circumstances.

Contents

1. Introduction

> *A reported value whose accuracy is entirely unknown is worthless.*
>
> — Churchill Eisenhart [2]

I designed a way and built a tool to analyse robustness of graph metrics by evaluating them on many generated graphs.

Graph metrics are often used to find and derive facts about important nodes or components of networks. However, real-world datasets may be inherently imprecise, or the procedure of obtaining networks from raw measured data may introduce (sometimes invisible) inaccuracies. When evaluating graph metrics on imprecise data, it is crucial to be able to reason about the reliability of the results, as with any other statistical procedures. The field of examining stability and reliability of graph metrics is relatively novel.

This project was inspired by, and is based on the paper "Measuring rank robustness in scored protein interaction networks" by L. V. Bozhilova et al. [1] (further referred to as "The Paper") which served as a kickoff point for ideas in this project. The Paper introduces ways to assess *robustness* of graph metrics specifically on protein interaction networks with confidence-scored edges.

In this dissertation I summarise the research done on this topic and build a command-line tool `graffs` that helps generalise the research of graph metric robustness and enables similar experiments on graphs of other kinds. The program is written in Kotlin from scratch, and wrapped up and published under GPLv3 as an open-source library that can further be used by future projects in this research area.

The goals of my work are, namely:

1. To implement a tool called `graffs` to automate the kind of experiments done in The Paper
2. To reproduce a subset of results from The Paper
3. To extend the idea of graph metric robustness on unscored networks
4. (Extension) to polish the tool and publish it under an open-source licence

Chapter 2 explains The Paper in more detail, revises the background material and explains method used in `graffs`. First, I generalise graph acquisition using edge score thresholding and random edge removal. Chapter 3 describes the tool implementation of the framework, which I then evaluate in chapter 4, by applying a number of robustness measures on some of the most common node-level graph metrics. Finally, I compare results with The Paper and derive new observations about unscored graph classes.

2. Preparation

This chapter consists of understanding the problem from the mathematical and research point of view. I explain in detail why measuring the robustness of graph metrics is an important and novel field, describing the work done in The Paper and similar academic studies. Then I introduce definitions of graph metrics and robustness measures used in the implementation.

2.1 Graphs and metrics

Ever-increasing applications of data drive researchers to use tools such as graph theory to model real-world problems, involving fields such as engineering, biology, chemistry, social systems, and many more [3]. Graph theory proved especially useful when analysing large structures of data which cannot simply be visualised. Graph *statistics* (e.g. size, diameter, radius) help describe properties of arbitrarily large graphs. Further, graph *metrics* (betweenness centrality, closeness centrality, eccentricity, etc.) quantify properties of individual nodes.

2.2 Introduction to confidence

Error measures are important in assessing how precise an observation is [4]. "A reported value whose accuracy is entirely unknown is worthless" [2]. Confidence scores, confidence intervals, and error ranges signal or approximate the extent to which the observed result is reliable.

Networks obtained from the real-world may contain errors too, such as missing components, false positives, uncertain nodes or edges, and falsely aggregated nodes. Network errors may stem from measurement errors, approximations, lack of knowledge, lack of experimental evidence or thresholding of scored networks [5]–[7]. These errors may then propagate to errors in graph metrics.

Can we measure precision or confidence of graph metrics? If an outcome is based on evaluating metrics on graphs which are themselves imprecise or possibly have errors in the structure, how can we assess the reliability of such result? Helping to answer these questions is the motivation behind building `graffs`.

Many attempts already tried to assess robustness or stability properties of graph metrics. Examples are: analysing the impact of errors on centrality measures in random networks [8] or, more specifically, evaluating the reproducibility of metrics on multi-temporal scans of the human brain using the coefficient of variation (CV), the repeatability coefficient (RC) and the intra-class correlation (ICC) [9], [10]. A significant amount of research analysing reproducibility of metrics in uncertain brain networks appeared around 2011, summarised in [11]. The latest research led to the development of methods for the analysis of such uncertain data, estimating a ground-truth network structure from imperfect data [12], [13], and even a method to mitigate sensitivity of metrics on the selection of a threshold in scored networks [14].

2.3 The Paper

This dissertation project builds on top of the ideas from The Paper by Bozhilova et al. [1], recent research on metric robustness. The study, with main concepts explained below, proposes ways to quantify the robustness of graph metrics in thresholded protein networks.

2.3.1 Protein networks

Protein interaction networks (PINs) are an example of a prominent research field relying on graph theory. Nodes represent proteins present in cells of organisms and edges represent interactions between proteins. Studying them helps in understanding the physiology of biological cells and developing drugs.[3]

Protein networks have a specific structure: nodes represent proteins and weighted (or scored) edges represent the presence of interactions between proteins. It is assumed that such interactions are mutual, therefore the formed graph is undirected. fig. 2.3 shows an example of a protein network.

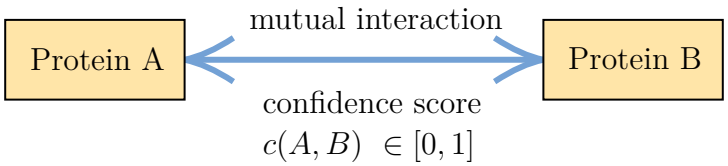


Figure 2.1: Schematic of an interaction between two proteins

The weight of each edge in protein networks represents a confidence score which may be a value or a set of values, describing some sort of *likelihood* that the interaction is true, given the available evidence (see fig. 2.2). Section 2.5.1 explains scored networks in detail.

2.3.2 Network thresholding

When researching protein interactions, in practice, a **threshold** is chosen and then a protein interaction network is formed from the database considering only interactions (edges) whose confidence score is greater than the selected threshold, to only take into account interactions of sufficiently high quality. Thresholding edges of sufficient confidence is a common and necessary step before further graph analysis.

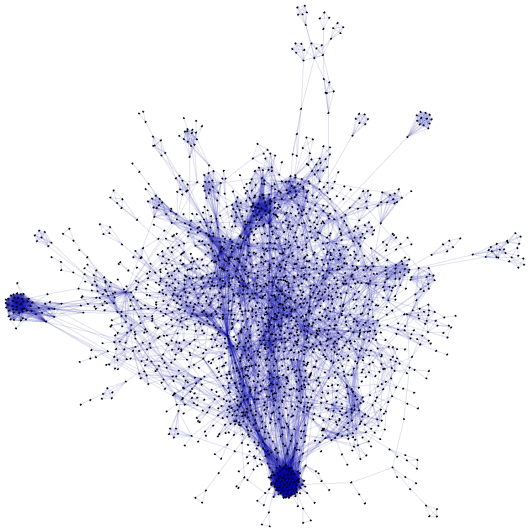
2.3.3 Metric robustness

As The Paper remarks, metric evaluation can be susceptible to the choice of the threshold. However, a useful metric should lead to similar results across networks obtained at different reasonable confidence score thresholds. Hence, the point of studying metric robustness, or stability, is a way to assess how reliable the results are on a graph derived from a given threshold.

When measuring the robustness of node metrics, numerical values of metrics will undoubtedly depend on the chosen threshold.[5] For this reason, The Paper assesses robustness based on the **rankings** of nodes induced by each metric.

[3]For example, highly connected proteins and hubs of proteins in a cell are shown to be most essential for its survival [15], [16].

[5]For example, the degree and the clustering coefficient of each node will monotonically decline with a declining density of the network, which monotonically declines as we increase the confidence threshold.



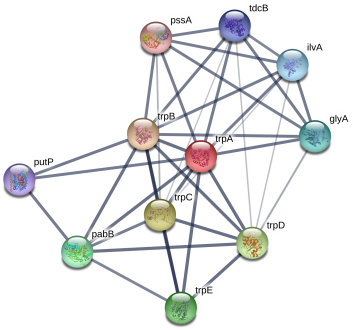


Figure 2.2: (right) Protein-protein interaction network visualised[4] by the STRING database [17]. Edge thickness indicates the overall confidence score, i.e. the strength of data support.

Figure 2.3: (left) A protein interaction network from the *Escherichia coli* organism from the STRING database, thresholded at the 0.9 score (high confidence). The giant component has 2382 nodes and 12071 edges.

Rank robustness

The Paper introduces three measures assessing the robustness of node metrics (defined later):

- rank continuity (comparing highly ranked nodes at similar thresholds),
- rank identifiability (comparing ranks at various thresholds and the overall ranking),
- rank instability (quantifying variation of ranks of top 1% of nodes).

All three measures are based on rankings of nodes induced by node metrics, and not on the exact values, to mitigate the density bias described above.

The Paper studies robustness of 25 different *node* metrics only, i.e. graph metrics which return one numerical value per node.

2.4 Mathematical background

I review background material on graph theory (based mostly on [18]) and fix terms to later avoid ambiguity. Later, I introduce graph metrics, ranking, and metric robustness.

Graphs (or networks) from graph theory consist of *nodes* (or vertices, components) and *edges* between them (links). Formally, let **graph** G be a pair (V, E) of a finite set of n **nodes** V and a set of **edges** $E \subseteq V \times V$. In this work, I also use standard definitions of graph properties, relations on graphs, node adjacency and connectedness [18]. Those are omitted here for brevity but included for reference in appendix A.

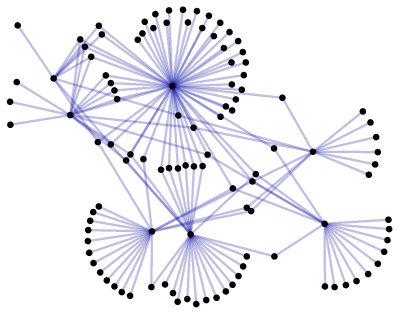


Figure 2.4: A small example of a *connected simple* graph (*undirected* and *anti-reflexive*).

Graphs I use in this project are all *simple graphs*, i.e., edges are undirected (or bi-directional) and without loops. Figure 2.4 is an illustration of a connected simple graph with 110 nodes and 151 edges, one of those I used for testing purposes when developing `graffs`.

I may interchange the words *graph*, *network* and *dataset*, but generally: *dataset* is data stored in its raw form, such as a file downloaded from the Internet; *network* is the real-world

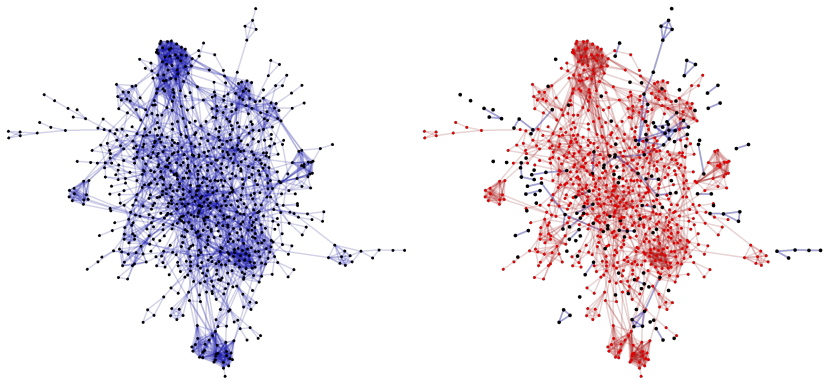


Figure 2.5: Illustrated is how thresholding (a subgraph of) a protein network (left) results in one giant component (red) and multiple small disconnected components. The left is a certain subgraph of the *ecoli* dataset (all edges with confidence > 0.4), the right graph has only edges with confidence > 0.5.

graph-like data conveyed by a dataset; and *graph* is a hypothetical structure, a mathematical representation, and an interface for algorithms.

Most of the real-world networks (such as social networks, citation networks or even protein interaction networks) contain a single giant component and some small isolated components. For example, in Figure 2.5, the red nodes form a giant component of the graph.

2.4.1 Graph metrics

Metrics are essentially functions of graphs, assigning a real value to each node. They are used to quantify various properties of graphs, identify important nodes in different contexts, describe graph structures, and more.

Mathematically speaking, if $\mathbb{G}$ is the set of all graphs $G = (V, E)$ with $V \subset \mathbb{V}$, then we can define the domain of metrics to be generally functions of graphs:

$$\mathbb{M} \stackrel{\text{def}}{=} \mathbb{G} \Rightarrow (\mathbb{V} \Rightarrow \mathbb{R}) \tag{2.1}$$

Evaluation of a metric on a graph then results in a *mapping* of nodes to real numbers. Therefore, for some metric $M \in \mathbb{M}$, some graph $G = (V, E)$, and a node $v \in V$: [6]

$$M(G) : \mathbb{V} \to \mathbb{R} \tag{2.2}$$

$$M(G)(v) \in \mathbb{R} \tag{2.3}$$

The metrics introduced below are those I chose to implement in the `graffs` tool. Their definitions are based on The Paper [1] (preferably, so that results can be reproduced), and on a paper summarising graph metrics [19]. They are all defined for *undirected* graphs only.

Centralities

One of the simplest metrics is the *degree* nodes, often considered as a centrality measure.

$$\text{Degree:} \quad DC(v) \stackrel{\text{def}}{=} \deg(v) = |\{\, v' \in V \mid (v', v) \in E \,\}| \tag{2.4}$$

[6] However, for convenience, in the notation the graph argument is often left out when it is understood which graph the metric is applied to, resulting in $M(v) \in \mathbb{R}$.

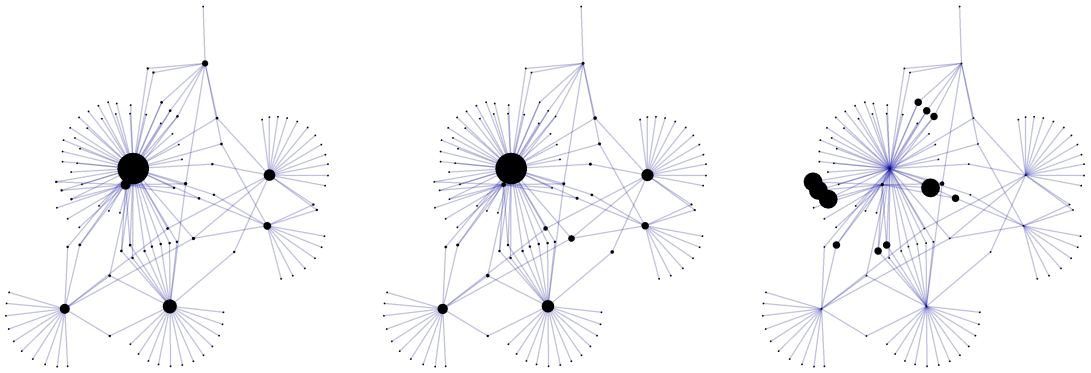


Figure 2.6: A simple graph with each node's diameter proportional to its *degree* (1), *betweenness centrality* (2), and *local clustering* (3). In this particular graph, (1) and (2) show similar characteristics (greater value for more "central" nodes), whereas local clustering is significantly different.

Betweenness centrality is calculated for each node v as the number of all shortest paths passing through that node, for all pairs of nodes. If there are multiple shortest paths between a pair of nodes, the fraction of those passing through v are considered.

$$\text{Betweenness:} \quad B(v) \stackrel{\text{def}}{=} \sum_{\substack{s,t \in V \\ s \neq v \neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{2.5}$$

where σ_{st} is the number of shortest paths between s and t, and $\sigma_{st}(v)$ is the number of those that pass through v.

Closeness centrality of a node v is the reciprocal value of the sum of $d(v,i)$, i.e., the distances from that node to all other nodes i in the graph. It measures the reciprocal of the *farness* of each node to other nodes.

$$\text{Closeness:} \quad CC(v) \stackrel{\text{def}}{=} \frac{1}{\sum_{i \neq v} d(v,i)} \tag{2.6}$$

This is sound for connected graphs, however, $d(v,i)$ is undefined if v, i belong to two different components of the graph. For disconnected graphs, I set $d(v,i) = |V|$, following the approach in The Paper.

Harmonic centrality is a similar measure of *farness*.

$$\text{Harmonic:} \quad HC(v) \stackrel{\text{def}}{=} \sum_{i \neq v} \frac{1}{d(v,i)}, \tag{2.7}$$

with $1/d(v,i) = 0$ for disconnected nodes v, i (as in [20]).

Both *closeness* and *harmonic centrality* require the computation of distances between all pairs of nodes. I used an implementation of the Floyd–Warshall algorithm [21] of time complexity $O(|V|^3)$, which came bundled in the graph library I used (section 3.4.1). A significant improvement could be achieved by running Dijkstra's algorithm [22] with time complexity $O(|E| + |V| \log |V|)$ starting in each node, or even better using Seidel's APD algorithm [23], but that was not the scope of my work.

Ego network measures

Ego networks are local subgraphs centred around a particular node in the network. Step-n ego network (also called ego-n network) of a node v is a subgraph including v (the *ego* node), all nodes reachable from v in at most n hops (the *alter* nodes), and all edges among these (see Figure 2.7). For simplicity, sets ego_n below are defined just as a subset of nodes V.

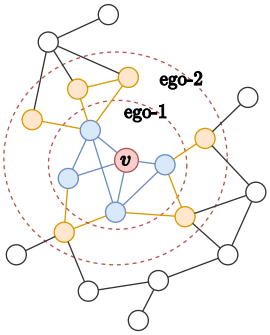


Figure 2.7: An illustration of ego-1 and ego-2 networks of a node v

For a graph $G = (V, E)$, define inductively $\text{ego}_n : V \to \mathcal{P}(V)$:

ego-**n** network:

$$\text{ego}_0(v) \stackrel{\text{def}}{=} \{ v \} \tag{2.8}$$

$$\forall n \in \mathbb{N}.\ \text{ego}_{n+1}(v) \stackrel{\text{def}}{=} \text{ego}_n \cup \{ v' \in V \mid \exists v'' \in \text{ego}_n(v).\ (v', v'') \in E \} \tag{2.9}$$

Ego metrics are node metrics whose value depends only on properties of local ego network(s). I chose and implemented 2 interesting ego metrics used also in The Paper: ego-**1** edges, and ego-**2** nodes.

$$\text{Ego1Edges:} \quad E1E(v) = |\{ (v', v'') \in E \mid v' \in \text{ego}_1(v) \wedge v'' \in \text{ego}_1(v) \}| \tag{2.10}$$

$$\text{Ego2Nodes:} \quad E2N(v) = |\text{ego}_2(v)| \tag{2.11}$$

Local clustering measures the proportion of existing and all possible edges among neighbours of v, defined as in [24]. Following that, *redundancy* node metric is defined as the extent that v's neighbours are adjacent to each other as well, according to [25].

Let $k = \deg(v)$, the degree of v, then

LocalClustering:

$$LC(v) \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if } k = 0, 1 \\ \dfrac{|\{ (v', v'') \in E \mid v' \in \text{ego}_1(v) \wedge v'' \in \text{ego}_1(v) \}|}{k(k-1)/2} & \textit{otherwise} \end{cases}$$

$$\text{Redundancy:} \quad R(v) \stackrel{\text{def}}{=} LC(v) \times (\deg(v) - 1) \tag{2.12}$$

Note that for $k \leq 1$, $LC(v) = R(v) = 0$.

Page rank

A different, interesting node metric is *PageRank* [26], first developed for Google Search to rate the importance of websites. For brevity, the full definition is not included here. I used an implementation of PageRank that is described in [27].

2.4.2 Ranking

It is clear that the values of most metrics will naturally change with changing graph structure. So to measure the robustness of a metric on a higher level, The Paper defines robustness measures which only depend on **ranking** of nodes induced by given metric. The important property of a stable metric is then to induce similar node rankings, not necessarily similar absolute values.

For example, in protein networks, the graph itself depends on the *confidence threshold* – high thresholds lead to sparse graphs. The degree of nodes will principally decrease with increasing threshold, however, degree happens to be a relatively stable metric in general, which means the sets of highest-degree nodes will be relatively similar across graphs of similar thresholds.

Here I assume a standard definition of the total order (included in appendix A). A less-than-or-equal relation ($\leq$) on real numbers $\mathbb{R}$ is an example of a total order.

Definition 2.4.1 (Ranking relation)

*A binary relation $\preceq$ on the set of nodes V of some graph $G = (V, E)$ is a **ranking relation induced by a metric** M **on a graph** G iff it is a total order satisfying:*

$$\forall v_1, v_2 \in V.\ M(v_1) < M(v_2) \implies v_1 \preceq v_2$$

By convention, one can consider a ranking to be a bijection between nodes V and the set $\{1, \ldots, |V|\}$, with the associated integers being called **ranks**. This definition means that ranking induced by a metric is *one of possible* orderings of nodes such that nodes with high metric values correspond to high ranks.

Definition 2.4.2 (Ranking (vector))

*A **ranking vector** $A^{\preceq}$, or a **ranking**, induced by a metric M on a graph G is a bijection $V \leftrightarrow \{1, \ldots, |V|\}$ such that*

$$\forall v_1, v_2 \in V.\ v_1 \preceq v_2 \implies A^{\preceq}(v_1) \leq A^{\preceq}(v_2),$$

where $\preceq$ is a ranking relation of graph $G = (V, E)$ with respect to metric M.

Also, allow $A^{M(G)}$ as a shortened notation of a ranking induced by the metric M on the graph G. Therefore,

$$A^{M(G)} : V \leftrightarrow \{1, \ldots, |V|\}$$

Note, there may be multiple valid rankings if multiple nodes have the same metric value, in which case any mutual order of such nodes is permitted in the ranking. A ranking just assigns each node a rank (or a position) such that it does not violate the ordering by metric values. Also, note that each node must be assigned a *unique* rank, due to the antisymmetry property.

2.4.3 Robustness

At the highest level where metrics are functions (Equation 2.1), robustness measures can be regarded as functions from $\mathbb{M}$ (the domain of all metrics) to real numbers.

$$\mathcal{R} \stackrel{\text{def}}{=} \mathbb{M} \Rightarrow \mathbb{R} \tag{2.13}$$

There are two relevant remarks to be made about robustness functions.

Robustness is empirical. The point of this dissertation is not to analyse the metrics purely as functions (which would indeed be difficult and not generalisable), instead, I analyse robustness **empirically**, by evaluating metrics on many graphs and deducing clues about how stable the values are across these multiple graphs. A reader can find more on this is in the Methods section.

Per-graph robustness. Robustness measures are calculated by evaluating metrics on graphs, and so values inherently depend on the graphs chosen (e.g. a metric may be more stable on dense graphs in general, where another metric may be unstable). The Paper calculates robustness by experiments specifically on a few protein interaction networks (which presumably follow a similar structure), and then even dares to calculate the averaged robustness.

Per-graph robustness is, indeed, not helpful in answering the question "Which metrics can I rely on when analysing my custom dataset X?" which is a problem my project tackles. Instead of coming up with more-general results, I defined **robustness** of a metric **on a certain graph** and rather built a *tool* that can perform this analysis generally on any given graph:

$$\mathcal{R} \stackrel{\text{def}}{=} \mathbb{M} \times \mathbb{G} \Rightarrow \mathbb{R} \tag{2.14}$$

2.5 Methods

Taking previously defined concepts, here I introduce the specific methods that I used to implement the `graffs` tool. In particular, this section will discuss: datasets and their kinds, small perturbations on graphs, evaluating metrics, ranking nodes, and definitions of robustness measures. Overall rank, k-similarity, and α-relaxed k-similarity are the building blocks of the robustness measures, and will be defined throughout this section.

2.5.1 Datasets

In my work I distinguish two types of networks: scored and unscored. Later in this section, I present different publicly available graph datasets used in this project.

Scored networks

In scored networks, edges are weighted, with each weight (also called **score**) representing the likelihood that the edge is true in the real world. The score is usually a real value between 0 and 1, possibly scaled (see fig. 2.8). Scored networks naturally provide a way to introduce perturbations to the networks, by adjusting the confidence threshold.

Using a scored network as-is just by discarding the confidence score is usually unacceptable as many low-confidence edges would be false positives. Obtaining a functional network from an almost fully connected clique needs thresholding at a reasonable value, and scored databases often state reasonable ranges of the confidence scores.

Table 2.1: A summary of *scored* datasets used in the Evaluation.

name	network	nodes	edges
`pvivax`	Plasmodium vivax, STRING	3255	344691
`ecoli`	Escherichia coli, STRING	4144	583440
`yeast`	Saccharomyces cerevisiae, STRING	6418	939998

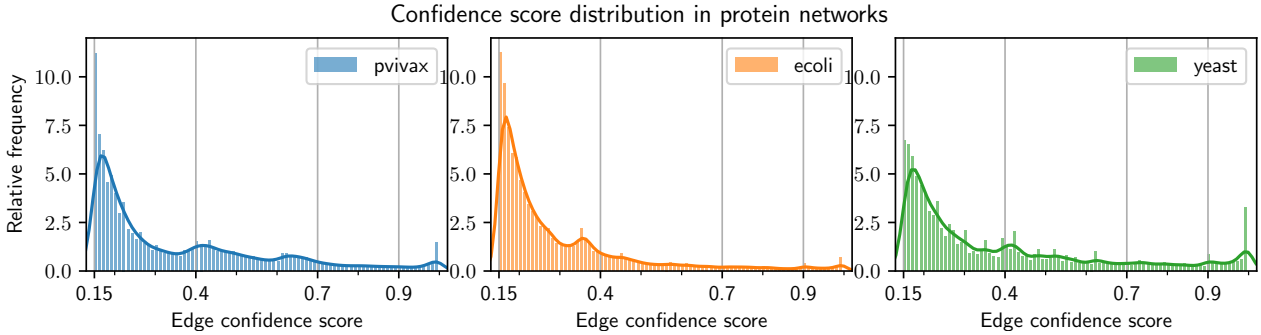


Figure 2.9: Distribution of the confidence scores of edges in 3 scored protein interaction networks.

STRING database [17]**.** An open database of scored interactions between proteins. At the time of writing, the database contains over 24M different proteins from over 5K organisms, accounting for over $2 * 10^9$ interactions. It provides confidence scores for various types of evidence, such as scientific experiments, genetic observations, statistical predictions, text mining of scientific articles, or other existing knowledge.

An overall confidence score is calculated for each edge, conveying "the approximate probability that a predicted link exists between two enzymes (proteins)". The Paper and my dissertation both use for simplicity only the overall score, disregarding biological details. The database offers the following guidance on the reasonable score values:

- low confidence – 0.15 (or better)
- medium confidence – 0.4
- high confidence – 0.7
- highest confidence – 0.9

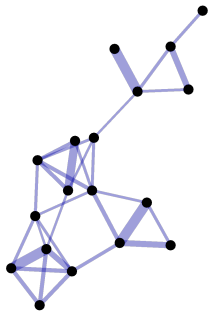


Figure 2.8: A tiny subgraph of the `ecoli` dataset with the width of edges proportional to their scores

A full-organism network is a subgraph of the huge database, containing only proteins of the given organism. In my work, I evaluate robustness on 3 full-organism protein networks from the STRING database[7], listed in table 2.1. Figure 2.9 shows the distribution of confidence scores in these networks. Databases HitPredict [28] and IntAct [29] are another examples of scored datasets, but not used in this project.

Unscored networks

Unscored networks are unweighted graphs, not conveying any information about the confidence of edges. Thresholding a scored network and discarding the scores results in an unscored network. There are numerous online sources providing interesting unscored network datasets.

[7]Datasets are taken from the source files of The Paper's analysis instead of the STRING database directly, to match those in The Paper. Available at `https://github.com/lbozhilova/measuring_rank_robustness`

Table 2.2: A summary of *unscored* datasets used in the Evaluation.

name	network	nodes	edges
`airports`	Airport–airport flights in the US in 2010 [34], [35], KONECT	1574	17215
`citation`	High-energy physics theory citation network [36], [37], SNAP	27770	352285
`collab`	Collaboration network of Arxiv General Relativity category [38], SNAP	5242	14484
`facebook`	Social circles (friends) from Facebook [39], SNAP	4039	88234
`internet`	Internet topology network of autonomous systems [40], [41], KONECT	34761	107720

SNAP datasets. Stanford Large Network Dataset Collection [30] of the Stanford Network Analysis Project [31] provides open datasets obtained from real-world data such as social networks, citation networks, web graphs, internet networks, road networks and many more.

KONECT datasets. The Koblenz Network Collection [32] presents hundreds of network datasets of various types, covering areas such as "social networks, hyperlink networks, authorship networks, physical networks, interaction networks, and communication networks".

I picked a few datasets (listed in table 2.2) to demonstrate the concept of `graffs` (see the Evaluation chapter), considering mainly their supposed structure and size (number of nodes and density; due to associated computational cost). Other huge online repositories such as the Network Repository [33] provide access to thousands similar networks (outside of the scope of this work), which are also compatible with `graffs`.

2.5.2 Perturbing graphs

Many datasets are observations or approximations of real-world structures, not their exact representation (e.g. protein networks). Analysing robustness of a metric means studying how different the metric values *could be* if we performed the same experiment on other datasets coming from the same source – datasets possibly measured by different techniques, at different times, or in a different universe.

Often, there is often only one instance of a graph to work with, so to simulate having multiple [8] graphs, we generate multiple samples with small perturbations.

Definition 2.5.1 (Graph generator)

A graph generator ξ is a function that takes a graph G and a number n[a] and generates a sequence of n graphs based on G.

$$\xi(G) \overset{n}{\hookrightarrow} G_1, G_2, \ldots, G_n \tag{2.15}$$

Thresholding scored networks

Various methods were proposed to choose a good baseline threshold, from fixing the value upfront [42] or after examining a range of values [43], [44], through maximising the threshold given some constraints [45], to choosing a value optimising some criterion such as classification rate [46]. Still, the threshold is a hyper-parameter of the graph-obtaining process and robust metrics should respond similarly to all reasonable values of the chosen threshold.

[8] Here, "similar" means that they convey the same real-world structure.

[a] The exact value of n is conceptually unimportant and is chosen individually in the Evaluation chapter.

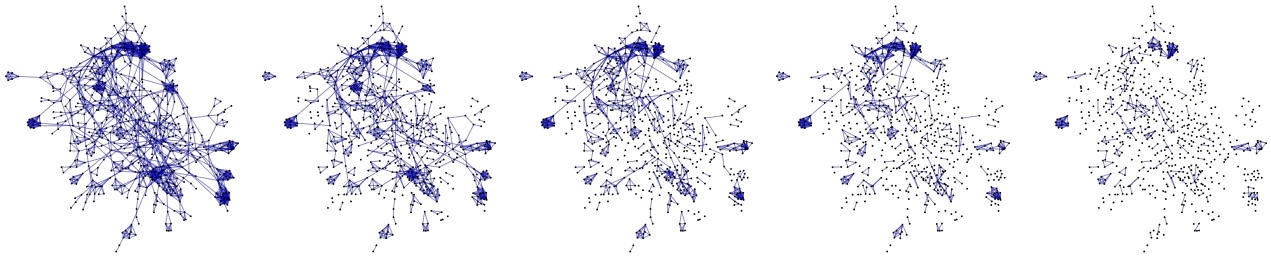


Figure 2.10: Visualised effect of thresholding confidence scores. The 5 graphs correspond to a small subset of the *ecoli* dataset thresholded at linearly spaced values between 0.55 and 0.95.

For scored networks, we generate perturbed graphs by considering different confidence thresholds (in a reasonable range) of the same scored network. Resulting is a sequence of graphs with decreasing density as the threshold increases. Define a **linear thresholding** graph generator, given a minimum threshold μ and a maximum threshold ν, as follows:

$$
\begin{aligned}
&{}^{T}\xi_{\mu}^{\nu}(G) \;\overset{n}{\hookrightarrow}\; (V, E_i) \text{ for } i \in \{1, \ldots, n\}\,, \qquad (2.16)\\
&\text{where } E_i = \left\{ (v_1, v_2) \in E \mid c(v_1, v_2) > \mu + \frac{i}{n-1}(\nu - \mu) \right\},\\
&G = (V, E) \text{ is the base graph,}\\
&c(v_1, v_2) \text{ is the confidence score of the edge } v_1 \text{—} v_2
\end{aligned}
$$

Randomly deleting edges

For unscored networks, we generate a sequence of graphs by repeatedly deleting a random subset of edges. One study [8] proposes four kinds of random error in networks: deleting edges, deleting nodes, adding edges, adding nodes (and connecting them to the graph by random edges); all with a given small percentage of nodes or edges to be affected.

The two adding methods both depend on the particular score distribution for sampling new edges, which cannot easily be generalised to all graphs, hence I use random deletion. Further, the 3 robustness measures I used are based on comparing sets of the highest-ranked nodes, so deleting edges and preserving the set of nodes is more suitable than deleting nodes.[9]

Define an **edge removing** graph generator, producing a sequence of graph, with a proportion of edges to delete δ (deletion rate) at each step:[10]

$$
\begin{aligned}
&{}^{ER}\xi_{\delta}(G) \;\overset{n}{\hookrightarrow}\; G_1, G_2, \ldots, G_n, \qquad (2.17)\\
&\text{where } G_1 = G\,,\\
&\forall i \in \{2, \ldots, n\}\,.\ \forall e \in E.\ \chi_e \sim \mathcal{U}(0,1)\,,\\
&\qquad\qquad G_i = (V_{i-1}, \{\, e \in E_{i-1} \mid \chi_e > \delta \,\})
\end{aligned}
$$

Unlike linear thresholding, edge removing applies to unscored networks. However, in practice, it is useful to specify an *initial threshold* so that this generator can work on scored networks, too, by turning them into unscored first.[11]

[9] Moreover, deleting nodes could introduce more drastic perturbations as randomly deleting a high-degree node is a more extreme change than deleting a low-degree node, thus causing additional bias in robustness.

[10] Section 4.3 shows how to choose a specific δ

[11] Keeping all edges of scored networks and purely discarding the confidence scores would be detrimental to the graph structure, as low-scored edges would make scored networks appear falsely dense.

2.5.3 Evaluating metrics

Given a metric M, a base graph G_0, a graph generator ξ, and the number of perturbed graphs n, let **metric evaluation** refer to the application of a metric to a sequence of generated graphs:

$$M \overset{\xi}{\vdash} G_0 \overset{n}{\hookrightarrow} M(G_1), M(G_2), \ldots, M(G_n), \tag{2.18}$$

$$\text{with} \qquad \xi(G_0) \overset{n}{\hookrightarrow} G_1, G_2, \ldots, G_n$$

Further, define *overall ranking* on a sequence of perturbed graphs that is calculated by first averaging node ranks across all induced rankings and then ranking the averaged values. This will be used in the following robustness definitions.

Definition 2.5.2 (Overall ranking)

*An **overall ranking** A^* of a metric M on n perturbed graphs generated using ξ from a base graph G_0, is defined as:*

$$A^*(M \mid G_0, \xi, n) = A^{\mathrm{Avg}(G_0)},$$

$$\textit{where } \mathrm{Avg}(G): \ \forall v \in V.\ v \mapsto \frac{1}{n}\sum_{i=1}^{n} A^{M(G_i)}(v)\,,$$

$$\textit{where } \xi(G) \overset{n}{\hookrightarrow} G_1, \ldots, G_n\,,$$

$$G = (V, E)$$

In the definition, $A^{M(G_i)}$ is the ranking induced on one generated graph. For each node v, we calculate the average of v's rank across all rankings and call it a "metric" Avg. Then the overall ranking is just the ranking of the locally defined Avg metric.

2.5.4 Robustness measures

I define the 3 robustness measures: rank continuity, rank identifiability, and rank instability. These were proposed in The Paper, however, the original definitions only work for linearly thresholded scored networks. I generalised their definitions and extended them to unscored networks, too.

The Paper argues that in the context of bioinformatics, metrics are often used to identify key nodes of graphs, therefore it is natural to focus on the highest-ranking nodes only. This is also true for networks that are just too large for each node to be inspected individually. Hence, in principle, these robustness measures test how reliably a metric can identify a certain number of highest-ranked nodes across perturbed graphs.

Let us first define k-similarity and α-relaxed k-similarity as in The Paper. Both similarities quantify the correlation between two rankings (similar to Spearman or Kendall rank correlation coefficients) but only take into account the highest-ranked nodes, thus suiting mainly applications where identification of highest-ranking nodes is important.

Definition 2.5.3 (*k*-similarity)

*The **k-similarity** of two rankings A_θ, A_μ is the overlap between their $100k\%$ highest ranking nodes:*

$$\mathrm{Sim}_k(A_\theta, A_\mu) = \frac{|\{\, v \in V^\cap \mid A_\theta(v) > N(1-k) \wedge A_\mu(v) > N(1-k) \,\}|}{Nk}$$

where $k \in (0, 1]$ and $V^\cap$ is the intersection of domains of A_θ, A_μ.
Note that this measure of rank similarity is symmetric.

The following α-relaxed k-similarity accounts for situations where the two rankings are to be interpreted differently (e.g. comparing a perturbed graph's ranking with an overall ranking).

Definition 2.5.4 (α-relaxed *k*-similarity)

*The **α-relaxed k-similarity** of two rankings A_θ, A is the proportion of the $100k\%$ highest ranking nodes in A which are also within $100k\boldsymbol{\alpha}\%$ highest ranking nodes in A_θ:*

$$\mathrm{Sim}_k^\alpha(A_\theta, A) = \frac{|\{\, v \in V^\cap \mid A_\theta(v) > N(1-k\boldsymbol{\alpha}) \wedge A_\mu(v) > N(1-k) \,\}|}{Nk}$$

where $\alpha > 0$, $k, k\alpha \in (0, 1]$ and $V^\cap$ is the intersection of domains of A_θ, A.

For example, if A_θ is obtained by ranking a perturbed graph and A is an overall ranking (of multiple perturbed graphs), relaxed k-similarity may be used to identify whether the top 10 nodes overall (in G), are among the top 15 for the particular observed threshold (in G_θ).

Following are definitions of rank robustness measures, given a metric M, a base graph G, a graph generator ξ, and the number of perturbed graphs to generate n.

Rank continuity is defined as the proportion of pairs of consecutively generated graphs (e.g. thresholded at consecutive values), for which the k-similarity of node rankings is above 0.9.

Definition 2.5.5 (Rank continuity)

$$\begin{aligned}&\mathrm{RankContinuity}\,(M \mid G, \xi, n) \overset{\mathrm{def}}{=} \\ &\qquad \frac{\left|\left\{\, (i,k) \in \{1,\ldots,n-1\} \times \mathcal{K} \mid \mathrm{Sim}_k\left(A^{M(G_i)}, A^{M(G_{i+1})}\right) \geq 0.9 \,\right\}\right|}{(n-1)\,|\mathcal{K}|}\end{aligned} \tag{2.19}$$

where $\mathcal{K} = \{\, 0.001, 0.002, \ldots, 0.05 \,\}$,

$\xi(G) \overset{n}{\hookrightarrow} G_1, G_{i+1}, \ldots, G_n$,

$\mathrm{Sim}_k(\,\cdot\,,\,\cdot\,)$ *is k-similarity of two rankings, by Definition 2.5.3*

Secondly, *rank identifiability* attempts to quantify, how well ranks induced by a metric on perturbed graphs resemble the overall ranking. Taking α-relaxed k-similarity[12] values between

[12]The Paper uses slightly different value of k for each dataset, so in order to generalise this process I set k to be a constant value calculated as the average of the values used in The Paper: $k \approx 0.023478$.

ranks of perturbed graphs and an overall ranking (induced by a metric), rank identifiability is defined as their minimum. The role of the α coefficient is "to allow for more user control when the rankings compared are not interpreted in the same way" [1].

Definition 2.5.6 (Rank identifiability)

$$\text{RankIdentifiability}\,(M \mid G, \xi, n) \stackrel{\text{def}}{=} \min_{i \in \{1,\ldots,n\}} \text{Sim}_k^{\alpha}(A^{M(G_i)}, A^*) \tag{2.20}$$

where A^* *is an overall ranking, by Definition 2.5.2,*

$\xi(G) \overset{n}{\hookrightarrow} G_1, G_{i+1}, \ldots, G_n\,,$

$k \approx 0.023478,$

$\alpha = 1.5,$

$\text{Sim}_k^{\alpha}(\,\cdot\,,\,\cdot\,)$ *is α-relaxed k-similarity of two rankings, by Definition 2.5.3*

Finally, *rank instability* measures how much the rank of each node changes. U is the set of the 1% highest-ranked nodes according to the overall ranking $A^*(M \mid G, \xi, n)$. Then, for each node $v \in U$, we calculate a range of *ranks* that the node attains across the generated graphs. Rank instability is then the average scaled range for the nodes in U.

Definition 2.5.7 (Rank instability)

$$\text{RankInstability}\,(M \mid G, \xi, n) \stackrel{\text{def}}{=} \frac{1}{|U|} \sum_{v \in U} \frac{\text{range}(v)}{n} \tag{2.21}$$

where $U = \{\, v \in V \mid A^*(v) > 99\%n \,\}\,,$

$$\text{range}: \begin{array}{l} V \to \mathbb{R} \\ \quad v \mapsto \max\limits_{i \in \{1,\ldots,n\}} \left(A^{M(G_i)}(v)\right) - \min\limits_{i \in \{1,\ldots,n\}} \left(A^{M(G_i)}(v)\right) \end{array},$$

A^* *is an overall ranking, by Definition 2.5.2,*

$G = (V, E),$

$\xi(G) \overset{n}{\hookrightarrow} G_1, G_{i+1}, \ldots, G_n$

High values of RankContinuity and RankIdentifiability mean high robustness of a metric, whereas high values of RankInstability mean low robustness.

Apart from the robustness measures above, there are other approaches one could investigate. For example, Borgatti et al. [8] suggest "Number of nodes in both the top 10% of the true network and of the observed network, divided by the number of nodes", and "Square of the Pearson correlation between true centralities and observed centralities" as robustness measures – these could be further investigated in future research, or easily embedded in `graffs`, too.

3. Implementation

This chapter describes first the overall structure of the project, its technical requirements and then dives into the implementation of individual modules of `graffs`.

3.1 Overview

The purpose of this project is to develop a methodology and framework, i.e. a tool, to help study graph metrics, and empirically analyse their robustness in particular.

`graffs` is a command-line tool written in Kotlin that can load/store datasets of different formats, generate perturbed graphs, evaluate metrics, and calculate robustness values. Figure 3.1 explains the natural flow of the program, i.e. the *main pipeline* starting with graph datasets and ending up with deductions about each metric's *robustness*.

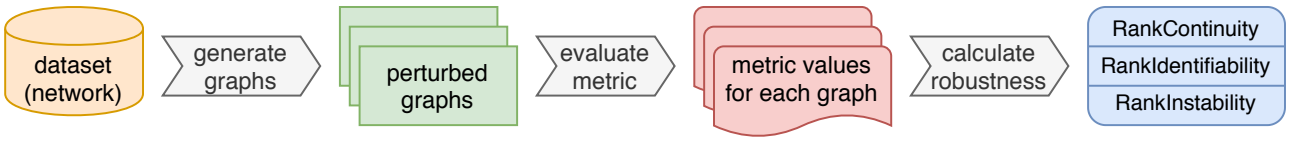


Figure 3.1: A brief illustration of the evaluation process. A generator is used to generate perturbed graphs from an input dataset. A chosen metric is evaluated on all petrurbed graphs, producing per-node values. Finally, each robustness measure's value of the metric is calculated. This is referred to as the "Main pipeline", explained further in fig. 3.6.

3.2 Design goals

The following are technical requirements I set for the project. Overall, I aim this tool to be reusable for similar projects, either by directly invoking the compiled binary, or by using it as a dependency, or by forking and extending it. The target user is a researcher or anyone who would benefit from assessing the stability of graph metrics evaluated on arbitrary graphs.

3.2.1 Supported features

`graffs` supports the following features:

1. Load input graphs in various formats, and represent them in a unified memory structure
2. Run algorithms that compute metrics on graphs
3. Generate graphs by applying perturbations to given input graphs
4. Run experiments by evaluating metrics on generated graphs in a systematic manner
5. Calculate robustness of metrics based on experiments
6. Possibly, produce visual output from the results

3.2.2 Scalability

According to The Paper [1], calculating natural connectivity for a single node for a graph with 7000 nodes takes 88 seconds on a standard computer. In one of my toy examples, calculating average betweenness centrality of 2500 nodes took 8 minutes on my personal computer. Thus, assuming computing a (computation-heavy) metric(s) on an input graph of average size 5000 nodes takes 30 minutes, the pure computation time suggested by the Project Proposal would take the following time on a standard personal computer (approximated in the order of magnitude)

$$(\sim 6 \text{ datasets}) \times (\sim 6 \text{ metrics}) \times (\sim 50 \text{ derived graphs}) \times (\sim 30 \text{ minutes}) \approx 38 \text{ days}$$

For this reason, one of the goals is to make the program efficient and runnable on a supercomputer, utilising the power of high-performance **multi-core systems** for **parallel execution**.

3.2.3 Reproducibility

It is important for results and outcomes in this field to be reproducible. Reproducibility of `graffs` is the guarantee that, given the same input, one can exactly reproduce any results produced by the program including the same robustness values, images, tables, numbers up to a bit-wise match where applicable. `graffs` solves challenges the following areas:

- **Stochastic processes**

  Methods based on stochastic processes or randomness must be reproducible. Examples of stochastic methods are graph generators.

  `graffs` makes these reproducible by generating all randomness starting off with a given seed for the generator.

- **Resolving ties**

  Methods that require pseudo-randomness to resolve ties must also be reproducible. An example is ranking nodes of a graph according to metric values, or generating a visual layout for graphs such as in Figure 2.4. Again, a solution is to base such flow on an input seed.

- **External factors**

  Unpredictability introduced by external factors such as the machine, operating system, thread scheduling, time or other factors, must be accounted for. These issues are resolved using a robust programming language and by appropriately synchronising parts of the program, where this is relevant.

3.2.4 Flexibility

The program must be flexible enough, in particular, the following:

1. Usable on all widely used machines and operating systems
2. Accepting input datasets (and any input parameters) in common formats, in particular, those datasets mentioned in section 2.5.1
3. As a library, it must provide modular access to individual parts of the program, so that it is easy to use `graffs` as a dependency in future projects of a similar kind

3.3 Architecture

I built `graffs` in the Kotlin programming language, with the help of Git and Gradle, and here I explain why.

3.3.1 Kotlin language

Kotlin [47] was chosen mainly for the following reasons.

- Safer, preventing a significant number of errors
- Concise, reducing the amount of boilerplate code
- IDE-friendly, allowing the IDE to help with software engineering
- Employs functional programming paradigms [48], abstracting high-level concepts
- Compiles to Java byte code and so preserves other important benefits of Java: Object-Oriented, multi-threaded, platform-independent, secure and easily extensible.

Kotlin is by nature similar to Java and can be used together with other Java code or libraries in a single project, as Kotlin compiler compiles `.kt` files to Java-bytecode `.class` files. It adopted most concepts from Java, such as classes, polymorphism, inheritance, which I heavily use in the project. Performance-wise, Kotlin is comparable to Java.

There is one notable feature of Kotlin: properties, which are an abstraction of fields and getters/setters in classes and help decouple the implementation of the class from its interface even more. In UML diagrams below, properties are presented instead of fields.

3.3.2 Build automation

The project uses Gradle [49] for project management. Split into different modules, Gradle also helps to keep the structure well defined and manages builds of each module separately.

Project configuration rules are set up using the `build.gradle` files (one in the root directory, then one within each module) with a number of plugins to facilitate the following and more:

1. Defines the structure of the project, such as source and build directories for each module
2. Manages and automatically downloads dependencies
3. Automates the process of code compilation, unit testing and producing deployable `jar`s
4. Generates HTML API documentation from the `KDoc` language[13]

Unit tests are written in the JUnit 5 testing framework (42 tests in total), making sure that the functionality of managing graphs and evaluating metrics does not diverge from what is expected. `graffs` can easily be build from sources by running `gradle build`, which downloads dependencies, compiles modules, runs tests and produces the `cli/build/libs/graffs-last.jar` file that has all necessary libraries bundled and can be run directly.

The commit hash, for example, is used to produce a commit-specific version number when packaging `graffs`.

CircleCI, a continuous integration service, is triggered with each commit to test the build pipeline including tests.

[13] Using the `Dokka` tool, similar to `JavaDoc`

3.3.3 Licensing

In this project I used libraries with different open-source licenses: LGPLv3 (the GraphStream library, described later), LGPLv2.1 (Hibernate ORM), APSL 2.0 (Clikt, Tablesaw, and some Apache libraries `commons-*`), MPLv2.0 (H2), GPLv2 or later (Renjin), MIT (java-express).

A working version of `graffs` is published on GitHub along with its source code under the GPLv3 licence [GPL] (open-source), which is in line with licences of used libraries.

3.3.4 Computing cluster

The `graffs` tool is programmed to allow parallel computing using multiple threads on a single machine. Having experimented with a number of frameworks for high-performance computing, I chose the `kotlinx.coroutines` library [50] (see section 3.6), to make `graffs` portable and easy to set up. It is also straightforward to manually run multiple simultaneous instances of `graffs` at different machines with a shared database, with each instance contributing to a different part of the evaluation.[14]

I used two remote high-performance computing servers[15]:

`rio.cl.cam.ac.uk` running *Ubuntu 18*, using *4x 8-Core AMD Opteron 6128* with 16 MiB L2 cache (32 cores in total), 128 GiB RAM

`nile.cl.cam.ac.uk` running *Ubuntu 18*, using *2x 12-Core AMD Opteron 6168* with 12 MiB L2 cache (24 cores in total), 128 GiB RAM

3.3.5 Project modules

The project is structured in the following modules, using multi-project builds in Gradle:

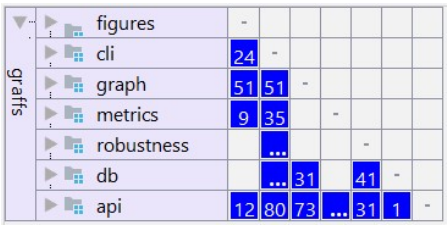


Figure 3.2: Dependency matrix of modules. Cell in the i-th row and j-th column corresponds to the dependency of the j-th module on the i-th module.

- `api` - interfaces of structures, metrics, generators
- `db` - database data model
- `metrics` - graph metrics
- `robustness` - node ranking, rank similarities, robustness measures
- `graph` - graph generators, loading datasets from filesystem
- `cli` - command-line interface: creating and evaluating experiments, visualising graphs, generating rank similarity plots, etc.
- `figures` - code for automating generation of some figures in this report

Figure 3.2 shows inter-module dependencies. An extended dependency matrix in fig. B.2 in appendix B contains individual project files.

[14] Off-the-shelf products such Apache Spark [51], Apache Storm, and Spring Boot have native support for computation across multiple machines, but are heavyweight and require user setup.

[15] sponsored by Dr Andrew Moore, from the System Research Group at the Computer Laboratory, University of Cambridge: `https://www.cl.cam.ac.uk/research/srg/`

3.4 Data model

The main concept is based on the three persisted entities:

- *Graph* storing its nodes, edges (and their attributes)
- *Graph generator* is an object capable of producing a number of graphs, given a dataset.
- *Experiment* is a description of a computational task involving:
  1. generating graphs from given input datasets using a given graph generator
  2. evaluating a given set of metrics on all perturbed graphs
  3. evaluating given robustness measures on all metrics over all datasets

A user can create and manage *graph generators* and *experiments* using the command line interface (see section 3.7). The `graffs` tool uses several libraries to represent graphs in memory, define a data model, and store data in a relational database.

3.4.1 Using GraphStream

Graphs in memory are stored and manipulated by the GraphStream library [52], a "Java library for the modelling and analysis of dynamic graphs".[16] The library consists of 3 modules: `gs-core` defining the API and underlying structures, `gs-algo` with various algorithms from graph theory, and `gs-ui` containing components for visualising graphs. Even though library allows working with dynamic graphs (changing over time), this project only uses static graphs.

The GraphStream library provides a solid base for numerous features in this project such as loading graph files, storing them in various formats, and visualisations. The library also contains an implementation of some graph metrics. In practice, the library's interfaces contain many more links and methods (e.g. for handling directed graphs) that are not relevant in the context of this project.

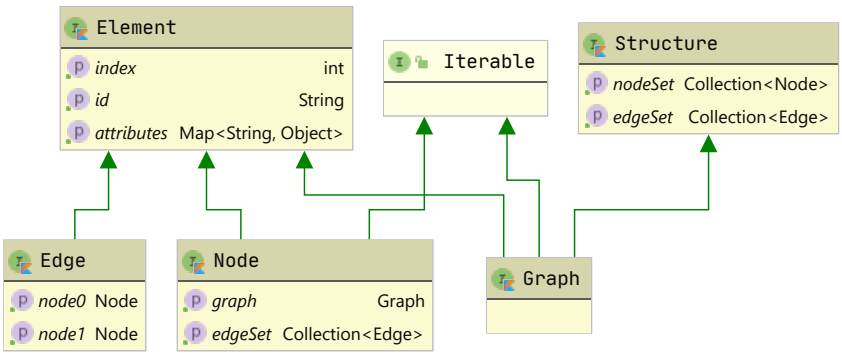


Figure 3.3: A simplified diagram of key interfaces from the GraphStream library

The library is based around the `Graph` interface, which provides access to `Node`s and `Edge`s of each graph. The most relevant interfaces are illustrated in Figure 3.3 (heavily simplified). All `Element`s (i.e. `Node`s, `Edge`s and even `Graph`s) have an `id` field which will be used for matching

[16] As alternatives, I also investigated and considered library JGraphT [53], which is algorithm-focused, however, the algorithms relate mostly to walking on graph nodes and are not relevant for evaluating graph metrics. GraphStream also provides visualisations as opposed to JGraphT.

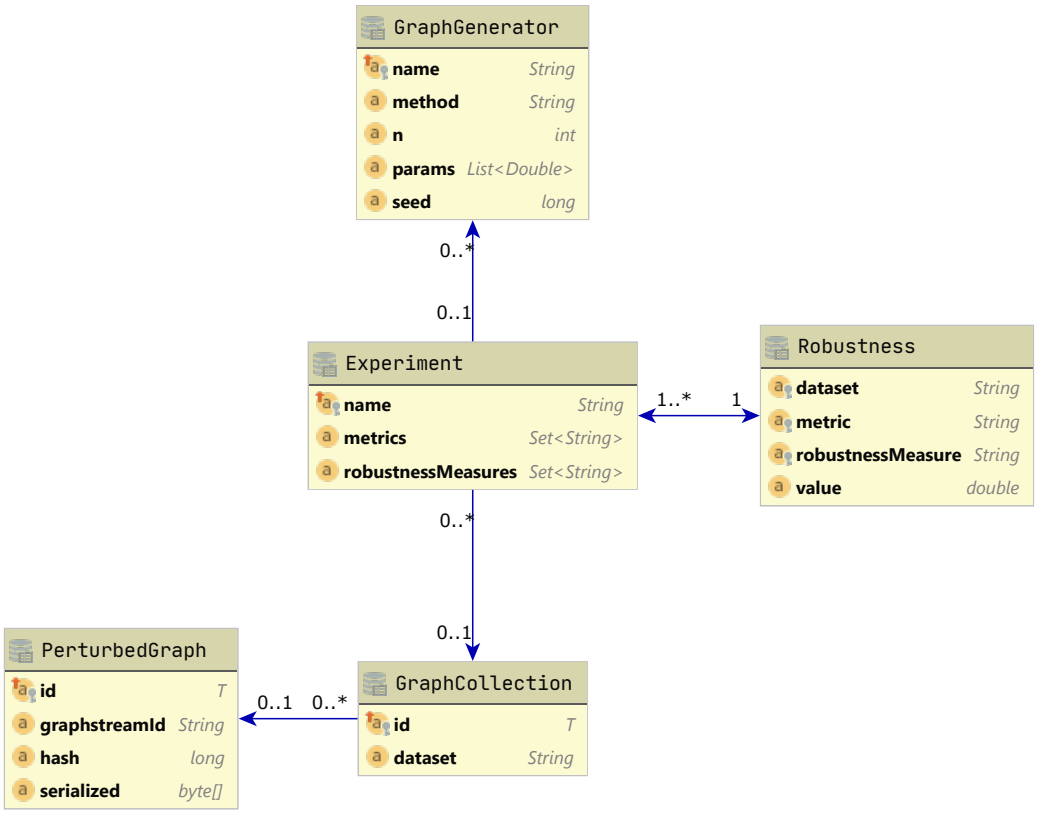


Figure 3.4: Data model diagram showing *persistence schema*, i.e. entities stored in the database. The arrows indicate *association* links, i.e. "has a" or "refers to" relationships. The diagram is created from the Java Persistence API schema inferred from the source code.

corresponding nodes between base and perturbed graphs. The interfaces also provide methods for changing graphs (adding/removing nodes, edges, attributes, etc.).

The `Graph` object from the GraphStream library stores (references to) all nodes and edges, along with their attributes. Calculated metric values of each node, as well as some metadata, is stored as each `Elements`'s attribute, with the attribute key being the metric name.

3.4.2 Relational model

Data such as generated graphs, evaluated metrics, robustness measure results as well as any user-defined hyper-parameters are stored in a relational database system. The Java Persistence API (JPA) [54] provides an abstraction for accessing relational data from Java, Hibernate [55], [56] is a framework that implements the interface. I used specifically the H2 database engine [57] as the underlying storage for Hibernate. This abstraction is later illustrated in Figure 3.5.

Figure 3.4 shows the entities that the program persists in the database, as explained below. Named entities (`Experiment`, `GraphGenerator`) are those that the user creates and later refers to with their name.

`PerturbedGraph` stores a deflated serialised version of the `Graph` object, with its metadata (how it was generated), including all so-far evaluated metric values of nodes, and possibly a weight associated with each edge (for scored networks).

`GraphCollection` represents an (ordered) collection of `PerturbedGraph`s, and keeps track of which datasets the graphs were generated from.

`GraphGenerator` stores user-defined rules for generating graphs from a dataset. The parameters include the `method` to use (such as `linear-thresholding`), number `n` of perturbed graphs to generate from each input dataset, seed, and any additional numeric parameters specific to each generator. One graph generator can be used across multiple experiments.

`Experiment` encapsulates a concept of evaluating multiple *robustness measures* of multiple *metrics* on multiple *datasets*, using a specific graph generator.

`Robustness` stores a single result of evaluating a *robustness measure* of a *metric*, on a set of perturbed graphs originating from a certain *dataset*. Note that the four fields `experiment`, `dataset`, `metric`, `robustnessMeasure` together form the primary key (Figure 3.4).

3.4.3 Java Persistence API

The Java Persistence API [54] (JPA) is an API specification for the management of relational data in Java. It describes ways in Java to specify a schema of a relational database and an interface to manage and access data of a relational model (i.e. entities in tables, relations, first-order logic). *Persistence* is a term referring to accessing, managing, and storing entities.

The Hibernate framework [55], an object-relational mapping tool, provides a concrete implementation of JPA [56]. I use Hibernate as the intermediate layer between the `core` module of `graffs` and the underlying H2 database that is completely abstracted away from the `graffs` code (Figure 3.5).

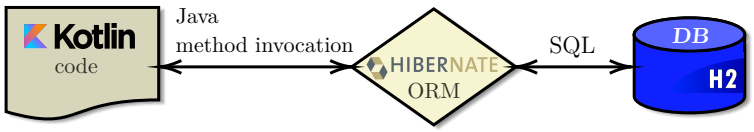


Figure 3.5: Hibernate operates as a middle layer between Kotlin and the underlying H2 database

Further, Figure B.1 shows the underlying *entity classes* written in Kotlin that have the following functions:

→ **Data model definition**, seen in Figure 3.4

JPA provides a number of Java annotations to define entities (`@Entity`), their fields (`@Column`), constraints such as foreign key constraint (`@OneToMany`, `@ManyToOne` and others), and metadata such as rules for fetching data from database (e.g. `@Basic(fetch = FetchType.LAZY)` for lazy fetching of a field). The data model is inferred from these annotations during compile time.

Hibernate then abstracts away operations such as creating and updating the database schema. Listing 3.1 shows the `Experiment` entity, using JPA annotations.

→ **Metamodel generation**

Taking the annotated entity classes during compile time, Hibernate generates a *metamodel* class for each entity to allow writing type-safe queries. For example, the `Experiment_` class is generated from `Experiment`, with corresponding fields (such as `Experiment_.name` of type `SingularAttribute<NamedEntity, String>`, or `Experiment_.generator` of type `SingularAttribute<Experiment, GraphGenerator>`). See Listing B.1 for an illustration how generated metamodel classes aid in writing type-safe queries.

Listing 3.1: The `Experiment` class written in Kotlin. Note especially the annotations which are enough to define the JPA data model.

```kotlin
/**
 * An evaluation of a _set of metrics_ using a set of _robustness measures_, generating
 * graphs from a set of _datasets_ using a given _graph producer_.
 */
@Entity
class Experiment(
        name: String,

        @ManyToOne(fetch = FetchType.EAGER)
        var generator: GraphGenerator,

        @ElementCollection
        @LazyCollection(LazyCollectionOption.FALSE)
        var metrics: List<MetricId> = listOf(),

        @ElementCollection
        @LazyCollection(LazyCollectionOption.FALSE)
        var robustnessMeasures: List<RobustnessMeasureId> = listOf(),

        datasets: Collection<GraphDatasetId> = listOf()
) : NamedEntity(name) {

    /* GraphCollections are created from the [datasets] constructor param */
    @OneToMany(mappedBy = "experiment", cascade = [CascadeType.ALL], orphanRemoval = true)
    @OrderColumn
    @LazyCollection(LazyCollectionOption.FALSE)
    var graphCollections: MutableList<GraphCollection> = datasets.map { GraphCollection(it,
        this) }.toMutableList()

    @OneToMany(mappedBy = "experiment", cascade = [CascadeType.REMOVE], orphanRemoval = true
        )
    @LazyCollection(LazyCollectionOption.FALSE)
    val robustnessResults: MutableList<Robustness> = mutableListOf()

    val datasets get() = graphCollections.map { it.dataset }
}
```

→ **Object-relational mapping**

The classes (Figure B.1) themselves carry the data managed by Hibernate. Other modules such as `robustness` and `cli` can use entity classes, while they are also managed by Hibernate. Hibernate is responsible for loading data into entity objects as well as persisting such objects. See Listing 3.2 for an illustration.

3.4.4 H2 Database

I employed the H2 relational database [57] (based on SQL language) for storing entities, considering the following:

- Very fast; small footprint on the system in terms of installation complexity, program size, and cost of background maintenance
- Easily embeddable, as it implements the Java JDBC API (used by Hibernate, too)
- Supports in-memory databases (good for testing)
- Written purely in Java, so can be bundled in `graffs`, requiring no standalone installation

Listing 3.2: A toy example of Hibernate loading and persisting an `Experiment` object.

```
// Load entity from database
val experiment: Experiment = session.get(Experiment::class.java, "experimentName")

// Modify and persist
experiment.metrics += "Degree"
session.update(experiment)
```

Listing 3.3: An example of a SQL query for retrieving robustness values of the three experiments described in the Evaluation chapter.

```
SELECT DISTINCT EXPERIMENT, DATASET, METRIC, ROBUSTNESSMEASURE, VALUE
FROM ROBUSTNESS
WHERE EXPERIMENT IN ('reproduce', 'random-edges', 'unscored')
```

Graphs are stored in the database as deflated serialised `Graph` objects, including their `Node`s, `Edge`s and their attributes. One an experiment is done, SQL queries such as Listing 3.3 can be used to manually retrieve the data.

3.5 Main pipeline

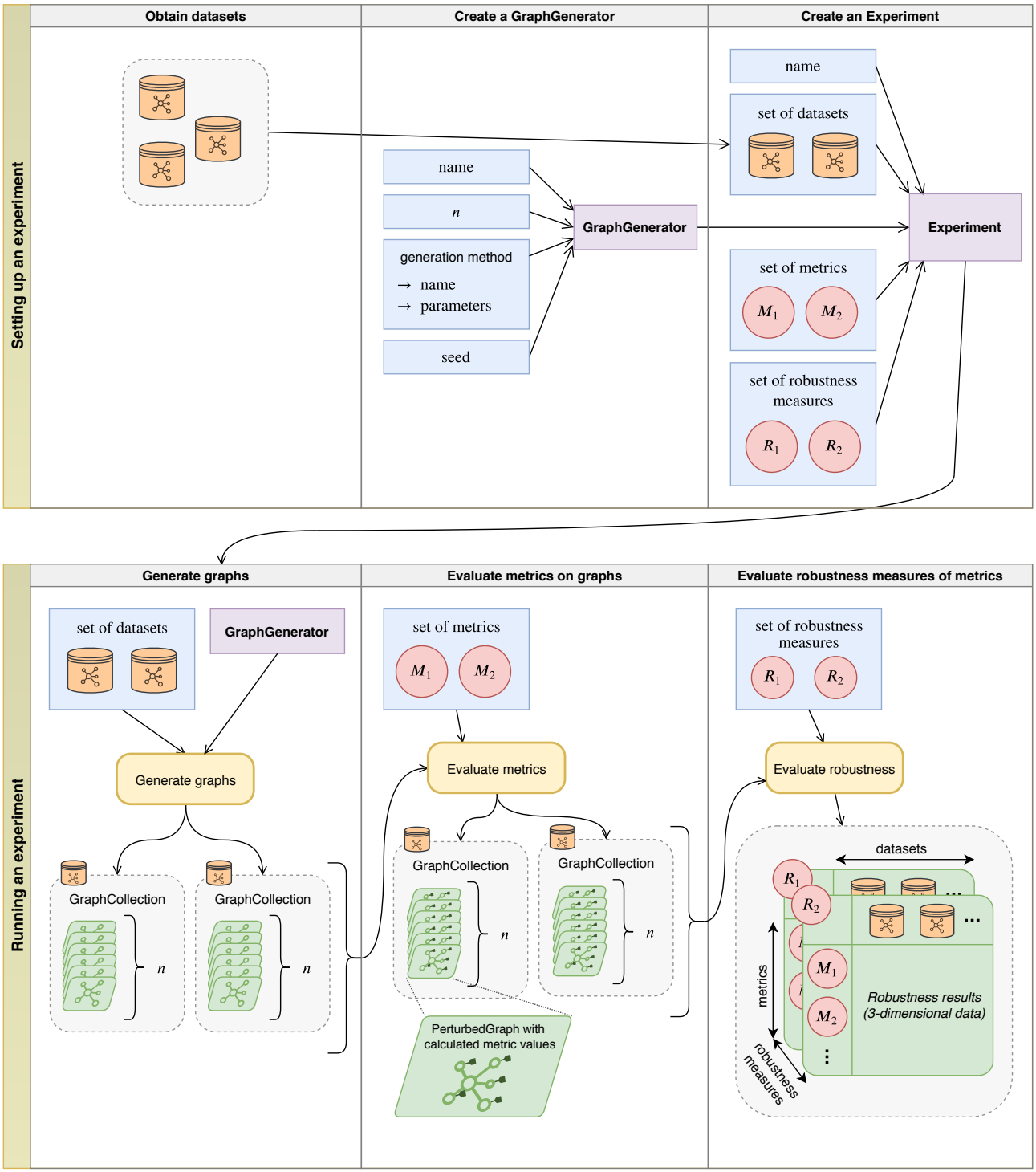
In this section I explain the steps a user needs to invoke to calculate robustness of metrics:

1. **Obtain datasets**, i.e., use demo datasets downloadable by the `graffs` tool or provide custom datasets
2. **Define graph generator**, a way to generate new graphs from these datasets
3. **Define an experiment** specifying a set of datasets, a graph generator to use, a set of metrics, and a set of robustness measures to assess stability of those metrics.
4. **Run the experiment**, which encompasses the following:
    a. **Generate perturbed graphs** (according to subsection 2.5.2).
    b. **Calculate metric values** on the generated graphs, i.e., calculate a real number ($\mathbb{R}$) for each metric for each node in each generated graph (according to subsection 2.5.3).
    c. **Calculate robustness measures** for each metric, on collated perturbed graphs of each dataset (according to subsection 2.5.4).

Running the experiment is a computationally intensive task, which may, depending on the inputs and the environment, run in the order of magnitude of hours. This is mainly due to metric evaluation (which certainly depends on the chosen metrics). To mitigate this, I employed the following:

- Optimised the program for computation speed by applying performance engineering principles
- Allowed parallel computation
- ran experiments on a high-performance computing facility (from subsection 3.3.4)

Figure 3.6 illustrates the flow of data across the above-mentioned steps.



Legend: orange – datasets; blue – user input; violet – user-defined persisted entity; red – program-defined operations; yellow – processes; green – program-generated output; arrows → represent the flow of data.

Figure 3.6: Diagram showing the data flow of an experiment, in two phases: 1. setting up an experiment (the user collects/inputs data), 2. running an experiment (the program is running the computations as described in section 3.5). The result is a 3-dimensional data with a calculated robustness value for each from *datasets*, *metrics*, *robustness measures*.

3.5.1 Loading datasets

`graffs` supports loading graphs from files of two types. In each case, the `GraphLoader` abstract class (Listing 3.4) is implemented.

Edge files – text format, pair of adjacent nodes per line

`RData` files – binary files with objects created in the R language [58]

Datasets are to be located in the 'data' directory in the `graffs`'s working directory, each with a separate subfolder. Datasets may additionally contain an `info.txt` file holding a human-readable description of the dataset.

Listing 3.4: The `GraphLoader` abstract class, providing a contract for implementations to load graphs

```
internal abstract class GraphLoader(private val fileFilter: File.() -> Boolean) {
    fun supportsFile(file: File): Boolean = fileFilter(file)
    abstract fun load(file: File, id: String): Graph
}
```

Edge files

Listing a pair of nodes per line is a common pure format of storing network structure (without any metadata). The set of nodes is implicitly defined by the set of all distinct node identifiers in the file. Figure 3.7 shows sample edge files that are accepted by `graffs`.

Figure 3.7: Three examples of dataset files containing one edge definition per line.

`edges.txt`, a sample graph file

```
0 1
0 2
0 3
0 4
0 5
0 6
0 7
1 2
1 4
1 8
1 9
1 10
...
```

`Cit-HepTh.txt` citation network from Stanford Large Network Dataset Collection[17]

```
# Directed graph (each unordered pair of nodes is saved once): Cit-HepTh.txt
# Paper citation network of Arxiv High Energy Physics Theory category
# Nodes: 27770 Edges: 352807
# FromNodeId ToNodeId
1001 9304045
1001 9308122
1001 9309097
1001 9311042
...
```

`362663.protein.links.v11.0.txt` (Escherichia coli) from the STRING database[18]

```
protein1 protein2 combined_score
362663.ECP_0001 362663.ECP_0006 518
362663.ECP_0001 362663.ECP_0003 606
362663.ECP_0001 362663.ECP_0002 606
362663.ECP_0001 362663.ECP_0005 518
362663.ECP_0001 362663.ECP_0004 606
362663.ECP_0002 362663.ECP_0798 158
362663.ECP_0002 362663.ECP_1284 175
362663.ECP_0002 362663.ECP_1357 170
362663.ECP_0002 362663.ECP_0918 621
362663.ECP_0002 362663.ECP_1773 175
362663.ECP_0002 362663.ECP_2824 342
...
```

To make `graffs` as versatile as possible, I implemented `FileSourceEdgeOptionalWeight` parser, extending GraphStream's `FileSourceEdge` simple LR parser, to allow for the following:

- Modify the grammar to correctly parse node IDs containing dots (`.`), such as `362663.ECP_0001` (to allow datasets from the STRING database)
- Ignore comment lines starting with `#` or `%` characters
- Ignore headers, if present
- Recognise if the edges have weights provided

[17] `https://snap.stanford.edu/data/cit-HepTh.html`

[18] `https://string-db.org/cgi/download.pl?species_text=Escherichia+coli+536`

RData files

The Paper provides the dataset files that were used in their research[19], in the `RData` format, a binary file capturing an R language session, storing objects from the `igraph` R package.

To load such files, an option is to convert them manually into another `graffs`-readable format or let `graffs` call R to do this conversion under the hood. To make `graffs` versatile and not require the R software environment to be installed on the client's computer, I employed the Renjin library[20], a "JVM-based interpreter for the R programming language". Renjin can execute R code (as well as load RData files) within Java, without the need to install any additional platform.

When an RData file is to be loaded in `graffs`, a Renjin engine starts under the hood and loads the desired file. If an object whose name ends with `.df`[21] is found, it is converted into a GraphStream's `Graph` object.

3.5.2 Generating graphs

First, there is the `GraphProducer` interface, which has a single method: `fun produce(sourceGraph: Graph, n: Int): List<Deferred<PerturbedGraph>>`. It returns a list of `Deferred` objects from the `kotlinx.coroutines` library, so that the generation can be parallelised at a higher level in the program using *Kotlin coroutines*, as described in section 3.6. Each `GraphProducer` implementation provides a corresponding `GraphProducerFactory` object, defined as:

```
typealias GraphProducerFactory =
    (seed: Long, params: List<Number>, coroutineScope: CoroutineScope) -> GraphProducer
```

The `method` field of `GraphGenerator`s reference a `GraphProducer` to use, which can be either `threshold-linear` or `removing-edges` (as described in subsection 2.5.2). Implementations of the graph-producing methods are straightforward. Each generated graph is in some way a memory-copied subset of the input graph, with some of the edges deleted.

3.5.3 Metric robustness

Metrics are classes (or rather, Kotlin `object`s) implementing the `Metric` interface, with a single method `fun evaluate(graph: Graph): MetricResult?`. In the database, each `PerturbedGraph` entry stores a single serialised `Graph` object, whose nodes hold values for all metrics. A metric's value at each node is stored in the node's attribute map. An `Experiment` entry (those are created by the user) has a `Set<String>` field specifying which metrics to evaluate on each `PerturbedGraph`, and those metrics are again referenced by a `String` identifier.

Similar to other concepts, robustness measure classes implement the `RobustnessMeasure` interface, with a single method `suspend fun evaluate(metric: MetricInfo, graphCollection: GraphCollection, metadata: GraphCollectionMetadata): Double`. `MetricInfo` uniquely identifies the `Metric` whose robustness to calculate. `GraphCollection` is an entity corresponding to a dataset, with a list of `PerturbedGraph`s.

[19] Available at `http://opig.stats.ox.ac.uk/resources`

[20] Rejin official website: `https://www.renjin.org/`

[21] This is to follow convention from `https://github.com/lbozhilova/measuring_rank_robustness/blob/c39bfed/data_prep.R`

`GraphCollectionMetadata` provides cached[22] access to intermediate results regarding the particular `GraphCollection` in the context of the current `Metric`, such as the *overall ranking* of the given metric over all the perturbed graphs (see Definition 2.5.2). Each `RobustnessMeasure` returns a single `Double` value, which is the numerical result, calculated as described in subsection 2.5.4. The `evaluate` function has the `suspend` modifier so that it can suspend and finish asynchronously within the context of Kotlin coroutines (see section 3.6).

Resolving ties

An important detail when using ranking vectors (e.g. for the overall ranking) is specifying how ties are resolved. According to the definition of ranking vector (definition 2.4.2), if we have $v_1, v_2 \in V$ with $M(v_1) = M(v_2)$ for some metric M, then $\preceq$ is a valid ranking relation with either $v_1 \preceq v_2$ or $v_2 \preceq v_1$. In The Paper, such ties were resolved "at random, independently across different metrics and different thresholds". This may, however, lead to inappropriately lower robustness of metrics due to random oscillations of ranks of nodes, as well as a high degree of variability between results (especially for graphs with a small number of nodes).

In `graffs`, I assured that ties are resolved arbitrarily but deterministically and constantly across all metrics and thresholds, and even constantly across multiple perturbed graphs with the same nodes. This follows the Reproducibility principle set in Design goals.

3.5.4 Visualising graphs & producing figures

This dissertation has charts, diagrams and visualisations of graphs from different sources.

- Plots that depend on computations within the `graffs` program can be exported using the command line interface.
- Other figures relying on the code of `graffs` (such as fig. 2.4) are generated by a custom code in the `figures` module.
- Other, independent plots, such as those in the Evaluation chapter, are generated using Python and SQL, and that code is kept with the LaTeX sources of this dissertation.

3.6 Parallelism using Kotlin coroutines

Kotlin supports the concept of *suspending functions*, i.e., functions which can pause their execution voluntarily (storing its *current continuation*) and return the control of the flow to its caller, and later continue by resuming the continuation at the suspension point. The context in which a suspending function suspends is called *coroutine*. Coroutines are a well-known control abstraction in programming languages [59] that can easily be implemented in higher-order languages using continuations[60]. For example, asynchronous computations, delays, and IO operations can be programmed as suspending calls.

Coroutines can be used even within a single thread (thus allowing a concurrent execution), but the case of `graffs` we are interested in running computation in parallel, to utilise the power of multiple CPU cores.

[22] cached, as the overall ranking is calculated only once for each `GraphCollection` and each `Metric`, and used for more robustness measures

Kotlin coroutines are *lightweight threads* – spawning coroutines is fast and cheap [61], as opposed to Java `Thread`s. They have a similar life cycle and are executed within regular threads, although they are not bound to them [50]. Coroutines are also safer, e.g., a failure of one operation within a *coroutine scope* triggers cancellation of other coroutines in the scope.

Suspending functions in Kotlin, when compiled to Java bytecode, are regular functions accepting `Continuation<Int>` as one of the arguments. The `async` function (or its alternatives) takes a suspending function, creates an asynchronous coroutine executing the function and immediately returns its future result (as `Deferred<T>`). Then the `await` suspending function called on a `Deferred` object waits for the resulting value without blocking and resumes when the deferred computation is complete.

Kotlin also provides other means to synchronise coroutines, such as using mutual exclusion or actors, when needed in parallel execution. Mutexes are used in `graffs` to ensure synchronised access when persisting entities in the database.

The following parts of `graffs` use parallel programming with coroutines:

- Generating graphs
- Evaluating metrics on graphs
- Evaluating robustness, including calculating overall ranks, which must be calculated for each `GraphCollection` before any robustness measure is evaluated on it
- Generating various figures, for which an intermediate

Listing 3.5: A simplified example of creating an asynchronous task for evaluating a set of metrics on a single generated graph.

```
/** Evaluate given metrics on the graph */
fun CoroutineScope.evaluateMetricsAsync(
        metrics: Collection<MetricInfo>,
        graph: Graph,
        callback: suspend (graph: Graph, results: List<MetricResult>) -> Unit
): Deferred<Unit> {
        // A list of metrics they need to be computed in, as some reuse values of others
    val metricsToCompute: List<MetricInfo> = metrics.topologicalOrderWithDependencies()
        return async {
        metricsToCompute
                // An instance of each [Metric] is created by invoking the [factory] field
                .map { metric -> metric.factory() }
                .run {
                // Run evaluation on each graph, clean up auxiliary data
                    val results: List<MetricResult> = map { it.evaluate(graph) }.
                        filterNotNull()
                    forEach { it.cleanup(graph) }
                    results // Return results: flags marking the outcome of each metric
                }
        // Run callback (only if there are any new results)
        .takeIf { it.isNotEmpty() }
        ?.let { callback(graph, it) }
    }
}
```

Listing 3.5 shows an example of creating an asynchronous task to evaluate metrics, although heavily simplified. The full version handles accessing the database, asynchronous deserialisation of `Graph`, logging, measuring time statistics, and preventing duplicate metric evaluation. `Graph`s are treated individually and in parallel, whereas the evaluation of metrics within one `Graph` is sequential as the GraphStream library does not support parallel access.

Figure 3.8: Hierarchy of supported command-line interface commands, and their brief description. The essential commands for running the main pipeline (Figure 3.6) are highlighted in bold.

```
graffs ..................................... Tool for evaluating Graph Metric Robustness
├── dataset ................................ Access available datasets
│   ├── list ............................... List all datasets available in the `data` directory
│   ├── load ............................... Check if datasets can be loaded from the `data` directory
│   ├── download-demos ..................... Download datasets for demonstration
│   └── viz ................................ Visualise graph
├── metric ................................. Access available metrics
│   └── list ............................... List available graph metrics
├── generator .............................. Create or list graph generators
│   ├── list ............................... List available graph generators
│   ├── create ............................. Create a new graph generator description
│   └── remove ............................. Remove a graph generator
├── experiment ............................. Manage experiments (evaluating metrics on generated graphs)
│   ├── list ............................... List created experiments and their properties
│   ├── create ............................. Create an experiment
│   ├── change ............................. Change an existing experiment
│   ├── clone .............................. Create a new experiment using parameters of existing experiment
│   ├── remove ............................. Remove an experiment, all its generated graph and any computed results
│   ├── run ................................ Run an experiment
│   └── prune .............................. Remove generated graphs and calculated robustness values of an experiment
├── plot ................................... Plot results
│   ├── rank-similarity
│   ├── rank-similarity-visual
│   └── relaxed-similarity
└── db ..................................... Manage the underlying database
    ├── test ............................... Test the database connection
    └── drop ............................... Reset the database schema and contents
```

3.7 Command line interface

The `graffs` tool is built as a program with a rich and intuitive command-line interface. The program is pre-packaged in a `.jar` file, built for Java 8 or higher, so running it involves running the shell command:

```
java -jar graffs.jar [ARGS]
```

Clikt is a "multiplatform command-line interface parsing for Kotlin"[23] library that I used for defining commands, their behaviour, description, and hierarchy, in a structured way. The library utilises several features of the Kotlin language, such as delegated properties, extension functions, and default arguments, resulting in a cleaner code that uses the library. With the help of Clikt, I generated scripts allowing automatic command and parameter completion within bash.

The `graffs` tool provides many commands (loading datasets, creating and accessing graph generators, creating and running experiments, and so on). Commands are grouped based on their context and the objects that they operate on, and can further have sub-commands. Those are together structured in a tree-like hierarchy. The hierarchy is shown in Figure 3.8. Listing B.2 shows an example output of some of the commands, with the printed help text. Further, the Evaluation chapter also shows commands that were used to set up each experiment.

[23] Available at `https://github.com/ajalt/clikt`

4. Evaluation

4.1 Success criteria

The project met all its success criteria:

- ☑ Implemented the experimental framework (`graffs`) to automate experiments of robustness of graph metrics
- ☑ Completed statistical analysis and compared results to The Paper
- ☑ Extended the idea from The Paper to unscored networks, and deduced empirical observations about graph metrics on interesting unscored datasets

I also completed one extension:

- ☑ Wrap up `graffs` and release it as a library under an open-source licence

For the evaluation of `graffs`, I designed the following 3 experiments:

`reproduce` focuses on reproducing some results from The Paper, following the setup from The Paper as closely as possible, carried out on the 3 protein interaction networks (`pvivax`, `ecoli`, `yeast`)

`random-edges` validates that random edge deletion is a suitable graph generation method, by applying the same pipeline as in `reproduce` but with random edge deletion

`unscored` applies the random edge deletion method to new, unscored, datasets

4.2 Validation against The Paper

One of the added values of `graffs` is the ability to experiment with *unscored* graphs. However, in order to validate whether the results produced by `graffs` are legit, I first constructed and ran a `reproduce` experiment trying to reproduce results from The Paper, by linearly thresholding 3 scored protein interaction networks in the same way as The Paper.

It is important to note that the robustness values from The Paper and `graffs` should not be compared directly as many details make the numerical values vary between implementations. For example, when ranking nodes according to a metric, The Paper resolved ties randomly for each perturbed graph, while my implementation resolves ties arbitrarily but predictably and constantly across all perturbed graphs of a dataset. The Paper also randomly resolved ties when calculating the overall ranking (Definition 2.5.2). Furthermore, other nuances may cause differences in numerical results, such as α, k coefficients used for k-similarity and α-relaxed k-similarity[24], definitions of graph metrics in special cases (such as for isolated nodes), and floating-point arithmetic. Instead of comparing results numerically, the overall trend is to be considered.

[24]Different values were used in `graffs` to generalise the process. See definitions 2.5.3 and 2.5.4

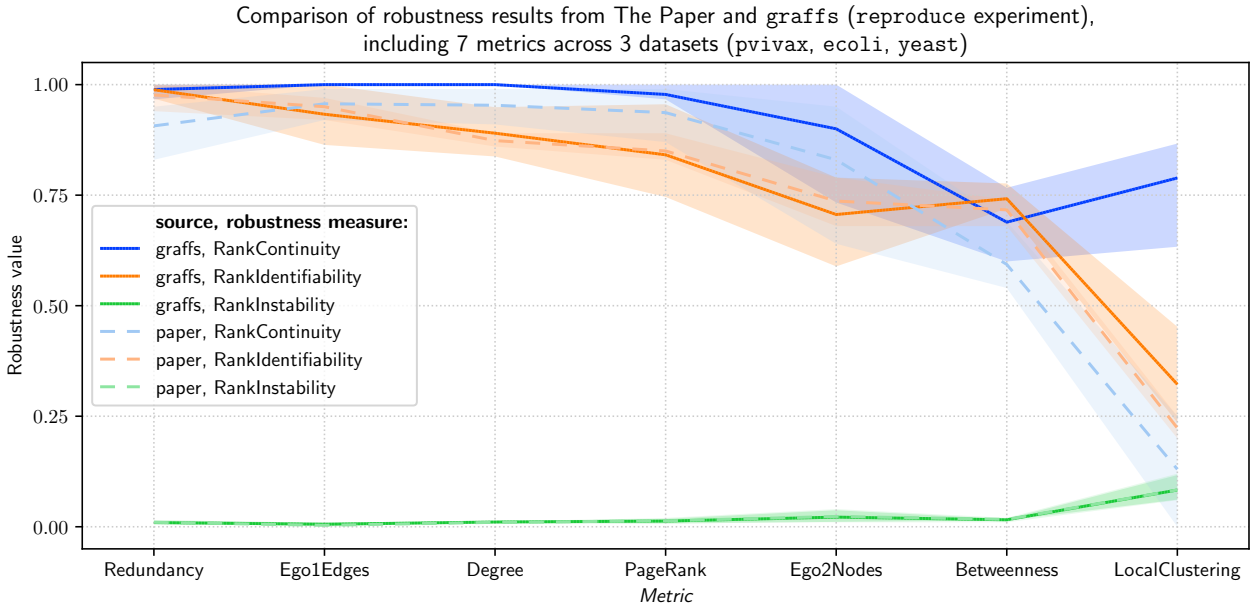


Figure 4.1: Comparison of results from The Paper and `graffs`. Each color corresponds to one robustness measure. Bands show the range of measured values across the 3 datasets, and lines show the average. Solid and dashed bands correspond to results obtained by `graffs` and The Paper, respectively.

High values of RankContinuity and RankIdentifiability mean high robustness, whereas high values of RankInstability mean low robustness. Metrics are sorted from left to right by their decreasing combined robustness, which is calculated just for the purpose of these charts as `RankContinuity` + `RankIdentifiability` − `RankInstability`, averaged across all datasets.

The `reproduce` experiment and the `thMedHigh` generator (producing 31 graphs at linearly spaced thresholds between 0.60 and 0.90 confidence values) were set up in the following way:

```
graffs dataset download-demos
graffs generator create --name thMedHigh --method threshold-linear --params 600,900 -n 31 --
    seed 7
graffs experiment create --name reproduce --datasets pvivax,ecoli,yeast --generator
    thMedHigh --metrics Betweenness,Degree,Ego1Edges,Ego2Nodes,LocalClustering,PageRank,
    Redundancy --robustnessMeasures RankIdentifiability,RankInstability,RankContinuity
graffs experiment run --name reproduce
```

As for the set of metrics, I evaluated all that were also evaluated in The Paper, apart from Closeness and Harmonic centrality (Definitions 2.6 and 2.7). These two depend on the all pair shortest paths algorithm with time complexity $O(|V|^3)$ that would take an unreasonable amount of time to compute on large graphs.

Averaging the robustness values across the 3 protein datasets with similar structure, Figure 4.1 demonstrates that `graffs` is successful in identifying similar robustness properties of metrics as in The Paper. The Paper identifies metrics `Ego1Edges`, `Redundancy`, `Degree`, and `PageRank` as very stable (i.e. robust). Those have reported RankContinuity and RankIdentifiability values above 0.75 on almost all evaluations on the 3 datasets, and RankInstability below ~ 0.1. `graffs` equally reports these metrics as relatively stable.

On the other hand, `Betweenness` and `LocalClustering` are considered unstable by The Paper (RankContinuity and RankIdentifiability significantly dropped while RankInstability increased), and the results from `graffs` copy the behaviour. Although the RankContinuity values of `LocalClustering` are not as low as reported by The Paper, one can conclude that these two metrics are equally identified as less robust by `graffs`.

For further thought, we can also consider the variance of robustness measures, concluding, for example, that `Redundancy` (a stable metric) has small variance in RankIdentifiability across the 3 datasets as reported by `graffs`, whereas the RankContinuity and RankIdentifiability of the `Ego2Nodes` metric led to more diverse values.

4.2.1 Rank similarity

Another way to validate intermediate results is to consider k-similarity of perturbed graphs between two consecutive thresholds, and α-relaxed k-similarity between graphs at individual thresholds and overall ranks. For each dataset `pvivax`, `ecoli`, `yeast`, I generated 85 graphs thresholded at values between 0.15 and 0.99, and calculated the needed metric values for each graph. Then I compare the results for 3 chosen metrics on the protein network datasets with the results of The Paper.

Figure 4.4 shows k-similarity of each pair of graphs at consecutive thresholds, along with plots of the same data from The Paper. The k-similarity values are used for calculating `RankContinuity`, which is equal to the proportion of such pairs of consecutive graphs whose k-similarity is above the 0.9 threshold (Definition 2.5.5).

Figure 4.5 shows α-relaxed k-similarity between overall ranking and rankings of graphs at individual thresholds. The α-relaxed k-similarity is used for calculating `RankIdentifiability`, which is equal to the minimum similarity value within the $[0.6, 0.9]$ confidence interval (Definition 2.5.6).

The visual plots of the three chosen metrics in each case demonstrate that `graffs`'s similarity values match those of The Paper closely, in both k-similarity and α-relaxed k-similarity.

4.3 Validation of random edge deletion

Randomly deleting a small subset of edges allows us to evaluate the robustness of metrics on unscored graphs. The purpose of the `random-edges` experiment is to justify reliability and accuracy of this approach, comparing robustness results of the same metrics, on the same datasets, but with a different graph-generating method.

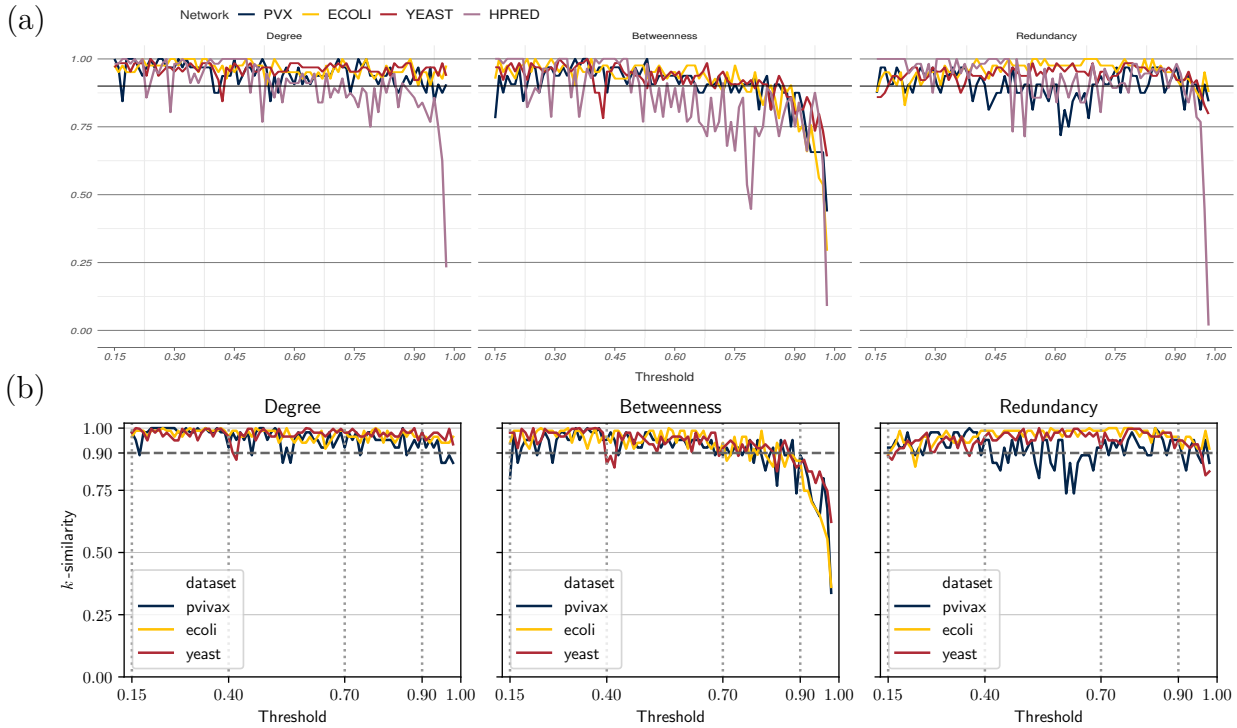


Figure 4.4: k-similarity plots between ranks of graphs derived from protein networksfig. 4.5 at consecutive thresholds, taking results from The Paper (a) and `graffs` (b).fig. 4.5

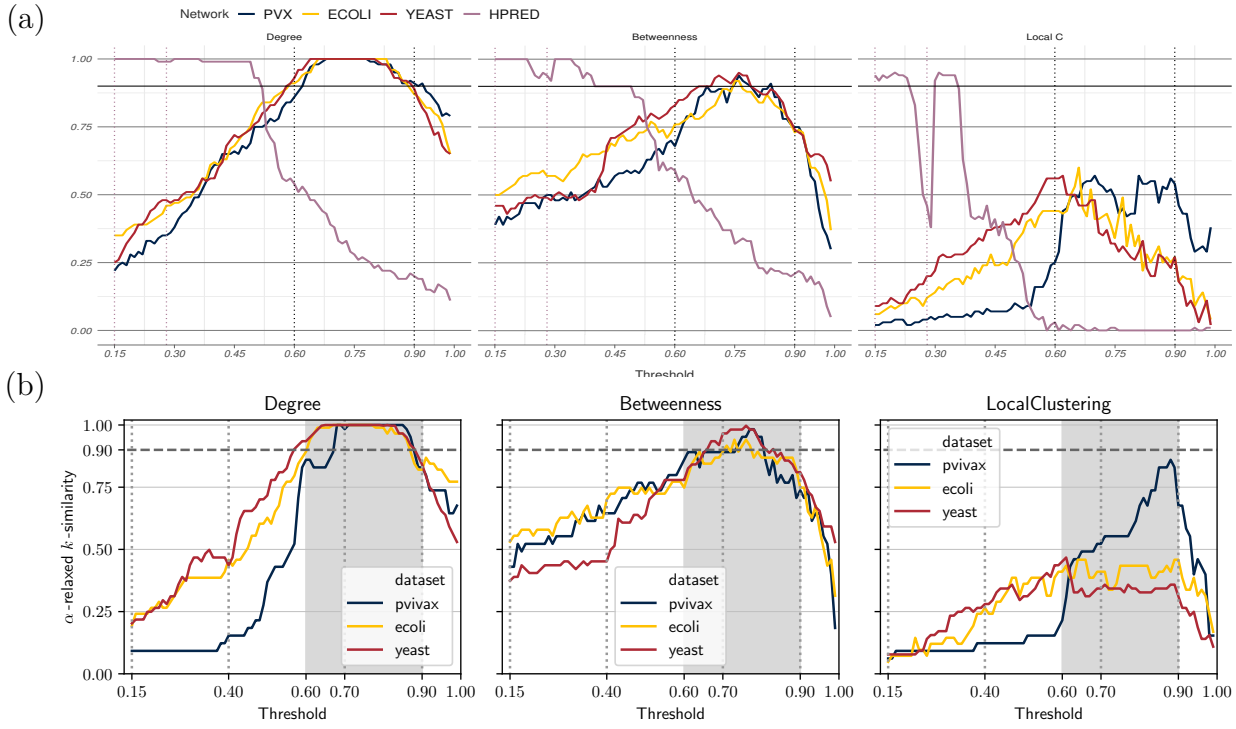


Figure 4.5: α-relaxed k-similarity plots between overall rank and ranks at individual thresholds, taking results from The Paper (a) and `graffs` (b) on protein networks[25]. The overall rank for each of 3 metrics was calculated within the "wide medium-high confidence region" $[0.6, 0.9]$.[26]

[25]The HPRED dataset is not used in this dissertation and so is irrelevant

[26]Coefficients were set to $k = 0.01, \alpha = 1.5$, as in the source code of experiments done in The Paper: `https://github.com/lbozhilova/measuring_rank_robustness`

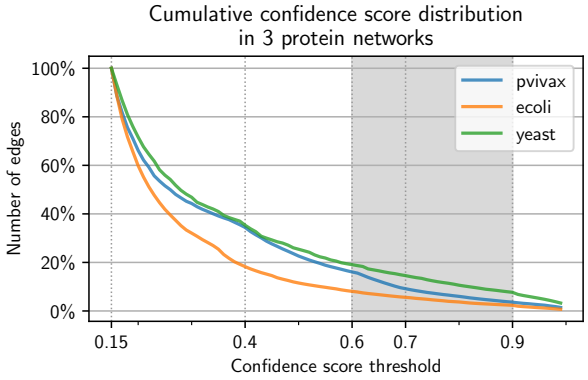


Figure 4.2: Cumulative (right-to-left) distribution of the confidence scores of edges in 3 scored protein interaction networks. The decrease rate in the highlighted interval $[0.6, 0.7]$ is used to determine the average ratio of deleted edges at each threshold, seen in Figure 4.3.

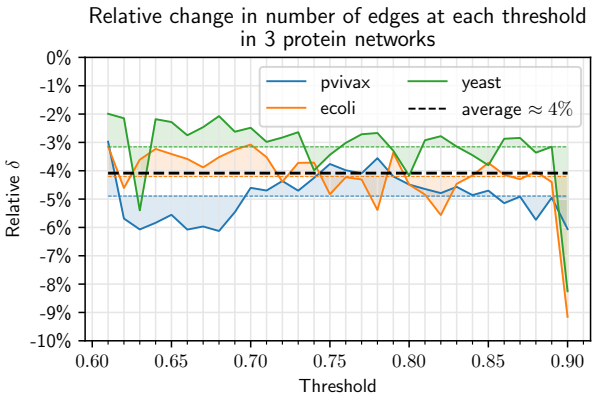


Figure 4.3: Relative change in the number of edges at each confidence threshold, in 3 protein networks. Thresholds are spaced at 0.01 intervals. Overall, around 4% edges are deleted at each threshold.

To start, I needed to choose the δ parameter for ${}^{ER}\xi_{\delta}$, the edge-removing graph generator (see Equation 2.17). Considering the distribution of confidence scores (Figure 2.9), I derived its right-to-left cumulative distribution (Figure 4.2) and differentiated to get a decrease rate at each threshold, as seen in Figure 4.3. As calculated, the relative amount of edges deleted when increasing the threshold in 0.01 increments (as practised in The Paper), is 4% on average. Thus, I set $\delta = 0.04$ (proportion of edges to delete) in the random edge deletion algorithm.

To make this graph generator work on unscored graphs comparably to scored graphs as in the previous experiment, I also set the *initial threshold* of each scored graph to the confidence of 0.6.

The experiment was set up using the following commands:

```
graffs generator create --name random04 --method removing-edges --params 0.04,600 -n 31 --
    seed 7
graffs experiment create --name random-edges --datasets pvivax,ecoli,yeast --generator
    random04 --metrics Betweenness,Degree,Ego1Edges,Ego2Nodes,LocalClustering,PageRank,
    Redundancy --robustnessMeasures RankIdentifiability,RankInstability,RankContinuity
graffs experiment run --name random-edges
```

Results are shown in Figure 4.6. We can see that all 3 robustness measures follow the expected behaviour for the two less-stable metrics `Ego2Nodes` and `LocalClustering`. However, RankContinuity is not able to mark `Betweenness` as less stable, and even shows zero variance for `Betweenness`.

RankContinuity is different from the other two metrics, in that it explicitly uses perturbed graphs in a sequence, considering differences between pairs of consecutive graphs. The different behaviour of RankContinuity can perhaps be explained by that the `random-edges` graph generator removes edges uniformly at each step, whereas `threshold-linear` removes edges according to a non-uniform distribution of edge scores.[27]

Note again, that the results of the experiments are not expected to match numerically, due to numerous changes and generalisations in `graffs` as opposed to The Paper, only to

[27] In protein networks used in this experiment, edges likely do not have all of their scores independent of each other, but rather dependent on local proximity of each edge.

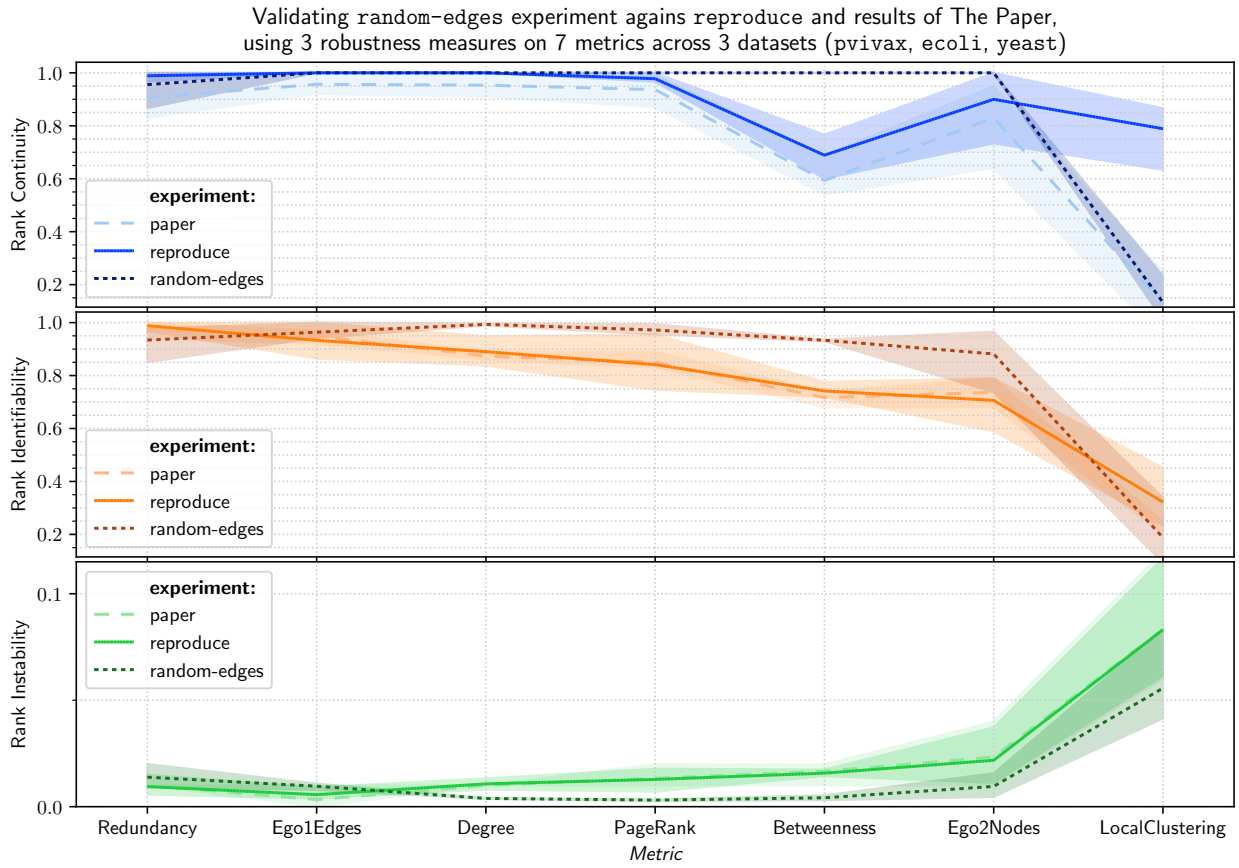


Figure 4.6: Validation of the `random-edges` experiment against `reproduce` and results of The Paper, using 3 robustness measures (displayed in rows) on 7 metrics across 3 datasets. Each color corresponds to a robustness measure. Bands show the range of measured values across the 3 datasets, and lines show the average. Each line style (solid, long dashes, short dashes) corresponds to results obtained from different experiment or source.

High values of RankContinuity and RankIdentifiability mean high robustness, whereas high values of RankInstability mean low robustness. Metrics are sorted from left to right by their decreasing combined robustness.

yield similar behaviour (as explained in section 4.2). Overall, the results show that randomly removing edges is a suitable substitute for linear thresholding.

4.4 Extending to unscored datasets

Having validated that randomly removing edges is a reasonable method for generating small perturbations in unscored graphs, we can now proceed to evaluating the robustness of multiple metrics on multiple interesting datasets. For this, I used the 5 unscored datasets (described in Figure 2.5.1): `airports`, `citation`, `collab`, `facebook`, `internet`.

The experiment is set up as follows:

```
graffs generator create --name random04-unscored --method removing-edges --params 0.04 -n 31
    --seed 7
graffs experiment create --name unscored --generator random04-unscored \
    --datasets airports,citation,collab,facebook,internet \
    --metrics Betweenness,Degree,Ego1Edges,Ego2Nodes,LocalClustering,PageRank,Redundancy \
    --robustnessMeasures RankIdentifiability,RankInstability,RankContinuity
graffs experiment run --name unscored
```

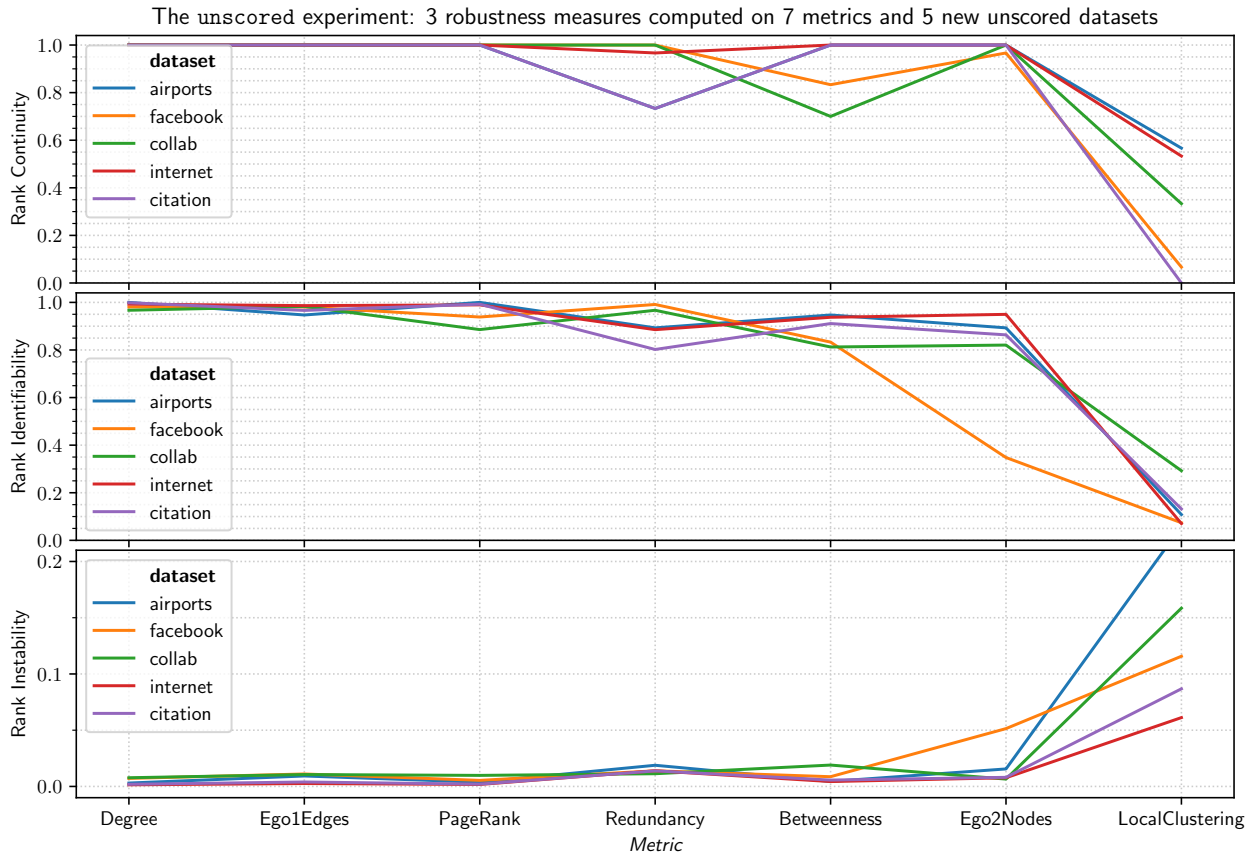


Figure 4.7: Evaluating RankContinuity, RankIdentifiability and RankInstability on 5 new unscored datasets, generating graphs by randomly deleting 4% of edges.

High values of RankContinuity and RankIdentifiability mean high robustness, whereas high values of RankInstability mean low robustness. Metrics are sorted from left to right by their decreasing combined robustness.

Figure 4.7 shows 3 robustness measures evaluated on the 7 metrics using 5 unscored datasets. Overall, the tendency of robustness measures identify the key results from previous experiments: metrics `Degree`, `Ego1Edges`, and `PageRank` are mostly stable on all datasets, while `LocalClustering` is drastically less stable on all datasets.

Tables C.1 to C.3 included in appendix C show the numerical values of all the results presented here and in the previous `random-edges` experiment.

There are several interesting observations regarding the new datasets, some of them posing questions difficul to answer without further research.

1. While `Redundancy` was the most stable metric in the previous two experiments, it is now reported as much less stable due to lower values in the `citation` dataset. What special is there about this dataset?

2. `RankIdentifiability` robustness of the `Ego2Nodes` metric is much lower specifically in the `facebook` (value 0.35, while other datasets are > 0.8). One of the explanations might be by comparing this metric with `Degree`. An impact of a person with many connections will likely not decline steeply when some of the connections are lost. However, considering the ego -2 network, one may loose a lot of impact when a few of the most important connections (to *influential people*) are lost.

Table 4.1: CPU Computation time of the 3 experiments evaluated by `graffs`, run on the `rio` computing cluster (see subsection 3.3.4). *Total CPU time* is the sum of all times of individual CPU cores spent on evaluating the experiment, and *Avg CPU time per graph* is that divided by (number of datasets) × (number of graphs generated from each dataset).

Experiment	Total CPU time	Avg CPU time per graph
`reproduce`	13 hours, 28 mins	9 mins
`random-edges`	14 hours, 28 mins	9 mins
`unscored`	6 days, 4 hours, 39 mins	58 mins

3. It is interesting to note that the variance of robustness values across datasets is different for each metric. In other words, for some metrics the measured robustness values are more *predictable* than for others. Knowing this variance of metrics for a larger number of datasets would be useful for assessing the reliability of robustness results themselves.

4. `RankContinuity` measure of `Degree` and `Ego1Nodes` is > 0.995 for *all* of the datasets. This means that the measure might not be *critical* enough to observe tangible differences of ranks within pairs of consecutively generated graphs. For future experiments, there are a number of parameters that can be tuned: increasing δ, changing the set $\mathcal{K}$, or redefining `RankContinuity` with a higher number substituted for 0.9 (definition 2.5.5), can all lead to the measure being more critical towards changing metric values.

An provocative thought for the future is finding a correlation between the robustness of metrics on graphs and their structure, or the values of some other graph metrics. Ideally, one would be able to reason about the robustness of a graph metrics in a graph purely based on the values of some other metrics in the graph (which themselves may be stable or unstable, too), without the need to simulate graph perturbations. My work does not include and was not expected to include this kind of analysis due to the limits of its extent – having enough of data finding correlations between metrics values and robustness of metrics would require an order of magnitude more evaluations. However, the `graffs` tool I developed is powerful enough to be able to facilitate this kind of analysis, given more time, computing power and resources for further investigation, and it can easily and modularly extended to work with new concepts.

4.5 Performance

Table 4.1 compares the time it took to evaluate the experiments: `reproduce`, `random-edges`, `unscored`; and the time spent on evaluating each graph. The performance of the experiments was measured on the `rio` computing cluster (see subsection 3.3.4). The table measures total CPU times, therefore assuming that n cores run in parallel and the machine executes no other processes in the meantime, the real (wall-clock) duration of each experiment is then n-times shorter. This value is useful to compare the *absolute computational complexity* of each experiment, as it does not depend on the number of cores, and neither the fraction of the program that is perfectly parallelised.

Figure 4.8 compares the logarithm of the time it took to evaluate each metric on a single graph, in average, taking measurements from the 3 experiments `reproduce`, `random-edges`, and `unscored`. We can see that the betweenness centrality is most computation-heavy, with

the computation taking more than an hour on average for a single graph generated from the `citation` dataset. `internet` is similarly computation-heavy, but with different relative complexities of metrics. These hold presumably because the `citation` and `internet` datasets are much larger in the number of nodes and edges compared to the other used datasets.

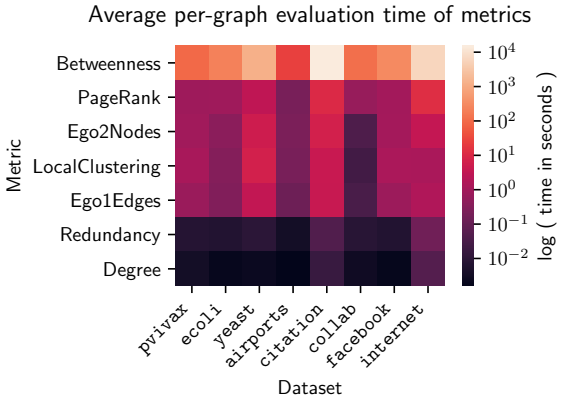


Figure 4.8: Per-graph evaluation time of metrics, averaged over graphs generated from various datasets

5. Conclusion

This project was successful, meeting its criteria. I designed a way to analyse the robustness of graph metrics on unscored networks. First I followed the approach from The Paper by Bozhilova et al. [1] and built a tool called `graffs` that is capable of carrying out similar kinds of experiments of evaluating rank robustness of scored networks, such as protein interaction networks. Further, building on these ideas, I advanced the concept of rank robustness and made it applicable to unscored networks too, by defining a way to generate multiple graphs from a source dataset by deleting a random proportion of edges each time.

As for the implementation, the command-line tool `graffs` is built from scratch in Kotlin, making use of existing Java libraries. It allows large-scale experiments in an efficient, flexible, and reproducible way. Computation-heavy parts of the program are parallelisable using Kotlin coroutines, so that computations can run on a supercomputer, utilising the power of high-performance multi-core systems. `graffs` is published as open-source, ready to be reused and extended by future projects.

Finally, I set up a high-performance computing environment and reproduced expected results from The Paper, as well as concluded new results of metric robustness on unscored datasets, which mostly follow those on scored protein networks.

Lessons learned

The project allowed me to investigate not only the approaches commonly practised, but also the reasoning why robustness measures and related ideas are important. Apart from the field of bioinformatics and graph theory, building `graffs` taught me a lot from the software engineering field. I improved my skills in technologies Kotlin, Gradle, as well as Python and LaTeX. Especially interesting was learning the Java Persistence API and the Hibernate framework.

Future work

In this novel field of measuring the robustness of metrics, there are many questions yet to be answered and approaches to be explored. One can extend the concept of metric robustness to different kinds of graphs (e.g. temporal), reason about different ways to perturb graphs and use different robustness measures such as calculating Pearson correlations between metric values. Several papers in the field of bioinformatics proposed evaluating the reproducibility of graph metrics in uncertain brain networks using the coefficient of variation, intra-class correlation and other statistical measures. These could be extended to (unscored) datasets of other kinds, studying which robustness measures suit each purpose best.

Implementation-wise, `graffs` is built in a modular way and can easily be extended to include other metrics and robustness measures. It would be beneficial to dive deep into inspecting performance bottlenecks of the program to optimise them or use faster algorithms where appropriate. Alternatively, it should be relatively easy to turn `graffs` into a distributed program that can run synchronously across multiple machines.

Bibliography

[1] L. V. Bozhilova, A. V. Whitmore, J. Wray, G. Reinert, and C. M. Deane, "Measuring rank robustness in scored protein interaction networks," *BMC Bioinformatics*, vol. 20, no. 1, Dec. 2019. DOI: 10.1186/s12859-019-3036-6.

[2] C. Eisenhart, "Expression of the Uncertainties of Final Results: Clear statements of the uncertainties of reported values are needed for their critical evaluation," *Science*, vol. 160, no. 3833, pp. 1201–1204, Jun. 14, 1968, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.160.3833.1201. pmid: 17818734.

[3] L. R. Foulds, *Graph Theory Applications*. Springer Science & Business Media, Dec. 6, 2012, 389 pp., ISBN: 978-1-4612-0933-1. Google Books: 5G4QBwAAQBAJ.

[4] D. Curran-Everett, "Explorations in statistics: Standard deviations and standard errors," *Advances in Physiology Education*, vol. 32, no. 3, pp. 203–208, Sep. 1, 2008, ISSN: 1043-4046. DOI: 10.1152/advan.90123.2008.

[5] D. J. Wang, X. Shi, D. A. McFarland, and J. Leskovec, "Measurement error in network data: A re-classification," *Social Networks*, 2012, ISSN: 03788733. DOI: 10.1016/j.socnet.2012.01.003.

[6] P. V. Marsden, "Network Data and Measurement," *Annual Review of Sociology*, vol. 16, no. 1, pp. 435–463, 1990. DOI: 10.1146/annurev.so.16.080190.002251.

[7] D. K. Jones, "Challenges and limitations of quantifying brain connectivity in vivo with diffusion MRI," *Imaging in Medicine*, vol. 2, no. 3, pp. 341–355, Jun. 5, 2010, ISSN: 1755-5191.

[8] S. P. Borgatti, K. M. Carley, and D. Krackhardt, "On the robustness of centrality measures under conditions of imperfect data," *Social Networks*, vol. 28, no. 2, pp. 124–136, May 1, 2006, ISSN: 0378-8733. DOI: 10.1016/j.socnet.2005.05.001.

[9] M. J. Vaessen, P. A. M. Hofman, H. N. Tijssen, A. P. Aldenkamp, J. F. A. Jansen, and W. H. Backes, "The effect and reproducibility of different clinical DTI gradient sets on small world brain connectivity measures," *NeuroImage*, vol. 51, no. 3, pp. 1106–1116, Jul. 1, 2010, ISSN: 1053-8119. DOI: 10.1016/j.neuroimage.2010.03.011.

[10] E. L. Dennis, N. Jahanshad, A. W. Toga, K. L. McMahon, G. I. de Zubicaray, N. G. Martin, M. J. Wright, and P. M. Thompson, "Test-Retest Reliability of Graph Theory Measures of Structural Brain Connectivity," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2012*, N. Ayache, H. Delingette, P. Golland, and K. Mori, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2012, pp. 305–312, ISBN: 978-3-642-33454-2. DOI: 10.1007/978-3-642-33454-2_38.

[11] Q. K. Telesford, J. H. Burdette, and P. J. Laurienti, "An exploration of graph metric reproducibility in complex brain networks," *Frontiers in Neuroscience*, vol. 7, 2013, ISSN: 1662-453X. DOI: 10.3389/fnins.2013.00067.

[12] T. Martin, B. Ball, and M. E. Newman, "Structural inference for uncertain networks," *Physical Review E*, vol. 93, no. 1, Jan. 15, 2016, ISSN: 24700053. DOI: 10.1103/PhysRevE.93.012306.

[13] M. E. Newman, "Network structure from rich but noisy data," *Nature Physics*, vol. 14, no. 6, pp. 542–545, Jun. 1, 2018, ISSN: 17452481. DOI: 10.1038/s41567-018-0076-1.

[14] M. Drakesmith, K. Caeyenberghs, A. Dutt, G. Lewis, A. S. David, and D. K. Jones, "Overcoming the effects of false positives and threshold bias in graph theoretical analyses of neuroimaging data," *NeuroImage*, vol. 118, pp. 313–333, Sep. 1, 2015, ISSN: 10959572. DOI: 10.1016/j.neuroimage.2015.05.011.

[15] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, no. 6833, pp. 41–42, 6833 May 2001, ISSN: 1476-4687. DOI: 10.1038/35075138.

[16] X. He and J. Zhang, "Why Do Hubs Tend to Be Essential in Protein Networks?" *PLoS Genetics*, vol. 2, no. 6, Jun. 2006, ISSN: 1553-7390. DOI: 10.1371/journal.pgen.0020088. pmid: 16751849.

[17] D. Szklarczyk, A. L. Gable, D. Lyon, A. Junge, S. Wyder, J. Huerta-Cepas, M. Simonovic, N. T. Doncheva, J. H. Morris, P. Bork, L. J. Jensen, and C. Von Mering, "STRING v11: Protein-protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets," *Nucleic Acids Research*, vol. 47, no. D1, pp. D607–D613, Jan. 8, 2019, ISSN: 13624962. DOI: 10.1093/nar/gky1131. pmid: 30476243.

[18] F. Menczer, S. Fortunato, and C. A. Davis, *A First Course in Network Science*. Oxford United Kingdom: Oxford University Press, 2020, 288 pp., ISBN: 978-0-19-872646-3.

[19] J. Martín Hernández and P. V. Mieghem, "Classification of graph metrics," 2011.

[20] M. Marchiori and V. Latora, "Harmony in the small-world," *Physica A: Statistical Mechanics and its Applications*, vol. 285, no. 3-4, pp. 539–546, Oct. 2000, ISSN: 03784371. DOI: 10.1016/S0378-4371(00)00311-3.

[21] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, Jun. 1, 1962, ISSN: 0001-0782. DOI: 10.1145/367766.368168.

[22] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[23] R. Seidel, "On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs," *Journal of Computer and System Sciences*, vol. 51, no. 3, pp. 400–403, Dec. 1, 1995, ISSN: 0022-0000. DOI: 10.1006/jcss.1995.1078.

[24] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 6684 Jun. 1998, ISSN: 1476-4687. DOI: 10.1038/30918.

[25] S. P. Borgatti, "Structural holes: Unpacking burt's redundancy measures," *Connections*, vol. 20, no. 1, pp. 35–38, 1997.

[26] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, Proceedings of the Seventh International World Wide Web Conference, vol. 30, no. 1, pp. 107–117, Apr. 1, 1998, ISSN: 0169-7552. DOI: 10.1016/S0169-7552(98)00110-X.

[27] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web.," Stanford InfoLab / Stanford InfoLab, Technical report 1999-66, Nov. 1999.

[28] Y. López, K. Nakai, and A. Patil, "HitPredict version 4: Comprehensive reliability scoring of physical protein–protein interactions from more than 100 species," *Database: The Journal of Biological Databases and Curation*, vol. 2015, Dec. 21, 2015, ISSN: 1758-0463. DOI: 10.1093/database/bav117. pmid: 26708988.

[29] S. Orchard, M. Ammari, B. Aranda, L. Breuza, L. Briganti, F. Broackes-Carter, N. H. Campbell, G. Chavali, C. Chen, N. del-Toro, M. Duesbury, M. Dumousseau, E. Galeota, U. Hinz, M. Iannuccelli, S. Jagannathan, R. Jimenez, J. Khadake, A. Lagreid, L. Licata, R. C. Lovering, B. Meldal, A. N. Melidoni, M. Milagros, D. Peluso, L. Perfetto, P. Porras, A. Raghunath, S. Ricard-Blum, B. Roechert, A. Stutz, M. Tognolli, K. van Roey, G. Cesareni, and H. Hermjakob, "The MIntAct project—IntAct as a common curation platform for 11 molecular interaction databases," *Nucleic Acids Research*, vol. 42, no. D1, pp. D358–D363, Jan. 1, 2014, ISSN: 0305-1048. DOI: 10.1093/nar/gkt1115.

[30] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford Large Network Dataset Collection*, 2014.

[31] J. Leskovec and R. Sosic, "SNAP: A General Purpose Network Analysis and Graph Mining Library," Jun. 23, 2016. arXiv: 1606.07550 [physics].

[32] J. Kunegis, "KONECT - The koblenz network collection," in *WWW 2013 Companion - Proceedings of the 22nd International Conference on World Wide Web*, 2013, pp. 1343–1350, ISBN: 978-1-4503-2038-2.

[33] R. Rossi and N. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, Mar. 4, 2015.

[34] *US airports network dataset – KONECT*, Sep. 2016.

[35] T. Opsahl. (2011). "Why anchorage is not (that) important: Binary ties and sample selection," [Online]. Available: http://wp.me/poFcY-Vw.

[36] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05, Chicago, Illinois, USA: Association for Computing Machinery, Aug. 21, 2005, pp. 177–187, ISBN: 978-1-59593-135-1. DOI: 10.1145/1081870.1081893.

[37] J. Gehrke, P. Ginsparg, and J. Kleinberg, "Overview of the 2003 KDD Cup," *ACM SIGKDD Explorations Newsletter*, vol. 5, no. 2, pp. 149–151, Dec. 1, 2003, ISSN: 1931-0145. DOI: 10.1145/980972.980992.

[38] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Transactions on Knowledge Discovery from Data*, vol. 1, no. 1, 2–es, Mar. 1, 2007, ISSN: 1556-4681. DOI: 10.1145/1217299.1217301.

[39] J. Leskovec and J. J. Mcauley, "Learning to discover social circles in ego networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 539–547.

[40] B. Zhang, R. Liu, D. Massey, and L. Zhang, "Collecting the Internet AS-level topology," *SIGCOMM Computer Communication Review*, vol. 35, no. 1, pp. 53–61, 2005.

[41] *Internet topology network dataset – KONECT*, Sep. 2016.

[42] D. Meunier, S. Achard, A. Morcom, and E. Bullmore, "Age-related changes in modular organization of human brain functional networks," *NeuroImage*, vol. 44, no. 3, pp. 715–723, Feb. 1, 2009, ISSN: 1053-8119. DOI: 10.1016/j.neuroimage.2008.09.062.

[43] B. C. M. van Wijk, C. J. Stam, and A. Daffertshofer, "Comparing Brain Networks of Different Size and Connectivity Density Using Graph Theory," *PLoS ONE*, vol. 5, no. 10, Oct. 28, 2010, ISSN: 1932-6203. DOI: 10.1371/journal.pone.0013701. pmid: 21060892.

[44] M.-T. Horstmann, S. Bialonski, N. Noennig, H. Mai, J. Prusseit, J. Wellmer, H. Hinrichs, and K. Lehnertz, "State dependent properties of epileptic brain networks: Comparative graph–theoretical analyses of simultaneously recorded EEG and MEG," *Clinical Neurophysiology*, vol. 121, no. 2, pp. 172–185, Feb. 1, 2010, ISSN: 1388-2457. DOI: 10.1016/j.clinph.2009.10.013.

[45] D. S. Bassett, A. Meyer-Lindenberg, S. Achard, T. Duke, and E. Bullmore, "Adaptive reconfiguration of fractal small-world human brain functional networks," *Proceedings of the National Academy of Sciences*, vol. 103, no. 51, pp. 19 518–19 523, Dec. 19, 2006, ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.0606005103.

[46] M. Zanin, P. Sousa, D. Papo, R. Bajo, J. García-Prieto, F. del Pozo, E. Menasalvas, and S. Boccaletti, "Optimizing Functional Network Representation of Multivariate Time Series," *Scientific Reports*, vol. 2, no. 1, pp. 1–6, 1 Sep. 5, 2012, ISSN: 2045-2322. DOI: 10.1038/srep00630.

[47] D. Jemerov and S. Isakova, *Kotlin in Action*, 1st edition. Shelter Island, NY: Manning Publications, Feb. 19, 2017, 360 pp., ISBN: 978-1-61729-329-0.

[48] S. Bonev and J. Galletly, "COMPUTER SCIENCE Functional programming features supported by Kotlin and Swift."

[49] T. Berglund and M. McCullough, *Building and Testing with Gradle*. "O'Reilly Media, Inc.", Jul. 13, 2011, 111 pp., ISBN: 978-1-4493-0463-8. Google Books: g2K2zBZWf0wC.

[GPL] Free Software Foundation, *GNU General Public License*, version 3, Free Software Foundation, Jun. 29, 2007.

[50] M. A. C. Torres, *Learning Concurrency in Kotlin: Build Highly Efficient and Robust Applications*. Birmingham, UK: Packt Publishing, Jul. 30, 2018, 266 pp., ISBN: 978-1-78862-716-0.

[51] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, Oct. 28, 2016, ISSN: 0001-0782. DOI: 10.1145/2934664.

[52] A. Dutot, F. Guinand, D. Olivier, and Y. Pigné, "GraphStream: A Tool for bridging the gap between Complex Systems and Dynamic Graphs," in *Emergent Properties in Natural and Artificial Complex Systems. Satellite Conference within the 4th European Conference on Complex Systems (ECCS'2007)*, Dresden, Germany, Oct. 2007.

[53] D. Michail, J. Kinable, B. Naveh, and J. V. Sichi, "JGraphT – A Java library for graph data structures and algorithms," Apr. 17, 2019. arXiv: 1904.08355.

[54] R. Biswas and O. Ed, "The Java Persistence API - A Simpler Programming Model for Entity Persistence," May 2016.

[55] J. Elliott, *Hibernate: A Developer's Notebook*. "O'Reilly Media, Inc.", 2004, 190 pp., ISBN: 978-0-596-00696-9. Google Books: uDizTW0jXjMC.

[56] C. Bauer, G. King, and G. Gregory, *Java Persistence with Hibernate*, 2nd ed. USA: Manning Publications Co., 2015, 608 pp., ISBN: 978-1-61729-045-9.

[57] T. Mueller. (2006). "H2 Database Engine," [Online]. Available: http://www.h2database.com/html/main.html (visited on 01/11/2020).

[58] R Core Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2009.

[59] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 2, 6:1–6:31, Feb. 20, 2009, ISSN: 0164-0925. DOI: 10.1145/1462166.1462167.

[60] C. T. Haynes, D. P. Friedman, and M. Wand, "Continuations and coroutines," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP '84, Austin, Texas, USA: Association for Computing Machinery, Aug. 6, 1984, pp. 293–298, ISBN: 978-0-89791-142-9. DOI: 10.1145/800055.802046.

[61] M. Arias and R. Chakraborty, *Functional Kotlin: Extend Your OOP Skills and Implement Functional Techniques in Kotlin and Arrow*. Packt Publishing, 2018, ISBN: 978-1-78847-648-5.

A. Mathematical definitions

A.1 Graph theory

Standard definitions of relevant concepts from graph theory, based mostly on *A first course in network theory* [18].

Definition A.1.1 (Graph)

*Let V be a finite set of n nodes (or vertices), and let $E \subseteq V \times V$ be a set of edges. A **graph** G is a pair (V, E).*
*V is the **node set** of G, and E is the **edge set** of G.*

Definition A.1.2 (Graph properties)

- *G is **reflexive** iff $\forall v \in V.\ (v, v) \in E$*
- *G is **anti-reflexive** iff $\forall v \in V.\ (v, v) \notin E$*
- *G is **undirected** (or equally **symmetric**) iff $\forall v_1, v_2 \in V.\ (v_1, v_2) \in E \Rightarrow (v_2, v_1) \in E$*
- *G is **directed** iff it is not undirected*
- *G is a **simple graph** iff it is undirected and anti-reflexive*

Definition A.1.3 (Relations on graphs)

- *$G' = (V', E')$ is a **subgraph** of $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$*
- *Let $G_1 = (V_1, E_1)$ and $G_2 = (v_2, E_2)$, then $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$ is the **union** of the two graphs, and $G_1 \cap G_2 = (V_1 \cap V_2, E_1 \cap E_2)$ is the **intersection** of the two graphs.*

Definition A.1.4 (Node adjacency)

- *In an undirected graph, v_1 is **adjacent** to v_2 iff $(v_1, v_2) \in E$*
- *A **loop** is an edge of the form (v, v). Note that simple graphs have no loops.*
- *Define $\deg^-(v) = |\{\, v' \in V \mid (v', v) \in E \,\}|$ to be the **indegree** of v, and $\deg^+(v) = |\{\, v' \in V \mid (v, v') \in E \,\}|$ to be the **outdegree** of v, in other words, the number of incoming and outcoming edges, respectively. If the graph is undirected, then $\deg^-(v) = \deg^+(v) = \deg(v)$ is called **degree**.*
- *For $G = (V, E)$ with $V = \{1, \ldots, n\}$, define $A = (a_{ij})$ to be the **adjacency matrix** of G where*

$$a_{ij} = \begin{cases} 1, & \text{if } (i, j) \in E \\ 0, & \text{if } (i, j) \notin E \end{cases}$$

for $1 \leq i, j \leq n$.

Definition A.1.5 (Connectedness)

- u, v *are* ***connected nodes*** *is there exists a path between* u, v *in* G*, i.e. if* $(A^k)_{uv} = 1$ *for some* $k \in \mathbb{N}$.
- $G = (V, E)$ *is* ***connected graph*** *if* $\forall v_1, v_2 \in V. v_1, v_2$ *are connected nodes.*
- *A subgraph* $G' \subseteq G$ *is a* ***(connected) component*** *iff* G' *is a maximal connected subgraph of* G.

  Note that all components of a graph are disjoint, therefore each node belongs to exactly one component.

A ***giant component*** *of a graph is a single component that has more nodes than any other component of the graph.*

Note that a giant component may not exist if more components have the maximum number of nodes.

A.2 Ranking

These definitions precede those in section 2.4.2.

Definition A.2.1 (Partial order)

A binary relation $\sqsubseteq$ *on some set* P *is a* ***partial order*** *iff it is antisymmetric preorder, i.e.:*

- $\forall a, b \in P.\ a \sqsubseteq a$ *(reflexivity)*
- $\forall a, b, c \in P.\ (a \sqsubseteq b \wedge b \sqsubseteq c) \implies a \sqsubseteq c$ *(transitivity)*
- $\forall a, b \in P.\ (a \sqsubseteq b \wedge b \sqsubseteq a) \implies a = b$ *(antisymmetry)*

Definition A.2.2 (Total order)

A binary relation $\sqsubseteq$ *on some set* P *is a* ***total order*** *iff it is a partial order and a connex relation, i.e.:*

- $\forall a, b, c \in P.\ (a \sqsubseteq b \wedge b \sqsubseteq c) \implies a \sqsubseteq c$ *(transitivity)*
- $\forall a, b \in P.\ (a \sqsubseteq b \wedge b \sqsubseteq a) \implies a = b$ *(antisymmetry)*
- $\forall a, b \in P.\ a \sqsubseteq b \vee b \sqsubseteq a$ *(connexity, implies reflexivity)*

B. Kotlin figures

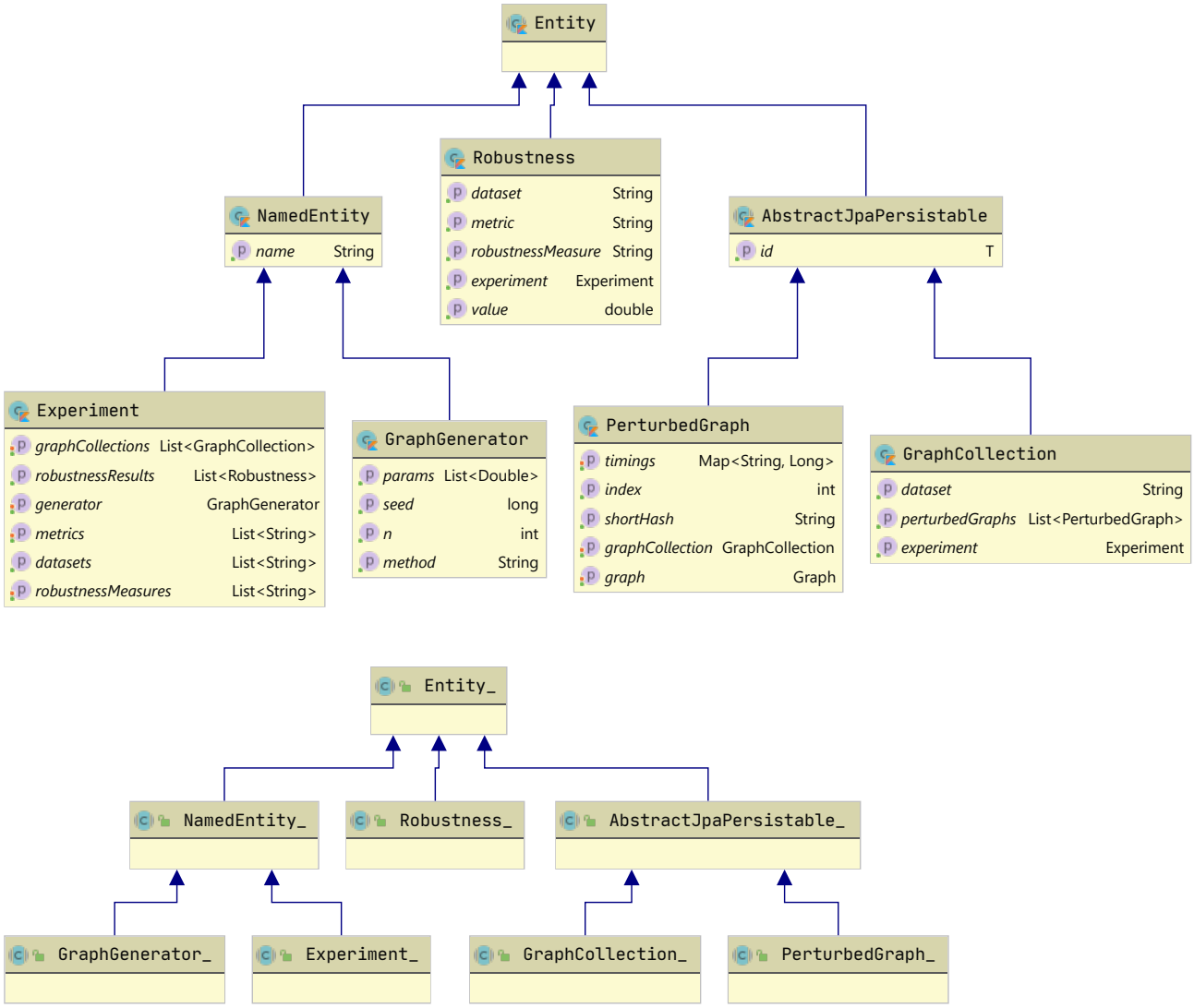


Figure B.1: *Entity classes* (i.e. classes defining the persistence model presented in Figure 3.4), and their inheritance hierarchy. The arrows indicate *inheritance* ("is a") relationships between classes. Below are displayed JPA metamodel classes automatically generated from the entities by Hibernate.

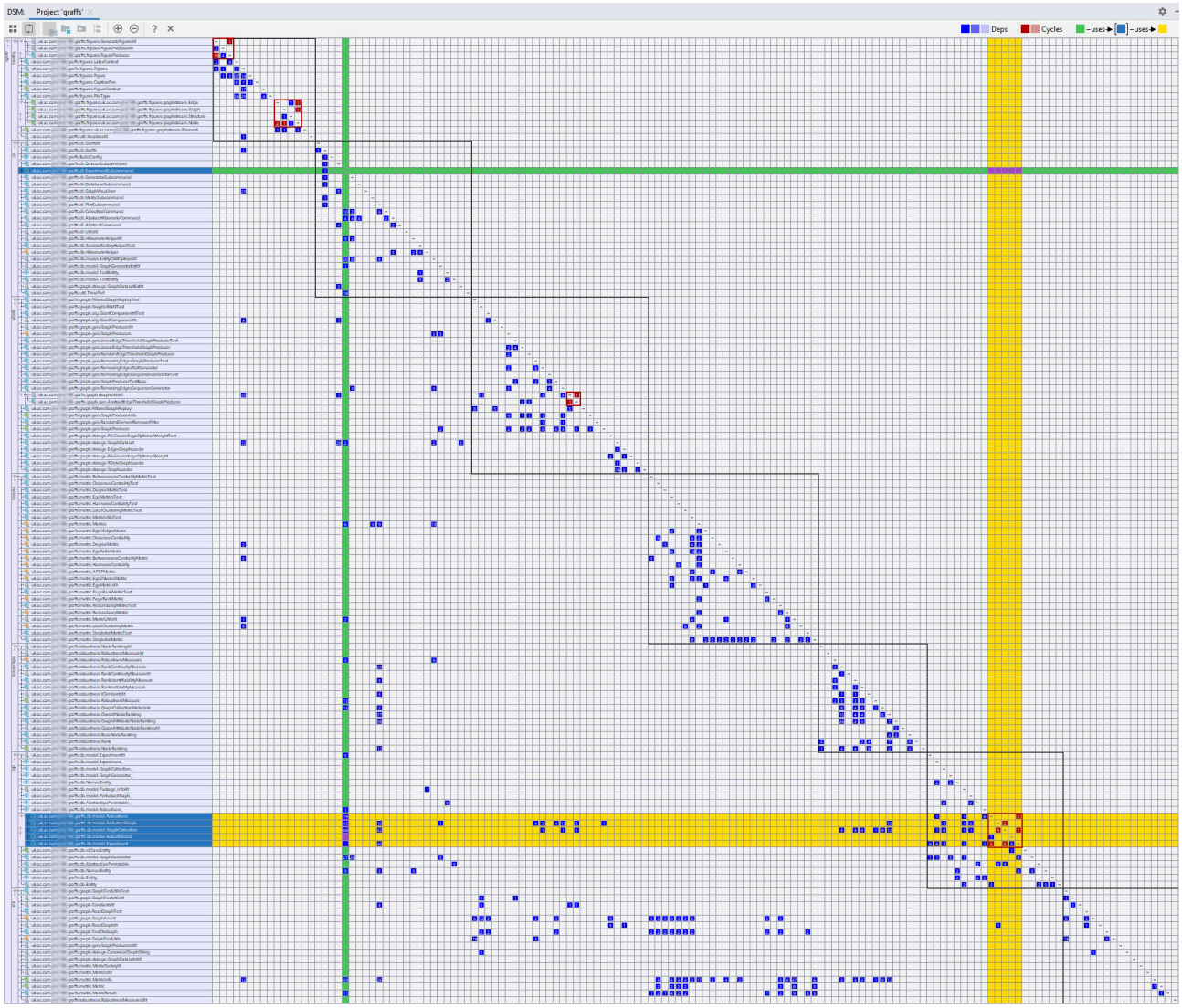


Figure B.2: Dependency matrix between Kotlin files of `graffs` (excluding tests), visualised by the IntelliJ IDEA editor. Each of the 7 groups corresponds to a module, namely: `figures`, `cli`, `graph`, `metric`, `robustness`, `db`, `api`. Cell in i-th row from the top and j-th column from the left corresponds to the dependency of j-th file on i-th file. Modules and files are ordered from more dependent (top) to less dependent (bottom).

The highlighted rows and columns show the directed dependency of the `ExperimentSubcommand` class (green), which provides the `graffs experiment` command-line subcommands, on the 5 JPA entity classes from fig. B.1 (yellow).

Listing B.1: A toy example of using typed Java Persistence API queries. Note the `Experiment_` metamodel class generated from the `Experiment` entity by Hibernate, with (meta)fields corresponding to entities' fields.

```
9  // A type-checked CriteriaQuery<String> (expected to result in String)
10 val query = session.criteriaBuilder.createQuery(String::class.java)
11
12 // The following will compile (Experiment::name is of type String)
13 query.select(query.from(Experiment::class.java).get(Experiment_.name))
14 // The following will not compile because Experiment::generator is not of type String
15 query.select(query.from(Experiment::class.java).get(Experiment_.generator))
16
17 // The result is checked to be String
18 val result: String = session.createQuery(query).singleResult
```

Listing B.2: A demonstration of the command-line help text of some subcommands, printed by `graffs`

```
$ ./graffs --help
Usage: graffs [OPTIONS] COMMAND [ARGS]...

  Tool for evaluating Graph Metric Robustness

Options:
  -l, --license  Show the license
  -v, --version  Show the version and exit
  -h, --help     Show this message and exit

Commands:
  dataset     Access available datasets
  metric      Access available metrics
  generator   Create or list graph generators
  experiment  Manage experiments (evaluating metrics on generated graphs)
  plot        Plot results
  db          Manage the underlying database

Examples:
> graffs db drop
> graffs dataset list
> graffs dataset load social-network
> graffs generator create --help
> graffs generator create --name g1 -n 10 --method removing-edges --params 0.05
> graffs experiment create --help
> graffs experiment create --name e1 --datasets test,social-network --generator g1 --metrics Degree,PageRank,
    Betweenness --robustnessMeasures RankInstability
> graffs experiment run --name e1

$ ./graffs experiment --help
Usage: graffs experiment [OPTIONS] COMMAND [ARGS]...

  Manage experiments (evaluating metrics on generated graphs)

Options:
  -h, --help  Show this message and exit

Commands:
  list    List created experiments and their properties
  create  Create an experiment
  change  Change an existing experiment
  clone   Create a new experiment using parameters of existing experiment
  remove  Remove an experiment, all its generated graph and any computed
          results
  run     Run an experiment
  prune   Remove generated graphs and calculated robustness values of an
          experiment

$ ./graffs dataset --help
Usage: graffs dataset [OPTIONS] COMMAND [ARGS]...

  Access available datasets

  Datasets are stored in the `data` folder, each dataset in a subfolder.
  Supported files are:

  - edges.txt (or *.txt) with lines describing edges with optional weight in the format: NODE1 NODE2 [WEIGHT]
  - *.RData with an R dataframe object named *.df

Options:
  -h, --help  Show this message and exit

Commands:
  list            List all datasets available in the `data` directory
  load            Check if datasets can be loaded from the `data` directory
  download-demos  Download datasets for demonstration
  viz             Visualise graph
```

C. Robustness results

Table C.1: RankContinuity of 7 metrics on 8 datasets (experiments `random-edges` and `unscored`)

RankContinuity	pvivax	ecoli	yeast	airports	collab	citation	facebook	internet
Betweenness	1.00	1.00	1.00	1.00	0.70	1.00	0.83	1.00
Degree	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Ego1Edges	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Ego2Nodes	1.00	1.00	1.00	1.00	1.00	1.00	0.97	1.00
LocalClustering	0.23	0.10	0.07	0.57	0.33	0.00	0.07	0.53
PageRank	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Redundancy	0.87	1.00	1.00	0.73	1.00	0.73	1.00	0.97

Table C.2: RankIdentifiability of 7 metrics on 8 datasets (experiments `random-edges` and `unscored`)

RankIdentifiability	pvivax	ecoli	yeast	airports	collab	citation	facebook	internet
Betweenness	0.93	0.94	0.94	0.95	0.81	0.91	0.83	0.94
Degree	0.99	1.00	0.99	1.00	0.97	1.00	0.98	0.99
Ego1Edges	0.94	1.00	0.95	0.95	0.98	0.97	0.98	0.99
Ego2Nodes	0.94	0.97	0.74	0.89	0.82	0.86	0.35	0.95
LocalClustering	0.09	0.13	0.34	0.11	0.29	0.13	0.07	0.07
PageRank	0.99	0.97	0.96	1.00	0.89	0.99	0.94	0.99
Redundancy	0.85	0.98	0.98	0.89	0.97	0.80	0.99	0.89

Table C.3: RankInstability of 7 metrics on 8 datasets (experiments `random-edges` and `unscored`)

RankInstability	pvivax	ecoli	yeast	airports	collab	citation	facebook	internet
Betweenness	0.00	0.00	0.01	0.00	0.02	0.01	0.01	0.00
Degree	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00
Ego1Edges	0.01	0.01	0.01	0.01	0.01	0.00	0.01	0.00
Ego2Nodes	0.01	0.00	0.02	0.02	0.01	0.01	0.05	0.01
LocalClustering	0.04	0.08	0.04	0.23	0.16	0.09	0.12	0.06
PageRank	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00
Redundancy	0.02	0.01	0.01	0.02	0.01	0.01	0.01	0.01

D. Project Proposal

Computer Science Tripos – Part II – Project Proposal

Framework for empirical analysis of graph metric robustness

Candidate 2343F, (college removed)
Originator: Dr Timothy Griffin
25 October 2019

Project Supervisor: Dr Timothy Griffin
Director of Studies: (name removed)
Project Overseers: Dr Rafal Mantiuk, Prof Andrew Pitts

Introduction

There exist numerous graphs representing the real world, such as proteins and their interactions, social networks, citation networks, web graphs, road networks and more. Those graphs are huge so are often analysed and described by studying different metrics (i.e. functions of graphs or graph nodes such as radius, degree, betweenness centrality, etc.; not necessarily numerical). Some graph databases provide edge weights indicating the available experimental evidence or confidence. Further networks are then built by thresholding on these weights, which is sensitive to the choice of such threshold.

In my work, I will study and compare ways to obtain multiple graphs of the similar nature; define and analyse 'robustness' of graph metrics when applied to such graphs, i.e. study how sensitive to some perturbations in the input graph these metrics are; and evaluate which graph metrics are better suited for describing graphs of respective application areas. As an extension to a standalone project, this can be turned into a library or a plugin to the graph visualisation software Gephi.

Starting point

The idea of this project stems from the following research paper:

> Bozhilova, L. V., Whitmore, A. V., Wray, J., Reinert, G., & Deane, C. M. (2019). Measuring rank robustness in scored protein interaction networks. BMC Bioinformatics, 20(1). https://doi.org/10.1186/s12859-019-3036-6

The study discusses at a few possible ways to analyse robustness of graph metrics specifically on protein graphs consisting of weighted edges. The authors came up with three different measures of robustness of a *graph metric applied to a graph within a specific confidence range*: Rank continuity, Rank identifiability, Rank instability, three different ways to assess how much the metric values change when changing the threshold. The weight of an edge between two proteins signals the amount of evidence for the specific protein interaction. Methods in this research paper require graphs with such weights, but the idea of metric robustness could be generalised.

There exist numerous libraries for working with graphs, such as Apache Giraph, JGraphT, JUNG and others, each having a different focus: iterative graph processing, algorithms, visualisation and others. I plan to base the project on one of such libraries and extend it by implementing various algorithms for computing graph metrics. I will also decide on a way to store the graphs in the file system.

I have theoretical knowledge of Java, algorithms, data science and other relevant topics from respective courses from the Computer Science Tripos. I also have some practical and software engineering skills from the Part IB project, my internships and other personal projects. I will need to study a lot of materials about graphs, networks, graph metrics, algorithms and similar topics.

This project will also use graph datasets from sources mentioned in .

Work to be done

Robustness

The goal of this project is to devise, study and compare 'robustness' of graph metrics when applied to specific graphs and conclude whether some metrics are more robust than others for describing particular types of graphs. The essence of this project is primarily the experiment itself and the concluding comparison of graph metrics rather than the platform needed to perform such an experiment.

Generate graphs

In order to test the robustness, a metric has to be evaluated over a set of similar graphs that are 'derived' from the same source dataset. In the abovementioned paper by Bozhilova, L. V. et al., the source datasets contained weights indicating the amount of evidence for each protein interaction, and so an arbitrary number of similar graphs could be produced by thresholding on different confidence values. Not all datasets contain this information and so I will need to devise a way to take an input dataset and generate similar datasets with small perturbations. The procedure may be as simple as pseudo-randomly deleting a small percentage ($\sim 5\%$) of edges, to not destruct the structure of the original graph; or taking a pseudo-random subset ($\sim 95\%$) of the nodes; or pseudo-randomly generating a confidence value for each edge and then thresholding on this value.

Evaluation

As values of node metrics in derived graphs may be affected by the derivation process (e.g. node degrees decrease with decreasing graph density), it is unhelpful to compare absolute values of the metrics. Instead, it is reasonable to compare ranks of nodes between derived graphs, thus implicitly taking into account the relative value of the metric. The authors Bozhilova et al. used a robustness analysis based on a rank similarity measure proposed by the following paper

> Trajanovski, S., Martín-Hernández, J., Winterbach, W., & Van Mieghem, P. (2013). Robustness envelopes of networks. Journal of Complex Networks, 1(1), 44–62. `https://doi.org/10.1093/comnet/cnt004`

In the paper by Bozhilova et al, three measures were defined and used to analyse robustness: Rank continuity (comparing sets of highly ranked nodes between graphs derived from similar confidence threshold), Rank identifiability (considering overall ranks of all nodes and quantifying how much these ranks differ from ranks observed in graphs derived from various confidence thresholds), Rank instability (quantifying variation of ranks of top 1% of nodes over a certain confidence range).

Furthermore, on a given graph, robustness measure of each metric may be compared to the absolute value of some (possibly other) metrics, to find out whether there is any correlation between the robustness of a metric and values of some metrics. We hope to see to see some interesting patterns in the statistical analysis.

The specific evaluation scheme will mainly depend on the robustness function, which may output more than just a single numerical value. Figure D.1 shows an illustration of how the whole evaluation process may look like.

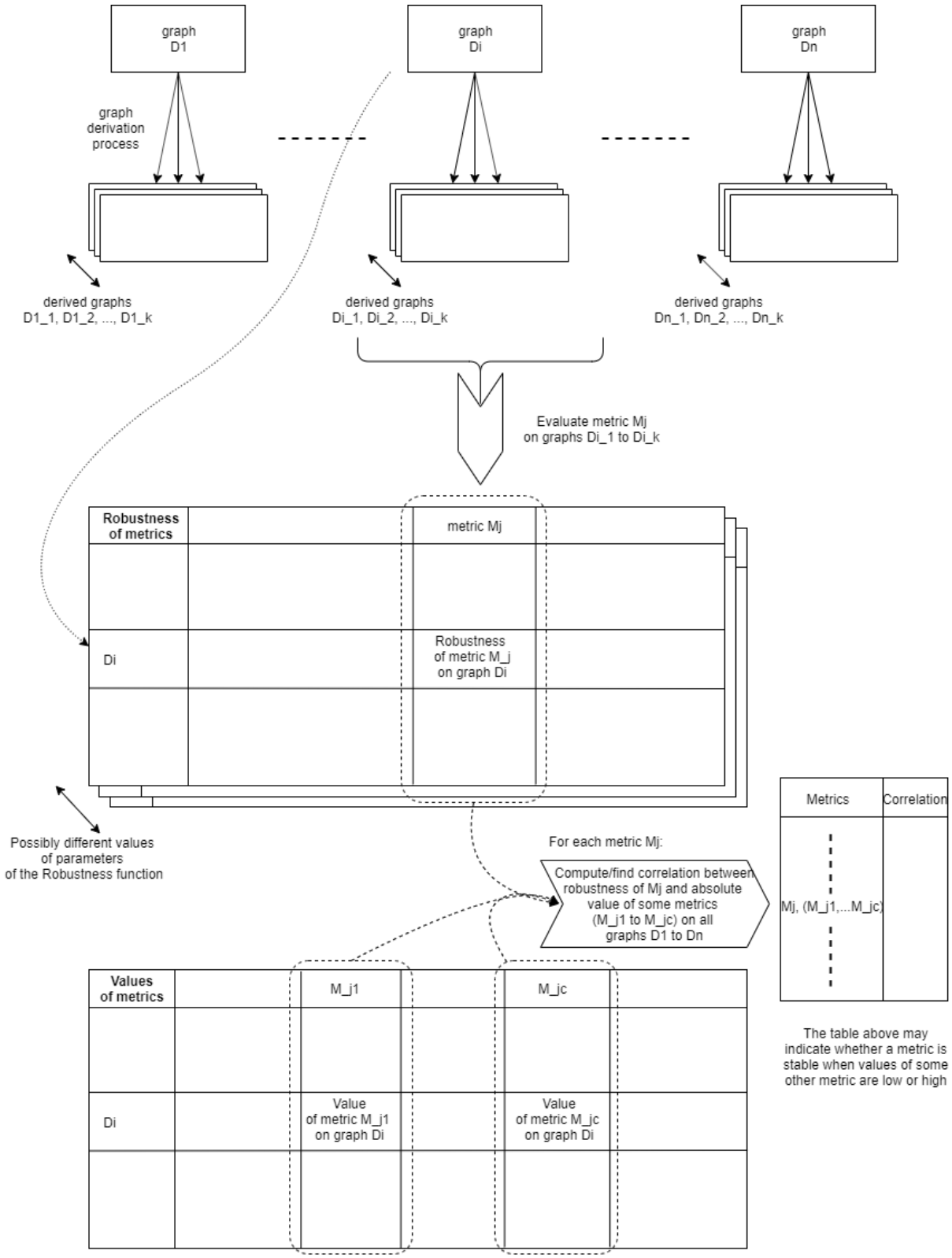


Figure D.1: An example of the evaluation process

Tasks

The following tasks need to be carried out to complete this project to the state described above.

1. Decide on the underlying graph framework, the programming language, tools and additional libraries to use for developing the software.

2. Study graphs and their metrics to decide which metrics to carry out the described experiment on. This may depend on the complexity of the algorithm of the metric, usefulness of the metric in the graph research context, computing feasibility to evaluate the metric on large datasets and other factors.

3. Study possible graph datasets and decide which graphs will be used in the experiment. Again this will take into account multiple factors such as availability of such datasets, commonness of the respective application area and others. This project intends to study robustness of graph metrics on graphs from different application areas.

4. Devise a 'robustness' measure of a metric applied to a graph (possibly more measures for different aspects of robustness). This will likely be a function of a metric and an input graph, and will likely involve generating multiple graphs from the input graph that the robustness will be calculated for, for example by randomly deleting a small percentage of edges. For this, I will need to decide on what 'small perturbations' mean, which may as well be left as a parameter of the robustness measure.

5. Build the platform to do the following:

   - Store, load and represent input graphs
   - Implement (or integrate existing) algorithms to compute graph metrics
   - Generate similar graphs, given an input graph, by making small changes or deletions
   - Implement the robustness function
   - Possibly, if suitable, produce visual results directly from the program

6. Execute the result, i.e. compute the robustness function value for chosen graph metrics and chosen input graphs, trying different values of parameters if appropriate.

7. Evaluate the outcome and conclude the findings. Compare graph metrics in terms of their robustness based on the results of the experiment. Possibly compare the results of this experiment to the results of the paper by Bozhilova et al.

8. Write up the dissertation.

This project may additionally involve the use of secondary languages and platforms (such as Python, Jupyter notebook) for statistical analysis etc.

Success criteria

This project will be considered a success if I manage to complete the steps described above, primarily to execute the experiment and conclude any observations about graph metrics.

1. Implement the experimental framework outlined above, using interesting datasets
2. Complete statistical analysis, as described in the section
3. Deduce empirical observations about graph metrics

The results of the experiment may also be compared to the results of the originating paper by Bozhilova, L. V., et al.

Possible extensions

The following ideas could be developed in case there is spare time during the implementation phase.

1. Wrap up and publish the project as an open-source library that can further be used by future projects.
2. Develop a plugin to the Gephi graph visualisation software, to compute robustness of different metrics within the Gephi user interface.
3. Produce a program, that will, given an input graph, be able to advise which metrics are more suited for describing such graph

Timetable

Planned starting date is 28 October 2019.

Weeks 1 to 2

Research and read papers on similar topics. Read about graphs and network from relevant books.

*Milestone **8 November**: Fully understand the problem this project is solving and approaches possible.*

Weeks 3 to 4

Set up the working environment, decide on programming language and tools needed, repositories for code and dissertation, find and integrate relevant libraries, familiarise myself by creating toy programs on graphs.

By taking various factors into account, decide on which metrics to analyse and which graph datasets to choose for the experiment. Use toy programs to estimate the complexity and feasibility of different algorithms on the datasets.

*Milestone **22 November**: Have a fully working setup of the programming environment with backup plans.*

Weeks 5 to 6

Devise the robustness measure(s) and mathematics behind the experiment. Write up the beginning of the dissertation.

Beginning of the implementation: storing, loading and representing input graphs. Also, be able to generate pseudo-random graphs derived from an input graph.

Milestone 6 December: Fully understand what robustness function the program will calculate and how.

Weeks 7 to 8

Implement algorithms of the metrics to evaluate. Document the implementation.

*Milestone **20 December**: Working algorithm for each chosen metric*

Weeks 9 to 10

Winter break, no work planned.

Weeks 11 to 12

Start implementation of the robustness measure(s).

Weeks 13 to 14

Finish implementing the robustness measure(s) and a way to compare metrics. Write the Progress Report.

*Milestone **31 January**: Be able to calculate the robustness measure, for a given collection of similar graphs and a graph metric. Submit the Progress Report.*

Weeks 15 to 16

Refine the robustness measure implementation. Integrate the individual parts of the implementation to create an executable program taking input parameters. If suitable, produce a statistical output from the program.

*Milestone **14 February**: Executable program*

Weeks 17 to 18

Execute the evaluation (this may be very computationally expensive). Compare individual metrics and conclude results. Start writing the dissertation.

*Milestone **28 February**: Have some results of the evaluation*

Weeks 19 to 20

Continue with carrying out the experiment. Over this period, focus mainly on writing the dissertation.

Weeks 21 to 22

Finish evaluation. Conclude experiment results. Continue writing the dissertation.

*Milestone **27 March**: Have the majority of the dissertation written up.*

Weeks 23 to 24

Ideally, work on extensions, or finish the implementation/evaluation if needed. Collect thorough feedback on the dissertation.

*Milestone **10 April**: Have a finished dissertation.*

Weeks 25 to 26

The dissertation should be done by now. Polishing of the work if needed.

Weeks 27 to 28

No work planned.

*Milestone **8 May**: Submit the dissertation*

Resource Declaration

Personal resources

For the development and writing the dissertation, I will primarily use my personal laptop with an IDE such as IntelliJ IDEA. Source code and the written work will regularly be version-controlled in a git repository, backed up on GitHub and possibly other relevant online storage places. I accept full responsibility for this machine and the software and I

have made contingency plans to protect myself against hardware and/or software failure, such as to fallback to using the MCS computers.

Computing resources

This project requires the use of a high-performance computing resource, to run algorithms of evaluating metrics and their robustness on large (real-world or derived) graphs.

Based on the paper by Bozhilova et al., calculating natural connectivity for a single node for a graph with 7000 nodes takes 88 seconds on a standard computer. Based on my toy example, calculating average betweenness centrality of 2500 nodes takes 8 minutes on my computer. Thus, assuming that computing a metric for an input graph of average size 5000 nodes takes 1 hour, the pure computation time suggested by this Project Proposal would take the following time on a standard personal computer (approximated in the order of magnitude)

$$(\sim 6 \text{ datasets}) \times (\sim 6 \text{ metrics}) \times (\sim 50 \text{ derived graphs}) \times (\sim 1 \text{ hour}) \approx 75 \text{ days}$$

The following resource has been granted permission for and will be used to parallelise the process and thus significantly speed up the computation time:

> Resource: a computing facility (such as one of the servers *yellow, nile,* or *rio*)
> Institution: Systems Research Group (`https://www.cl.cam.ac.uk/research/srg/`)
> Sponsor: Dr Andrew Moore (`andrew.moore@cl.cam.ac.uk`)
>
> This requires setting up a Computer Laborarory account.

An alternative to this is Amazon Web Services or Google Cloud with free credits for students.

Datasets

Graph datasets are available and will be obtained from either of the following sources (or others):

- Stanford Large Network Dataset Collection: `https://snap.stanford.edu/data/`
- STRING Database: `https://string-db.org/`