Daniel Hopper

Methods for Calculating Random Walk Betweenness

Computer Science Tripos – Part II

Jesus College

May 7, 2020

Declaration of Originality

I, Daniel Hopper 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.

Signed Daniel Hopper

Date 06/05/2020

Proforma

Candidate Number:	2320D
Project Title:	Methods for Calculating Random Walk Betweenness
Examination:	Computer Science Tripos – Part II, 2020
Word Count:	9576[1]
Lines Count:	1147[2]
Project Originator:	The Candidate
Supervisor:	Andrej Ivašković

Original Aims of the Project

The core aim of the project was to implement and evaluate the performance of two algorithms for calculating random walk betweenness. In order to facilitate the evaluation, I was to collect a selection of networks to run the algorithms on, including both real world datasets and synthetic ones. I also aimed to create visualisations of random walk betweenness, and to create a tool that would enable me to invoke the algorithms from the command line. As an extension, I implemented and evaluated the performance of an algorithm that approximates random walk betweenness.

Work Completed

The project has been a success; I implemented all three algorithms and evaluated them based on their run time and accuracy. I also produced one tool for visualisation and another tool for running the algorithms from the command line. My implementations of the Brandes-Fleischer algorithm and the approximate Brandes-Fleischer algorithm have been demonstrated to outperform existing Python implementations found in the `NetworkX` library. In particular, my implementation of the Brandes-Fleischer algorithm is approximately 18 times faster than the existing `NetworkX` implementation.

Special Difficulties

None.

[1]This word count was computed using `TeXcount`.
[2]This line count was computed using `cloc`.

Contents

Chapter 1

Introduction

In the field of network analysis, it is often of interest to identify the nodes that are most important to the spread of information through a network. This project concerns a particular way of measuring importance. **Random walk betweenness** is suited for the analysis of networks where information moves randomly around the network rather than being guided to take an efficient route. In this project I implement algorithms that calculate random walk betweenness, and I evaluate their performance. My evaluation shows that my implementation of the Brandes-Fleischer algorithm [5] is approximately 18 times faster than the existing state of the art. As an extension, I implement and evaluate the performance of the approximate Brandes-Fleischer algorithm, which approximates random walk betweenness.

In this chapter, I explain the motivation for using random walk betweenness as a measure of centrality and give a summary of the work that I have undertaken within this project.

1.1 Motivation

Network analysis is a broad discipline dedicated to extracting useful information from data that takes the form of a network – a collection of nodes and edges between nodes. One common task is to find nodes in a network that are considered important (for some notion of importance). There are many different ways to measure how important a node is, which are known as **measures of centrality**. Examples include node degree, PageRank [21], and **betweenness centrality** [8], which have been used to analyse networks in a wide range of domains from social networks [26] to biological networks [1]. This project considers a less commonly used measure of centrality: **random walk betweenness**, as proposed by Newman [20].

From packets routed through a computer network, to vehicles navigating through a traffic network, it is often the case that there is some sort of information travelling through a network. Betweenness centrality attempts to measure how important a node is to the spread of this information. The betweenness centrality of node n is defined as the proportion of shortest paths from node s to node t that pass through n, averaged over all pairs (s, t).

The definition of random walk betweenness is motivated by the observation that betweenness centrality ignores contributions from paths that are not shortest paths. However, in many real world examples, such as the flow of rumours in a social network or the spread of infectious diseases through a population, the assumption that information only moves along shortest paths is unjustified. In these cases, an approach that assumes that information traverses the network randomly is a better fit [6] [23].

Random walk betweenness of a node n can be thought of roughly as the average number of times a random walk from node s to node t passes through n, averaged over all pairs (s, t). Intuitively, this gives a very similar metric, adjusted for different assumptions about how information travels through the network. A more rigorous definition of random walk betweenness can be found in Section 2.1.3.

1.2 Achievements

The core aims of this project are to implement and evaluate the performance of two algorithms for calculating random walk betweenness: the Newman algorithm [20] and the Brandes-Fleischer algorithm [5]. As an extension of the project, I implement an approximating algorithm developed by Brandes and Fleischer that delivers a trade-off of accuracy for speed. This trade-off is key for very large graphs that exact approaches cannot handle. In addition to implementing all of these algorithms, I test them extensively.

In the evaluation section of this project, I measure the running times of each of the aforementioned algorithms on various networks. There are existing implementations of some of these algorithms in the Python `NetworkX` library, so I measure running times for those as well. I then compare the running times of these various algorithms. Evaluation is done on a wide range of networks, including some that are created synthetically, and others that come from real world datasets. The evaluation shows that my implementations achieve the same results as those in the `NetworkX` library, but in significantly less time. In the case of the Brandes-Fleischer algorithm, the speedup is by a constant factor of approximately 18.

Chapter 2

Preparation

In this chapter I first outline the theoretical background of the project – I give a rigorous definition of random walk betweenness and introduce the areas of randomised algorithms, measures of centrality, and random graphs. I then go on to discuss the requirements and success criteria of the project, giving a detailed breakdown of work to be completed. I discuss the methodology and the tooling used in the project. Finally, I describe the existing code and knowledge that I had before commencing work.

2.1 Theoretical background

2.1.1 Randomised algorithms

Here, I give formal definitions of the concepts of a **randomised algorithm** and of a **fully polynomial randomised approximation scheme** (FPRAS). I do this specifically because of the relevance to one of the extension algorithms which I introduce later in this section, however the concepts do not play a key part in the project as a whole.

Randomised algorithms A randomised algorithm is one that uses random bits in its execution. Because of this randomness, the algorithm may give different outputs or take different amounts of time when run twice on the same input. One example of a randomised algorithm is the variant of the quicksort algorithm in which the pivots are chosen uniformly at random. Whereas with a fixed pivot there is a clearly defined class of lists that produce a degenerate case that runs in $\mathcal{O}(n^2)$ time, for the randomised version of the algorithm it can be proved that for any input list the algorithm will run in $\mathcal{O}(n \log n)$ time *with high probability* [12].

Fully Polynomial Randomised Approximation Scheme A Fully Polynomial Randomised Approximation Scheme $\mathcal{A}$ is a randomised algorithm with the following properties [25]:

- approximates a function f, with the *coarseness* of the approximation controlled by a parameter $\epsilon > 0$, that is, satisfies $\mathbb{P}[|\mathcal{A}(x) - f(x)| < \epsilon \cdot f(x)] \geq \frac{3}{4} \;\; \forall x \in \text{domain}(f)$;
- runs in time that is polynomial in $|x|$ and in $\frac{1}{\epsilon}$.

This means that an algorithm that runs in $\mathcal{O}(\frac{1}{\epsilon^3}n^2)$ time is permitted, but not one that runs in $\mathcal{O}(n^{\frac{1}{\epsilon}})$ time. An example of a problem that admits an FPRAS is the *DNF counting problem* [14] – the problem of finding the number of satisfying truth assignments for a formula given in disjunctive normal form.

Above is the standard definition of an FPRAS, however some authors (in particular Brandes and Fleischer [5]) use an alternative definition that considers absolute error rather than relative error. Under this definition, $\mathcal{A}$ must satisfy $\mathbb{P}[|\mathcal{A}(x) - f(x)| < \epsilon] \geq \frac{3}{4}$.

2.1.2 Measures of centrality

Measures of centrality aim to provide quantitative methods to describe how important a node is within a graph. In his essay "Centrality in social networks conceptual clarification" [9], Freeman splits measures of centrality into three categories: degree, betweenness, and closeness.

- *Degree-based* measures of centrality say that a node is central if it is connected to a large number of other nodes. An example of this is node degree (i.e. the number of neighbours of a node).
- *Betweenness-based* measures of centrality say that a node is central if it lies on a large number of paths between other nodes. Examples of this include betweenness centrality [8] and random walk betweenness.
- *Closeness-based* measures of centrality say that a node is central if it is only a short distance from the other nodes in the graph. An example of this is the measure of centrality given in the paper "An improved index of centrality" by Murray A. Beauchamp [2].

2.1.3 Random walk betweenness

Definition Random walk betweenness of a node n is the net number of times that a random walk from node s to node t passes through n, averaged over all pairs (s, t). The word 'net' is used to mean that a walk crossing an edge that has already been crossed in the opposite direction is considered to cancel the previous crossing out, giving no contribution to the net number of times the nodes at each end of the edge were travelled through. Figure 2.1 shows the random walk betweenness of nodes in an example graph.

To build up a more rigorous definition of random walk betweenness, consider an undirected, connected graph $G(V, E)$. We define a random walk on G starting at $s \in V$ and terminating at $t \in V$. At each step, the random walk moves to a node chosen uniformly

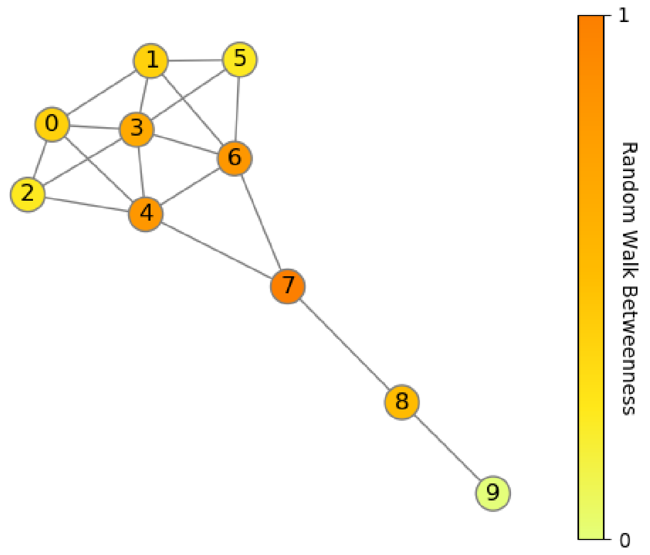


Figure 2.1: Random walk betweenness of nodes in a small graph.

at random from the nodes adjacent to the current node. The random walk continues until we reach node t, at which point it immediately stops. Mathematically, we define a random walk as an infinite sequence of random variables W_i over graph vertices such that:

$$W_0 = s \tag{2.1}$$

$$W_{i+1} = \begin{cases} t, & \text{if } W_i = t \\ \text{uniformly distributed on } \{k | (k, W_i) \in E\}, & \text{otherwise} \end{cases} \tag{2.2}$$

We use the notation $X^{s,t}(v, w)$ for the number of times that the random walk travels through an edge from v to w (which will necessarily equal 0 if there is no edge between v and w). Now, we define the net flow through the edge (v, w):

$$V^{s,t}(v, w) = \left|\mathbb{E}(X^{s,t}(v, w)) - \mathbb{E}(X^{s,t}(v, w))\right| \tag{2.3}$$

And now the net flow through a node, $v \in V$

$$F^{s,t}(v) = \frac{1}{2} \sum_{w \in V} V^{s,t}(v, w) \tag{2.4}$$

Finally, the random walk betweenness of a node is the mean of $F^{s,t}(v)$ across all pairs (s, t), $s \neq t$, $s \neq v$, $t \neq v$:

Algorithm	Running Time	Space	
Newman algorithm	$\mathcal{O}(I(n-1)+mn^2)$	$\mathcal{O}(n^2)$	[20]
Brandes-Fleischer algorithm	$\mathcal{O}(I(n-1)+mn\log n)$	$\mathcal{O}(n^2)$	[5]
Approximate Brandes-Fleischer algorithm	$\mathcal{O}(\frac{\log n}{\epsilon^2}(S(n-1)+m))$	$\mathcal{O}(m)$	[5]

Figure 2.2: Summary of all algorithms introduced in Section 2.1.3

$$b(v) = \frac{1}{(|V|-1)(|V|-2)} \sum_{\substack{s,t \in V \setminus \{v\} \\ s \neq t}} F^{s,t}(v) \tag{2.5}$$

Algorithms The concept of random walk betweenness was first described by M.E.J Newman in his paper "A measure of betweenness centrality based on random walks" [20]. In this paper, Newman also describes another measure of centrality known as **current flow betweenness**, which considers a graph to be an electrical network where edges are unit resistors. Current flow betweenness of a node n is the current that flows through n when one unit of current is injected into s and one unit of current is extracted from t, averaged over all pairs (s,t). Newman shows that this is mathematically equivalent to random walk betweenness. It is from this from this physical understanding that Newman derives an algorithm for calculating random walk betweenness using matrix methods. This algorithm is referred to in this dissertation as the **Newman algorithm**. It runs in $\mathcal{O}(I(n-1)+mn^2)$ time and $\mathcal{O}(n^2)$ space where n is the number of nodes in the network, m is the number of edges in the network, and $I(n)$ is the running time of inverting an $n \times n$ symmetric, positive-definite matrix. The $(n-1)\times(n-1)$ matrix that is inverted is the constrained Laplacian of the graph (see Section 3.3.1).

Brandes et al. [5] give an improved exact algorithm for calculating random walk betweenness, which is referred to in this dissertation as the **Brandes-Fleischer algorithm**. This algorithm runs in $\mathcal{O}(I(n-1)+mn\log n)$ time and $\mathcal{O}(n^2)$ space. The paper also describes an FPRAS that approximates random walk betweenness to within an absolute error of ϵ with probability $\frac{2}{n^l}$ in $\mathcal{O}(\frac{\log n}{\epsilon^2}(S(n-1)+m))$ time and $\mathcal{O}(m)$ space. $S(n)$ is the time taken to solve the linear system $\mathbf{Ax}=\mathbf{b}$ where $\mathbf{A}$ is an $n \times n$ symmetric, positive-definite matrix and $\mathbf{b}$ is an n length column vector. This FPRAS is referred to in this dissertation as the **approximate Brandes-Fleischer algorithm**.

2.1.4 Random graphs

A random graph is a graph where nodes, edges, or both, are generated by a random procedure [13]. One classical model of random graph is the **Erdös-Rényi** model. It has two parameters: n and p. The graph is produced by creating nodes labelled $\{1,2,...,n\}$ and for each pair of nodes, connecting them with an edge with probability p.

In this project, I am not interested in random graphs for their own particular mathematical intrigue, but for a practical purpose – it is difficult to get a large number of real world datasets with which to test the algorithms implemented in this project. Instead, I generate some of my own datasets using random graphs.

Ideally these synthetic datasets should mimic some of the characteristics of real world networks. Some commonly observed properties of real world networks are:

- clustering – networks are made up of highly connected clusters;
- the "small world property" – the average length of the shortest path between two nodes is low [24];
- the degrees of nodes are distributed approximately as a power law distribution, meaning that most nodes have a low degree but there are a small number with a very high degree.

The **Watts-Strogatz model** [27] for generating random graphs creates graphs with clustering and the small world property. It works by first creating a ring graph with N nodes each connected to their $\frac{K}{2}$ nearest neighbours on each side. Then, for each node in the graph, its $\frac{K}{2}$ rightmost neighbours are rewired with probability β. To rewire an edge we change it from (i, j) to (i, k) where k is chosen uniformly at random whilst avoiding connecting a node to itself or duplicating an edge that already exists. The values N, K, and β act as parameters in the creation of the graph.

The **Holme-Kim model** [11] for generating random graphs creates graphs with all three of the properties listed above. This model is fairly complex, so here I give only an overview of how it works rather than a detailed description. One by one, N nodes are added to the graph with m edges each. New edges added are more likely to be attached to nodes with higher degrees, giving roughly a power law distribution of node degrees. When we add an edge from i to j, with probability p we also add an edge from i to a random neighbour of j. This increases the clustering of the graph by adding more connected triads of nodes.

The Erdös-Rényi model, Watts-Strogatz model, and Holme-Kim model are used in the Evaluation chapter of this project.

2.2 Requirements analysis

In this section, I set out the project's success criteria, before breaking the project down into the individual tasks to be completed and the rough order that they are to be completed in. I then go on to give further detail about what individual components of the project entail.

2.2.1 Success criteria

There are two core algorithms that must be implemented as part of my success criteria: the Newman algorithm and the Brandes-Fleischer algorithm. In addition to these, I implement an extension algorithm: the approximate Brandes-Fleischer algorithm. The project revolves around these algorithms – the work I do other than implementing them is either creating infrastructure to help me to improve them, or it is evaluating them.

My core success criteria are as follows:

- implementations of the core algorithms exist and have sufficient tests to ensure proper functioning;
- core algorithms have been evaluated based on their running time when run on real world and synthetic graphs (See Section 2.2.3);
- real world graphs have been collected (See Section 2.2.4) and synthetic graphs have been generated (See Section 2.1.4);
- a visualisation tool has been created (See Section 2.2.5);
- a command-line tool has been created (See Section 2.2.6).

My extension goals are:

- implementation of extension algorithm exists, and has sufficient tests to ensure proper functioning;
- extension algorithm has been evaluated based on its running time and accuracy when run on real world and synthetic graphs.

2.2.2 Overview

In Figure 2.3, I outline the tasks that make up the project, all of which have been successfully completed. I divide the project into four phases:

- Phase 1 – Set up infrastructure so that I have graphs to run my algorithms on once they are implemented, and have a test suite in place so that I will know when my algorithms are working correctly.
- Phase 2 – Implement the core algorithms. Create a visualisation tool and perform bench-marking to help myself bug-fix and improve the core algorithms.
- Phase 3 – Write evaluation scripts, and create a command-line tool that will aid me and any users of the library in evaluating the performance of the algorithms.
- Phase 4 – Implement and evaluate the extension algorithm.

Task	Difficulty	Phase	Extension
Read/write graphs	Low	Phase 1	
Generate random graphs	Low	...	
Test core algorithms	Low	...	
Implement core algorithms	High	Phase 2	
Visualise graphs	Low	...	
Do bench-marking and timing	Medium	...	
Collect real world graphs	Low	Phase 3	
Implement command-line tool	Low	...	
Evaluate core algorithms	Medium	...	
Test extension algorithm	Low	Phase 4	✓
Implement extension algorithm	High	...	✓
Evaluate extension algorithm	Medium	...	✓

Figure 2.3: Summary of tasks completed in the project.

2.2.3 Evaluation

In the evaluation, I time the execution of different algorithms on a variety of networks both from real world datasets, and those generated synthetically. The algorithms considered are:

- my implementation of the Newman algorithm;
- my implementation of the Brandes-Fleischer algorithm;
- the implementation of the Brandes-Fleischer algorithm built into the `NetworkX` library;
- my implementation of the approximate Brandes-Fleischer algorithm;
- the implementation of the approximate Brandes-Fleischer algorithm built into the `NetworkX` library.

In evaluating the extension algorithms, it is important to evaluate the accuracy of the result as well as the time taken to produce it. Particularly with the approximate Brandes-Fleischer algorithm, there are trade-offs between accuracy and speed depending on what parameters the algorithm is run with. A notion of accuracy is defined in Section 4.3.

2.2.4 Real world networks

The real world datasets used for evaluation are:

- **General Relativity and Quantum Cosmology Collaboration Network** (GRQC-Collab) [16] – A network where nodes are authors of papers in the General Relativity and Quantum Cosmology category of the arXiv website, which is a repository of preprint scientific papers. An edge connects two nodes if their respective authors have collaborated on a paper. The network has a large component with 4,158 nodes and 13,428 edges. It has 5,242 nodes and 14,496 edges total, spread across 355 connected components.

- **Enron Email Network** (Enron-Email) [15] – A network where nodes are email addresses and each edge connects two email addresses where one has sent an email to the other. The dataset was made public by the Federal Energy Regulatory Commission during a legal investigation concerning the Enron corporation. The network has a large component with 36,692 nodes and 183,831 edges. It has 33,696 nodes and 180,811 edges total, spread across $1,065$ connected components.
- **European Research Institution Email Network** (EU-Email) [16] – Another email network, this time from internal emails between members of a European research institution. The network has a large component with 986 nodes and 16,687 edges. It has 1,005 nodes and 16,706 edges total, spread across 20 connected components.
- **Gnutella Peer-to-Peer File Sharing Network** (Gnutella-P2P) [19] – A network where nodes represent hosts on a peer-to-peer file sharing platform, and edges represent connections between these hosts. The network is made up of a single connected component with 10,876 nodes and 39,994 edges.

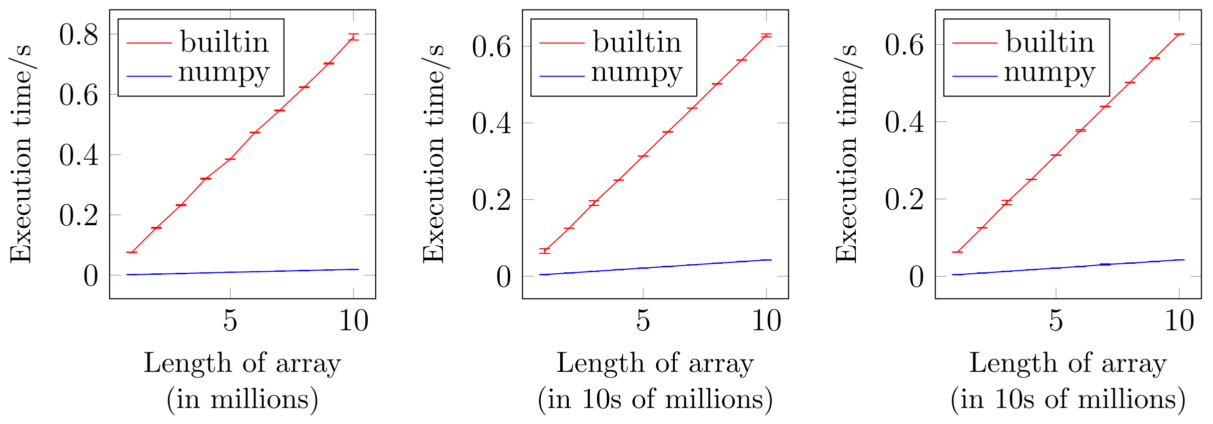
All of these datasets are available for download in an adjacency list format, albeit some with different characters used as delimiters. Therefore, they can all be read using the `NetworkX` `read_adjlist` function with the only individual processing required being tagging each of them with the delimiter they use.

2.2.5 Visualisation tool

The visualisation tool creates plots showing the structure of a graph as well as indicating the random walk betweenness of each node. This is useful for bug-fixing code, and for creating figures such as Figure 2.1. This tool is exposed to users via the command-line tool.

2.2.6 Command-line tool

The command-line tool allows the user to run any of the random walk betweenness algorithms implemented in this project on any graph. The output can take the form of a `.csv` file mapping nodes to random walk betweenness, a visualisation of the graph created by the visualisation tool, or a measurement of the time taken to execute the algorithm.



(a) Absolute Value (b) Sum (c) If-Then-Else

Figure 2.4: Comparison of execution times of built-in Python functions versus similar functions from the `NumPy` library. Error bars are $\pm 2 \times$ standard error of mean. Subfigure (a) compares the execution of the `numpy.abs()` function to that of the built-in Python `abs()` function. Subfigure (b) compares the execution of the `numpy.sum()` function to that of the built-in Python `sum()` function. Subfigure (c) compares the execution of the `numpy.where()` function to that of the built-in Python ternary operator (`x if y else z`).

2.3 Tooling

2.3.1 Python

I use Python as the programming language for this project because of my familiarity with it, and because of its strong support for dealing with networks and matrices. Here, I describe the main Python libraries that are used in the project:

NumPy is a Python library that provides sophisticated functions for creating and manipulating multidimensional arrays. `NumPy` is a staple of high-performance Python code. It does most of its work through calls to highly optimised C code. This results in code using `NumPy` arrays being significantly faster than code using normal Python lists.

In this project, `NumPy` is used for speeding up a variety of operations. Figure 2.4 shows that some common `NumPy` functions are much faster than built-in Python functions when applied to large arrays. For example, the first plot compares the use of `numpy.abs(x)` to `[abs(i) for i in x]`.

NetworkX is a Python library that provides functions for dealing with networks. This project makes use of it for:

- reading and writing networks to and from files in the adjacency list format;
- returning the adjacency matrix representation of a network;

- getting arrays of nodes and edges in a network;
- generating random graphs for evaluation purposes.

`NetworkX` also provides implementations of many graph algorithms, crucially including the Brandes-Fleischer algorithm and the approximate Brandes-Fleischer algorithm. In the Evaluation chapter, I compare these implementations to my own implementations of the same algorithms.

SciPy is a Python library that provides a wide array of functionality for scientific computing. This project makes use of two of its functions in particular:

- `scipy.stats.rankdata()` – used in my implementation of the Brandes-Fleischer algorithm to get the positions that each element of a list would be in were the list sorted.
- `scipy.sparse.linalg.spilu()` – used in my implementation of the approximate Brandes-Fleischer algorithm to efficiently solve linear equations of the form $\mathbf{Ax} = \mathbf{b}$ (where $\mathbf{A}$ is an $n \times n$ matrix, and $\mathbf{x}$ and $\mathbf{b}$ are n length column vectors). This function uses an incomplete sparse LU decomposition as a preconditioner for an iterative method of solving $\mathbf{Ax} = \mathbf{b}$ (See Section 3.2.1).

timeit is a simple library for measuring the execution time of code snippets. This is useful for measuring execution times for the Evaluation chapter, and for finding bottlenecks when trying to improve performance.

pytest is a Python library for writing unit tests. In this project, it is used to produce unit tests for each of the algorithms implemented.

2.3.2 Development environment and version control

The PyCharm IDE is used to write all of the code in the project. PyCharm is a full-featured IDE that I find to be very convenient and usable. It provides integration with Git, which makes regularly pushing my work to a GitHub repository very painless. Git is used for version control as well as backups.

2.4 Starting point

Going into this project, I had existing knowledge about probability theory and about betweenness centrality:

- *IA Probability* (maths department) gave a strong foundation in statistics. It contains a small amount of material about simple random walks in 1 dimension and about Markov chains.

- *IB Foundations of Data Science* reinforced my knowledge of probability.
- *IB Machine Learning and Real World Data* has a small amount of material about random graphs and introduced me to the concept of betweenness centrality.

However, I knew little about random walks, which I supplemented by reading “Random Walks on Graphs: A Survey” by L. Lovász [17]. I also had no prior knowledge about random walk betweenness, so I explored the literature. In particular, Newman, 2005 [20] and Brandes et al., 2005 [5] were of interest.

2.5 Summary

There is a natural waterfall-like structure to the project, which is split into a number of tasks that can be completed one by one. Unit tests for the core algorithms are written before the algorithms themselves, taking lessons from the philosophy of Test Driven Development [3]. This project uses Python, which provides a large number of libraries that serve to make my life easier in many aspects of this project.

Chapter 3

Implementation

In this chapter, I describe how I have implemented the various components of the project. I give an overview of the file and the class structures that I use. I then elaborate on details of each of the components of the project. For each algorithm I have implemented, I describe the mathematical background of the algorithm, give the algorithm itself in a pseudocode form, and discuss specific, notable details of my Python implementation of the algorithm.

3.1 Project structure

In this section, I give an overview of the different directories within the project and their purposes. I also describe the class structure of the project.

3.1.1 File and directory structure

Figure 3.1 shows the layout of the project's directories in a directory tree format. The core components of the project are:

- the command-line tool, implemented in `main.py`;
- the implementations of the algorithms, found in the `random_walk_betweenness` directory;
- the evaluation, for which scripts can be found in the `scipts/evaluation` directory;
- timing code, which can be found in the `scripts/timing` directory;
- the visualisation tool, which can be found in the `scripts/vis` directory;
- the suite of unit tests, which can be found in the `tests` directory.

I use the `data` directory as an easy way to back up important non-code files such as data collected or figures produced.

```
project
├── main.py .............................................. see section 3.4.3
├── random_walk_betweenness
│   ├── calculate.py ..................................... see section 3.1.2
│   ├── RandomWalkBetweennessSolver.py ................... see section 3.1.2
│   ├── NewmanSolver.py .................................. see section 3.3.1
│   ├── BrandesSolver.py ................................. see section 3.3.2
│   ├── ApproxSolver.py .................................. see section 3.3.3
│   ├── NXApproxSolver.py
│   └── helper_functions.py
├── graphs ............................................... see section 3.4.1
│   ├── read_write.py
│   ├── random_graphs.py
│   ├── small_graphs.py
│   └── resources
│       └── ...
├── scripts
│   ├── evaluation
│   │   └── ...
│   ├── timing ........................................... see section 3.4.4
│   │   ├── Profiler.py
│   │   └── time_algorithm_execution.py
│   └── vis .............................................. see section 3.4.2
│       └── draw_graph.py
├── tests
│   └── ...
└── data
    └── ...
```

Figure 3.1: Diagram showing the various directories and files in the project.

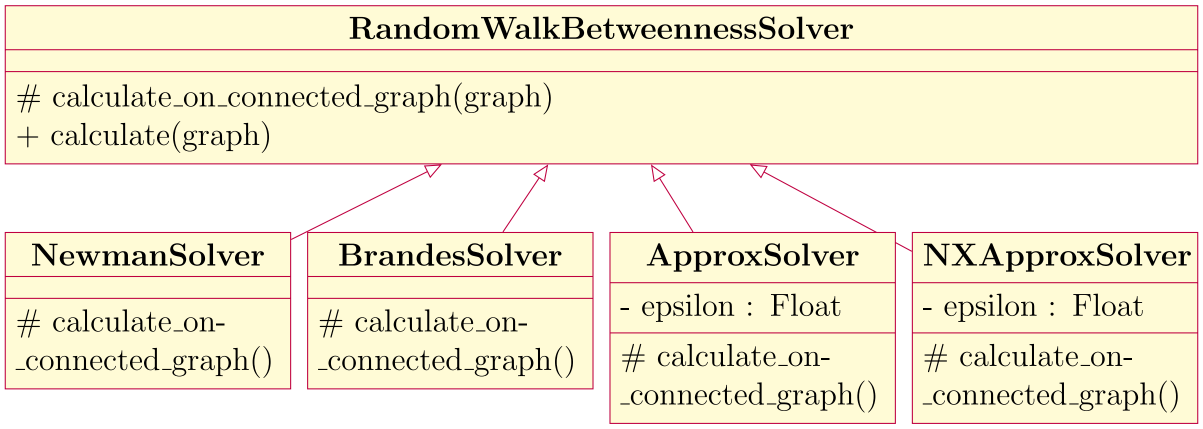

RandomWalkBetweennessSolver
# calculate_on_connected_graph(graph)
+ calculate(graph)
NewmanSolver
# calculate_on-_connected_graph()
BrandesSolver
# calculate_on-_connected_graph()
ApproxSolver
- epsilon : Float
# calculate_on-_connected_graph()
NXApproxSolver
- epsilon : Float
# calculate_on-_connected_graph()

Figure 3.2: Class diagram for `RandomWalkBetweennessSolver` and the classes that inherit from it.

3.1.2 Class hierarchy

Since the main aim of the project is implementing various algorithms that do the same thing in different ways, it is natural to use the Strategy design pattern. Each of the algorithms that I implement, I implement as the `calculate_on_connected_graph` method of a class that inherits from `RandomWalkBetweennessSolver`. This structure is shown in Figure 3.2.

- `RandomWalkBetweennessSolver` implements the `NetworkX` version of the Brandes-Fleischer algorithm
- `BrandesSolver` implements my version of the Brandes-Fleischer algorithm
- `NewmanSolver` implements the Newman algorithm
- `NXApproxSolver` implements the `NetworkX` version of the approximate Brandes-Fleischer algorithm
- `ApproxSolver` implements my version of the approximate Brandes-Fleischer algorithm

Note that the approximate algorithm classes have a member *epsilon*, which controls how precise the approximation is (discussed further in Section 3.3.3).

The `RandomWalkBetweennessSolver.calculate` function can be called with any graph as an argument and will return a Python dictionary mapping each node in the graph to the random walk betweenness of that node. This function addresses a number of potential sources of repeated code that I would have suffered from had I not used the Strategy pattern. Firstly, random walk betweenness is calculated for a disconnected graph by calculating it individually for each connected component of the graph. I avoid having to repeat the code that detects connected components and runs a random walk betweenness calculation algorithm on each component by putting this code in the `calculate` function. Secondly, it simplifies some of the logic in the random walk betweenness algorithms if the nodes of their input graph are labelled $\{0, 1, 2, \ldots, n-1\}$. Therefore I relabel the nodes of each connected component in the `calculate` function before passing the connected component to the `calculate_on_connected_graph` function (and then undoing the relabeling before returning). Algorithm 1 shows the logic of the `calculate` function.

Algorithm 1: `calculate()` function logic

```
Input: An undirected graph $G(V, E)$, which may be disconnected.
Output: A dictionary $d$ where $d(v) = b$ if $b$ is the random walk betweenness of
        node $v$, and $d$ contains an entry for each $v \in V$.
$output \leftarrow \{\}$
for $component \in connected\_components(G)$ do
    $mapping \leftarrow$ mapping from $nodes(component)$ to $\{0, 1, 2, \ldots\}$
    $inverse\_mapping \leftarrow inverse(mapping)$
    $relabelled\_component \leftarrow component.relabel\_nodes(mapping)$
    $relabelled\_result \leftarrow calculate\_on\_connected\_graph(relabelled\_component)$
    $result \leftarrow relabelled\_result.relabel\_nodes(inverse\_mapping)$
    $output.update(result)$
end
return $output$
```

3.2 Investigation of efficiency of different approaches

In this section I explain the design decisions I take with regards to methods for matrix inversion and solving linear systems. I take a quantitative approach to decision-making, bench-marking several possibilities and choosing the solution with the best running time.

3.2.1 Methods of inverting matrices

The Newman algorithm and the Brandes-Fleischer algorithm require the inversion of the constrained Laplacian of the graph they are run on (defined in Section 3.3.1), which is an $(n-1) \times (n-1)$ symmetric, positive-definite matrix. In particular, the time complexity of the Brandes-Fleischer algorithm is dominated by this matrix inversion. There are many highly optimised functions to do this in the `NumPy` and `SciPy` libraries, so I decide not to take up the fool's errand of trying to improve on these functions by re-implementing them. Instead, I simply decide which of them to use.

I investigate three different options: the `NumPy` `inverse()` function (for use on `NumPy` arrays), the `M.I` attribute of `NumPy` matrices, and the `SciPy` `spilu()` function.

Brandes et al. [5] suggest using an incomplete LU decomposition as a preconditioner for the conjugate gradient method. This is roughly what `spilu()` does.

- The **conjugate gradient method** (CGM) and **GMRES** are examples of iterative methods for solving linear systems of the form $\mathbf{Ax} = \mathbf{b}$ (where $\mathbf{A}$ is an $n \times n$ matrix, and $\mathbf{x}$ and $\mathbf{b}$ are n length column vectors). They can be used to invert matrices because $\mathbf{A}^{-1}$ is the solution to the equation $\mathbf{AX} = \mathbf{I}$, which can be split into a system of n linear equations of the form $\mathbf{Ax} = \mathbf{b}$, each of which can be solved independently.

- A **preconditioner** is a matrix (usually an approximation of $\mathbf{A}^{-1}$) that is used to improve the convergence speed of an iterative method.

- An **incomplete LU decomposition** finds sparse triangular matrices $\mathbf{L}$ and $\mathbf{U}$ such that $\mathbf{A} \approx \mathbf{LU}$. These can be easily inverted to acquire $\mathbf{U}^{-1}\mathbf{L}^{-1} \approx \mathbf{A}^{-1}$, which can be used as a preconditioner.

The `spilu()` function performs an incomplete LU decomposition and returns an object with a `solve()` method. Calling this solve method with b as an argument solves the linear system $\mathbf{Ax} = \mathbf{b}$. `SciPy` does this using calls to `SuperLU`, which is a `C` library for LU decompositions. It is undocumented which iterative method this library uses, but it is likely to be GMRES because GMRES is used in the paper [22] referenced by `SuperLU`'s documentation. That said, the `spilu()` function is the closest I was able to find to Brandes et al.'s recommendation without creating my own implementation of an incomplete LU decomposition, which I consider to be beyond the scope of this project.

The `inverse()` function and the `M.I` attribute both use Gaussian elimination. In Figure 3.3, I compare the three methods. The analysis shows that the `spilu()` method is significantly slower than the other methods. `M.I` is shown to be slightly faster than `inverse()`, so I use `M.I` for matrix inversion in my implementations of the Newman algorithm and the Brandes-Fleischer algorithm.

3.2.2 Methods of solving linear systems

The approximate Brandes-Fleischer algorithm requires solving linear systems of the form $\mathbf{Ax} = \mathbf{b}$. The algorithm is of the form:

```
$\mathbf{A} \leftarrow \dots$
**for** $i \leftarrow 1$ **to** $k$ **do**
    $\mathbf{b} \leftarrow \dots$
    $\mathbf{x} \leftarrow$ solution to $\mathbf{Ax} = \mathbf{b}$
**end**
```

There are broadly two possible approaches – either solve the linear systems independently inside the **for** loop, or do some preprocessing outside of the **for** loop that will accelerate the individual calculations that are made inside the loop. The approaches I compare are:

- use `numpy.linalg.solve()` to solve each linear system;

- use `scipy.linalg.solve()` to solve each linear system;

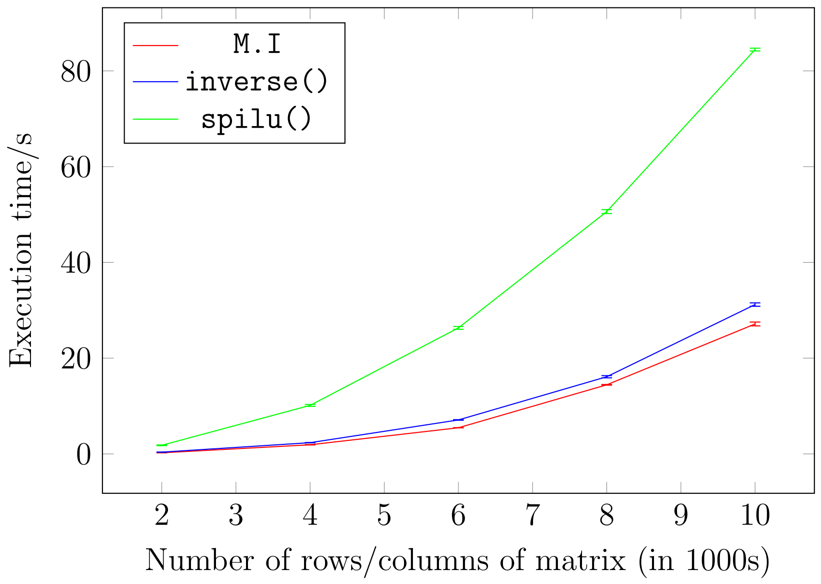


Figure 3.3: Graph comparing execution times of three methods of inverting an $N \times N$ symmetric, positive-definite matrix. Each matrix is the constrained Laplacian of an Erdös-Rényi graph with parameters $n = N + 1$ and $p = 10/(N + 1)$ (chosen to give roughly the same sparsity and structure of matrix that we expect in the execution of the algorithms). The error bars are $\pm 2 \times$ standard error of mean.

- use the `spilu()` function, calculating the preconditioner outside of the loop and using an iterative method inside the loop;
- calculate $\mathbf{A}^{-1}$ outside of the loop (using `M.I`), then use matrix multiplication to calculate $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ inside the loop.

Figure 3.4 shows a comparison between the first three of these methods. The `spilu()` method is much much faster than the other two methods per loop, and the time taken upfront is comparable to the time per loop for the other methods. Therefore I rule out the use of the `numpy.linalg.solve()` and `scipy.linalg.solve()` methods because if $k > 1$ then they will be slower than `spilu()`.

Figure 3.5 compares the remaining two methods: `spilu()` and `M.I`. The `spilu()` method is faster both for both the upfront cost and the per loop cost, therefore I use the `spilu()` method in my implementation of the approximate Brandes-Fleischer algorithm.

3.3 Calculating random walk betweenness

In this section, I describe each of the three algorithms that I implement in this project: the Newman algorithm, the Brandes-Fleischer algorithm, and the approximate Brandes-Fleischer algorithm. For each algorithm, I first describe the mathematical background of

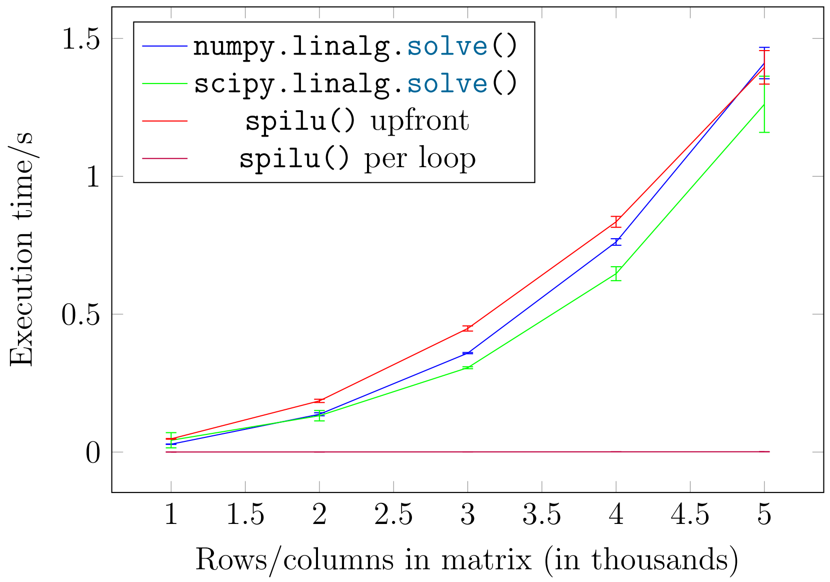


Figure 3.4: Graph comparing different methods of solving the $\mathbf{Ax} = \mathbf{b}$ system. Each matrix $\mathbf{A}$ is the constrained Laplacian of an Erdös-Rényi graph with parameters $n = N + 1$ and $p = 10/(N + 1)$. Note that `spilu()` appears twice because it has both an upfront cost and a cost per iteration of the loop. The other two methods only have a per loop cost. The error bars are $\pm 2 \times$ standard error of mean.

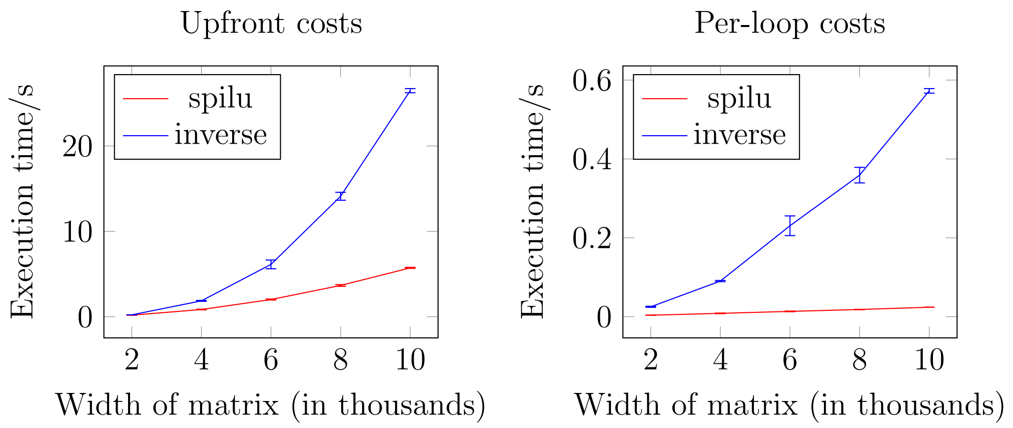


Figure 3.5: Graphs comparing two methods for repeatedly solving the $\mathbf{Ax} = \mathbf{b}$ system. These methods are (1) calculating a sparse LU decomposition and using it as a preconditioner for an iterative method (labelled *spilu*); and (2) finding $\mathbf{A}^{-1}$ and then using matrix multiplication to solve each instance of the system (labelled *inverse*). The left graph shows the execution time of the "one-off" calculation of the preconditioner or the inverse, and the right graph shows the execution time taken to solve each instance of the system. The error bars are $\pm 2 \times$ standard error of mean.

the algorithm, then describe the algorithm itself, then finally discuss specific details of my Python implementation of the algorithm.

3.3.1 The Newman algorithm

Mathematical definitions

For a graph $G(V, E)$, we let $n = |V|$ and $m = |E|$. The Laplacian matrix $\mathbf{L}$ of G is defined as:

$$\mathbf{L} = \mathbf{D} - \mathbf{A} \tag{3.1}$$

where $\mathbf{D}$ is a diagonal matrix of node degrees (i.e. $\mathbf{D}_{v,v} = \text{degree}(v)$) and $\mathbf{A}$ is the adjacency matrix (i.e. $\mathbf{A}_{v,w} = 1$ if $(v, w) \in E$ and 0 otherwise).
We define the matrix $\mathbf{T}$ as:

$$\mathbf{T} = \begin{pmatrix} 0 & \mathbf{0}^T \\ \mathbf{0} & \widetilde{\mathbf{L}}^{-1} \end{pmatrix} \tag{3.2}$$

where $\mathbf{0}$ is an $(n-1)$ length column vector with each element equal to 0, and the constrained Laplacian $\widetilde{\mathbf{L}}$ is $\mathbf{L}$ with the first row and column removed. The choice of which row and column to remove is arbitrary, but we pick the first for simplicity.
We define I_v^{st} as:

$$I_v^{st} = \begin{cases} \dfrac{1}{2} \displaystyle\sum_{\substack{w \text{ s.t.} \\ (v,w) \in E}} |T_{vs} - T_{vt} - T_{ws} + T_{wt}| & \text{if } v \neq (s, t) \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

Lastly, we produce an expression for random walk betweenness b_v:

$$b_v = \frac{\displaystyle\sum_{s<t} I_v^{st}}{(1/2)(n-1)(n-2)} \tag{3.4}$$

I have made some slight changes to the equations above: some to make them easier to follow, and others in the name of consistency with the other algorithms. In the original formulation $I_s^{st} = I_t^{st} = 1$, which corresponds to counting s and t as lying on the path from s to t. As a result, a constant factor is subtracted from the random walk betweenness of each node. This means that the minimum random walk betweenness is 0 rather than $2/(n-1)$. Having changed that, I also change the normalising factor in Equation 3.4 from $n(n-1)$ to $(n-1)(n-2)$ to account for the fact that we now want to average over all pairs s, t excluding those where $v = s$ or $v = t$. Making this change means that a random walk betweenness of 1 is still possible.

The algorithm

The Newman algorithm can be executed in $\mathcal{O}((I(n-1)+mn^2)$ time, where $I(n)$ is the time taken to invert a symmetric positive-definite $n \times n$ matrix. The major bottlenecks of the algorithm are (1) inverting $\widetilde{\mathbf{L}}$, and (2) calculating each I_v^{st}, which involves summing a total of $\mathcal{O}(mn^2)$ elements. In my implementation, I use the `M.I` method of matrix inversion as discussed in Section 3.2.1. This implies that $I(n) = \mathcal{O}(n^3)$.

I rearrange Equations 3.4 and 3.3 to give:

$$b_v = \frac{1}{(n-1)(n-2)} \sum_{(v,w)\in E} \sum_{\substack{s<t \\ s,t\neq v}} |T_{vs} - T_{vt} - T_{ws} + T_{wt}| \tag{3.5}$$

From this I derive Algorithm 2, which is what I implement in the `NewmanSolver` class. Note that I make use of the fact that the term for edge (v,w) in I_v^{st} is equal to the term for the same edge in I_w^{st}. Thus, rather than iterate other all the edges in both directions, I do both directions at once, roughly halving the number of times $|T_{vs} - T_{vt} - T_{ws} + T_{wt}|$ needs to be computed.

Algorithm 2: The Newman algorithm

```
Input: An undirected connected graph $G(V,E)$ where $V = \{0,1,2,\ldots,n-1\}$.
Output: A dictionary mapping nodes to their random walk betweenness.
compute $\widetilde{\mathbf{L}}$
$\mathbf{T} \leftarrow \begin{pmatrix} 0 & \mathbf{0}^T \\ \mathbf{0} & \widetilde{\mathbf{L}}^{-1} \end{pmatrix}$
initialise $b[v] \leftarrow 0$ $(\forall v \in V)$
for $s < t$ do
    for $(v,w) \in E$ do
        $temp \leftarrow |T_{vs} - T_{vt} - T_{ws} + T_{wt}|$
        if $v \neq s,t$ then
            $b[v] \leftarrow b[v] + temp$
        end
        if $w \neq s,t$ then
            $b[w] \leftarrow b[w] + temp$
        end
    end
end
for $v = 0$ to $(n-1)$ do
    $b[v] \leftarrow \frac{b[v]}{(n-1)(n-2)}$
end
return $b$
```

Python implementation

In Python, I am able to use `NumPy` to eliminate the outer **for** loop, instead turning it into significantly faster vector operations. I use the following statement to create two `NumPy` arrays s and t, which when taken together contain all the pairs of nodes such that $s < t$. The `itertools` library is used to generate all of the combinations of two elements taken from the list $[0, 1, \ldots, n-1]$ (generated using `numpy.arange(n)`). There is then some conversion required to convert the result of this into a `NumPy` array with dimensions $(2, \frac{n(n-1)}{2})$.

```
s, t = numpy.fromiter(
    itertools.combinations(numpy.arange(n), 2), dtype='i,i'
).view(numpy.int).reshape(-1, 2).transpose()
```

These `NumPy` arrays are then used as follows. The first line creates an array of temps with an entry for each $s < t$. The second line filters the temp array by indexing into it with a Boolean array so that the entries remaining are the ones where $v \neq s, t$, then sums up the resulting array and increments $b[v]$.

```
temp = numpy.abs(T[v, s] - T[v, t] - T[w, s] + T[w, t])
b[v] += numpy.sum(temp[(s != v) & (t != v)])
b[w] += numpy.sum(temp[(s != w) & (t != w)])
```

3.3.2 The Brandes-Fleischer algorithm

Mathematical definitions

We carry over the definition of $\mathbf{T}$ from Section 3.3.1.
Define the matrix $\mathbf{B}_{ve}$ for $v \in V$ and $e \in E$:

$$\mathbf{B}_{ve} = \begin{cases} +1 & \text{if } e = (v, w) \\ -1 & \text{if } e = (w, v) \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

Now let $\mathbf{F}$ be:

$$\mathbf{F} = \mathbf{BT} \tag{3.7}$$

We use $\mathbf{F}_{e\_}$ as notation for the row of $\mathbf{F}$ corresponding to edge e. We define $\text{pos}(e, v)$ as the position of $\mathbf{F}_{ev}$ within the list constructed by sorting $\mathbf{F}_{e\_}$ in decreasing order using a stable sorting algorithm. Figure 3.6 gives an illustrative example of how $\text{pos}(e, v)$ is calculated.

We give each edge an arbitrary direction and define source and target such that for $e = (v, w)$ we have $\text{source}(e) = v$ and $\text{target}(e) = w$.

$$\begin{aligned} \text{if } \mathbf{F}_{e\_} &= [1,8,3,5,3] \\ \text{then sorted}(\mathbf{F}_{e\_}) &= [8,5,3,3,1] \\ \text{and pos}(e,\_) &= [5,1,3,2,4] \end{aligned}$$

Figure 3.6: Example of how $\text{pos}(e, v)$ is calculated. Note that there are two instances of the number 3 in the list and they result in the $\text{pos}(e, v)$ values 2 and 3 because those are the positions they are moved to by the stable sort.

The algorithm

This algorithm runs in time $\mathcal{O}(I(n-1) + mn\log n)$ time. Again, I use the `M.I` method of matrix inversion, so that $I(n) = n^3$. Calculating $\text{pos}(e, v)$ takes $\mathcal{O}(mn \log n)$ time because it requires sorting $\mathbf{F}_{e,\_}$ (an array of length n) for each edge e.

Algorithm 3 differs slightly from that of the original paper [5], which used labeling of $\{1, 2, \ldots, n\}$ for the nodes. For consistency with the other algorithms, I use indexing starting at 0, which requires some small off-by-one tweaks. The original paper also made the decision to remove the last row and column from $\mathbf{L}$ rather than the first. This decision is arbitrary, so I have kept all of the algorithms the same for consistency, and so that they can share some of the code for calculating $\widetilde{\mathbf{L}}$ and $\mathbf{T}$.

Python implementation

Fortunately, the `SciPy` library provides a function that can be used to find $\text{pos}(e, v)$ – `scipy.stats.rankdata()`. The `method="ordinal"` mode tells the function to break ties between two equal elements based on their position in the original list. I pass the function `-row` rather than `row` in order to "trick" it into ranking highest to lowest rather than lowest to highest.

```
    row = F[e, :]
    pos = scipy.stats.rankdata(-row, method="ordinal")
```

The inner **for** loop is neatly converted into `NumPy` vector operations as follows. Here, `v` is a `NumPy` array of the form $[0, 1, \ldots, n-1]$. The `dot()` function is used to multiply two arrays element-wise, that is, it computes the dot product.

```
    v = numpy.arange(n)
    b[source] += numpy.sum((v + 1 - pos).dot(row))
    b[target] += numpy.sum((n - v - pos).dot(row))
```

Algorithm 3: The Brandes-Fleischer algorithm

```
Input: An undirected connected graph $G(V, E)$ where V = {0, 1, 2, ..., n-1}.
Output: A dictionary mapping nodes to their random walk betweenness.

compute $\widetilde{\mathbf{L}}$
$\mathbf{T} \leftarrow \begin{pmatrix} 0 & \mathbf{0}^T \\ \mathbf{0} & \widetilde{\mathbf{L}}^{-1} \end{pmatrix}$
$\mathbf{F} = \mathbf{BT}$
initialise $b[v] = 0$ $(\forall v \in V)$
for $e \in E$ do
    $row \leftarrow \mathbf{F}_{e_-}$
    calculate $pos(e, v)$ by sorting $row$
    for $v \leftarrow 0$ to $(n-1)$ do
        increase $b[source(e)]$ by $(v + 1 - pos(e, v)) \cdot row[v]$
        increase $b[target(e)]$ by $(n - v - pos(e, v)) \cdot row[v]$
    end
end
for $v = 0$ to $(n-1)$ do
    $b[v] \leftarrow \frac{2(b[v] + 1 - v)}{(n-1)(n-2)}$
end
return $b$
```

3.3.3 The approximate Brandes-Fleischer algorithm

Mathematical definitions

The approximate Brandes-Fleischer algorithm has a parameter ϵ and a constant l. The rough format of the algorithm is to select a pair $s, t \in V$ at random and calculate the contribution of this pair towards the random walk betweenness of each node. Repeating this gradually converges on the true values of random walk betweenness.

We define k, which is used to decide how many iterations of the main loop of the algorithm need to be performed in order to achieve the required accuracy of approximation.

$$c^* = \frac{n}{n-2} \tag{3.8}$$

$$k = l \cdot \lceil (c^*/\epsilon)^2 \log n \rceil \tag{3.9}$$

Similarly to the matrix $\mathbf{B}$ used in the Brandes-Fleischer algorithm, we define the n length column vector $\mathbf{b}_{st}$:

$$\mathbf{b}_{(s,t)}(v) = \begin{cases} +1 & \text{if } v = s \\ -1 & \text{if } v = t \\ 0 & \text{otherwise} \end{cases} \tag{3.10}$$

Let $\widetilde{\mathbf{b}}_{(s,t)}$ be an $n-1$ length column vector constructed by removing the first entry from $\mathbf{b}_{(s,t)}(v)$.

The algorithm

Given parameter ϵ and constant l, the approximate Brandes-Fleischer algorithm returns $\mathbf{b}^*$ such that for any $\mathbf{v} \in V$:

$$\mathbb{P}\left(|\mathbf{b}^*[v] - \mathbf{b}[v]| \geq \epsilon\right) \leq \frac{1}{n^{2l}} \tag{3.11}$$

where $\mathbf{b}[v]$ is the true random walk betweenness of node v. Thus this algorithm meets the criteria for the absolute error definition of an FPRAS as defined in Section 2.1.1.

A full pseudocode description of the approximate Brandes-Fleischer algorithm is found in Algorithm 4.

Algorithm 4: The approximate Brandes-Fleischer algorithm

```
Input: An undirected connected graph $G(V,E)$ where V = $\{0,1,2,\ldots,n-1\}$.
Output: A dictionary mapping nodes to an approximation of their random walk
        betweenness.

initialise $b[v] = 0$ $(\forall v \in V)$
$k \leftarrow l \cdot \lceil (c^*/\epsilon)^2 \log n \rceil$
for $i \leftarrow 1$ to $k$ do
    select $s \neq t$ uniformly at random from $V$
    solve $\widetilde{\mathbf{L}}\widetilde{\mathbf{p}} = \widetilde{\mathbf{b}}_{st}$
    for $v \in V \setminus \{s,t\}$ do
        for $e = (v,w) \in E$ do
            increase $b[v]$ by $|\widetilde{\mathbf{p}}(v) - \widetilde{\mathbf{p}}(w)| \cdot (c^*/2k)$
        end
    end
end
return $b$
```

Python implementation

I make `epsilon` a parameter that can be set when the `ApproxSolver` class is initialised, defaulting to 0.05. I let the constant l be equal to 1 in order to remain consistent with the `NetworkX` implementation of the same algorithm.

```
class ApproxSolver(RandomWalkBetweennessSolver):
    def __init__(self, epsilon=0.05):
        self.epsilon = epsilon
```

I am able to eliminate both of the inner loops using `NumPy`. Most of the techniques used are similar to those used in the last two algorithms, with the exception of the use of the `numpy.bincount()` function that doesn't feature in either. The following snippet takes an array of nodes `v` that contains repeats and a list of values `val`. It sums up the values for each unique node and adds the result to the betweenness of the node.

```
betweennesses += numpy.bincount(v, val, minlength=B.size)
```

The `val` array contains terms calculated with the formula `val` $= |\widetilde{\mathbf{p}}(v) - \widetilde{\mathbf{p}}(w)|$.

3.4 Supplementary tools

3.4.1 Graphs

I maintain a library of graphs, both those from real world data and those that are created synthetically. They are stored in the `graphs/resources` directory in an adjacency list format. They can be read by file name using the `read_graph()` function, for example `read_graph("kite_graph")` will return the kite graph (visualised in Figure 2.1) in the form of a `NetworkX Graph`.

The real world graphs described in Section 2.2.4 were put into the `graphs/resources` directory manually. The synthetic graphs; however, were created by scripts utilising existing `NetworkX` functions. Random graphs using the Erdös-Rényi, Watts-Strogatz, and Holme-Kim models (as discussed in Section 2.1.4) are generated by scripts in the `random_graphs.py` file. For example calling `get_erdos_renyi(n=100, average_degree =10)` returns an Erdös-Rényi graph generated with parameters $n = 100$ and $p = 0.1$ as a `NetworkX Graph`. There are other similar functions that generate random graphs and write them to the `graphs/resources` directory.

Small graphs of just a few nodes, which I use for testing and debugging purposes, are generated by scripts in the `small_graphs.py` file. For example, the `generate_kite_graph` function generates the kite graph (visualised in Figure 2.1) and stores it to the `graphs/resources` directory. These small graphs are also generated using existing `NetworkX` functions.

```
python main.py
            [--graph_name {bull_graph, erdos_renyi, kite_graph, ...}]
            [--method_name {nx, newman, brandes, approx, nxapprox}]
            [--output {csv, vis, time}]
```

Figure 3.7: All available options for running the command-line tool.

3.4.2 Visualisation tool

The visualisation tool produces a diagram from a graph, with nodes represented by circles and edges represented by lines between the circles. The nodes are coloured depending on their random walk betweenness. The nodes are positioned using a force-directed placement algorithm due to Fruchterman and Reingold [10] that is implemented in the `NetworkX` library. This algorithm models edges as springs that pull nodes closer to their neighbours, and models nodes as repellent to each other. In order to produce this tool, I made use of `NetworkX`'s strong integration with `Matplotlib`, which is a visualisation library.

The visualisation tool can be used in two ways: via the command-line tool, or by using the `draw_graph(graph, strategy="nx")` function. The `graph` parameter can either be a string representing the file name of a graph in the library, or a `NetworkX Graph`. The strategy parameter can be used to select which random walk betweenness algorithm is used to calculate the colours of the nodes. Calling `draw_graph("kite_graph")` results in the diagram in Figure 2.1.

3.4.3 Command-line tool

The command-line tool can be used to run any of the algorithms I have implemented on any of the graphs in the library of graphs. It has three modes of output: putting the results in a `.csv` file, displaying a visualisation of the graph created using the visualisation tool, and printing the time taken to run the algorithm to the terminal. Figure 3.7 shows the available options for running the tool. If any of the options are not put into the command line, then the user will be prompted to choose a value for that option from a list of possibilities.

If the user wanted to create the visualisation pictured in Figure 2.1, they could run the command:

```
$ python main.py --graph_name kite_graph --method_name brandes --output vis
```

Equally, if the user wanted to save the results of the approximate Brandes-Fleischer algorithm running on the kite graph to a file, they could run the command:

```
$ python main.py --graph_name kite_graph --method_name approx --output csv
```

and enter a suitable file name for the result when prompted.

3.4.4 Timing

There are two different aspects to timing in this project – timing the code as a whole, and profiling it to see which parts of it are fast and which are slow. For the former, I

```
p = Profiler()
# thing to time 1
p.mark("thing1")
for i in range(1000):
    # thing to time 2
    p.mark("thing2")
    # thing to time 3
    p.mark("thing3")

print(p.get_data())
# output: {"thing1": 0.42, "thing2": 1.21, "thing3": 0.02}
```

Figure 3.8: Example usage of `Profiler` class

use the `time_random_walk_betweenness_algorithm` function, and for the later I use the `Profiler` class. Both of these use measurements of the current time given to them by the `timeit` library.

The `time_random_walk_betweenness_algorithm` function is used throughout the Evaluation chapter to measure execution times. It takes two parameters: a graph and a strategy. The graph can be either be a string that is the name of a graph in the library, or it can be a `NetworkX` graph. The strategy must be a string that refers to one of the random walk betweenness algorithms. This function returns a dictionary containing the following data about the execution of the algorithm: the name of the graph (if it was one from the library), the strategy that was used, the time that was taken, and the number of nodes and edges in the graph.

I use the `Profiler` class to compare the relative speeds of different parts of an algorithm in order to find opportunities for improvement. Using it involves inserting code into an algorithm: first initialising the `Profiler`, then marking the end of each section of the algorithm that you wish to time, then finally getting the data out of the `Profiler`. Figure 3.8 gives an example of how the `Profiler` class is used. The output of the `get_data` function is a dictionary where the values are the time taken for each section of code in seconds.

3.5 Summary

The approach to this project has been consistently scientific and professional. When there have been multiple possible methods of achieving the desired result, I have performed quantitative tests in order to find the best method. I have made use of software engineering techniques such as using the Strategy design pattern, and the `NumPy` library has been utilised to translate pseudocode algorithms into efficient code.

Chapter 4

Evaluation

In this chapter, I evaluate the performance of the three algorithms that I have implemented. I also compare their performance to existing implementations within the `NetworkX` library. Firstly, I compare execution times when each algorithm is run on an assortment of random graphs and real world datasets. I then investigate the accuracy of the approximating algorithm that I have implemented. Finally, I perform partial stress-testing to find the maximum size of graph that each algorithm can withstand.

All experiments in this chapter are run on my personal laptop, which is an MSI GP62 7RD Leopard (i7 core, 2GB graphics card, 8GB RAM, 128GB SSD) running Windows 10. Experiments are run with the machine plugged in to the mains and no other programs running. These conditions aim to make comparisons between results valid and relevant.

4.1 Execution time on random graphs

In this section, I compare the execution time of the algorithms implemented in this project when run on random graphs of various sizes. The results show that my implementations of the Brandes-Fleischer algorithm and the approximate Brandes-Fleischer algorithm outperform their respective `NetworkX` counterparts by a significant margin.

4.1.1 Exact methods

The exact methods in this project are the Newman algorithm, my implementation of the Brandes-Fleischer algorithm and the `NetworkX` implementation of the Brandes-Fleischer algorithm. For brevity, these are sometimes referred to as **newman**, **brandes**, and **nx** respectively throughout the Evaluation chapter.

Figure 4.1 shows execution times of these exact algorithms when they are run on the three types of random graph discussed in Section 2.1.4 – Erdös-Rényi, Watts-Strogatz, and Holme-Kim random graphs.

To generate a graph of N nodes, I use parameters $n = N,\ p = 10/N$ for Erdös-Rényi graphs; $n = N,\ K = 10,\ p = 0.1$ for Watts-Strogatz graphs; and $n = N,\ m =$

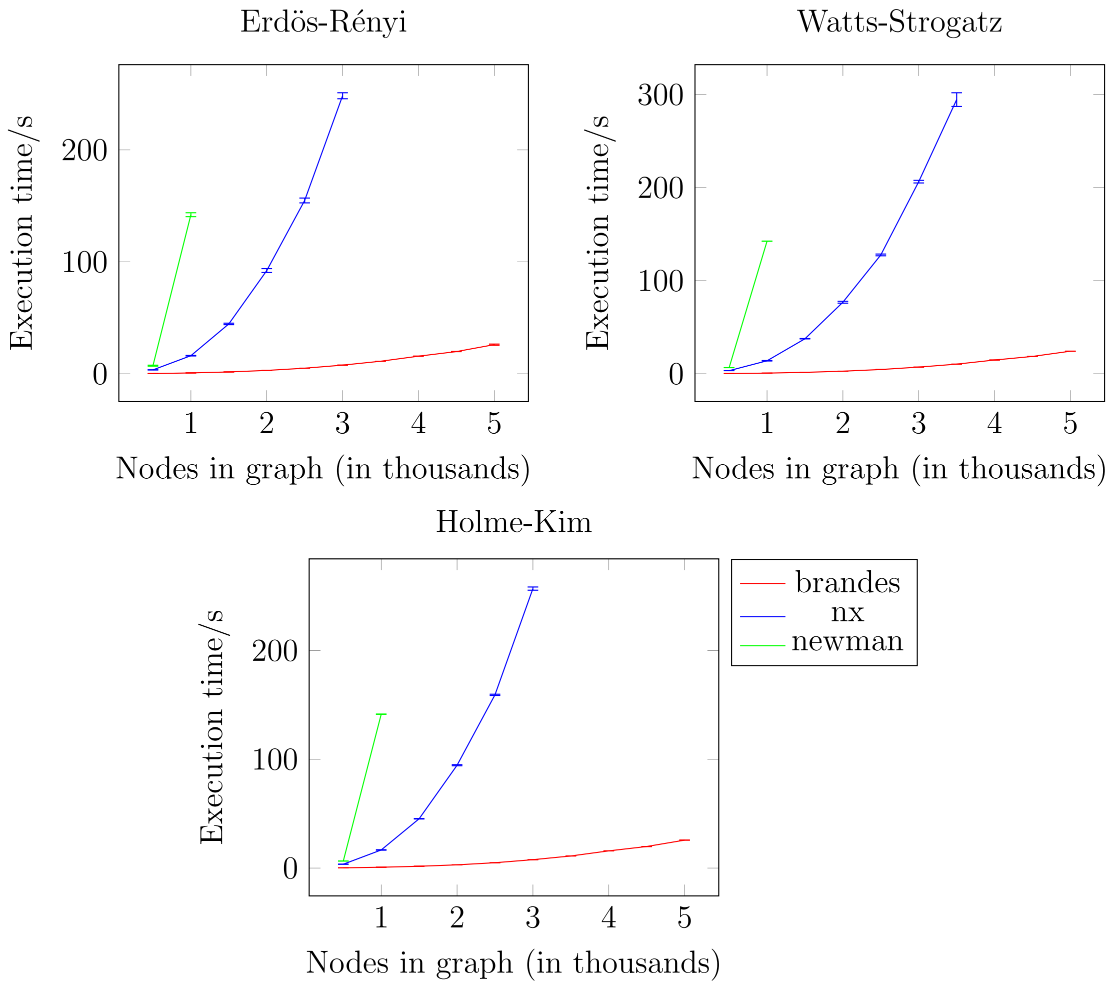


Figure 4.1: Plots comparing execution time of each of the exact methods on three types of random graph. Error bars are $\pm 2 \times$ standard error of mean.

5, $p = 0.1$ for Holme-Kim graphs. These parameters are selected to keep the total number of edges approximately equal between graphs with the same number of nodes. I limit my investigation to sparse graphs here because in the real world almost all networks are sparse.

I expect each of these graphs to behave as if they are made up of a single connected component. This is because there are so few nodes outside of the largest component that the time spent calculating random walk betweenness of these nodes is negligible.

- An Erdös-Rényi graph with $np = 10$ has a giant component with expected size sn where s is the non-zero solution to $1 - s = e^{-10s}$ ($s \approx 0.99996$) [7]. Therefore, there will be very few nodes outside of the largest component.
- A Watts-Strogatz graph with $K > \log n$ is almost certainly connected [27]. The graphs we consider here have sufficiently few nodes that this is the case for $K = 10$, so the Watts-Strogatz graphs each consist a single connected component.
- There are no results in the literature about the connectedness of Holme-Kim graphs. In lieu of a proof I have generated 1,000 graphs with $N = 1{,}000$ and 1,000 with $N = 10{,}000$. All of these graphs were connected.

For each type of random graph and number of nodes, five random graphs are generated. Then each algorithm is run on each of these graphs, and the execution time measured using the `time_random_walk_betweenness_algorithm` function (see Section 3.4.4). The number of nodes is increased from 500 to 5,000 in increments of 500. If the execution of any algorithm takes longer than 300s, it is considered a failure and that algorithm is not run on any larger graphs.

The plots show a clear hierarchy of the algorithms – **brandes** is the fastest, followed by **nx**, with **newman** being the slowest. A question of interest is why **brandes** is so much faster than **nx** given that they are both implementations of the Brandes-Fleischer algorithm. Looking at the code in the `NetworkX` library, it is likely that the difference is due to the `NetworkX` implementation being a more direct translation of what is written in Brandes et al.'s paper [5]. The `NetworkX` implementation uses **for** loops where I have used much faster vector operations courtesy of `NumPy`.

4.1.2 Approximate methods

The approximate methods in this project are my implementation of the approximate Brandes-Fleischer algorithm, and the `NetworkX` implementation of the approximate Brandes-Fleischer algorithm. For brevity, these are sometimes referred to as **approx** and **nxapprox** respectively throughout the Evaluation chapter.

Figure 4.2 shows execution times of these approximate algorithms when they are run on random graphs, using the same methodology as Figure 4.1. The parameter *epsilon* is fixed at 0.05 for both **approx** and **nxapprox**.

These plots show that **approx** vastly outperforms **nxapprox** despite them being implementations of the same algorithm. Similarly to the difference between **brandes** and

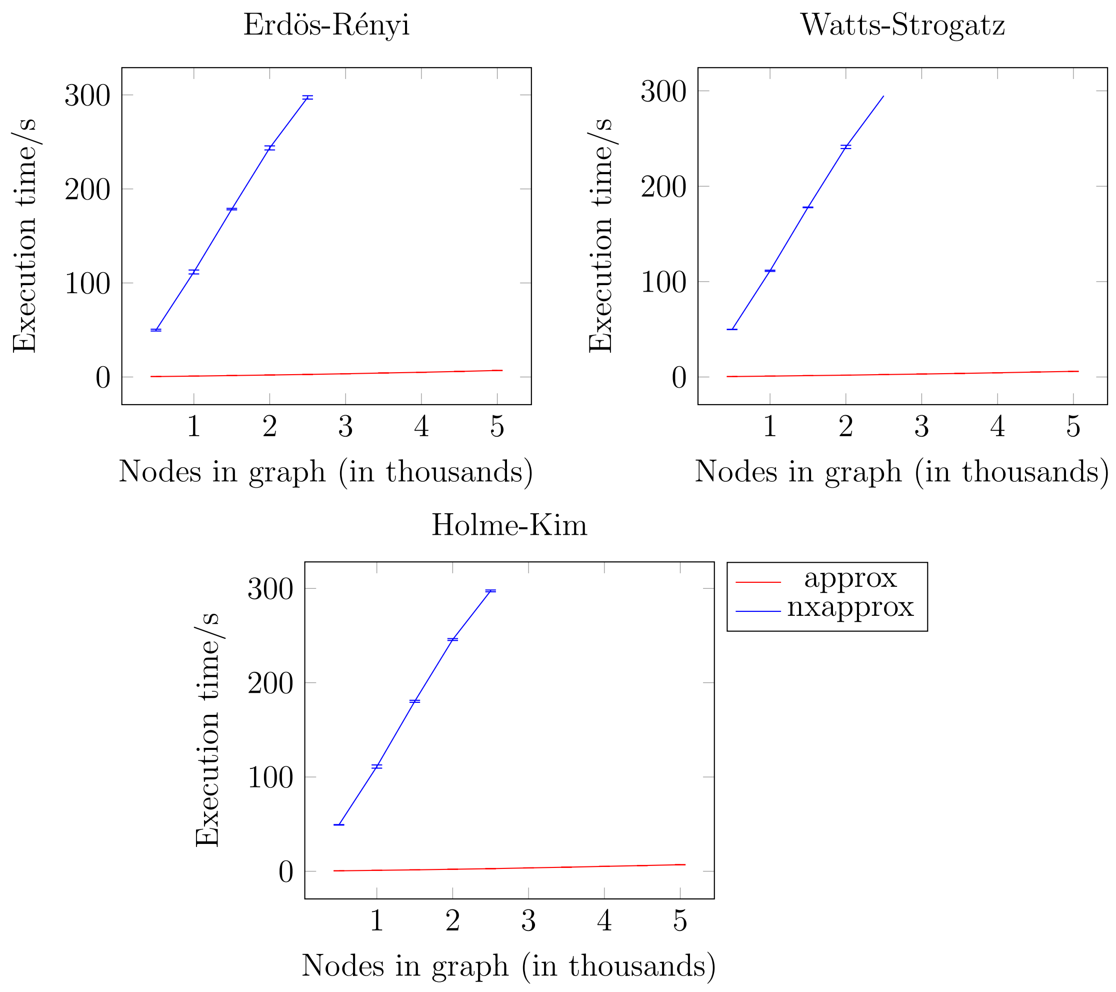


Figure 4.2: Plots comparing execution time of each of the approximate methods on three types of random graph. Error bars are ±2 × standard error of mean.

Dataset	Nodes	Edges	**brandes** run time	**approx** run time
Email-EU	1,005	16,064	2.6s (±0.1)	2.34s (±0.03)
GRQC-Collab	5,242	14,484	11.7s (±0.4)	22.2s (±0.2)
Gnutella-P2P	10,876	39,994	790s (±50)	17.5s (±0.8)

Figure 4.3: Table showing execution times of **brandes** and **approx** on real world datasets. The parameter `epsilon` was set to 0.05 for the **approx** method. 5 trials were performed for each combination of method and dataset. Error is $\pm 2 \times$ standard error of mean.

nx, this is likely due to a lack of vectorisation in **nxapprox**. I hasten to clarify that the reason the authors of **nx** and **nxapprox** wrote these functions the way they did is unlikely to be due to their lack of knowledge or expertise. It is much more likely that they simply prioritised minimising their time spent writing this code since they predicted that it would not be commonly used.

4.2 Execution time on real world datasets

My implementation of the Brandes-Fleischer algorithm is significantly faster than the `NetworkX` implementation of the same algorithm. On the Email-EU dataset (1005 nodes), my implementation runs in 2.26s (±0.02) seconds versus the 41.2s (±0.2) seconds taken by the `NetworkX` implementation, a speed-up of 18.2 times (calculated across 10 samples, error is $2 \times$ standard error of mean).

Figure 4.3 compares the run times of the **brandes** and **approx** methods on each of the real world datasets. The Enron-Email dataset is excluded since it is too large for the **brandes** method to be run on it.

4.3 Accuracy

A key feature of the approximate Brandes-Fleischer algorithm that must be evaluated is the accuracy of the results it gives, that is, how far from the true values of random betweenness centrality the output of the algorithm is. In this section, I consider absolute error to be my primary measure of accuracy. Absolute error can be calculated for each node in a graph using true values of random walk betweenness from an exact method and approximate values from an approximate method as follows:

$$residual(v) = true\_value(v) - approximate\_value(v) \tag{4.1}$$

$$absolute\_error(v) = |residual(v)| \tag{4.2}$$

The only two approximating algorithms in this project are **approx** and **nxapprox**, which are both implementations of the same algorithm. Since these will have the same accuracy, there is no suitable comparison to be made between different approximate methods.

Dataset	Number of nodes	Mean absolute error	99th percentile absolute error	99.9th percentile absolute error	Maximum absolute error
EU-Email	1,005	0.0004	0.0032	0.0060	0.0060
GRQC-Collab	5,242	0.0009	0.0163	0.0370	0.0460
Gnutella-P2P	10,876	0.0003	0.0013	0.0021	0.0045

Figure 4.4: Table showing statistics pertaining to the accuracy of the approximate Brandes-Fleischer algorithm when run on real world datasets, with parameter *epsilon* $= 0.05$. All columns are given with a precision of 10^{-4}, with the exception of the number of nodes, which is given exactly.

Figure 4.4 shows statistics about absolute errors for the real world datasets introduced in Section 2.2.4. The Enron-Email dataset is excluded because it is too large to run any of the exact algorithms on, so there is no way to find the true value of random walk betweenness of a node. The data in the table was collected by running **approx** and **brandes** on each dataset, where the results produced by **brandes** were used as the true values. A value of 0.05 was used for the parameter of **approx**, meaning that (recalling Section 3.3.3) we expect that a high proportion ($\geq 1 - 1/n^2$) of the absolute errors will be less than 0.05. All of the absolute errors are smaller than 0.05, fulfilling this expectation.

Figure 4.5 shows a histogram of the residuals for the Gnutella-P2P dataset (again calculated with parameter *epsilon* $= 0.05$). The distribution we observe has a large spike at zero, which represents the 1990 nodes with random walk betweenness of zero, which **approx** always gets exactly right. 98% (2 s.f.) of the remaining residuals are overestimates. Creating the same plot with different datasets produces histograms with different skews. For example, for the EU-Email dataset, 80% of the approximations of non-zero values are underestimates.

4.4 Comparison of exact to approximating methods

It is clear from the comparisons in Section 4.1 that the fastest exact algorithm is **brandes** and the fastest approximate algorithm is **approx**. In this section I compare these two approaches, considering in particular the trade-off between accuracy and speed (controlled by the **approx** method's parameter *epsilon*).

Figure 4.6 shows a comparison between the execution times of these two methods for varying sizes of graph, and for varying values of the parameter *epsilon*. A graph with N nodes is generated by creating an Erdös-Rényi graph with parameters $n = N$ and $p = 10/N$. All measurements are taken from a mean of five trials. A score is calculated using the formula:

$$\text{score} = \log_2 \frac{\text{time taken for approximate algorithm}}{\text{time taken for exact algorithm}}$$

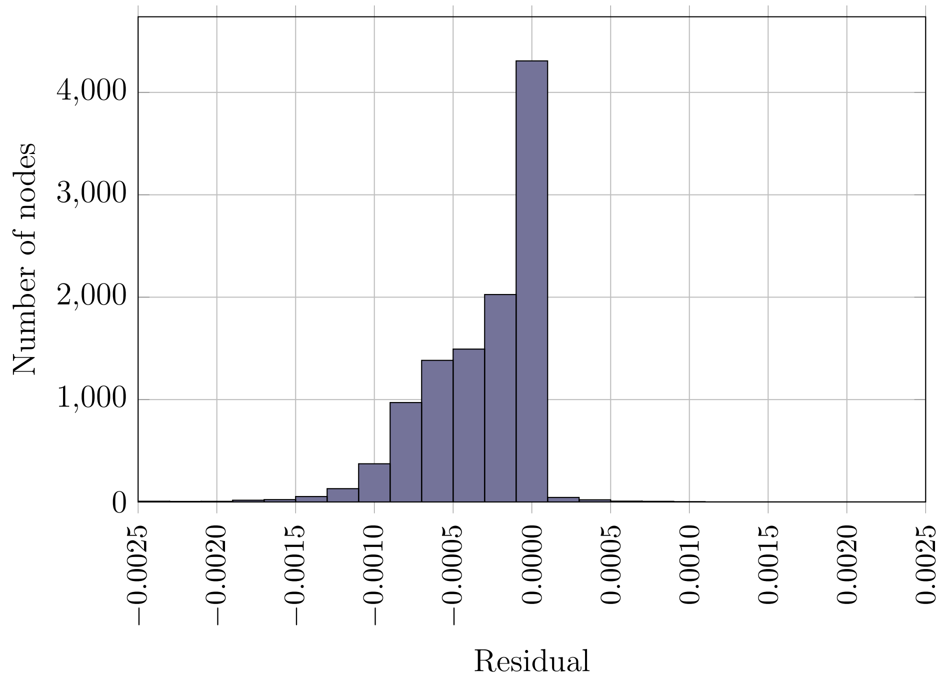


Figure 4.5: Histogram showing the distribution of residuals when the approximate Brandes-Fleischer algorithm is run on the Gnutella-P2P dataset (10,876 nodes) with parameter $epsilon = 0.05$.

so that a positive value means that **brandes** was faster and a negative value means that **approx** was faster. The scale is logarithmic, so a score of $+1$ means that the exact algorithm was 2 times faster and a score of -3 means that the approximate algorithm was $2^3 = 8$ times faster.

As we would expect from the asymptotic complexities of the algorithms, **approx** performs better for larger graphs, and when a lower precision is required. In many cases, it is several orders of magnitude faster.

4.5 Maximum graph size

In this section, I investigate the maximum size of graph that can be run with the **brandes** and **approx** algorithms.

The **brandes** method takes $\mathcal{O}(n^2)$ space. As a result of this, it is not suitable for very large graphs. For example, when run on a graph of $18,000$ nodes and $90,000$ edges, the algorithm failed with a `MemoryError`. However, I have successfully run **brandes** on a graph of 12,000 nodes and 60,000 edges, which took 2 hours and 33 minutes.

The **approx** method, on the other hand, only uses $\mathcal{O}(m)$ space. Therefore, graphs would have to get extremely large before it would cause a `MemoryError`. Running **approx** on the Email-Enron dataset (which is significantly bigger than the **brandes** method can

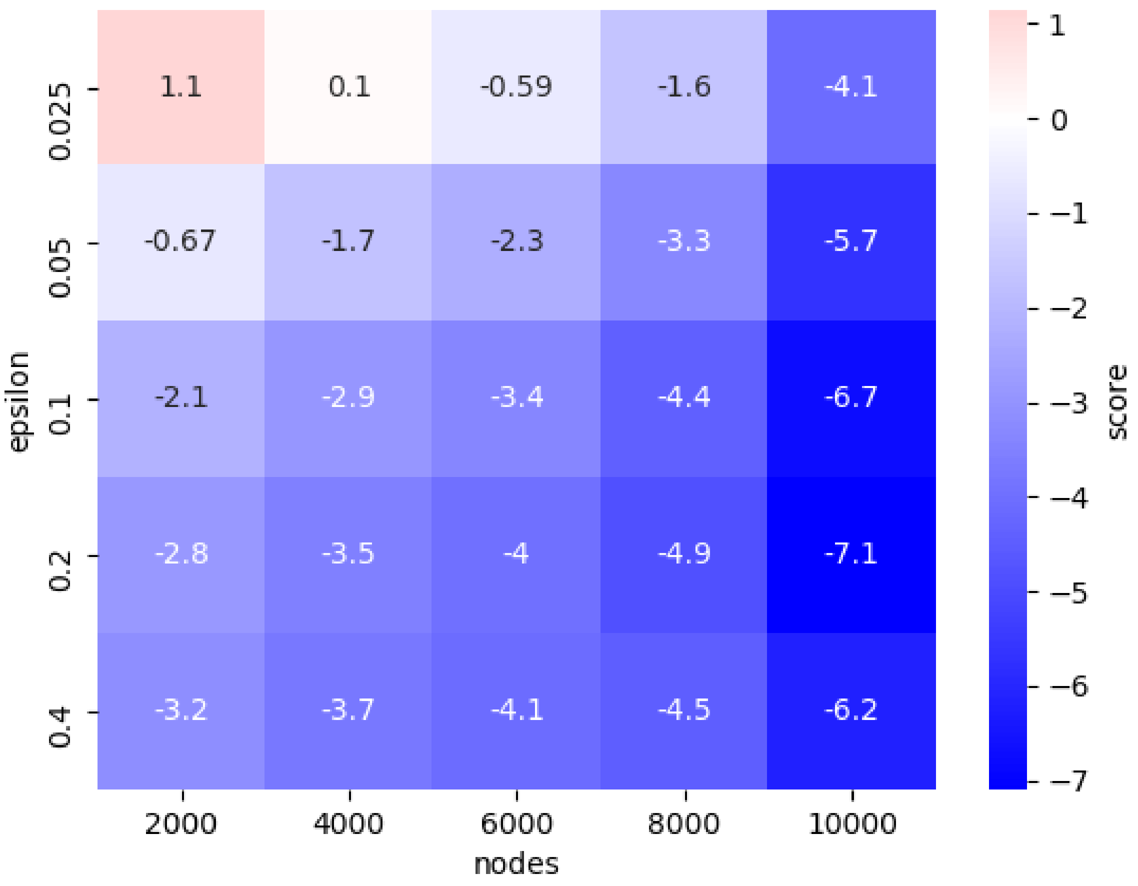


Figure 4.6: Heat map comparing run times of **brandes** with run times of **approx**. The approximate algorithm's parameter *epsilon* was varied along with the number of nodes in the graph the algorithms were run on.

handle at 36,692 nodes) takes 41.8s (± 0.1) using parameter $epsilon = 0.5$ (5 trials, error is $2 \times$ standard error of mean). Since changing *epsilon* only changes the number of times the main loop is run, this can obtain an upper bound on the running time on this dataset of about 70 minutes were we to use $epsilon = 0.05$.

I have successfully run the **approx** method on a graph with 200,000 nodes and 1,000,000 edges, which took 1 hour and 58 minutes with $epsilon = 0.5$. Using a smaller value of epsilon would take an unreasonably long time, suggesting that the usefulness of the **approx** method is bounded for very large graphs by the time it takes rather than the space it uses.

4.6 Summary

My implementations of the Brandes-Fleischer algorithm and the approximate Brandes-Fleischer algorithm outperform their `NetworkX` counterparts handily. The **brandes** method provides exact values of random walk betweenness, but cannot be run on graphs of above a certain size (somewhere between 12,000 and 18,000 nodes on my machine) due to memory constraints. The **approx** method is suitable when either graphs are sufficiently large, or the accuracy required is sufficiently low.

Chapter 5

Conclusions

5.1 Achievements

The project has been a success. I have achieved all the success criteria set out in Section 2.2.1; I implemented three algorithms and evaluated them based on their run time and accuracy, and I produced one tool for visualisation and another tool for running the algorithms from the command line.

Moreover, my implementations of the Brandes-Fleischer algorithm and the approximate Brandes-Fleischer algorithm have been demonstrated to outperform existing Python implementations found in the `NetworkX` library. This broadens the horizons of what networks it is feasible to analyse using random walk betweenness within a Python environment. The approximate Brandes-Fleischer algorithm has been shown to meet and exceed its promised accuracy, so can be used as a trustworthy approximation where calculating random walk betweenness exactly would be slow or impossible.

After this project, I intend to adapt my code so that it can be added as a contribution to the `NetworkX` library. I'm very pleased to have the opportunity to make a contribution to the state of the art in Python network analysis.

5.2 Lessons learnt

Over the course of this project, I have learnt a great deal about writing good code using `NumPy`, especially about the importance of following the principle of "keep it simple, stupid". Each of the algorithms I implemented went through several stages of being rewritten throughout their life-cycles. At the end of the day, each of the final implementations was the version that was the clearest, most readable, and least complex.

Were I to do this project again, I would leave more time for the evaluation section of the project. With more time, I could have investigated what properties of a graph cause the skew of the approximations given by the approximate Brandes-Fleischer algorithm. I could also have done a better job of accurately measuring the maximum size of graph each algorithm can handle – those kinds of experiments take a very long time to run and prevent me from using my laptop whilst they're running.

5.3 Potential for future improvement

It is possible to make further improvements to the process of calculating random walk betweenness.

One way of doing this is through improving the implementation further. I suggest that it might be possible to find faster sparse LU implementation that can be used as a preconditioner to the conjugate gradient method by looking outside of the Python ecosystem. Also, undoubtedly one can improve upon my own use of the `NumPy` tool which is hugely powerful and which I do not claim to have a complete mastery of.

Another way to make progress is through finding and implementing new algorithms. Generally, there is theoretical research to be done in order to come up with new algorithms, however some work has already been done on other approximating algorithms. Lulli et al. [18] describe a distributed algorithm for approximating random walk betweenness. Bozzo et al. [4] describe an algorithm for approximating the inverse of $\widetilde{\mathbf{L}}$, which can be applied to the Newman algorithm and the Brandes-Fleischer algorithm. Implementing and evaluating these approaches is a natural extension to this project.

Bibliography

[1] D. Arrell and A. Terzic. Network systems biology for drug discovery. *Clinical Pharmacology & Therapeutics*, 88(1):120–125, 2010.

[2] M. A. Beauchamp. An improved index of centrality. *Behavioral Science*, 10(2):161–163, 1965.

[3] K. Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[4] E. Bozzo and M. Franceschet. Approximations of the generalized inverse of the graph laplacian matrix. *Internet Mathematics*, 8(4):456–481, 2012.

[5] U. Brandes and D. Fleischer. Centrality measures based on current flow. In *Annual symposium on theoretical aspects of computer science*, pages 533–544. Springer, 2005.

[6] D. J. Daley and D. G. Kendall. Stochastic Rumours. *IMA Journal of Applied Mathematics*, 1(1):42–55, 03 1965.

[7] P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[8] L. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40:35–41, 03 1977.

[9] L. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215 – 239, 1978.

[10] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[11] P. Holme and B. J. Kim. Growing scale-free networks with tunable clustering. *Physical Review E*, 65(2):026107, 2002.

[12] J. JaJa. A perspective on quicksort. *Computing in Science & Engineering*, 2(1):43–49, 2000.

[13] S. Janson, T. Luczak, and A. Rucinski. *Random Graphs*, chapter 5, pages 103–138. John Wiley & Sons, Ltd, 2011.

[14] R. M. Karp, M. Luby, and N. Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429 – 448, 1989.

[15] B. Klimt and Y. Yang. Introducing the enron corpus. In *Conference on Email and Anti-Spam*, 2004.

[16] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2–es, 03 2007.

[17] L. Lovász. Random walks on graphs: A survey. *Bolyai Society Mathematical Studies*, 2, 01 1993.

[18] A. Lulli, L. Ricci, E. Carlini, and P. Dazzi. Distributed current flow betweenness centrality. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 71–80. IEEE, 2015.

[19] R. Matei, A. Iamnitchi, and P. Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, 2002.

[20] M. Newman. A measure of betweenness centrality based on random walks. *Social Networks*, 27:39–54, 2005.

[21] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[22] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical linear algebra with applications*, 1(4):387–402, 1994.

[23] M. D. Shirley and S. P. Rushton. The impacts of network topology on disease spread. *Ecological Complexity*, 2(3):287 – 299, 2005.

[24] J. Travers and S. Milgram. An experimental study of the small world problem. In S. Leinhardt, editor, *Social Networks*, pages 179 – 197. Academic Press, 1977.

[25] V. V. Vazirani. *Appromixation algorithms.* Springer, 2001.

[26] S. Wasserman and K. Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge University Press, 1994.

[27] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440, 1998.

Appendix A

Source Code Listings

A.1 Random walk betweenness function

The contents of `random_walk_betweenness/calculate.py`:

```
from random_walk_betweenness.RandomWalkBetweennessSolver \
    import RandomWalkBetweennessSolver
from random_walk_betweenness.NewmanSolver import NewmanSolver
from random_walk_betweenness.BrandesSolver import BrandesSolver
from random_walk_betweenness.ApproxSolver import ApproxSolver
from random_walk_betweenness.NXApproxSolver import NXApproxSolver


random_walk_betweenness_strategies = {
    "nx": RandomWalkBetweennessSolver,
    "newman": NewmanSolver,
    "brandes": BrandesSolver,
    "approx": ApproxSolver,
    "nxapprox": NXApproxSolver,
}


# Given a NetworkX Graph, returns a dictionary mapping
# [node] -> [random walk betweenness]
def random_walk_betweenness(g, strategy="nx", epsilon=None):
    if strategy not in random_walk_betweenness_strategies:
        raise ValueError(
            "You must pick one of the following strategies: "
            + str(list(random_walk_betweenness_strategies.keys()))
        )

    if epsilon is None:
        solver = random_walk_betweenness_strategies[strategy]()
    else:
        solver = random_walk_betweenness_strategies[strategy](epsilon)

    return solver.calculate(g)
```

A.2 The Newman algorithm

The contents of `random_walk_betweenness/NewmanSolver.py`

```python
import numpy as np
from random_walk_betweenness.helper_functions import \
                                          construct_newman_T_matrix
from random_walk_betweenness.RandomWalkBetweennessSolver \
    import RandomWalkBetweennessSolver
from itertools import combinations


class NewmanSolver(RandomWalkBetweennessSolver):
    # Implements algorithm described in 'A measure of betweenness
    # centrality based on random walks', M.E.J. Newman (2004)
    def calculate_on_connected_graph(self, g):
        n = g.number_of_nodes()

        T = construct_newman_T_matrix(g)
        b = np.zeros(n)  # Initialise array of betweennesses
        s, t = np.fromiter(
            combinations(np.arange(n), 2), dtype='i,i'
        ).view(np.int).reshape(-1, 2).transpose()  # Find all pairs s<t

        for v, w in g.edges:
            temp = np.abs(T[v, s] - T[v, t] - T[w, s] + T[w, t])

            # Exclude values where (v =/= s,t)
            b[v] += np.sum(temp[(s != v) & (t != v)])
            b[w] += np.sum(temp[(s != w) & (t != w)])

        b /= ((n-1)*(n-2))  # normalise

        # Return the result as a dictionary mapping
        # [node]->[random walk betweenness]
        return dict(zip(range(n), b))
```

A.3 The Brandes-Fleischer algorithm

The contents of `random_walk_betweenness/BrandesSolver.py`

```python
import numpy as np
from scipy.stats import rankdata
from random_walk_betweenness.helper_functions import \
                                          construct_newman_T_matrix
from random_walk_betweenness.RandomWalkBetweennessSolver \
    import RandomWalkBetweennessSolver


# Constructs the (n by m) matrix B where:
# B(v,e) =  1, if e = (v,w)
#          -1, if e = (w,v)
#           0, otherwise
def construct_B(g, n):
    # Helper function defines a single row of the matrix
    def row(i, j):
        a = np.zeros(n)
        a[i] = 1
        a[j] = -1
        return a
    return np.vstack((row(i, j) for (i, j) in g.edges))


class BrandesSolver(RandomWalkBetweennessSolver):
    # Algorithm as described in 'Centrality Measures Based on
    # Current Flow', Ulrik Brandes and Daniel Fleischer (2005)
    def calculate_on_connected_graph(self, g):
        n = g.number_of_nodes()
        v = np.arange(n)

        B = construct_B(g, n)
        C = construct_newman_T_matrix(g)
        BC = np.matmul(B, C)

        b = np.zeros(n)  # Initialise array of betweennesses
        for edge_number, e in enumerate(g.edges):
            source, target = e
            row = BC[edge_number, :]
            pos = rankdata(-row, method="ordinal")

            b[source] += np.sum((v + 1 - pos).dot(row))
            b[target] += np.sum((n - v - pos).dot(row))

        b = (b - v) * (2 / ((n-1) * (n-2)))

        # Return the result as a dictionary mapping
        # [node]->[random walk betweenness]
        return dict(zip(range(n), b))
```

Project Proposal
Methods for Calculating Random Walk Betweenness

May 8, 2020

Introduction

Measures of centrality are indicators of which nodes in a graph are the most important. Among them are PageRank [1] and betweenness centrality [2], which have both been used to analyse networks in a wide range of domains from social networks to biological networks. This project will consider another measure of centrality: 'random walk betweenness', as proposed by Newman [3].

The concept of random walk betweenness is motivated by the observation that the formulation of betweenness centrality ignores contributions from paths that are not shortest paths. There are many examples, such as the flow of rumours in a social networks, or the spread of infectious diseases, in which the assumption that information only moves along shortest paths seems unlikely to hold. In these cases an approach that assumes that information traverses the graph randomly may be a better fit. This is where random walk centrality comes in.

The betweenness centrality of a node, i on a graph $G(V, E)$ is given by the following expression:

$$b_i = \frac{1}{n(n-1)} \sum_{(s,t)\in V^2} \frac{\text{number of shortest paths from } s \text{ to } t \text{ that pass through node } i}{\text{total number of shortest paths from } s \text{ to } t}$$

The random walk centrality of a node, i on a graph $G(V, E)$ is given by the following expression:

$$r_i = \frac{1}{n(n-1)} \sum_{(s,t)\in V^2} \mathbb{E}[\text{net number of times random walk from } s \text{ visits } i \text{ before visiting } t]$$

where *net* means that when a walk passes through a node and then later passes through it again in the opposite direction, these visits cancel out. Furthermore, if a random walk is equally likely to go either way through a node, these cancel out as well.

Newman gives an algorithm to calculate betweenness centrality using matrix methods [3]. I will implement this algorithm and evaluate its performance based on run time when run on different graphs. These graphs will be divided into two categories: real-world graphs, and synthetic ones. I will conduct research in order to select suitable real-world graphs. Ideally, they should be from areas where the aforementioned assumption made by betweenness centrality that information travels only along shortest paths is thought to be incorrect. As for the synthetic graphs, I will explore various common types of random graph such as Erdős-Rényi graphs and Watts-Strogatz graphs.

Betweenness centrality can be calculated in $\mathcal{O}(nm)$ time for weighted graphs and $\mathcal{O}(nm+n^2\log(n))$ time for unweighted graphs using Brande's Algorithm [2]. Work has also been done to improve execution time by approximating betweenness centrality rather than directly calculating it [4], resulting in a speedup where precision is not necessary (which it often isn't – knowing the top n most central nodes may be more interesting than their exact ordering). Newman's algorithm takes $\mathcal{O}((m+n)n^2)$ time, largely due to the cost of inverting $n \times n$ matrices. As an extension of my project, I will explore whether good approximations to random walk betweenness can be found quickly.

Project Structure

Research

Before starting the rest of the project, some research and planning will be required:

- I will research into different representations of graphs, with the aim of deciding on a file format to store graphs in. This format should either be directly usable by the algorithm, or require minimal conversion.

- I will research python libraries for dealing with graphs and matrices. Adjacency matrices are used in Newman's algorithm, which use $\mathcal{O}(n^2)$ space. I will explore more memory-efficient representations of these matrices, which may be already readily available in existing libraries. Newman's algorithm uses matrix multiplication and inversion, so finding libraries that can do these quickly will also be important.

- I will research python libraries for visualising graphs. I will also research methods of laying out graphs – I'm aware of algorithms such as Force Atlas 2 being used for this [5]. I will aim to lean heavily on existing frameworks, minimising time I spend working on visualisation.

- I will review the literature around measures of centrality and will decide on which types of random graph will be suitable for evaluating my implementation. Erdős-Rényi graphs and Watts-Strogatz graphs are likely candidates.

Network Generation

- **Real-world graphs** – In selecting real-world graphs for evaluation, I will aim to select graphs from areas where the assumption made by 'shortest-path betweenness' that information travels only along shortest paths is thought to be incorrect. Furthermore, I will aim to have some of these graphs be large enough to come close to the limits of what is tractable. If these graphs are found in a file format different to the one that I will be using, I may have to write a script to perform a conversion from one format to another.

  When Newman wrote his paper about random walk betweenness in 2004 [3], the limit of tractability was graphs of approximately 10^4 nodes, with memory being the main limiting factor. Nowadays, this limit will have increased slightly due to machines having on average more memory, however graphs of 10^5 or more nodes will not be feasible without improvements to the algorithm. This is because inverting a $10^5 \times 10^5$ matrix still being a highly challenging problem even on sophisticated hardware and software [6].

- **Synthetic graphs** – I will write scripts to generate synthetic graphs. These scripts will allow me to create graphs of different sizes and with different properties so that I can evaluate my implementation on a variety of graphs.

Core Algorithm Implementation

I will implement the algorithm described in [3]. I will test this implementation to ensure that it is giving the expected results.

Extension Algorithm Implementation

I will explore different potential avenues for creating an algorithm that approximates random walk betweenness, but takes less time to run than computing it exactly:

- A naïve approach is to simply simulate a number of random walks. It is likely that the convergence properties of this algorithm will be undesirable, but at worst it will act as a baseline for other approaches.

- Another approach is to consolidate communities of tightly connected nodes into 'super nodes', then run the algorithm on the remaining graph of super nodes. This will reduce the running time of the algorithm because it will be running on a smaller number of nodes.

Visualisation

I will write scripts to create visualisations of graphs based on how nodes are connected, and on how central they are. This will be used to visually display the results of Newman's algorithm. This will be useful because it's hard for humans to understand a graph from a non-visual representation.

Evaluation

I will write scripts to run my implementation on the real-world and synthetic graphs. I will investigate how the run time scales with the size of the networks and compare this to the theoretical result given in [3]. In order to measure running times I will perform multiple trials, and will report the results with error bars.

The extension algorithms, as approximations, should naturally be evaluated based on how accurate the approximation is and how much faster they run. There is also potential to evaluate the memory usage of these algorithms, which may be able to improve upon the $\mathcal{O}(n^2)$ space that Newman's algorithm uses. Furthermore, any improvements that I am able to make to the memory usage of Newman's algorithm can be evaluated by measuring memory usage.

Success Criteria

The following should be achieved:

- Implementation of core algorithm exists and has sufficient tests to ensure proper functioning.
- Real-world graphs have been collected.
- Synthetic graphs have been generated.
- Core algorithm has been evaluated based on its running time when run on the real-world and synthetic graphs.
- Visualisations of the graphs are produced.

Resources Required

For this project, I will do most of the work on my laptop, which is an MSI GP62 7RD Leopard (i7 core, NVIDIA GeForce GTX 1050 2GB graphics card, 8GB RAM, 1TB HDD, 128GB SSD) running Windows 10. Should my primary machine fail, I can fall back on using the MCS machines.

Backups and version control will be handled by using git to maintain a private repository of my code. Any documents that I produce in LaTeX (including this one and my final dissertation) will be written in Overleaf, which will mean there will always be a copy of my work available online. I will also regularly save snapshots to disk.

My code will be written in python, using existing libraries for matrices, graphs, and visualisation.

Starting Point

Knowledge from university courses:

- IA Probability (maths department) gave a strong foundation in statistics. It contains a small amount of material about simple random walks in 1 dimension and about markov chains.
- IB Foundations of Data Science reinforced my knowledge of probability.
- IB Machine Learning and Real-world Data has a small amount of material about random graphs and introduced me to the concept of betweenness centrality.
- II Information Theory will teach me more about probability.

Before researching for this project, I knew very little about random walks. How I have a bit more of a baseline level of knowledge from having read [7].

Timetable and Milestones

- **Michaelmas Weeks 3 – 4 (25th October – 6th November)**

  Deadline: Final project proposal by Friday Week 3 (25th October)

  Research into various python libraries and methods for working with graphs and matrices (as discussed in the research section above).

- **Michaelmas Weeks 5 – 6 (7th November – 20th November)**

  Generate toy example graphs. Set up simple visualisation. Later on, the graphs can be used as test cases to ensure that the core algorithm is working correctly. Having some visualisation available will also make debugging and figuring out whether stuff is working correctly easier.

  These first tasks being fairly easy will provide me with an opportunity to learn how to use the libraries that I haven't used before.

  Milestone: Simple graphs generated and visualisations of these graphs produced.

- **Michaelmas Weeks 7 – 8 (21st November – 4th December)**

  Implement core algorithm.

- **First Half of Christmas Holiday (5th December – 20th December)**

  Create tests for core algorithm and fix any issues with it. Iterate on and improve the algorithm.

  Milestone: Core algorithm complete and functional with sufficient test suite.

- **Second Half of Christmas Holiday (21st December – 8th January)**

  Add more methods for generating synthetic graphs and collect some real-world graphs. Improve visualisation.

  Milestone: Have a more complete set of graphs for evaluation. Visualisations are now good enough that they could become figures in a dissertation.

- **Lent Weeks 0 – 1 (9th January – 22nd January)**

  Write progress report and presentation.

  Slack time. Ensure all objectives so far have been delivered.

- **Lent Weeks 2 – 4 (23rd January – 12th February)**

  Deadline – Progress Report by Friday Week 3 (31st January)

  Deadline – Progress Report Presentation in Week 4 (6th – 11th February)

  Make usable evaluation scripts.

  Milestone: Have scripts that run the algorithm and collect data about its running time.

- **Lent Weeks 5 – 6 (13th February – 26th February)**

  Implement extension algorithms.

- **Lent Weeks 7 – 8 (27th February – 11th March)**

  Improve extension algorithms. Produce first draft of introduction section of dissertation

  Milestone: first draft of introduction section.

- **Easter Holiday (12th March – 1st April)**

  Produce first draft of preparation and implementation sections of dissertation.

  Milestone: first draft of preparation and implementation sections.

- **Easter Holiday (2nd April – 15th April)**

  Produce first draft of evaluation and conclusion sections of dissertation.

  Milestone: First draft of full dissertation

- **Easter Weeks 0 – 1 (16th April – 29th April)**

  Edit dissertation and send off to supervisor and director of studies for feedback.

- **Easter Weeks 2 – 3 (30th April – 8th May)**

  Edit dissertation.

  Deadline - Final dissertation hand-in by Friday Week 3 (8th May)

References

[1] Lawrence Page et al. *The pagerank citation ranking: Bringing order to the web.* Tech. rep. Stanford InfoLab, 1999.

[2] Ulrik Brandes. “A faster algorithm for betweenness centrality”. In: *The Journal of Mathematical Sociology* 25.2 (2001), pp. 163–177. DOI: `10.1080/0022250X.2001.9990249`. eprint: `https://doi.org/10.1080/0022250X.2001.9990249`. URL: `https://doi.org/10.1080/0022250X.2001.9990249`.

[3] M.E.J. Newman. “A measure of betweenness centrality based on random walks”. In: *Social Networks* 27 (2005), pp. 39–54.

[4] Nicolas Kourtellis et al. “Identifying high betweenness centrality nodes in large social networks”. In: *Social Network Analysis and Mining* 3.4 (July 2012), pp. 899–914. ISSN: 1869-5469. DOI: `10.1007/s13278-012-0076-6`. URL: `http://dx.doi.org/10.1007/s13278-012-0076-6`.

[5] Mathieu Jacomy et al. “ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software”. In: *PLOS ONE* 9.6 (June 2014), pp. 1–12. DOI: `10.1371/journal.pone.0098679`. URL: `https://doi.org/10.1371/journal.pone.0098679`.

[6] Yang Liang et al. “Spark-based large-scale matrix inversion for big data processing”. In: *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Apr. 2016, pp. 718–723. DOI: `10.1109/INFCOMW.2016.7562171`.

[7] László Lovász. “Random Walks on Graphs: A Survey”. In: *Bolyai Society Mathematical Studies* 2 (Jan. 1993).