**Albert Kutsyy**

# Evaluating Betweenness Centrality Algorithms on Real World Datasets

Computer Science Tripos – Part II

Trinity College

April 25, 2021

# Proforma

| | |
|---|---|
| Name: | **Albert Kutsyy** |
| College: | **Trinity College** |
| Project Title: | **Evaluating Betweenness Centrality Algorithms on Real V** |
| Examination: | **Computer Science Tripos – Part II, July 2021** |
| Word Count: | **TODO**[1] **(well less than the 12000 limit)** |
| Project Originator: | Albert Kutsyy |
| Supervisor: | Dr Timothy Griffin |

## Original Aims of the Project

TODO To write a demonstration dissertation[2] using LaTeX to save student's time when writing their own dissertations. The dissertation should illustrate how to use the more common LaTeX constructs. It should include pictures and diagrams to show how these can be incorporated into the dissertation. It should contain the entire LaTeX source of the dissertation and the makefile. It should explain how to construct an MSDOS disk of the dissertation in Postscript format that can be used by the book shop for printing, and, finally, it should have the prescribed layout and format of a diploma dissertation.

## Work Completed

TODO All that has been completed appears in this dissertation.

## Special Difficulties

TODO Learning how to incorporate encapulated postscript into a LaTeX document on both Ubuntu Linux and OS X.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

[2] A normal footnote without the complication of being in a table.

# Declaration

I, Albert of Trinity 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 Albert Kutsyy

Date April 25, 2021

# Contents

# List of Figures

# Acknowledgements

# Chapter 1

# Introduction

## 1.1 Motivation

This project aims to evaluate algorithms for analyzing graphs. Graphs (also known as networks) represent entities (nodes) and the connections between them (edges). Their generality lends them to a vast number of applications, and the analysis of large graphs has lead to interesting results in disease modeling, sociology, supply chain management, and biology [22][21][20][3].

One method of analyzing graphs is through the use of graph statistics. In contrast to *metrics*, which are single numbers that describe the entire graph (such as average degree, connectivity, or diameter), *statistics* assign a value to each node. One widely used and important statistic is betweenness centrality, which measures the importance of a node. While there are several other measures of importance, betweenness centrality is by far the most frequently used [14] TODO: mention applications?. Further, algorithms for computing betweenness centrality can be trivially extended to compute several other centrality statistics, including closeness centrality and stress centrality [9].

The betweenness centrality of a node $v$ is defined as the number of shortest paths from any node to any other node which pass through $v$. Formally, we can write:

$$C(v) := \sum_{s \neq v \neq t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)} \tag{1.1}$$

where $\sigma(s,t|v)$ is the number of shortest paths from $s$ to $t$ that pass through $v$, and $\sigma(s,t)$ is the total number of shortest paths from $s$ to $t$.

Betweenness centrality was first defined in 1977 by Linton Freeman [13]. Naïve implementations can compute betweenness centrality for all nodes in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space on a graph of $n$ nodes and $m$ edges by solving the all-pairs shortest paths problem. In 2001, Brandes improved this to $\mathcal{O}(nm)$ time and $\mathcal{O}(n+m)$ space for unweighted graphs, and $\mathcal{O}(nm+n^2\log(n))$ time and $\mathcal{O}(n+m)$ space for weighted graphs [9]. No asymptotic improvement has been found since, and $\mathcal{O}(nm)$ time is still too high to compute betweenness centrality for large graphs, which may have millions of nodes and edges.

Despite the lack of asymptotic improvements, there have been dozens of proposed algorithms to compute betweenness centrality faster or to use statistical methods to rapidly compute approximations to it. However, papers proposing new algorithms have just compared them to the algorithm by Brandes, and direct comparisons between them have been limited (see *Related Work*). In this project, I will examine and evaluate five promising algorithms for computing betweenness centrality, detailed in *Selected Algorithms*.

## 1.2 Related Work

Despite the importance of efficient betweenness centrality algorithms, relatively little work has been done to compare the various existing algorithms.

Of the examined algorithms, no author evaluates their algorithm in a comparable way. Bader et al. does not compare the performance of their algorithm to any other [5]. Borassi and Natale [7] compare the performance of their algorithm to the `RK`, `ABRA-Aut`, and `ABRA-1.2` algorithms. Geisberger et al. [14] and Erdős et al. [11] compare their algorithms with `Brandes`. Brandes [9], in turn, compares his algorithm to the now-obsolete `Floyd-Warshall` algorithm.

There are two major studies which attempt to compare the performance of multiple betweenness centrality algorithms. Al-Ghamdi et al. [1] create a system for benchmarking betweenness centrality algorithms on a supercomputer. Their paper focuses primarily on the creation of the benchmark system and the process of benchmarking, and the resulting comparison is treated as a corollary. They compare the performance of the algorithms by Bader et al. and Geisberger et al., as well as others algorithms not discussed in this project.

The other study, by Matta et al. [19] compares a smaller number of algorithms by using them to solve two "real world" problems – clustering a graph and iteratively removing the most important nodes. They run the tests on more typical hardware than Al-Ghamdi et al., but the implementations they test have been parallelized to different degrees. Matta et al. thus conclude that the algorithms which have parallelized implementations are the fastest. They evaluated the algorithms by Borassi and Natale, Brandes, and Geisberger et al., among others.

## 1.3 Project Aims

In this project, I will create efficient implementations of five algorithms (see Selected Algorithms) and a testing harness to evaluate their performance. I will instrument each of these algorithms to compute performance metrics (time per node, total time, memory usage, the number of graph reads). I will run each of these on large graphs using a high performance computing server and determine the accuracy and efficiency trade-offs each algorithm offers.

## 1.4 Overview of selected algorithms

Table 1.1: Overview of Selected Algorithms
TODO: self calculated in appendix or footnotes

| Algorithm | Brief Overview |
|---|---|
| `Brandes` [9] | Published in 2001, this was the first (and only) algorithm that improves upon the $\mathcal{O}(n^3)$ `Floyd-Warshall` algorithm for computing Betweenness Centrality |
| `BP2007` [10] | This was a follow up to Brandes' 2001 paper, and was motivated by the infeasability of running `Brandes` on very large graphs. It was one of the first approximate algorithms for computing betweenness centrality. |
| `Geisberger Linear`[14] | Geisberger et al. found that the `BP2007` algorithm over-estimated the betweenness of neighbors to sampled nodes. `Geisberger Linear` was developed as a relatively straightforward modification to reduce this effect. |
| `Geisberger Bisection`[14] | This algorithm was originally developed for use on graphs where there are very few shortest paths between any two nodes. Although it has a very high worst case complexity (which occurs when there are many shortest paths, such as in a grid), it was found to perform well on many real world graphs. It more aggressively corrects for the over-estimation issue. |
| `Geisberger Bisection Sampling` [14] | This algorithm addresses the worst-case performance issues with `Geisberger Bisection` by only considering one (randomly chosen) shortest path between any two nodes. It takes an additional parameter for the number of shortest path samples to take. |
| Bader et al. [5] | All approximate algorithms considered so far use a fixed number of samples. Bader et al. created their algorithm to base the number of samples on the desired accuracy (represented by a parameter $\alpha$). |
| `KADABRA`[7] | Borassi and Natale developed their adaptive algorithm to give tight accuracy guarantees, and use a different sampling strategy (directly sampling shortest paths) than most other algorithms. They develop a framework for using different statistical guarantees, although I only consider the version that guarantees absolute error of each centrality estimate. |
| `Brandes Subset` [11] | In some situations, such as TODO: which?, it is more appropriate to consider only shortest paths between some subset $S \subseteq V$ of nodes. In this situation, Erdős et al. describe a simple modification to the `Brandes` algorithm. |
| `Brandes++` [11] | Erdős et al. use a divide-and-conquer paradigm to calculate subset betweenness centrality more efficiently than `Brandes Subset`. |

Table 1.3: Worst Case Complexities
TODO: self calculated in appendix or footnotes

| **Algorithm** | **Type** | **Worst Case Complexity** |
|---|---|---|
| `Brandes` [9] | Exact | $\mathcal{O}(n(m + n\log(n)))$ |
| `BP2007` [10] | Approximate | $\mathcal{O}(s(m + n\log(n)))$ |
| `Geisberger Linear`[14] | Aproximate | $\mathcal{O}(s(m + n\log(n)))$ |
| `Geisberger Bisection`[14] | Approximate | $\mathcal{O}(2^n)$ |
| `Geisberger Bisection Sampling` [14] | Approximate | $\mathcal{O}(s(m + n\log(n) + nh))$ |
| Bader et al. [5] | Approximate Adaptive | $\mathcal{O}(n(m + n\log(n)))$ |
| `KADABRA`[7] | Approximate Adaptive | $\mathcal{O}(\frac{\log(n)-\log(\delta)}{\lambda^2}(m + n\log(n)))$ |
| `Brandes Subset` [11] | Exact Subset | $\mathcal{O}(p(m + n\log(n)))$ |
| `Brandes++` [11] | Exact Subset | $\mathcal{O}(p(m + \sum_{i=1}^{k}(\lvert F_i(\lvert F_i\rvert + \lvert V_i \setminus F_i\rvert))) + \sum_{i=1}^{k}(\lvert F_i\rvert\lvert E_i\rvert + \lvert F_i\rvert\lvert V_i\rvert\log\lvert V_i\rvert))$ |

## 1.5 Reasons for selecting algorithms

Table 1.5: Reasons for Selecting Each Algorithm

| **Algorithm** | **Reason** |
|---|---|
| `Brandes` [9] | This is the most commonly used algorithm for computing betweenness centrality[11], and remains the fastest known algorithm for exactly computing it. |
| `BP2007` [10] | This is the simplest algorithm for approximating betweenness centrality and is the base for the work by Geisberger et al. and Bader et al. |
| Geisberger et al.[14] `Linear`, `Bisection`, `Bisection Sampling`) | `Geisberger Linear` was the algorithm selected by Matta et al.[19] as the best to use if one does not need accuracy guarantees. Tests by Geisberger et al. indicate that their other two algorithms perform even better than `Geisberger Linear` |
| Bader et al. [5] | AlGhamdi et al. find that Bader is their fastest algorithm they test, and it uses the same ideas as `BP2007` with modifications to be adaptive. |
| `KADABRA`[7] | This algorithm hasn't been compared to other algorithms outside of experiments done by its authors, which place it as a substantial improvement to other adaptive algorithms of the same type, including the algorithm by Riondato and Kornaropoulos [23]. This algorithm was one of the best that Matta et al. tested, making any faster algorithm one of the fastest known. |
| `Brandes Subset` [11] | I implement this algorithm primarily to compare to `Brandes++`. |
| `Brandes++` [11] | Erdős et al. claim that this algorithm is up to 100 times faster than `Brandes Subset`, making it appealing to test. |

# Chapter 2

# Preparation

## 2.1 Selected Algorithms

### 2.1.1 Brandes

When betweenness centrality was first described by Freeman in 1977, there were no known approaches to calculate it other than the naïve approach of calculating all shortest paths and doing the summation in equation 1.1. By using the `Floyd-Warshall` algorithm, it is possible to do this in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space for a graph with $n$ nodes.

The first major improvement to this was described by Ulrik Brandes in his 2001 paper "A Faster Algorithm for Betweenness Centrality" [9]. Brandes introduces *pair-dependency*, representing the proportion of shortest paths between $s$ and $t$ that pass throubh $v$, defined as

$$\delta(s,t|v) = \frac{\sigma(s,t|v)}{\sigma(s,t)} \tag{2.1}$$

Further, he defines the *dependency* of $s$ on $v$ as

$$\delta(s|v) = \sum_{t \in V} \delta(s,t|V) \tag{2.2}$$

From equation 1.1, can see that

$$C(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)} = \sum_{s \neq v \neq t \in V} \delta(s,t|V) = \sum_{s \neq v \in V} \delta(s|v) \tag{2.3}$$

The crucial observation in Brandes's paper is that $\delta(s|v)$ follows the following recursive relation:

$$\delta(s|v) = \sum_{w : v \in pred(w)} \frac{\sigma(s,v)}{\sigma(s,w)} \cdot (1 + \delta(s|w)) \tag{2.4}$$

Where $pred(v)$ is the set of immediate predecessors of $v$ on all shortest paths from s to any node $t$ that pass through $v$.

With this, Brandes proved that we can calculate $\delta(s|v)$ in two phases. First, compute the solution to the single-source shortest paths (SSSP) problem for $s$. That is, compute all shortest paths from $s$ to any node $t$, storing $pred(v)$ for all $v \in V$ and a list of nodes in non-ascending order of distance from $s$. This can be done by running a breadth-first search (BFS) (for unweighted graphs) or Djikstra's algorithm (for weighted graphs) starting at $s$. While exploring the graph,

add the immediate predecessor of each explored node $v$ to $pred(v)$ and add the node to a stack. This operation takes $\mathcal{O}(n)$ time for unweighted graphs, $\mathcal{O}(m + n\log(v))$ time for weighted graphs, and takes $\mathcal{O}(n + m)$ space.

Additionally, augment the SSSP to also calculate $\sigma(s, v)$ by adding $\sigma(s, w)$ to $\sigma(s, v)$ when exploring the edge $(w, v)$.

Next, accumulate dependencies by iteratively popping elements $w$ off of the stack and incrementing each $v \in pred(w)$ by $\frac{\sigma(s,v)}{\sigma(s,w)} \cdot (1 + \delta(s|w))$. If all $\delta(s|v)$ are initialized to 0, then each $\delta$ will be the value prescribed by equation 2.4 once the stack is empty. This step takes $\mathcal{O}(m)$ time and $\mathcal{O}(n + m)$ space.

Finally, increment $C(v)$ by $\delta(s|v)$ for all $v \in V$. By iterating over all $s \in V$, this will compute $C(v)$ for all $v \in V$.

Overall, this algorithm takes $\mathcal{O}(nm)$ time for unweighted graphs, $\mathcal{O}(nm + n^2\log(n))$ time for weighted graphs, and $\mathcal{O}(n + m)$ space in either case.

This algorithm, denoted here as `Brandes`, is the de facto standard algorithm uses to compute betweenness centrality. In fact, it has been proven that the complexity of computing the betweenness centrality of a single vertex is at least $\mathcal{O}(n^2)$ if the Strong Exponential Time Hypothesis holds [6]. Therefore if the graph is sparse (that is, $m \sim n$), then the `Brandes` algorithm has optimal asymptotic performance, even for computing the centrality of a single node.

### 2.1.2 Brandes Subset

In "A Divide-and-Conquer Algorithm for Betweenness Centrality" [11] Erdős et al. describe a slightly different metric - target set betweenness centrality. It is defined as follows: For a set $S \subseteq V$ such that $2 \leq |S| \leq |V|$, the target set betweenness centrality is defined as the betweenness centrality considering only shortest paths between nodes in the target set, so

$$C^S(v) = \sum_{s \neq v \neq t \in S} \frac{\sigma(s, t|v)}{\sigma(s, t)}. \tag{2.5}$$

Observe that if $S = V$ then $\forall v \in V.\ C^S(v) = C(v)$.

Target set betweenness centrality can be calculated with simple modifications to the `Brandes` algorithm: SSSP is only run from each $s \in S$ and equation 2.4 is replaced by

$$\delta(s|v) = \sum_{w:v \in pred(w)} \frac{\sigma(s, v)}{\sigma(s, w)} \cdot (I_{w \in S} + \delta(s|w)), \tag{2.6}$$

where $I_{w \in S}$ is an indicator that is 1 if $w \in S$ and 0 otherwise.

### 2.1.3 Brandes++

#### Target Set Betweenness Centrality

Erdős et al. aim to speed up `Brandes Subset` by using techniques similar to those used in network routing - splitting the graph into clusters, running a computations on each cluster, and aggregating the results.

### The Algorithm

Erdős et al. detail a new algorithm for efficiently computing target set betweenness centrality. Their algorithm, which they call `Brandes++` consists of five main steps: clustering the graph, adjusting the clustering, constructing a 'skeleton graph', calculating the betweenness centrality of nodes in the skeleton graph, and finally calculating the betweenness centrality of nodes not in the skeleton graph.

I will describe each step individually.

1. **Clustering**

   First, we split the overall graph $G$ into a set of sub-graphs $\{G_1, G_2 \dots G_k\}$ in a way that minimizes the number of edges between sub-graphs. Erdős et al. evaluate the performance of several graph partitioning algorithms and find that using the `METIS` software package results in the best better performance of `Brandes++`. Erdős et al. encounter the issue that the speed of the highly optimized `METIS` package can't be compared to their Python implementation of `Brandes++`. In order to keep the clustering algorithm's performance comparable to the other algorithms, I re-implement the paper that `METIS` is based off of. "A fast and high quality multilevel scheme for partitioning irregular graphs"[15] describes the algorithm used by `METIS`, a multilevel graph partition algorithm we will refer to as `MLGP`.

   MLGP diagram `MLGP` recursively splits a graph into two partitions by using a three-step process. The algorithm aims to minimize the edge cut, which is the sum of the weights of edges between nodes in different partitions.

   1. First, the graph is 'coarsened' a pre-determined $k$ times by combining nodes. Each node is combined with with the neighbor connected by the heaviest edge. This step reduces the size of the graph to approximately $\frac{|V|}{2^k}$, significantly speeding up the next step. Note that `MLGP` interprets weights as closeness while `Brandes++` interprets them as distance, so all weights must be inverted before running `MLGP`.
   2. Next, the graph is actually partitioned. Of the various partitioning algorithms tested by Karypis and Kumar, `GGGP` consistently achieves the best performance and results [15].

      `GGGP` begins by adding a random vertex to a set $T$. Then for each neighbor, `GGGP` calculates the change in the edge cut if that node were to be added to $T$. This is called the gain. GGGP then iteratively adds the neighbor with the lowest gain to $T$, and updates the gain of that node's neighbors. This iteration continues until half of the graph has been added. Then, we define the partitions as the two sets $T$ and $V \setminus T$.

      Since the starting node greatly affects the quality of the partition, `MLGP` runs `GGGP` multiple times and selects the partitioning with the lowest edge cut.
   3. Finally, the graph is uncoarsened and refined. For each of the $k$ times the graph was coarsened we do the following:
      i. Create a new graph where nodes that were merged in the $j^{\text{th}}$ iteration are unmerged
      ii. Refine the graph: `MLGP` uses a modification of the KernighanLin algorithm, where we calculate the gains (as described above) of all nodes with neighbors in the

opposite partition. Then, two nodes in different partitions which would reduce the edge cut the most when swapped are swapped. This process is repeated until no progress has been made in a constant number of swaps or if there are no more nodes to be swapped. Then the iteration with the best edge-cut (computed adding the gains of executed swaps) is selected and output as the refined partition.
This is made efficient by using a specialized data structure to store the gains and only selecting from all combinations of the best 3 nodes from each partition.

**2. Adjustment of Partitions**

Step 4 requires that each target node $s \in S$ is a frontier node (has a neighbor in another partition). To satisfy this, we iterate over all $s \in S$ and if $s$ is not a frontier node, we remove it from its partition and create a new partition containing only $s$. If $S = V$ then every node will be a frontier node and step 3 will take the same time as `Brandes`, but if $S$ is small and the clustering is good, not every node will be a frontier node.

**3. Skeleton Graph**

Once the graph has been partitioned, we construct a simplified representation of the graph, called SKELETON, with the following properties:

If $u$ and $v$ are frontier nodes in different partitions and the edge $(u, v) \in G$ with weight $w$, then $(u, v) \in$ SKELETON with weight $w$.

If $u$ and $v$ are frontier nodes in the same partition $P$, and there exists a shortest path from $u$ to $v$ of length $l$ only going through non-frontier nodes (to avoid double-counting), then $(u, v) \in$ SKELETON with weight $l$. We can calculate this by doing an SSSP from each frontier node $u$, skipping any $v \notin P$ and not adding the neighbors of any other frontier node $v \in P$.

We also associate a value $\theta$ with each edge, defined as the number of shortest paths that the edge represents. If $u$ and $v$ are in different partitions, then $\omega(u, v) = 1$. If $u$ and $v$ are in the same partition, we calculate $\theta$ when we run the SSSP to determine $l$.

**4. BRANDES_SK**

The target-set `Brandes` (described in Target Set Betweenness Centrality) is run on SKELETON, with a few modifications to account for the fact that paths between frontier nodes in the same partition can represent multiple paths (as given by $\theta$). The modifications are as follows.

Rather than incrementing $\sigma(s, v)$ by $\sigma(s, w)$ when exploring the edge $(w, v)$, we instead increment it by $\sigma(s, w) \cdot \theta(w, v)$

We must also account for the multiple paths when accumulating delta, so use the following instead of equation 2.4:

$$\delta(s|v) = \sum_{w} \sigma(v, w) \frac{\sigma(s, v)}{\sigma(s, w)} \cdot (I_{w \in S} + \delta(s|w)) \tag{2.7}$$

Note that if $|\text{SKELETON}| = |V|$ then this takes exactly as much time as `Brandes`, illustrating that we must have a target set $S$ such that $|S| < |V|$ for the algorithm to be effective, and we must have a good clustering that minimizes the number of frontier nodes.

### 5. CENTRALITY

Thus far, we have only computed the centrality of nodes in the skeleton graph (the frontier nodes). In order to compute the centralities of the remaining nodes, Erdős et al. describe one final algorithm, which they call the `centrality` algorithm.

This algorithm essentially calculates equation 2.7 for each $s \in S$, $w \in F_i$, and $v \in V_i/F_i$,

TODO: frontier notation

We already know $\delta(s, w)$ and $\sigma(s, w)$ for each $w \in F_i$, and it is trivial to determine the value of $I_{w \in S}$.

If the graph is undirected, then $\sigma(v, w) = \sigma(w, v)$. When constructing the skeleton graph, we run a modified version of `djikstra` from each $w \in F_i$, which can be simply modified to also compute $\sigma(w, v)$. If the graph is directed, then we can run the same modified algorithm to compute $\sigma(v, w)$.

Next, we need to compute $\sigma(s, v)$. The paper doesn't actually describe how to compute this, but the code released by the authors here implements this by doing the following:

TODO: pseudocode

1. For each $s \in S$:
2. For each partition $i$:
3. num_paths = 0, distance=$\infty$
4. For each $v \in V_i \ F_i$:
5. For each $f \in F_i$:
6. If $d(s \to f \to v) <$ distance:
7. distance = $d(s \to f \to v)$, num_paths = $\sigma(s, f) * \sigma(f, v)$
8. Else if $d(s \to f \to v) =$ distance:
9. num_paths $+ = \sigma(s, f) * \sigma(f, v)$

Finally, we need to apply equation 2.7 for each $w \in F_i$, $v \in V_i/F_i$ such that $d(s \to v \to f) = d(s \to f)$ (as calculated in `Brandes_SK`).

The paper by Erdős et al. claims this can be done in time

$$\mathcal{O}(\sum_{i=1}^{k} |F_i||V_i \ F_i|) \tag{2.8}$$

However, this process needs to be iterated over all $s \in S$, so this algorithm has the following running time (as confirmed by the authors after an email exchange).

$$\mathcal{O}(\sum_{i=1}^{k} |S||F_i||V_i \ F_i|) \tag{2.9}$$

### Brandes and Pich(2007)

While I didn't originally propose to implement the algorithm detailed by Brandes and Pich in 2007 [10], the algorithm by Geisberger et al. is a direct extension of it, making this algorithm useful both for comparing to other approximate algorithms and for understanding them.

Their algorithm (here called `BP2007`) is a simple extension of the `Brandes` algorithm. Rather than iterate over every source $s \in V$, we compute $n \leq |V|$ samples by randomly selecting source nodes and doing the same computation as in `Brandes`, accumulating the centrality. At the end, we multiply each centrality by $\frac{|V|}{n}$ to extrapolate from these n samples.

While Brandes and Pich test several different methods for randomly selecting sources, they find that simply selecting a random source with uniform probability $\frac{1}{|V|}$ results in the highest accuracy estimator.

## 2.1.4 Geisberger et al.

Geisberger et al. detail three different algorithms that use the same framework. First, they define a *scaling function* $f\colon [0,1] \to [0,1]$. This scaling function is what varies between their three algorithms.

Similar to `BP2007`, all of their algorithms do a number of iterations defined by an input parameter. At each iteration, they pick a random node with uniform probability. Then, they select whether to do a forward or backward search with equal probability. Forward searches are done by running `SSSP` on the graph from the selected node, and backward searches are done by running it on the *transpose* graph. The transpose graph is defined as $G = (V^T, E^T)$ where $V^T \coloneqq V$ and $E^T \coloneqq \{(u,v) \mid (v,u) \in E\}$. That is, all nodes are present and all edges are reversed.

Then, for each path $P = \langle \overbrace{s \dots t}^{Q} \dots t \rangle$ that the `SSSP` finds, they define a *scaled contribution*

$$\delta_P(v) \coloneqq \begin{cases} \frac{f(l(Q)/l(P)}{\sigma(t,s)} & \text{if forward search} \\ \frac{1-f(l(Q)/l(P)}{\sigma(s,t)} & \text{if backward search} \end{cases} \tag{2.10}$$

where $l(Q)$ is the total length of the path $Q$ (whether weighted or unweighted).

Then

$$\delta(v) \coloneqq \sum \sum \{\delta_P(v)\colon P \in SP_{st}(v)\}, \tag{2.11}$$

where the outer sum iterates over $t$ for a forward search and over $s$ for a backward search. $2n\delta(v)$ is an unbiased estimator for the betweenness centrality, so can be treated the same way as $\delta(v|s)$ is in `BP2007`. Each of the three algorithms simply varies the scaling function, and describes how to compute it.

#### Linear Scaling

The first algorithm described by Geisberger et al. uses $f(x) = x$ as the scaling function, which is why it is called the Linear Scaling algorithm.

It is a simple modification to `BP2007`, and we can simply use the following instead of equation 2.4:

$$\delta(s|v) = \sum_{w:v \in pred(w)} \frac{\mu(s,v)}{\mu(s,w)} \cdot \frac{\sigma(s,v)}{\sigma(s,w)} \cdot (1 + \delta(s|w)) \tag{2.12}$$

TODO: dist or $\mu$ TODO: not right? see implementation section

Where $\mu(s, v)$ is the distance from $s$ to $v$.

#### Bisection Scaling

The second algorithm uses

$$f(x) = \begin{cases} 0 & \text{for } x \in [0, 1/2) \\ 1 & \text{for } x \in [1/2, 1] \end{cases} \tag{2.13}$$

Implementing this scaling function is somewhat more complicated - each node $t$ only contributes to the $\delta$ of nodes more than halfway on the path from $s$ to $t$. The way Gesiberger et al. address this is by modifying the SSSP algorithm to store successors rather than predecessors. Then, they do a depth-first traversal of this shortest-paths directed acyclic graph (DAG). When a node $v$is visited, we call `Decrement_Half` (described below). Then, we continue the depth-first traversal and when all children have been explored, we can do $\delta(s|v) = \delta(s|v) + \frac{\sigma(s,v)}{\sigma(s,w)} \cdot (1 + \delta(s|w))$ for all children $w \in succ(v)$.

`Decrement_Half` decrements the node halfway on the path from $s$ to $v$ by $\frac{1}{\sigma(s,v)}$ Since this is called once for each path going to $v$, this entirely eliminates the contribution of node $v$ on the node halfway from $s$ to $v$. To determine which node to decrement, we maintain a list representing the current path from $s$ to $v$.

If the graph is unweighted and the stack has $k$ elements, we decrement the node in position $i$, where

$$i = \begin{cases} \max(0, \lfloor (k-2)/2 \rfloor) & \text{if forward search} \\ \lfloor (k-1)/2 \rfloor & \text{else} \end{cases} \tag{2.14}$$

If the graph is weighted, we can store the distances calculated when solving the SSSP and do a binary search for the last node with distance less than $d(v)/2$ (if a forward search) or the first node with a distance greater than $d(v)/2$ (if a backward search).

This algorithm visits each node $v$ $\sigma(s, v)$ times rather than once as the Linear Scaling algorithm does, but this performs well if most shortest paths are unique (there is only one way shortest way to reach a node), such as in road networks.

### 2.1.5 Bisection Sampling

To address the runtime problems of the Bisection Scaling algorithm, Geisberger et al. introduce the Bisection Sampling method. Here, they do the following procedure $k$ times for some constant $k$.

1. For each node $v$ in the shortest-paths DAG, randomly pick a parent $w$ with probability $\frac{\sigma(s,w)}{\sigma(s,v)}$.
2. For this new tree, run the accumulation step of Bisection Scaling
3. For each $v \in V \setminus \{s\}$, increment $C(v)$ by $\delta(s, v)/k$

This reduces the DAG to a tree, ensuring we visit each node only once. Since this is sensitive to the choice of parents, we run this sampling multiple times and average the results.

### 2.1.6 Bader et al.

This algorithm is a very simple modification of the approximate algorithm proposed by Brandes and Pich in 2007. Rather than stopping after a fixed number of iterations, this algorithm stops when the computed betweenness centrality of a chosen node reaches a threshold. The aim of the algorithm is to stop sampling once a particular node has a reasonably good estimate - thus guaranteeing that that node will be accurate.

In order to do this, the algorithm takes two parameters, a node $v$ and a value $c$ (Bader et al. use a value of 5 in their experiments). Then `BP2007` is run until the accummulated centrality for $v$ is greater than $c \cdot |G|$ where $|G|$ is the size of the graph. Bader et al. also use a maximum cutoff (as in `BP2007`) of $|G|/20$, as described in their "Methodology" section.

### 2.1.7 KADABRA

`KADABRA` is a substantially more advanced adaptive algorithm, and uses a different technique to the algorithms seen so far. The core principle is that it repeatedly selects two nodes at random, then uses a balanced bidirectional version of `BFS` or `Djiksra's` to find (uniformly at random) one of the shortest paths between them. The centrality of every node on the path (except for the start and end) is then incremented by one, and this process continues until a particular end condition is met.

Finally, the centralities are divided by the number of iterations, thus representing the probability of passing through a particular node when taking a shortest path between any two nodes. In order to keep consistent with the other algorithms, we multiply by $(|G| - 1)(|G| - 2)$ in order to de-normalize the centralities.

#### Balanced Search

In order to find a random shortest path between two nodes, `KADABRA` uses a balanced bidirectional breadth first search (`BB-BFS`). The paper only considers unweighted graphs, but suggests the algorithm could be adapted to weighted graphs by using `Djikstra's` instead of `BFS`.

The bidirectional search performs both a `BFS` from the start node $s$ and a backwards `BFS` from the end node $t$. In order to do the backwards search (where edges are followed backwards), we can simply do a forward search on the transpose graph (as discussed in Geisberger et al).

Once the two searches intersect, finding a midpoint of total distance $n$, the searches are continued until all midpoints of distance $n$ are found. A midpoint is then chosen at random, weighted proportionally to the number of shortest paths that pass through it. Finally, a path is chosen by backtracking to $s$ and $t$, where at each step a predecessor on the path back to $s$ is chosen at random, with the same weighting as the midpoint, and likewise for the path back to $t$. For this purpose, the `BFS` must track the predecessors of each node, as well as the number of paths that pass through it, just as we did in the `BFS` component of `Brandes`.

#### End Condition

Borassi and Natale spend the majority of their paper determining an end condition which guarantees that all centralities will have a maximum error of $\lambda$, with probability $1 - \frac{\delta}{2}$. They also define an end condition for determining the $k$ nodes with highest centralities, but I do not consider that.

First, Borassi and Natale define a maximum termination condition

$$\omega = \frac{c}{\lambda^2}(\lfloor \log_2(\text{VD-2}) \rfloor + 1 + \log(\frac{2}{\delta})) \tag{2.15}$$

where $c$ is described in [23] and estimated to be 0.5 by Löffler and Phillips in [18]. VD is an upper bound on the vertex diameter, and can be calculated as such: for each strongly connected component, do a search (`BFS` or `Djikstra's`) on both the graph and its transpose, adding together the distance of the furthest (distinct) nodes. The maximum value across all strongly connected components is likely to be an upper bound on the vertex diameter.

$\omega$ is the first of two stopping conditions, the second requires an array $\delta(v)$ computed by the function `computeDelta`

TODO: formatting

**computeDelta** First, perform a number (Borassi and Natale suggest $\frac{\omega}{100}$) of samples to compute preliminary betweennesses $\tilde{\boldsymbol{b}}(v)$ (which must then be divided by $\frac{\omega}{100}$)).

Then, perform a binary search t find C such that

$$\sum_{v \in V} 2 \cdot \exp(-\frac{C\lambda^2}{2\tilde{\boldsymbol{b}}(v)\omega}) = \frac{\delta}{2} - \epsilon\delta \tag{2.16}$$

for some small $\epsilon$.

This binary search can be performed with a min of 0 and a max of $\frac{1}{\lambda^2}\log(4 \cdot |G| \cdot (1-\epsilon)/\delta)$

Finally, for all $v \in V$ assign $\delta(v) = \exp(-\frac{C\lambda^2}{2\tilde{\boldsymbol{b}}(v)\omega}) + \frac{\epsilon\delta}{2 \cdot |G|}$

**haveToStop** On each iteration of the loop, we only continue if `haveToStop` returns false. `haveToStop` returns true only when two conditions are met:TODO: space after dots, general neatness

1. $\exists v \in V.\boldsymbol{b}(v) > 0$
2. $\forall v \in V.\frac{1}{\tau}\log\frac{1}{\delta}(\frac{1}{3} - \frac{\omega}{\tau} + \sqrt{(\frac{1}{3} - \frac{\omega}{\tau})^2 + \frac{2 \cdot \boldsymbol{b}(v)\omega}{\log\frac{1}{\delta}}}) < \lambda$

where $\boldsymbol{b}(v)$ is the current estimate of the betweenness centrality of $v$, and $\tau$ is the number of iterations done so far.

## 2.2 Requirements Analysis

TODO: Future or past tense?

### 2.2.1 Stakeholders

The primary stakeholder of the Part II project is myself - I need the code to meet the goals outlined in my proposal and here, and to deliver results that can be used in my dissertation. Additionally, the assessors are stakeholders as they may want to review the code. TODO: supervisor as stakeholder? Finally, as my project evaluates research, both the wider research community and developers who may want to implement betweenness centrality algorithms are also stakeholders.

### 2.2.2 Core Requirements

My code must meet the following core requirements:

1. To be able to run an arbitrary selected algorithm on an arbitrary graph
2. To handle both weighted and unweighted graphs
3. To parse graphs from common file types (`.edges`, `.mtx`, `.csv`, or `.txt` containing an adjacency list)
4. To output calculated betweenness centralities for each node
5. To compute accuracy metrics from the calculated centralities
6. To output useful performance statistics TODO: list them here?
7. To optionally accept or generate a subset to use with the `Brandes Subset` or `Brandes++` algorithms
8. It must be possible to verify to high certainty the correctness of the code. TODO: remove?
9. For the source code to be accessible by assessors, researchers, and the general public. TODO: supervisor?

I do *not* consider pseudographs - a graph where a node $u$ can have multiple edges to another node $v$, nor graphs with cycles that have a non-positive cost.

### 2.2.3 Distribution

As I originally planned to run my code on a high performance server, the code needs to be portable to various Linux systems, and must be runnable without root access. Additionally, as the implementations may be of interest to the wider community, it should straightforward for someone with technical aptitude to run the code on their own system, potentially after changing the source code.

### 2.2.4 Effectiveness Requirements

Since the project evaluates the speed of betweenness centrality algorithms, it is important for each implementation to be nearly as efficient as a known fast implementation. Due to differences between languages, compiler configurations and other factors, we simply define this to be the same order of magnitude.

What is more important is that the implementations are comparable to each other - if there is a direct comparison between two algorithms, my algorithms should have similar results. TODO: is this important/useful?

However, as parallel versions of most of the selected algorithms aren't available, and converting algorithms to parallel algorithms is beyond the scope of this dissertation, I will only consider performance when restricted to a single thread.

### 2.2.5 Utilization Environments

My project may be run in the following ways

- To evaluate the accuracy and performance of a single algorithm
- To evaluate the accuracy and performance of a series of algorithms
- To calculate the betweenness centrality for a particular graph (Which may be weighted or unweighted)
- To use my implementation of an algorithm as part of a larger system

### 2.2.6 Necessary Components

TODO: Chart? In order to meet the core requirements and utilization environments, the system should have the following components:

- A method for parsing graphs from files
- An internal graph representation
- Efficient implementations of all chosen algorithms
- A handler to feed one or more graphs to one or more algorithms
- A set of tests to validate correctness
- A method for returning calculated centrality and performance metrics
- A method for deriving accuracy statistics from the calculated centrality

## 2.3 Engineering Practices

### 2.3.1 Waterfall

At the time of writing the project proposal, I chose to use the Waterfall model of software development. My project is a large piece of software development with a well defined start and end points, verifiable requirements, and little to no maintenance after delivery. These make it a very good fit for the Waterfall model. Additionally, I can estimate the time required to develop each component (primarily, each algorithm), meaning the timeline I set out in my project proposal was accurate and useful, a key requirement for the waterfall model.

I considered using the Agile model since the testing during development is an attractive element, but my project has little scope for changing requirements. Additionally, I am the primary stakeholder, so a distinct feedback cycle is less important. Further, my project is easily broken down into separate components (algorithms) and would be difficult to break into iterative development cycle.

However, I did decide to adapt the testing regime from the Agile model and tested each component after implementing it, rather than solely testing at the end TODO: do I?.

### 2.3.2 Verification

For the purposes of this dissertation, I will define *verification* as the use of tests to check that my implementations are correct and efficient. This is distinct from *experiments*, in which I use my implementations to compare the speed and accuracy of different algorithms.

I will verify my code as I write it, one algorithm at a time. I will first verify the correctness of my implementations by manually stepping through each algorithm implementation on a small graph. I will also develop a testing package, containing both automated and semi-automated (which output an easy-to-verify description of the state) tests to verify the correctness of my implementations. Further, this testing package will include tests of performance.

Since many of the algorithms are approximate, I will verify that their accuracy is similar to that which is described by the authors.

I will write tests to verify the following components:

1. Graph representations
2. Complex data structures (i.e. priority heaps)
3. Experimentation harness TODO: keep?
4. Each algorithm
5. Experiment data analysis

### 2.3.3 Performance

My project concerns determining the relative performance of complex graph algorithms. To ensure that my results are accurate, I need every implementation to be an efficient implementation of the algorithm. Since there are large performance differences between different languages and tools, I will define an "efficient" implementation as one that is less than 10x slower than a known efficient implementation.

Since this is a relatively loose bound, I will also compare the relative performance of algorithms, where the expected relative performance is known.

#### Techniques to Ensure Performance

In order to ensure that performance is comparable between algorithms, I will use the following techniques:

- I will re-use functions wherever possible, and make minimal modifications where exact re-use isn't possible. (For example, I can re-use my implementation of Djikstra's algorithm many times, though some algorithms may require a slightly modified version).
- I will re-use data structures wherever I have a choice of which to use (i.e. priority heaps).
- I will test multiple data structures where I have a choice and select the one that results in the best overall performance.
- I will use similar constructions between different algorithms - re-using functions where possible and using the same data structures and paradigms across different functions.
- I will use the efficient graph representation for all algorithms

#### Data Validation

My project has very few requirements for the data I use. It must represent a valid graph with a maximum of one edge between any two nodes, and if it is weighted then all weights must be positive. In order for my code to parse the graph, all nodes must have integers associated with them. Graphs may be disconnected, but connectivity isn't a requirement for any of the chosen algorithms. Further, disconnected graphs appear in real-life scenarios so testing performance on them is useful.

In order to ensure these properties, the graph parser detects and warns about graphs with more than one edge between nodes or non-positive weights, I have written a python script to convert any node labels into integer values.

## 2.4 Tools

### 2.4.1 Languages

TODO: highlight choice I will highlight potential choices for language below.

Table 2.1: Language options

| **Language** | **Pros** | **Cons** |
|---|---|---|
| C | • Very fast<br>• Somewhat Portable | • Difficult to debug<br>• Not object-oriented<br>• Comparatively low-level |
| C++ | • Very fast<br>• Somewhat Portable<br>• Object-oriented<br>• Extensive library support | • Difficult to debug |
| Java **(chosen)** | • Fairly fast<br>• Highly portable (no re-compiling necessary)<br>• Object-oriented<br>• High-level libraries<br>• I have a lot of experience using it<br>• Easy debugging<br>• Some library support | • Slightly slower than C and C++<br>• Performance issues with auto-boxing |
| Python | • Object-oriented<br>• High level<br>• Flexible<br>• Fairly portable<br>• Extensive library support | • Vast performance disparity between native and library operations (can make ensuring comparability between algorithms difficult)<br>• Large projects result in many dependencies to install<br>• Making readable/reusable code requires special effort (typing hints etc.) |

I chose Java in the end because it is highly portable, (a distribution requirement), it's easy to write readable code in (a core requirement), and it is straightforward to write efficient code in (a core requirement). Additionally, because of my experience using Java, its high level nature, and the ease of debugging, using Java will speed up the development time and allow me to

ensure I can finish my project on time, potentially with extensions.

### 2.4.2 Datasets

I have chosen to use the following graphs for testing algorithms.

Table 2.3: Datasets

| Graph | Domain | Properties | Description |
|---|---|---|---|
| `4932.protein.links` [8] | Biology | 6,574 Nodes<br>1,845,966 Edges<br>Undirected<br>Weighted | Protein relations of Saccharomyces Cerevisiae (Brewer's Yeast). Nodes are proteins and edges are association (based on direct and indirect interaction) between nodes. The original dataset uses high scores to represent closely associated proteins, so I have inverted all weights such that short path mean close relations. |
| `wiki-vote` [17] | Social | 7,115 Nodes<br>103,689 Edges<br>Directed<br>Unweighted | Votes on requests by Wikipedia users to become administrators as of January 3 2008. Users vote on these requests. Nodes are users, and edges are votes. |
| `as-caida20071105` [24] | Technology | 26,475 Nodes<br>106,762 Edges<br>Undirected<br>Unweighted | Network of autonomous systems (AS), a component of the internet, collected by the CAIDA project in 2007. Nodes are AS's and edges are communication. |
| `slashdot0811` [17] | Social | 77,360 Nodes<br>905,468 Edges<br>Directed<br>Unweighted | Slashdot is a website containing user-submitted news. This is a network of users that have tagged others as friends or foes. Nodes are users and edges are tags (there is no distinction between friend and foe tags) |
| `com-amazon` [17] | Business | 334,863 Nodes<br>925,872 Edges<br>Undirected<br>Unweighted | This is a network of amazon products. If two products are frequently purchased together, there is an edge between them. Nodes are products and edges are frequent co-purchasing. |

### 2.4.3 Version Control

## 2.5 Starting Point

The literature available for each algorithm is dramatically different, so I will discuss them individually, as well as the starting point for the overall testing framework.

### 2.5.1 Brandes

`Brandes` is a very common algorithm, and there are dozens of implementations easily available. In fact, I have previously implemented it as part of the first year course 'Machine Learning and Real World Data'. That implementation was very slow, taking 43 seconds to run on a graph of 4000 nodes, but I did start with an intuition for how to think about `Brandes`.

### 2.5.2 Brandes++ and Brandes Subset

Dora Erdős has published the implementations of `Brandes_Subset` and `Brandes++` used in [11], which can be found here. The implementation is written using native python loops, and while it is an invaluable resource document, would be extremely difficult to transcribe into efficient Java code.

The published implementation of `Brandes++` does *not* include an implementation of the `METIS` algorithm, as Erdos et al. used a standalone software package. The sourcecode for the standalone `METIS` package is published here, and is high-performance parallelized C++code.

### 2.5.3 Brandes and Pich 2007

Implementations of `BP2007` are widely available, such as the version published by AlGhamdi et al. [1] here as "RAND1". This implementation is written in high-performance C++code.

### 2.5.4 Geisberger et al.

AlGhamdi et al. [1] tested one of the algorithms described by Geisberger et al., referring to it as RAND2. They don't specify which one they use, but their published source code (found here), calls `NetworKit::ApproxBetweenness2` when RAND2 is specified as the algorithm [2]. NetworKit is open source and can be found here TODO: netorkit.

The paper by AlGhamdi et al. was published June 2017, as was the github repository containing their code. The latest version of NetworKit at the time was version 4.2, published December 13, 2016 [4]. This release can be found in commit 8a2d77448e18a29a487eb9a2e642467733635f24. Here, `networkit/cpp/centrality/ApproxBetweenness2.cpp` runs the Linear Scaling algorithm described by Geisberger et al. It is written in C++.

I have not been able to find any implementation of the other algorithms described in [14], but Robert Geisberger's original student thesis (which can be found here) gives detailed information that is sometimes missing in the published paper. For example, where the published paper says to decrement a node, the original thesis specifies to decrement by $\frac{1}{\sigma(w)}$.

### 2.5.5 Bader et al.

AlGhamdi et al. [1] have published a version of the algorithm by Bader et al. as "GSIZE" here. It is high-performance C++code.

### 2.5.6 KADABRA

Borassi and Natale have published an implementation of their `KADABRA` algorithm here, which was written in C++.

### 2.5.7 A Benchmark for Betweenness Centrality Approximation Algorithms on Large Graphs

AlGhamdi et al. [1] compared the performance of `Brandes`, `Geisberger Linear`, `BP2007`, and `Bader`. They use the NetworKit version of `Geisberger Linear`, but their own version of the others. They only measured total runtime and several error statistics, and only did a single run of each algorithm, with a single set of parameters.

Their code, which has been optimized for use on a supercomputer, was published here.

### 2.5.8 Comparing the speed and accuracy of approaches to betweenness centrality approximation

Matta et al. compare `BP2007`, `Geisberger Linear`, `Bader`, and `KADABRA`. In all cases, they use published implementations of these algorithms, and have not published their own code.

TODO: How to discuss best practices?

# Chapter 3

# Implementation

## 3.1 Completed Code

My implementation consists of 5 main components - a framework that includes graph representations and necessary data structures, implementations of the 7 algorithms, harnesses evaluating the algorithms, tools to extract useful statistics from the evaluation outputs, and a testing suite with unit and integration tests.

### 3.1.1 Framework

#### Graph Representation

I evaluated four different graph representations and implemented three of them.

- **Adjacency Matrix:** One possibility is to store the graph as an $n$ by $n$ matrix where an entry $x_ij$ is nonzero if there is an edge from $i$ to $j$. This possibility was quickly discarded because graphs are often sparse - `com-amazon` has just under 1 million edges, but the matrix representation would need to have 100 billion entries.
- **Edge List:** Another possibility was to store a list of all edges in the graph. However, this leads to extremely poor performance for finding the neighbors of a node and was only implemented as an exercise in thinking about graphs.
- **Adjacency List:** This option stores a list of nodes, and associates with each node a list of edges from that node. This allows for relatively breadth first search and I actually implemented two different versions of it, one using complex Java objects and another simply using lists of integers.
- **Array Representation:** The final option is to use the representation described by Zardosht Kasheff in [16]. This consists of two arrays. The first (edges[]) is a list of the endpoints ($j$ in the edge $i \to j$) of all edges, arranged in order by the start node of the edge. The second (nodeList[]) is used to index into edges[], such that the edges from node $n$ lie in edges[] between nodeList[$2n$] and nodeList[$2n + 1$]. This does require the nodes to be numbered $0, 1, 2 \ldots n$, but this can be achieved simply by re-labeling nodes. TODO: figure in notes

I tested the two adjacency list implementations as well as my array representation implementation by running the `Brandes` algorithm on `wiki-vote` with each of the three. I found the array

representation to offer much higher performance, which is likely due to the simplicity and cache performance of simply iterating over an array. TODO: elaborate?

If a graph is undirected, this is represented simply by having both the edges $i \to j$ and $j \to i$ in the graph.

#### Heaps

I tested thee different priority queue data structures, the Binary Heap, the Fibonacci Heap, and the Rank Pair Heap [12]. I implemented the binary and rank pair heap, and adapted an implementation of the Fibonacci heap by Keith Schwarz [25]. I also tested the Java `PriorityQueue` and JGraphT `JRankPairHeap` classes by running `Brandes` with each. I also created a testing harness (`HeapTester`) to exercise common operations on these heaps, the results of which are below. I used my binary heap implementation due to it being by far the fastest in both tests. It is important to note that Java's `PriorityQueue` is only extremely slow at doing decreaseKey() operations.

Table 3.1: Time to Complete Heap Tests for 100,000 elements

| **Implementation** | **Time** |
|---|---|
| Binary Heap | 0.164s |
| Fibonacci Heap | 0.374s |
| Rank Pair Heap | 0.341s |
| JGraphT `RankPairingHeap` | 0.224s |
| Java `PriorityQueue` | 9.086s |

Table 3.2: Time to Complete Heap Tests for 10,000,000 elements

| **Implementation** | **Time** |
|---|---|
| Binary Heap | 25.71s |
| Fibonacci Heap | 50.24s |
| Rank Pair Heap | 57.44s |
| JGraphT `RankPairingHeap` | 56.91s |
| Java `PriorityQueue` | $>$ 3000s |

### 3.1.2 Algorithms

Although each algorithm was implemented following the paper describing them, each required a series of design decisions that I will now discuss. All algorithms use the Array Representation of a graph and return a Statistics object, which contains information about the execution. However, for reasons discussed later, this does not actually contain any information other than the centralities of each node.

All of the implemented algorithms work for both directed and undirected graphs, since undirected graphs are represented simply as directed ones with extra nodes. Further, they also work for both weighted and unweighted graphs, choosing at runtime whether to use `Djikstra's` or `BFS` to traverse the graph.

As per the definition of the Array Representation of the graph, I assume that all nodes in the graph are integers $1 \dots (|G| - 1)$. As such, I use arrays to represent mapping of nodes to values, such as the `centralities[]` array, which assigns a centrality to each node.

I also make extensive use of the `Trove` package TODO: cite, which contains implementations of many standard data structures. I use these because the Java standard libraries operate only on Objects, and wrap primitives inside an object. If objects are frequently being added and removed from these structures (such as a queue for `BFS` or `Djikstra`), this creates pressure on the garbage collector, drastically slowing down computations. In contrast, Trove implementations directly use primitives without wrapping them in an Object.

### Brandes

The `Brandes` algorithm is very well defined, leaving few design choices. Following a series of tests (See Heaps), I use my implementation of a binary heap as the priority queue used by `Djikstra's`.

The subset version of Brandes as defined in [11] takes a set of target nodes as a parameter, and simply implements the two minor changes described in Brandes Subset. As with all algorithms, it ensures that all nodes have a default centrality of 0, even if they are never visited.

### Brandes++

In contrast to `Brandes`, `Brandes++` leaves a huge amount up to the designer. At any place where there are multiple options for sub-algorithms, I use the one the authors test to have the consistently best performance, resulting in the series of algorithms described in Brandes++.

I vary the number of partitions in my tests TODO: write which values?, but leave the other parameters for `METIS` unchanged - a coarsening level proportional to $\log(|G|)$ to ensure that the size of the partitioned graph grows logarithmically with the size of $G$. I use 5 iterations of `GGGP` and a partition stop condition of 100, as recommended by TODO: author in TODO: cite. Since `METIS` interprets large weights as closely related nodes, and `Djikstra` uses weights as distance, I invert all weights before feeding the graph to `METIS`.

The algorithm presented by Erdős et al. only works for undirected graphs, so for directed graphs I also have to run `Djikstra` from all non-frontier nodes to all frontiers in each partition.

This was by far the most complicated algorithm to implement, requiring over 1,000 lines of code. TODO: use cloc for accuracy

TODO: discuss some of the performance issues here or in experimental results?

### Brandes and Pich 2007

This algorithm is a near-identical copy of `Brandes` with very minor modifications. While Brandes and Pich test many different sampling strategies, all result in high runtime or error for some subclass of graphs TODO: graphs. They conclude that uniform random sampling is the most robust sampling strategy, and is nearly the most accurate for all cases they test. My implementation follows this recommendation.

In my tests, I test with $25, 50, 100, 200, 400, 800, 1600$, and 3200 sampled nodes to determine how accuracy varies with the number of samples

**Geisberger et al.**

**Geisberger Linear** I use the modification discussed in the text of section **3.2** of [14], which is to do

$$\delta(s|v) = \sum_{w:v\in pred(w)} \frac{\sigma(s,v)}{\sigma(s,w)} \cdot (\frac{1}{\mu(s,w)} + \delta(s|w)), \tag{3.1}$$

rather than equation 2.4, and

$$C(v) = \sum_{s\neq v\in V} \delta(s|v) \cdot \mu(s,v) \tag{3.2}$$

rather than equation 2.3. Here, $\mu(s,v)$ is the shortest path distance (as calculated in `SSSP`) from $s$ to $v$.

In order to do backwards searches (as with all Geisberger versions), I construct the transpose graph as detailed in 2.1.7 and run `SSSP` on that.

**Geisberger Bisection** My implementation uses an iterative depth-first traversal of the shortest-path tree. As stated in Geisberger's thesis, when we visit a node $w$ on the way back down the traversal, we set $\delta(s|v) \leftarrow \delta(s|v) - \frac{1}{\sigma(s,w)}$. If the graph is unweighted, $v$ is simply the node halfway through the DFS stack. If the graph is weighted, I perform a binary search to find the node just under halfway from $s$ to $w$.

There are minor differences in rounding for iterations of the backwards search.

**Geisberger Bisection Sampling** This is a straightforward modification of `Geisberger Bisection` with no design choice.

**Bader et al.**

This is another simple modification to `Brandes`, and in my tests I try different methods of choosing the input node, and different values of $\alpha$. I use the same cutoff of $\frac{|G|}{20}$ that Bader et al. describe in their methodology section, as otherwise there is no guarantee of termination. I note in the Statistics object whether the cutoff was used for the termination condition.

### 3.1.3 Verification

TODO: Name consistency of this I designed my project for evaluation from the beginning - I started by creating the individual components (graphs, heaps, file parsing) required for my implementation. I was able to test all of these individually before beginning to implement any algorithms. Most of these unit tests can be found in the `testing` package, though some were conducted using the testing harnesses found in `framework/mains` and have since been overwritten.

I tested in three main ways. Where ground truth implementations (such as the Java priority queue or `Brandes` algorithm) existed, I created tests to run both on the same dataset and verify that the results are identical. Examples can be found in `framework/mains/Harness.java` and `testing/HeapTester.java`. In fact, one of the first things I did was create the `verification` package, which allows me to verify my implementation of `Brandes` against the `JGraphT` implementation.

I also tested the speed of different implementations to check that I am using the fastest implementation I can. This can also be found in the aforementioned files, and I tested implementations of Brandes with different graph representations in order to determine which is the fastest (see the `SetGraph` git branch).

Where no ground truth implementations exist, I used a small dataset (the `toy` or `toy2` graphs) and worked out a solution by hand, then ran my code with debug information printing. I verified my file parsing, `BrandesSubSet`, and `METIS` implementations this way, among other components. While some of these debug statements have been removed, many use my `print_debug` method and can be activated by running with a high (20+) debug level. TODO: do I mention that I removed some?

Additionally, one of my core requirements was to compare the speed of my graph algorithm implementations against fast published ones, and I have made sure to do so for all algorithms that I could find.

Finally, all of my graph algorithms have been developed for testability, with components such as `Djikstra's` separated out and easily testable independent of any other graph algorithm. All return a `Statistics` object, which can be augmented to include extra information about the run (such as which end condition `Bader` terminates with). This is important both for testing, and for actually running my experiments.

TODO: "Any design strategies that looked ahead to the testing stage should be described in order to demonstrate a professional approach was taken" - do I successfully do this?

### 3.1.4 Instrumentation

I instrumented my ArrayGraph representation to record the number of times an edge is followed. I make sure to always traverse edges with the following paradigm:

Because of this consistency, it is straightforward to augment `start(i)` and `end(i)` to track the number of times they've been called. Because some algorithms may use multiple graphs (`SKELETON` in `Brandes++` or the transpose graph in `KADABRA`), all graphs update a single global variable.

TODO: Should I have mentioned the metrics in Preparation?

Although I implemented instrumentation to record the time per node for `Brandes`, I made it an optional feature and didn't implement it for any other algorithm for reasons discussed in Extensions and Shortcomings.

### 3.1.5 Experimentation

I created three harnesses in order to run experiments on my graph algorithms. All three have necessary information about the run (which graphs and parameters to use, and for how many iterations) stored as variables. I chose to do this instead of taking them as parameters so that I wouldn't risk having typos when running the experiments and having different parameters than intended.

All three harnesses follow the same basic structure: TODO: pseudocode

Here, `warmup` just runs the `Brandes` algorithm on `wiki-vote` ten times, and serves to raise the CPU temperature to ensure the first few experiments don't have an advantage due to the colder system temperatures.

I re-construct the graph on each run in order to guarantee that a bug which alters the graph in one algorithm does not affect any others. Additionally, the `getGraph` subroutine has a `System.gc()` and a `Thread.sleep(2000)` command, which signal the system to run garbage collection and then pause for two seconds to allow the system to settle a little.

I save the full list of centralities for later processing, as well as statistics about the execution - total time and number of graph accesses.

### 3.1.6 Extensions and Shortcomings

The following changed from my project proposal to the final implementation

#### Algorithms

While I had originally only proposed implementing five algorithms, I extended my project to include `BrandesSubset`, `BP2007`, and two additional algorithms by Geisberger et al.

`BrandesSubset` was a necessary component for testing the performance and accuracy of `Brandes++`, so implementing it was mandatory.

`BP2007` is a simple algorithm that Bader et al. and Geisberger et al. base their algorithms off of, so implementing it allowed me to compare those to a 'baseline' approximate algorithm.

I originally only proposed to implement the algorithm by Geisberger et al. which Matta et al. tested, which was `Geisberger Linear`. However, after seeing the impressive results Geisberger et al. claim for `Geisberger Bisection` and `Geisberger Bisection Sampling`, I decided to implement all three algorithms.

### 3.1.7 METIS

While I had originally only proposed to study algorithms for betweenness centrality, I quickly realized that a major shortcoming in the paper by Erdős et al. was that they didn't implement a graph clustering algorithm themselves. Instead, they performed the graph clustering with a high performance package written in C++, and the rest of their implementations with native Python. This is a major weakness in their claims, since it is impossible to determine whether the speedup is due to the algorithm itself or them using a much faster implementation for part of it.

To avoid the same shortcoming, I implemented the paper `METIS` is based on.

### 3.1.8 Metrics

I originally proposed to collect 4 metrics - total time, number of graph reads, time per node, and memory usage.

I successfully implemented the first two, but collecting memory usage information was substantially more complicated. Java has a method that returns the current memory usage, but unless I called that near-continuously, I could not find the peak memory usage with it. I could use an external memory usage monitor, but I would need to record and extract the peak memory usage over a specific interval of time for 1000+ intervals. Perhaps more important, memory usage isn't useful for analyzing these algorithms. With the exception of `Brandes++`, the main place where the algorithms use large volumes of memory is storing the lists of predecessors when doing the `SSSP` problem. Thus, monitoring peak memory usage would just tell us the maximum size DAG that `SSSP` computes, added to a significant amount of noise from the Java garbage

collector. Since all of the approximate algorithms sample a small subset of nodes, the depth of the largest DAG can vary significantly due to change, adding even more noise to memory usage metrics. Finally, `SSSP` is a well-studied problem TODO: find citations and there are much better ways to profile it.

While I had originally proposed to measure the time per node, I quickly realized that it is a poorly defined metric. While `Brandes` iterates over all nodes and does a computation from each $v$, that computation (the `SSSP` and accumulation) doesn't actually affect the betweenness of $v$. Additionally, the approximate algorithms sample only a subset of nodes, making time per node only applicable to the nodes that are chosen. One way to define it is average time per node, which is just $time/|G|$. TODO: For later: do I end up using this in a graph?

### 3.1.9 Parallelizability

Parallelizing the betweenness centrality algorithms was a proposed extension that I did not do. However, I wanted to briefly make some notes about it. Matta et al. find that the best algorithms they test are those with parallel implementations. However, (TODO: from my informal analysis, should I still have this?) it is actually possible to effectively parallelize all of the algorithms I have discussed.

Apart from `Brandes++`, all algorithms spend the vast majority of their runtime in a loop that is essentially TODO: pseudocode, make sure to specify uniformally at random

For `Brandes`, `BP2007`, `Geisberger Linear`, `Geisberger Bisection`, and `Geisberger Bisection Sampling`, the end condition is simply the total number of iterations conducted. This means that there are no dependencies between iterations of the loop, so parallelization is straightforward. After all $k$ threads have completed, betweennesses can be aggregated with $\log_2(k)$ inter-thread messages. TODO: correct? keep?

`Bader` and `KADABRA` have the additional complication of having an end condition that depends on the aggregated betweenness. However, if one can efficiently check this end condition, then the vast majority of the work done by the algorithm can be parallelized.

Erdős et al. describe how to parallelize `Brandes++` - much of the work done in building the skeleton graph, and all of the work of the `Centralities` algorithm depends only on each cluster independent of any other. This means that those components could be parallelized up to the number of clusters. `Brandes_SK` can be parallelized in a similar manner to `Brandes`.

## 3.2 Repository Overview

Will do when all experiments etc. are done

## 3.3 Experimental Results

I would like to talk about how to present/discuss my results.

# Chapter 4

# Evaluation

TODO: What exactly should I discuss here - my main ways to do evaluation are the test scripts, though those are fairly incomplete, the experiments, and running my code and other people's on the same dataset

# Chapter 5

# Conclusion

TODO: What should go here?

# Bibliography


[1] Ziyad AlGhamdi, Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. A Benchmark for Betweenness Centrality Approximation Algorithms on Large Graphs. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, New York, NY, USA, 2017. Association for Computing Machinery. event-place: Chicago, IL, USA.

[2] Ziyad AlGhamdi, Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. A Benchmark for Betweenness Centrality Approximation Algorithms on Large Graphs, 2017.

[3] Gil Amitai, Arye Shemesh, Einat Sitbon, Maxim Shklar, Dvir Netanely, Ilya Venger, and Shmuel Pietrokovski. Network Analysis of Protein Structures Identifies Functional Residues. *Journal of Molecular Biology*, 344(4):1135 – 1146, 2004.

[4] Eugenio Angriman, Alexander van der Grinten, and Henning Meyerhenke. NetworKit, April 2021. Publication Title: NetworKit.

[5] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating Betweenness Centrality. In Anthony Bonato and Fan R. K. Chung, editors, *Algorithms and Models for the Web-Graph*, pages 124–137, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[6] Michele Borassi, Pierluigi Crescenzi, and Michel Habib. Into the Square - On the Complexity of Quadratic-Time Solvable Problems. *arXiv:1407.4972 [cs]*, July 2014.

[7] Michele Borassi and Emanuele Natale. KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation. *CoRR*, abs/1604.08553, 2016. _eprint: 1604.08553.

[8] Peer Bork, Lars Juhl Jensen, and Christian von Mering. *Welcome to STRING.*

[9] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001. Publisher: Routledge _eprint: https://doi.org/10.1080/0022250X.2001.9990249.

[10] Ulrik Brandes and Christian Pich. CENTRALITY ESTIMATION IN LARGE NETWORKS. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007. _eprint: https://doi.org/10.1142/S0218127407018403.

[11] Dora Erdos, Vatche Ishakian, Azer Bestavros, and Evimaria Terzi. *A Divide-and-Conquer Algorithm for Betweenness Centrality.* 2015. _eprint: 1406.4173.

[12] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1-4):111–129, November 1986.

[13] Linton C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, 1977. Publisher: [American Sociological Association, Sage Publications, Inc.].

[14] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better Approximation of Betweenness Centrality. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 90–100, USA, 2008. Society for Industrial and Applied Mathematics. event-place: San Francisco, California.

[15] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998. Place: USA Publisher: Society for Industrial and Applied Mathematics.

[16] Zardosht Kasheff. Partial Parallelization of Graph Partitioning Algorithm METIS Term Project, 2004.

[17] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. June 2014.

[18] Maarten Lffler and Jeff M. Phillips. Shape Fitting on Point Sets with Probability Distributions. *Lecture Notes in Computer Science Algorithms - ESA 2009*, pages 313–324, 2009.

[19] John Matta, Gunes Ercal, and Koushik Sinha. Comparing the speed and accuracy of approaches to betweenness centrality approximation. *Computational Social Networks*, 6(1):2, February 2019.

[20] Philip Nuss, T. E. Graedel, Elisa Alonso, and Adam Carroll. Mapping supply chain risk by network analysis of product platforms. *Sustainable Materials and Technologies*, 10:14 – 22, 2016.

[21] Evelien Otte and Ronald Rousseau. Social network analysis: a powerful strategy, also for the information sciences. *Journal of Information Science*, 28(6):441–453, 2002. _eprint: https://doi.org/10.1177/016555150202800601.

[22] Navavat Pipatsart, Charin Modchang, Wannapong Triampo, and Somkid Amornsamankul. Network Based Model of Infectious Disease Transmission in Macroalgae. *International Journal of Simulation: Systems, Science and Technology*, 19:11.1–11.8, 2018.

[23] Matteo Riondato and Evgenios M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery*, 30(2):438–475, 2015.

[24] Ryan A. Rossi and Nesreen K. Ahmed. *Network Data Repository: The First Interactive Network Data Repository*. 2020. Publication Title: Network Repository.

[25] Keith Schwarz. Archive of interesting code - FibonacciHeap.java.

# Appendix A

# Project Proposal