Albert Kutsyy

Evaluating Betweenness Centrality Algorithms on Real World Datasets

Computer Science Tripos – Part II

Trinity College

May 13, 2021

Declaration of Originality

I, Albert Kutsyy 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. I am content for my dissertation to be made available to the students and staff of the University.

Signed Albert Kutsyy

Date May 13, 2021

Proforma

Candidate Number:	**2400G**
Project Title:	**Evaluating Betweenness Centrality Algorithms on Real World Datasets**
Examination:	**Computer Science Tripos – Part II, 2021**
Word Count:	**11,017**[1]
Lines of code:	**7,549**[2]
Project Originator:	Dr. Timothy Griffin
Project Supervisor:	Dr. Timothy Griffin

Original Aims of the Project

The original aim of the project was to research, implement, instrument, and compare five different algorithms for computing betweenness centrality. All implementations were to be programmed with the same techniques and data structures to allow me to fairly compare them, something that has never been done before for betweenness centrality. I would measure time per node, total execution time, memory usage, and the number of edge traversals. All algorithms would be run on large datasets across a variety of parameters, and the data collected and analyzed.

Work Completed

I have created efficient implementations of nine different algorithms for computing betweenness centrality. Of these, six are approximate algorithms and two calculate a related metric, 'target set betweenness centrality'. Two have never had published implementations. This is the first evaluation that uses comparable implementations of each algorithm, making it novel research. I instrumented each implementation to measure execution time, edge traversals, and a variety of accuracy statistics, writing over 7,000 lines of code. I collected data from over 1,500 executions across five graphs and a variety of parameters, and analyzed the results here.

Special Difficulties

In order to get useful data, I had to run many algorithms on large datasets more than 1,500 times. I had to guarantee that hardware performance was consistent and that no other processes used a large portion of the CPU. This meant that I ran over 68 hours of experiments (not including those I discarded) on my laptop, which I could not use for any other purpose during that time. The consistency requirement prevented me from using cloud computing.

[1] Calculated using `texcount`, sections 1-5 only, including headers and captions.
[2] Calculated using `cloc`, excluding blank lines. 862 lines are scripts to generate charts.

Contents

Chapter 1

Introduction

1.1 Motivation

This project evaluates 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, biology, and more [1][2][3][4].

One method of analyzing graphs is through 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 statistic is *betweenness centrality*, which measures the importance of a node. While there are several measures of importance, betweenness centrality is by far the most frequently used [5]. Further, algorithms for computing betweenness centrality can trivially be extended to compute several other centrality statistics, including closeness centrality and stress centrality [6].

The betweenness centrality of a node v is defined as the number of shortest paths (between any two nodes) that 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 [7]. 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 computing the shortest paths between every pair of nodes (the all-pairs shortest path 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 [6]. No asymptotic improvement has been found since, and $\mathcal{O}(nm)$ time is still too large to compute betweenness centrality for large graphs, which have millions of nodes and edges.

To address this, researchers have proposed dozens of algorithms for rapidly approximating betweenness centrality. However, direct comparison between these algorithms has been limited (see Section 1.2). In this project, I examine and evaluate several promising algorithms for computing betweenness centrality, detailed in Section 2.1.

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 authors evaluate their algorithms in comparable ways. Bader et al. do not compare the performance of their algorithm to any other [8]. Borassi and Natale [9] compare the performance of their algorithm to the `RK`, `ABRA-Aut`, and `ABRA-1.2` algorithms. Geisberger et al. [5] and Erdős et al. [10] compare their algorithms to `Brandes`. Brandes [6], in turn, compares his algorithm to the now-obsolete `Floyd-Warshall` algorithm.

Two major studies attempt to compare the performance of multiple betweenness centrality algorithms. AlGhamdi et al. [11] create a system for bench-marking betweenness centrality algorithms on a supercomputer. They compare the performance of `Brandes`, `BP2007`, `Bader`, and `Geisberger Linear`, as well as other algorithms not discussed in this project. They implement `BP2007` and `Bader` themselves, as well as a parallelized version of `Brandes`. They compare these to the NetworKit [12] implementation of `Geisberger Linear`. AlGhamdi et al. only evaluate a single set of parameters for each algorithm and only do a single iteration for each. However, they do run these on very large (18 million node) graphs.

The other study, by Matta et al. [13], compares algorithms by using them to solve two "real world" problems — clustering a graph, and iteratively removing the most important nodes. They run the experiments on more typical hardware than AlGhamdi et al., and test a variety of parameters for each. They compared the AlGhamdi et al. implementation [14] of `BP2007`, the parallelized NetworKit implementation [12] of `Geisberger Linear`, and the parallel implementation of `KADABRA` published by Boarassi and Natale [15], among others not discussed in this paper. Matta et al. conclude that the parallelized algorithms are the fastest. However, not all of the implementations they tested are parallelized.

1.3 Project Aims

In this project, I create efficient implementations of nine algorithms (see Section 2.1) and an experimentation harness to evaluate their performance. I instrument each of these algorithms to record performance metrics (see Section 3.1.4). Additionally, I create a testing suite to validate the performance and correctness of my implementations. I then run each implementation on large graphs and compare the accuracy and speed trade-offs each algorithm offers (see Section 4.3). I implement more algorithms, record fewer performance metrics, and use a different computer than I originally proposed. Reasons for these changes are detailed in Sections 3.3 and 3.4.

1.4 Overview of Selected Algorithms

Table 1.1: Overview of Selected Algorithms

Algorithm	Brief Overview
`Brandes` [6]	Published in 2001, this is the first (and only) algorithm to improve on the $\mathcal{O}(n^3)$ `Floyd-Warshall` algorithm.
`BP2007` [16]	Published by Brandes and Pich in 2007, this is a direct follow up to Brandes' 2001 paper. It is one of the first approximate algorithms for computing betweenness centrality.
`Geisberger Linear` [5]	Geisberger et al. find that the `BP2007` algorithm over-estimated the centrality of some nodes. `Geisberger Linear` is a relatively straightforward modification of `BP2007` that reduces this effect.
`Geisberger Bisection` [5]	This algorithm, published in the same paper as `Geisberger Linear`, more aggressively corrects for the over-estimation issue. It was developed for graphs with few shortest paths and has an exponential worst case time complexity, which occurs on graphs with many shortest paths (such as grids). However, Geisberger et al. find it performs well on real-world graphs.
`Geisberger Bisection Sampling` [5]	This algorithm fixes the worst-case time complexity issues of `Geisberger Bisection` by doing an additional round of random sampling.
`Bader` [8]	All approximate algorithms considered so far use a fixed number of samples. Bader et al. describe an algorithm to adaptively determine the number of samples to take, using an accuracy parameter alpha.
`KADABRA` [9]	Borassi and Natale develop an adaptive algorithm to give tight statistical accuracy guarantees. It uses a different sampling strategy than most other algorithms — directly sampling shortest paths. They develop a framework for guaranteeing different statistical properties, although I only consider the version of their algorithm that guarantees absolute error of each estimate.
`Brandes` [10] `Subset`	In some situations, such as analyzing road networks between major cities, 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++` [10]	Erdős et al. use a divide-and-conquer paradigm to calculate target set betweenness centrality more efficiently than `Brandes Subset`.

1.5 Reasons for Selecting Algorithms

Table 1.3: Reasons for Selecting Each Algorithm

Algorithm	Reason
`Brandes` [6]	This is the most commonly used algorithm for computing betweenness centrality [10].
`BP2007` [16]	This is the simplest algorithm for approximating betweenness centrality and is the basis for the algorithms by Geisberger et al. and Bader et al.
`Geisberger Linear` [5]	Matta et al. [13] conclude that `Geisberger Linear` is one of the best algorithms they test.
`Geisberger Bisection` [5]	Geisberger et al. find that `Geisberger Bisection` performs even better than `Geisberger Linear` [5].
`Geisberger Bisection Sampling` [5]	This algorithm resolves the worst-case runtime of `Geisberger Bisection`.
`Bader` [8]	AlGhamdi et al. find that `Bader` is the fastest algorithm they test.
`KADABRA`[9]	Matta et al. [13] find that `KADABRA` is the fastest non-GPU algorithm that they test, and Borassi and Natale [9] find that `KADABRA` is significantly faster than any similar algorithm.
`Brandes Subset` [10]	This algorithm is the most common method for computing target set betweenness centrality [12], and serves as a baseline to evaluate `Brandes++`.
`Brandes++` [10]	Erdős et al. claim that this algorithm is up to 100 times faster than `Brandes Subset`.

1.6 Complexity of Selected Algorithms

I have included below the worst-case time complexity (on weighted graphs) of each of the selected algorithms. Here, n is the number of nodes in the graph, m is the number of edges, p is the number of samples (a user-defined parameter), and h is the number of accumulation samples used by `Geisberger Bisection Sampling` (another user parameter). I define r to be the value given by

$$\frac{0.5}{\lambda^2}(\lfloor \log_2(\text{VD} - 2) \rfloor + 1 + \log \frac{1}{\delta}), \tag{1.2}$$

where λ and δ are user parameters described in Section 2.1.5. VD is the vertex diameter - the maximum number of nodes in a shortest path in the graph.

Finally, q is the size of the subset used by `Brandes Subset` and `Brandes++`, V_i is the number of nodes in cluster i, and E_i is likewise the number of edges.

Table 1.5: Worst Case Complexities

Algorithm	**Algorithm Type**	**Worst Case Complexity**
`Brandes` [6]	Exact	$\mathcal{O}(n(m + n\log(n)))$
`BP2007` [16]	Approximate	$\mathcal{O}(p(m + n\log(n)))$
`Geisberger Linear` [5]	Aproximate	$\mathcal{O}(p(m + n\log(n)))$
`Geisberger Bisection` [5]	Approximate	$\mathcal{O}(2^n)$
`Geisberger Bisection Sampling` [5]	Approximate	$\mathcal{O}(p(m + n\log(n) + nh))$
`Bader` [8]	Approximate Adaptive	$\mathcal{O}(n(m + n\log(n)))$
`KADABRA` [9]	Approximate Adaptive	$\mathcal{O}(r(m + n\log(n)))$
`Brandes Subset` [10]	Exact Subset	$\mathcal{O}(q(m + n\log(n)))$
`Brandes++` [10]	Exact Subset	$\mathcal{O}\left(q\left(\sum_{i=1}^{k} \lvert V_i\rvert^2\right) + \sum_{i=1}^{k}(\lvert V_i\rvert\lvert E_i\rvert + \lvert V_i\rvert^2 \log(\lvert V_i\rvert))\right)$

Chapter 2

Preparation

2.1 Selected Algorithms

2.1.1 Brandes

When betweenness centrality was first described by Freeman in 1977, the best known approach was to calculate all shortest paths and count how many pass through each node. 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 $G = (V, E)$ with $n = |V|$ nodes.

A major improvement to this was described by Ulrik Brandes in his 2001 paper "A Faster Algorithm for Betweenness Centrality" [6]. Brandes introduces *pair-dependency*, representing the proportion of shortest paths between nodes s and t that pass through v, defined as

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

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

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

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

Thus, we can compute betweenness centrality by

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

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

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

where $\text{pred}(w)$ is the set of immediate predecessors of w on all shortest paths from s to any t.

Thus, we can calculate $\delta(s|v)$ in two phases. First, compute the solution to the single-source shortest paths problem for s. That is, compute all shortest paths from s to any node t, storing $\text{pred}(w)$ along the way. This can be done with breadth-first search (`BFS`) or Djikstra's algorithm (`Djikstra`), and I will refer to the algorithm for solving the single-source shortest paths problem as `SSSP`.

In order to be used in `Brandes`, `SSSP` needs to store a stack of explored nodes in non-ascending order of distance from s. `SSSP` must also calculate $\sigma(s, v)$ by setting $\sigma(s, v) \leftarrow \sigma(s, w) + \sigma(s, v)$ when exploring the edge (w, v).

Next, accumulate dependencies by popping elements w off of the stack and incrementing each $v \in \text{pred}(w)$ by $\frac{\sigma(s,v)}{\sigma(s,w)} \cdot (1 + \delta(s|w))$. If all $\delta(s|v)$ are initialized to 0, then each δ will be the value prescribed by equation 2.4 once the stack is empty. Finally, increment $C(v)$ by $\delta(s|v)$ for all $v \in V \setminus \{s\}$. When this procedure is done for all $s \in V$, this will compute $C(v)$.

This algorithm, denoted here as `Brandes`, takes $\mathcal{O}(nm)$ time for unweighted graphs and $\mathcal{O}(nm + n^2 \log(n))$ time for weighted graphs, on a graph with n nodes and m edges. It uses $\mathcal{O}(n + m)$ space in either case, and is the standard algorithm used to compute betweenness centrality.

Borassi et al. [17] prove 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 [18] holds. Thus, if the graph is sparse (that is, $m \sim n$), then the `Brandes` algorithm likely has optimal asymptotic performance, even for computing the centrality of a single node.

2.1.2 Brandes and Pich (2007)

The algorithm by Brandes and Pich (here called `BP2007`) is a simple extension of the `Brandes` algorithm [16]. Rather than iterating over every source $s \in V$, they compute $p \leq n$ samples by randomly selecting source nodes (pivots) and doing the same computation as in `Brandes`, accumulating the centrality. At the end, they multiply each centrality by $\frac{n}{p}$ to extrapolate from these p samples, where n is the number of nodes in the graph.

Brandes and Pich test several methods for randomly selecting pivots, and find that all almost all fail on some subset of graphs (such as in Figure 2.1). They conclude that the most robust method is selecting nodes at random with uniform probability.

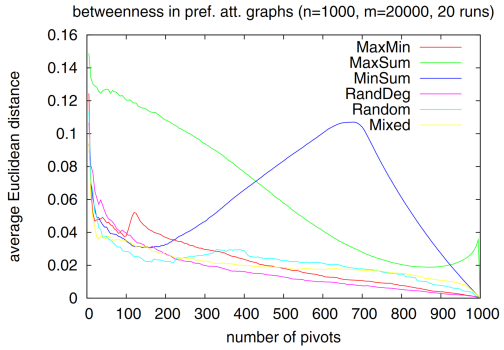

Figure 2.1: A test done by Brandes and Pich, pivots vs Euclidean distance (error), run on a particular type of graph. Note the high error of the MinSum and MaxSum sampling methods.

2.1.3 Geisberger et al. (Linear, Bisection, Bisection Sampling)

Geisberger et al. [5] detail three different algorithms that use the same framework and different *scaling functions* $f\colon [0,1] \to [0,1]$.

Similar to `BP2007`, all three algorithms randomly select p pivots with uniform probability. Then, they randomly select whether to do a forward or backward search. Forward searches are done by running `SSSP` on the graph from the selected pivot, and backward searches are done by running it on the *transpose* graph. The transpose graph is defined as $G^T := (V^T, E^T)$ where $V^T := V$ and $E^T := \{(u,v) \mid (v,u) \in E\}$. That is, all nodes are present and all edges are reversed.

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

$$\delta_P(v) := \begin{cases} \frac{f(l(Q)/l(P))}{\sigma(t,s)} & \text{if forward search} \\ \frac{1-f(l(Q)/l(P))}{\sigma(s,t)} & \text{if backward search,} \end{cases} \tag{2.5}$$

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

They then define $\delta(v)$ to be the sum of $\delta_P(v)$ over all shortest paths and endpoints. That is, letting SP_{st} be the set of all shortest paths from s to t,

$$\delta(v) \coloneqq \sum \sum_{P} \{\delta_P(v) \colon P \in \mathrm{SP}_{st}(v)\}. \tag{2.6}$$

The outer sum iterates over all values of t for a forward search, and over s for a backward search. Geisberger et al. prove that $2n\delta(v)$ is thus an unbiased estimator for the betweenness centrality. Each of the three algorithms simply varies the scaling function, and how to compute it.

Linear Scaling

The first algorithm described by Geisberger et al. uses the linear function $f(x) = x$ as the scaling function.

Utilizing this requires a simple modification to `BP2007`, to use the following instead of equation 2.4:

$$\delta(s|v) = \sum_{w:v\in\mathrm{pred}(w)} \frac{\mu(s,v)}{\mu(s,w)} \cdot \frac{\sigma(s,v)}{\sigma(s,w)} \cdot (1 + \delta(s|w)), \tag{2.7}$$

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

Bisection Scaling

The second algorithm uses the scaling function

$$f(x) \coloneqq \begin{cases} 0 & \text{for } x \in [0, 1/2) \\ 1 & \text{for } x \in [1/2, 1]. \end{cases} \tag{2.8}$$

Using this scaling function is more complicated, and Geisberger et al. describe the following procedure to implement it. First, modify `SSSP` to store successors rather than predecessors. Then, do a depth-first traversal of the resulting directed acyclic graph (DAG). When a node v is visited, call `Decrement_Half` (described below). Continue the depth-first traversal until all of v's children have been explored, then compute:

$$\delta(s|v) = \sum_{w\in\mathrm{succ}(v)} \frac{\sigma(s,v)}{\sigma(s,w)} \cdot (1 + \delta(s|w)). \tag{2.9}$$

`Decrement_Half` decrements the node halfway on the current path from s to v by $\frac{1}{\sigma(s,v)}$. Since this is called once for each path going through v, it eliminates the contribution of v on the node halfway from s to v. If the graph is unweighted and the stack has k elements, it decrements the node in position i, where

$$i \coloneqq \begin{cases} \max(0, \lfloor (k-2)/2 \rfloor) & \text{if forward search} \\ \lfloor (k-1)/2 \rfloor & \text{else.} \end{cases} \tag{2.10}$$

If the graph is weighted, `Decrement_Half` stores the distances calculated when running `SSSP`. In case of a forward search, it does a binary search for the last node with distance less than $\mu(s,v)/2$. For a backwards search, it finds the first node with distance greater than $\mu(s,v)/2$.

The `Bisection Scaling` algorithm visits each node $\sigma(s,v)$ times for each pivot s, which can take exponential time in the worst case. However, Geisberger et al. found it to perform well on real-world graphs.

Bisection Sampling

To address the worst-case runtime issue of `Bisection Scaling`, Geisberger et al. introduce the `Bisection Sampling` method, which take *accumulation samples* rather than traversing the entire DAG. The algorithm performs the following procedure k times for k accumulation samples.

1. For each node v in the shortest-paths DAG, pick one parent w with probability $\frac{\sigma(s,w)}{\sigma(s,v)}$.
2. For the resulting 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 modification uses the scaling in equation 2.10 without an exponential worst-case complexity.

2.1.4 Bader et al.

This algorithm is a very simple modification of the `BP2007` algorithm described in Section 2.1.2 [8]. Rather than stopping after a fixed number of iterations, this algorithm stops when the approximate 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 the chosen node will be accurate.

In order to do this, the algorithm takes two parameters, a node v and a value alpha (Bader et al. use a value of 5 in their experiments). Then `BP2007` is run until the accumulated centrality for v is greater than alpha $\cdot n$ where n is the number of nodes in the graph. Bader et al. also use a maximum cutoff to terminate after $n/20$ iterations.

2.1.5 KADABRA

`KADABRA` is an adaptive algorithm that uses statistical methods to give guarantees about the accuracy of the estimated centralities. The algorithm selects two nodes at random, then uses a balanced bidirectional version of `BFS` or `Djiksra's` to find a random shortest path 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 random shortest path between any two nodes. In order to keep consistent with the other algorithms, I multiply by $(n-1)(n-2)$ 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` 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), it does a forward search on the transpose graph (see Section 2.1.3). The searches meet at multiple midpoints and find all shortest paths between s and t. A path is then selected at random.

End Condition

Borassi and Natale describe an end condition that guarantees all centralities have a maximum error of λ with probability $1 - \frac{\delta}{2}$, for inputs λ and δ. They also define an end condition for determining the k nodes with highest centralities, which I do not evaluate.

First, Borassi and Natale define a maximum termination constant

$$\omega = \frac{c}{\lambda^2}\left(\lfloor \log_2(\mathrm{VD} - 2)\rfloor + 1 + \log\left(\frac{2}{\delta}\right)\right), \tag{2.11}$$

where c is estimated to be 0.5 by Löffler and Phillips in [19]. VD is an upper bound on the vertex diameter and can be simply estimated by the following procedure.

For each strongly connected component in the graph, run `SSSP` from a random node on both the graph and its transpose, adding together the distances of the furthest (distinct) nodes. The maximum value of this across all strongly connected components is likely to be an upper bound on the vertex diameter.

This is a maximum cutoff — the algorithm terminates if the number of iterations ever goes above ω. The adaptive stopping condition also requires an array $\delta(v)$ computed by `computeDelta`.

`computeDelta`

First, perform $\frac{\omega}{100}$ samples to compute preliminary betweenness centrality $\tilde{\boldsymbol{C}}(v)$. Then, for some small ϵ, perform a binary search to find K such that

$$\sum_{v \in V} 2 \cdot \exp\left(-\frac{K\lambda^2}{2\tilde{\boldsymbol{C}}(v)\omega}\right) = \frac{\delta}{2} - \epsilon\delta \tag{2.12}$$

This binary search can be performed with a min of 0 and a max of $\frac{1}{\lambda^2}\log(4 \cdot n \cdot (1-\epsilon)/\delta)$. Finally, for all $v \in V$, assign $\delta(v) = \exp\left(-\frac{K\lambda^2}{2\tilde{\boldsymbol{C}}(v)\omega}\right) + \frac{\epsilon\delta}{2n}$.

`haveToStop`

On each iteration of the loop, only continue if `haveToStop` returns false. `haveToStop` returns true only if both the following conditions are met:

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

where $\boldsymbol{C}(v)$ is the estimated centrality of node v, and τ is the current number of iterations.

2.1.6 Brandes Subset

Erdős et al. describe a new metric — target set betweenness centrality $C^S(v)$. It is defined as the number of shortest paths between any two nodes *in the target set* that pass through v. If the target set S is equal to the set of all nodes 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

$$\delta(s|v) = \sum_{w : v \in \mathrm{pred}(w)} \frac{\sigma(s,v)}{\sigma(s,w)} \cdot (I_{w\in S} + \delta(s|w)), \tag{2.13}$$

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

2.1.7 Brandes++

Erdős et al. [10] detail a new algorithm for efficiently computing target set betweenness centrality. Their algorithm, dubbed `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.

Clustering

First, split (*cluster*) the overall graph G into a set of sub-graphs $\{G_1, G_2, \ldots, G_k\}$ to minimize 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 overall performance of `Brandes++`.

`METIS` is a highly optimized software written in C++, making it incomparable to the rest of our codebase. Erdős et al. encountered this issue and were not able to determine the effect this step has on the overall performance. To counteract this issue, I re-implement the algorithm that `METIS` is based off of, which is described by Karypis and Kumar in [20]. This is a multilevel graph partitioning algorithm that I will refer to as `MLGP`.

`MLGP` recursively splits a graph into two partitions by using a three-step process (see Figure 2.2).

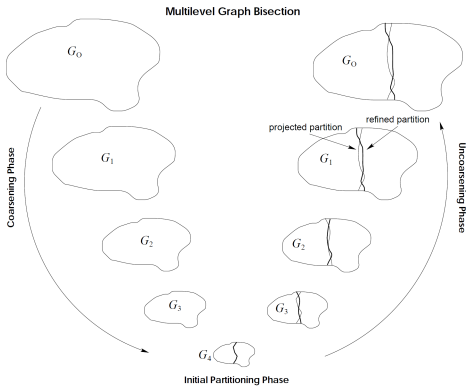

Figure 2.2: Multilevel Graph Partitioning Algorithm (`MLGP`)

1. 'Coarsen' the graph k times by combining nodes. On each iteration, combine every node with the closest neighbor (randomly if there are ties). This step reduces the size of the graph substantially, speeding up the partitioning. Note that `MLGP` interprets weights as closeness while `Brandes++` interprets them as distance, so all weights must be inverted before running `MLGP`.

2. Partition the graph. Of the various partitioning algorithms tested by Karypis and Kumar, the Greedy Graph Growing Partitioning (`GGGP`) algorithm consistently achieves the best performance and results [20].

Begin by adding a random vertex v to a set T. Then, for all neighbors of v, calculate the change in the edge-cut (the sum of the weights of all edges between partitions) if that neighbor were added to T. This is called the gain. Iteratively add the neighbor with the highest gain to T, and update the gain of that node's neighbors. Continue until half of the graph has been added, at which point the graph is partitioned into 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 overall edge-cut.

3. Uncoarsen and refine the partition. For each of the k times the graph was coarsened, do the following:

a. Create a new graph where nodes that were merged in the j^{th} iteration are un-merged.

b. Refine the graph: `MLGP` uses a modification of the Kernighan–Lin algorithm, which follows. Calculate the gains of all nodes with neighbors in the opposite partition. Then, swap the two nodes with highest gains. This process is repeated until progress stops or there are no more nodes to be swapped. The iteration with the best edge-cut is then selected and output as the refined partition.

   This is made efficient by using a specialized data structure (a `Gain Bucket`) to store the gains and only selecting from the best three nodes from each partition.

Partition Adjusting

`Brandes Skeleton` requires that each target node $s \in S$ is a frontier node (has a neighbor in another partition). To satisfy this, remove all nodes $s \in S$ which are not frontiers from their partitions and add them to a new partition that contains only s.

If $S = V$, every node will be a frontier node and `Brandes Skeleton` will take the same time as `Brandes`, making this algorithm unsuitable for calculating standard betweenness centrality.

SKELETON

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

- If u and v are frontier nodes in different partitions and there is an edge $(u, v) \in G$ with weight w, then add (u, v) to 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 add (u, v) to SKELETON with weight l. Calculate this by running `SSSP` from each frontier node u, skipping any $v \notin P$ and not adding the neighbors of any other frontier node $v \in P$.

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

Brandes Skeleton

`Brandes Subset` 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 θ). The modifications are as follows:

Rather than incrementing $\sigma(s, v)$ by $\sigma(s, w)$ when exploring the edge (w, v), increment it by $\sigma(s, w) \cdot \theta(w, v)$. Additionally, to account for the multiple paths when accumulating deltas, use following instead of equation 2.4:

$$\delta(s|v) = \sum_{w} \theta(v, w) \frac{\sigma(s, v)}{\sigma(s, w)} \cdot (I_{w \in S} + \delta(s|w)). \tag{2.14}$$

Note that if the number of nodes in SKELETON is close to $|V|$ due to a large number of frontier nodes, this will take the same time as `Brandes Subset`.

`Centrality`

Thus far, we have only computed the centrality of nodes in SKELETON (the frontier nodes). In order to compute the centralities of the remaining nodes, Erdős et al. describe one final algorithm, the `Centrality` algorithm. Here, I denote the set of frontiers of each cluster i as F_i, making the set of non-frontier nodes $V_i \setminus F_i$.

`Centrality` directly calculates equation 2.14 for each $s \in S$, $w \in F_i$, and $v \in V_i \setminus F_i$. As suggested by Erdős et al., store $\delta(s, w)$ and $\sigma(s, w)$ for each $s \in S$ and $w \in F_i$ when running `Brandes Skeleton`. This imposes a $\mathcal{O}(|S|n)$ space complexity that makes `Brandes++` infeasible for very large graphs. If the graph is undirected, then $\sigma(v, w) = \sigma(w, v)$, which was computed when constructing SKELETON for all (w, v). If the graph is directed, then run the modified `SSSP` used in constructing SKELETON from every $v \in V_i \setminus F_i$.

Next, we need to compute $\sigma(s, v)$. The paper does not actually describe how to compute this, but the code released by the authors in [21] implements this with the following algorithm. As in Section 2.1.3, μ denotes distance.

Algorithm 1: Computing $\sigma(s, v)$ for `Centralities`

Input: Distances μ, Partitions V_i, Frontiers F_i
Result: Centralities of all non-frontier nodes
foreach $s \in S$ **do**
  foreach partition i **do**
    num_paths $\leftarrow 0$
    distance $= \leftarrow \infty$
    foreach $v \in V_i \setminus F_i$ **do**
      foreach $f \in F_i$ **do**
        if $\mu(s, f) + \mu(f, v) <$ *distance* **then**
          distance $\leftarrow \mu(s, f) + \mu(f, v)$
          num_paths $\leftarrow \sigma(s, f) \cdot \sigma(f, v)$
        else if $\mu(s, f) + \mu(f, v) =$ *distance* **then**
          num_paths $\leftarrow$ num_paths $+\sigma(s, f) \cdot \sigma(f, v)$

Finally, apply equation 2.14 for each $s \in S$, $w \in F_i$ and $v \in V_i \setminus F_i$ such that $\mu(s, v) + \mu(v, f) = \mu(s, f)$ (as calculated in `Brandes Skeleton`).

`Centrality` takes $\mathcal{O}\left(\sum_{i=1}^{k} |S||F_i||V_i \setminus F_i|\right)$ time, as confirmed by the by the authors [22].

2.2 Requirements Analysis

2.2.1 Stakeholders

The primary stakeholder of the Part II project is myself — I need the code to meet the goals outlined here, and to deliver useful statistics to analyze. Additionally, the assessors are stakeholders as they may want to review the code. 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. Run any of my selected algorithms on any graph.
2. Handle both weighted and unweighted graphs.
3. Parse common graph file types (`.edges`, `.mtx`, `.csv`, and `.txt` adjacency lists).
4. Output calculated betweenness centralities for each node.
5. Compute accuracy statistics from the calculated centralities.
6. Output useful performance statistics, including total computation time, computation time per node, maximum memory usage, and number of times an edge in the graph is followed.
7. It must be possible to verify to high certainty the correctness of the exact algorithms.
8. The source code must be accessible by assessors, researchers, and the general public.

I do *not* consider pseudographs — a graph where a node u can have multiple edges to another node v — nor graphs with non-positive weights.

2.2.3 Distribution Requirements

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

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 some of the fastest published implementations. Due to differences between languages, compiler configurations and other factors, I simply define this to be the same order of magnitude.

However, parallel versions of most of the selected algorithms are not available, and converting algorithms to parallel algorithms is beyond the scope of this dissertation (see Appendix A). Therefore, 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 (weighted or unweighted);
- To use my implementation of an algorithm as part of a larger system.

2.2.6 Necessary Components

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 pass one or more graphs to one or more algorithms,
- A set of tests to validate correctness,
- A method of outputting the performance metrics and calculated centrality estimates,
- A method of deriving accuracy statistics from the calculated centrality,
- A method of visualizing these performance and accuracy statistics.

2.3 Engineering Practices

2.3.1 Waterfall

When proposing the project, I chose to use the Waterfall model of software development. My project is a large piece of software development with 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 roughly estimate the time required to develop each component (primarily, each algorithm), 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 an iterative development cycle.

However, I adapted the testing regime from the Agile model and tested sub-components after implementing them, rather than solely testing at the end.

2.3.2 Verification

I 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 verify my code as I write it, one algorithm at a time. I first verify the correctness of my implementations by manually stepping through each algorithm implementation on a small graph. I also develop a testing package, containing both automated and semi-automated (which output an easily verifiable description) tests to verify the correctness of my implementations, as well as performance tests. I verify the following components:

1. Graph representations,
2. Complex data structures (i.e. priority heaps),
3. Each algorithm,
4. Experiment data analysis.

2.3.3 Performance

In addition to the effectiveness requirements described in Section 2.2.4, each algorithm must be implemented in a way that makes comparisons between the implementations a useful reflection on the underlying algorithm. In order to ensure this, I:

- re-use functions wherever possible, and make minimal modifications where exact re-use is impossible;
- test multiple data structures where possible and select the one that results in the best overall performance;
- re-use data structure implementations wherever possible;
- use similar constructions and paradigms where possible; and
- use the same graph representation for all algorithms.

Data Validation

My project has very few requirements for the data I use. Datasets 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 be represented as integers.

To validate that these properties are met, I have augmented the graph parser to warn about graphs with multiple edges between nodes or non-positive weights. I also wrote a python script to convert any node labels into integer values.

My experiments do not depend on what the graph represents, so if the graphs have incorrect data, this does not affect my results. For example, if a graph represents social media friendships and omits some connections, this does not affect the claim that one algorithm is faster than another.

2.4 Tools

2.4.1 Languages

Table 2.1: Potential Choices for Programming Language

Language	Pros	Cons
C	• Very fast • Somewhat portable	• Difficult to debug • Not object-oriented • Comparatively low-level
C++	• Very fast • Somewhat portable • Object-oriented • Extensive library support	• Difficult to debug
Java **(chosen)**	• Fairly fast • Highly portable (no re-compiling necessary) • Object-oriented • High-level libraries • Easy debugging • Some library support	• Slightly slower than C and C++ • Performance issues with auto-boxing
Python	• Object-oriented • High-level • Flexible • Fairly portable • Extensive library support	• Vast performance disparity between native and library operations (can make ensuring comparability between algorithms difficult) • Large projects can have a large number of dependencies that must be installed • Large projects can quickly become unreadable

I chose Java because it is portable (distribution requirement), efficient (a core requirement), and high-level, making it easy to write readable code. Additionally, because of my experience with Java, its high-level nature, and the ease of debugging, using Java will speed up development and allow me to ensure I can finish my project on time.

2.4.2 Datasets

I used data from Stanford SNAP [23], NetworkRepository [24] and STRING [25], and from a variety of domains. I present details about each dataset in Section 4.3.3.

2.4.3 Version Control

I use Git for version control. It allows me to test different techniques for optimization on new branches, and allows me to access previous commits to help isolate bugs.

2.5 Starting Point

The literature available for each algorithm is dramatically different, so I discuss them individually. Additionally, I discuss previous attempts to compare betweenness centrality algorithms

2.5.1 Brandes

`Brandes` is a very common algorithm, and there are dozens of available implementations. 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 4,000 nodes, but I did begin with an understanding of `Brandes`.

2.5.2 Brandes and Pich 2007

Implementations of `BP2007` are widely available, such as the version published by AlGhamdi et al. in [14] as "RAND1." This implementation is written in high-performance C++ code [11].

2.5.3 Geisberger et al.

AlGhamdi et al. [11] test the `Geisberger Linear` algorithm under the name "RAND2." They use an implementation published by Networkit [12], written in C++.

I have not been able to find any implementation of the other algorithms described by Geisberger et al. Thus, publishing my implementations of these algorithms is a novel contribution.

2.5.4 Bader et al.

AlGhamdi et al. [11] published a version of the algorithm by Bader et al. as "GSIZE" in [14]. It is high-performance C++ code.

2.5.5 KADABRA

Borassi and Natale have published an implementation of their `KADABRA` algorithm in [15], which was written in C++.

2.5.6 Brandes Subset and Brandes++

The implementations of `Brandes Subset` and `Brandes++` used by Erdős et al. have been published in [21]. They are written in native Python.

The published implementation of `Brandes++` does *not* include an implementation of the `METIS` algorithm, as Erdős et al. used a standalone software package. The source code for the standalone `METIS` package is published in [26], and is high-performance parallelized C++ code [26].

2.5.7 Benchmarking

AlGhamdi et al. [11] compare 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 in [14].

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.

No comparison has ever exclusively compared algorithms from the same source (and thus programmed with the same techniques and optimizations), making my project novel work.

Chapter 3

Implementation

3.1 Completed Code

My implementation consists of five main components. First, I implemented a framework with graph representations and critical data structures. I then implemented the nine algorithms. Additionally, I designed a harness for evaluating them. Finally, I created tools to extract useful statistics from the evaluation outputs. I continuously developed a validation suite that contains both 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 $n \times n$ matrix, where an entry x_{ij} is nonzero if there is an edge from i to j. Real-world graphs are sparse, so the adjacency matrix would be almost entirely empty. As this is very memory inefficient, I discarded this option without implementing it.
- **Edge List:** Another possibility is to store a list of all edges in the graph. However, this leads to poor graph traversal performance.
- **Adjacency List:** This is the option to store a list of nodes and to associate outgoing edges with each node. This allows for fast breadth first searches, and I implemented two versions, one using complex Java objects and another simply using sets of integers.
- **Array Representation:** The final option is to use the representation described by Zardosht Kasheff in [27]. This consists of two arrays, `nodes` and `edges`. For any node v, represented by an integer $0 \ldots (n-1)$, `nodes`$[2v]$ and `nodes`$[2v+1]$ point to a range in `edges`. For every w in `edges` in this range, there is an edge (v, w) in the graph. Similarly, if the graph is weighted, then the same range in `weights` contains the weights of all edges from v.

I tested the two Adjacency List implementations as well as my Array Representation implementation by running the `Brandes` algorithm on the `ca-astroph` dataset with each of the three. I found that the Array Representation offered much higher performance, likely due to the simplicity and cache performance of iterating over an array.

I represent undirected graphs by having both the edges (i, j) and (j, i) in the graph.

Heaps

I tested three different priority queue data structures, the Binary Heap, Fibonacci Heap, and Rank Pair Heap [28]. I implemented the Binary Heap and Rank Pair Heap, and adapted an implementation of the Fibonacci Heap by Keith Schwarz [29]. I also tested the Java `PriorityQueue`

and JGraphT `JRankPairHeap` classes by running `Brandes` with each. I created a testing harness (`HeapTester`) to exercise common operations on these heaps, the results of which are below. I used my Binary Heap implementation because it was by far the fastest in both tests. It is important to note that Java's `PriorityQueue` is only slow at doing `decreaseKey()` operations.

Table 3.1: Time to Complete Heap Tests

Implementation	Time (100,000 elements)	Time (1,000,000 elements)
Binary Heap	0.164s	25.71s
Fibonacci Heap	0.374s	50.24s
Rank Pair Heap	0.341s	57.44s
JGraphT `RankPairingHeap`	0.224s	56.91s
Java `PriorityQueue`	9.086s	$>$ 3000s

3.1.2 Algorithms

I utilized a few techniques to ensure that all of my algorithm implementations were efficient. I used the Array Representation of graphs, which I found sped `Brandes` up significantly. Since all nodes are represented as integers $0 \ldots (n-1)$, I used arrays to represent mapping of nodes to values, such as the `centralities[]` array, which assigns a centrality to each node.

I also made extensive use of the Trove package [30], which contains primitive-based implementations of many standard data structures. I used 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 (as in `BFS` and `Djikstra`), this creates pressure on the garbage collector, drastically slowing down computations.

3.1.3 Verification

Throughout the project, I continuously developed a `testing` package, where I ran unit and integration tests of each component before moving onto the next. I developed the algorithms with testability in mind, and separated out components such as `Djikstra` so they can be tested independently.

I verified correctness against ground-truth implementations, and when these were not available I stepped through the code using the debugger and manually verified the steps taken. Additionally, I created debug messages that could be enabled or disabled by setting an integer "debug level". This allowed me to verify that the implementations performed as expected on large datasets.

I tested the speed of common components to ensure that my implementations were efficient, and compared the performance of my algorithm implementations to published ones (see Section 4.2).

3.1.4 Instrumentation

To meet the core requirements, I instrumented my code to record the number of times an edge is followed. This was made possible because I consistently used the following paradigm to traverse edges:

```
// v is the node we are traversing edges from
// For each vertex w such that (v,w) in E, do calculations
if (!g.empty(v)) {
   for (int j = g.start(v); j <= g.end(v); j++) {
      int w = g.adjacency[j];
      double weight = g.weights[j];
      // Do calculations
   }
}
```

By augmenting `start(i)` and `end(i)` to track the number of times they have been called, I can accurately track the number of edge traversals. Because some algorithms may use multiple graphs (SKELETON in `Brandes++` or the transpose graph in `KADABRA`), all graphs update a single global variable.

I also instrumented the harness to track the time each algorithm takes to run and instrumented `Brandes++` to track a breakdown of how time is spent on each sub-algorithm. Additionally, `KADABRA` and `Bader` record whether they finished due to their adaptive or cutoff conditions.

I implemented instrumentation in `Brandes` to record the time per node, but I made it an optional feature and did not implement it for any other algorithm. The reasons for this are discussed in Section 3.4.

3.1.5 Experimentation Harnesses

I created a harness in order to run experiments on my graph algorithms. All necessary information about each experiment (which graphs and parameters to use, and for how many iterations) is stored as constants. This eliminates the risk of mistyping parameters when running the program. The harness has the following structure:

Algorithm 2: Experimentation Harness

```
warmup()
foreach graph ∈ graphs do
    foreach algorithm ∈ algorithms do
        foreach i ∈ iterations do
            name ← name(graph, algorithm, i)
            if ¬ fileFound(name) then
                g ← constructGraph(graph)
                System.gc()
                Thread.sleep(2000)
                statistics, centralities ← algorithm(g)
                saveToFile(statistics, centralities, name)
```

Here, `warmup` runs the `Brandes` algorithm three times on the `ca-astroph` graph, and serves to raise the CPU temperature to ensure the first few experiments do not have an advantage due to the colder system temperatures.

Before each experiment, I call `System.gc()` and `Thread.sleep(2000)`, which signal the system to run garbage collection and then pause for two seconds. These ensure that garbage collection from the previous experiment does not interfere with the next one. I save the full list of centralities for later processing, as well as the performance metrics discussed in Section 3.1.4.

3.2 Git

I made frequent use of Git throughout the project. I occasionally tested new techniques (such as `SetGraph`) on new branches, and frequently used the git revision history to determine possible causes of bugs. For example, when restructuring my code I accidentally saved the number of edge traversals as an integer, which overflowed. When I later found the negative results, I was able to determine the cause of the error by examining the git revision history. Further, I purged the data from all experiments that took place after the bug was introduced, as they potentially had incorrect results.

3.3 Hardware

I originally planned to run my code on a high-performance server used by the Cambridge Systems Research Group. However, I tested running my code on it and found that it had significantly slower single-thread performance than my laptop. Additionally, it would be impossible for me to guarantee that no other CPU-intensive programs were running on the server. This is important because other programs can interfere with the performance of my tests (such as by using a portion of the power budget). Thus, I conducted all experiments on my Ubuntu 18.04 laptop with an Intel i7-9750H CPU running at 2.60GHz. The `top` command was used to verify that no process was using more than 5% of any of the eight cores.

I was able to run the experiments over multiple sessions by saving the results of completed experiments and automatically skipping previously completed experiments.

3.4 Extensions and Shortcomings

3.4.1 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 evaluating 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. that Matta et al. tested, `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.

3.4.2 METIS

While I 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 did not 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 the use of a significantly faster tool for part of it. As discussed in Section 2.1.7, I avoided this shortcoming by re-implementing the graph clustering algorithm Erdős et al. used.

3.4.3 Performance Metrics

I originally proposed to collect four metrics — total time, number of edge traversals, 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. However, unless I called it near-continuously, I would not be able to use it to find the peak memory. I could use an external memory usage monitor, but I would need to record and extract the peak memory usage for each of my 1,500+ experiments individually. More importantly, memory usage is not particularly useful for analyzing these algorithms. With the exception of `Brandes++`, the main source of memory usage is storing the lists of predecessors when running `SSSP`. Thus, monitoring peak memory usage would just reveal how many nodes `SSSP` traverses, with a significant amount of noise from the Java garbage collector. `SSSP` is a well-studied problem and there are much better ways to profile it.

Additionally, 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 s and runs a computation from each, that computation does not actually affect the centrality of s. Further, the approximate algorithms sample only a subset of nodes, making this definition of time per node meaningless for unsampled nodes. Another way to define this metric is simply average time per node. As I already record the total execution time and number of nodes, this is simply a per-graph scaling factor. This metric does not add new or useful information, so I do not use it.

3.5 Repository Overview

- `Source_Code`
  - `Algorithm_Testing`
    - `src`
      - `algorithm_evaluation`— Harnesses to run the experiments in Section 4.3
      - `algorithms` — Implementations of each selected algorithm, as well as `SSSP`
      - `data_processing` — A script to parse raw centralities and generate statistics files for each experiment
        - `NumberOfInversions.java` — This file contains `invCount`, a method written by Marek Kirejczyk in [31]
      - `framework` — A framework for my algorithms (see Section 3.1.1)
        - `graphs` — Graph representation implementations
        - `heaps` — Heap implementations
          - `fibonacci/SchwarzFibonacciHeap.java` — A fibonacci heap implementation written by Keith Schwarz in [29]
        - `parsing` — A framework to help parse graph files
      - `testing` — Unit and integration testing scripts to verify the correctness of my implementations
      - `utility` — Utility functions
  - `Data_Visualization` — A script to parse arbitrarily labeled graphs into a usable format, and scripts to generate graphs for this dissertation

Chapter 4

Evaluation

My project meets the core requirements detailed in Section 2.2.2. In my experiments (Section 4.3) I run all of the selected algorithms on many graphs (requirement 1), both weighted and unweighted (requirement 2). These experiments output betweenness centralities (4), accuracy statistics (5), and performance metrics (6). All required metrics are output apart from the maximum memory usage and computation time per node statistics (see Section 3.4). I have tested parsing graphs from a variety of common file types (3), and I will make my Git repository public after submitting this project (8).

I discuss how I verify that the project meets the criteria to 'verify to high certainty the correctness of exact algorithms' in Section 4.1 (7).

I have compiled my experimentation harness to a `.jar` file, which can be run without root access on any system with Java 8 or higher installed, meeting my distribution requirement.

In order to verify that the project meets my effectiveness requirements, I compare the performance of all of my algorithm implementations with known fast implementations in Section 4.2.

Finally, I run a series of experiments to compare the algorithms to each other. In many ways, this is the final product of my project, which aims to evaluate algorithms for betweenness centrality. However, it is also important for evaluation, as it allows me to determine whether my implementations are as fast and accurate as the authors of each algorithm claim they should be.

4.1 Verifying Exact Algorithm Correctness

To verify the correctness of my `Brandes` implementation, I compared the centralities it calculated to those computed by `JGraphT` and found that it had 100% accuracy on unweighted graphs. On weighted graphs, there was a tiny average normalized error of 1.50×10^{-6}, likely due to floating point errors when comparing near-equal distances. Because this discrepancy is much smaller than the error of any approximate algorithm I test, the quality of my experiments is not affected. Further, as any two programs that do floating point operations are likely to get slightly different results, this does not affect me meeting my core requirements.

I compared my implementation of `Brandes Subset` to the implementation by Erdős et al. by running both with the same target set. The outputs were identical, verifying that my `Brandes Subset` implementation is correct. I tested my implementation of `Brandes++` against my implementation of `Brandes Subset` and found that the results were near identical. There were a few (<5 on the `as-caida20071105` dataset) nodes with slightly different results due to a latent bug.

4.2 Comparison to Published Code

In order to determine whether my project has efficient implementations of each algorithm, I compare its runtimes to some of the fastest published implementations of the same algorithms.

I have tabulated the results in Table 4.1. I perform all experiments on the `as-caida20071105` dataset with no warmup. I use node 90 as the chosen node for `Bader`. I use the implementations tested by AlGhamdi et al. and Matta et al. (NetworKit) where possible, as well as the code published by each algorithm's authors, when available. All parallel implementations are restricted to a single thread.

Table 4.1: Comparison of the Runtimes of my Implementations vs Published Code

Algorithm	Language	Sources	Parameters	Completion Time (s)
`Brandes`	Java	**My Own**	N/A	50.04
	Java	JGraphT [32]	N/A	706.35
	C++	AlGhamdi et al. [14]	1 Thread	384.49
	C++	NetworKit [12]	N/A	35.70
`BP2007`	Java	**My Own**	800 samples	1.91
	C++	AlGhamdi et al.	800 samples	5.79
`Geisberger Linear`	Java	**My Own**	800 samples	2.01
	C++	NetworKit	800 samples	1.07
`Bader`	Java	**My Own**	Node=90 alpha = 5	0.42
	C++	AlGhamdi et al.	Node=90 alpha = 5	2.42
`KADABRA`	Java	**My Own**	$\lambda = 0.005$ $\delta = 0.1$	15.87
	C++	Borassi and Natale [9]	$\lambda = 0.005$ $\delta = 0.1$	0.34
`Brandes Subset`	Java	**My Own**	$\lvert S\rvert = 0.001 \cdot n = 26$	0.13
	Java	**My Own**	$\lvert S\rvert = 0.01 \cdot n = 264$	0.76
	Python	Erdős et al. [21]	$\lvert S\rvert = 0.001 \cdot n = 26$	8.53
	Python	Erdős et al.	$\lvert S\rvert = 0.01 \cdot n = 264$	308.58
`Brandes++`	Java	**My Own**	$\lvert S\rvert = 0.001 \cdot n = 26$ 100 Partitions	3.90
	Java	**My Own**	$\lvert S\rvert = 0.01 \cdot n = 264$ 100 Partitions	36.88
	Python	Erdős et al.	$\lvert S\rvert = 0.001 \cdot n = 26$ 100 Partitions	16.012
	Python	Erdős et al.	$\lvert S\rvert = 0.01 \cdot n = 264$ 100 Partitions	273.47

As is clearly evident, all of my implementations (except for `KADABRA`) are within an order of magnitude of the runtime of the fastest implementation, meeting my effectiveness requirements. For `KADABRA`, I am unsure what causes this discrepancy. I have verified that the structure of my implementation of `BB-BFS` is identical to the one published by Borassi and Natale. Further, as

shown in Section 4.3.10, the speed of `KADABRA` is comparable to the rest of my algorithms. It is possible that the implementation by Borassi and Natale is just extremely efficient, but I lack the means to be sure.

One interesting result to note is the variation in runtimes between different implementations of similar algorithms. `BP2007` and `Geisberger Linear` perform near-identical operations, and yet the AlGhamdi et al. implementation of `BP2007` takes three times longer than the NetworKit implementation of `Geisberger Linear`. This raises questions whether the results by Matta et al. and AlGhamdi et al. actually compare the algorithms or just the efficiency of different implementations, and highlights the motivation for this project.

4.3 Algorithm Comparison

4.3.1 Setup

I created a harness (see Section 3.1.5) to run every algorithm on every graph, with a variety of parameters. Each algorithm with multiple sets of parameters is run for each combination. All experiments are conducted three times, except for `Bader`, which is run ten times due to high variance.

`Bader` is run with nodes selected from the top *selection_range* percentage of nodes (as determined by `Brandes`). This allows me to determine the effect of node selection quality on runtime. `Brandes Subset` and `Brandes++` are run on the same randomly selected subsets, each containing *subset_size* percent of nodes.

All experiments are conducted on the computer described in Section 3.3.

Table 4.3: Parameters Tested for Each Algorithm

Algorithm	Parameters
`Brandes`	N/A
`BP2007`	samples = [25, 50, 100, 200, 400, 800, 1600, 3200]
`Geisberger Linear`	samples = [25, 50, 100, 200, 400, 800, 1600, 3200]
`Geisberger Bisection`	samples = [25, 50, 100, 200, 400, 800, 1600, 3200]
`Geisberger Bisection Sampling`	samples = [25, 50, 100, 200, 400, 800, 1600, 3200] accumulation samples = [1, 2, 4, 8, 16]
`Bader`	selection_range = [0.1%, 1%, 10%, 20%] alpha = [2, 5]
`KADABRA`	λ = [0.01, 0.025, 0.02, 0.015, 0.005, 0.001] δ = 0.1
`Brandes Subset`	subset_size = [0.1, 1, 5, 10]
`Brandes++`	subset_size = [0.1, 1, 5, 10] partitions= [2, 8, 32, 128, 512, 2048]

4.3.2 Statistics

All experiments save both the computed centralities, and a descriptor file. This file describes the parameters, number of edge traversals, total time taken, and any algorithm-specific instrumentation.

After all experiments concluded, I ran `ProcessFiles`, a Java program I created to extract the statistics detailed below. It compares the centralities generated by the experiment to the one output by `Brandes` (an exact algorithm) for the same graph.

These statistics represent important problems that each algorithm attempts to solve and are widely used in literature. For example, AlGhamdi et al. chose to use average error, maximum error, and top 1% correctness [11], and both Brandes and Pich and Geisberger et al. compare their algorithms using normalized Euclidean distance and inversion percentage [5][16].

1. **Average Error**

This is the average absolute error of the normalized centrality. Normalized centrality is computed by $\frac{C(v)}{(n-1)(n-2)}$, and scales centrality by graph size. It is a simple metric for absolute error.

2. **Maximum Error**

Similarly, this is the maximum absolute normalized error, and is useful for applications that need error bounds.

3. **Normalized Euclidean Distance**

Each list of centralities (the experiment's and `Brandes`'s) is treated as a vector, which are then normalized and the Euclidean distance between them is computed. This is a more robust error metric that more harshly penalizes large errors.

4. **Top 1% Correctness**

This is the percentage of nodes in the top 1% of betweenness centrality values (as given by `Brandes`) that the algorithm correctly identifies. It is important for applications that use betweenness centrality to find a set of important nodes.

5. **Inversion Percentage**

This statistic measures the accuracy of the algorithm in ranking nodes by centrality, and is computed by the following procedure:

(1). For both the experiment's and `Brandes`'s outputs, order the nodes by centrality.
(2). Calculate the number of inversions in the orderings.
(3). Normalize by the maximal number of possible inversions, given by $\binom{n}{2}$.

6. **Edge Traversals**

Edge traversals not an accuracy statistic, but rather a performance metric. It is calculated as the algorithm runs, and is a simple count of how many times an edge in a graph is followed (the ID of the node on the other end of an edge is queried).

4.3.3 Datasets

Table 4.5: Datasets Used

Graph	Domain	Properties	Description
`4932-protein` [25]	Biology	6,574 Nodes 1,845,966 Edges Undirected Weighted	Protein relations of *Saccharomyces cerevisiae* (Brewer's Yeast). Nodes are proteins and edges are association (based on direct and indirect interaction) between proteins. The original dataset uses high scores to represent closely associated proteins, so I have inverted all weights such that short paths mean close relations.
`ca-astroph` [24]	Collaboration	18,772 Nodes 396,160 Edges Directed Unweighted	Collaboration network of authors submitting to the Astro Physics category of the arXiv e-print. Nodes are authors and edges are collaborations between them.
`as-caida20071105` [23]	Technology	26,475 Nodes 106,762 Edges Undirected 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` [23]	Social	77,360 Nodes 905,468 Edges Directed Unweighted	User-defined tags of other users on Slashdot, a website featuring user-submitted news articles. Users tag other users as 'friends' or 'foes', but this network has no distinction between the two. Nodes are users and edges are tags.
`com-amazon` [23]	Business	334,863 Nodes 925,872 Edges Undirected 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.

4.3.4 Experimental Results

I ran all of the algorithms in Table 1.1 on all datasets in Table 4.5 with all parameters in Table 4.3, for three iterations, with the following exceptions. Due to excessive runtimes, I only ran `Brandes` once on `com-amazon` and did not run `BrandesSubset` or `Brandes++` on `com-amazon` or `slashdot0811`. Due to high variance, I ran `Bader` for ten iterations.

Brandes

`Brandes` is the ground truth algorithm, and all approximate algorithms are compared to it. As such, only *time* and *edge traversals* are meaningful metrics for `Brandes`. These are tabulated below, along with the standard deviation. There is no variation in the number of edge traversals, and `Brandes` is only run once on `com-amazon`.

Graph	Time (s), 95% confidence	Edge Traversals
4932-protein	86.49 ± 1.77	1.22×10^{10}
ca-astroph	50.05 ± 1.45	8.02×10^{9}
as-caida20071105	72.87 ± 0.31	4.93×10^{9}
slashdot0811	919.017 ± 8.23	8.00×10^{10}
com-amazon	31021.63	9.56×10^{11}

4.3.5 BP2007

`BP2007` approximates betweenness centrality by doing a user-defined number of samples. By varying this parameter, we can see that the computation time and number of edge traversals are highly correlated (Figure 4.1), with an R^2 value of 0.99996.

Additionally, we can see the effect the number of samples has on both the computation time and accuracy in Figure 4.2. Plotting this on a log-log plot in Figure 4.3, we can see that time and error are indeed inversely correlated.

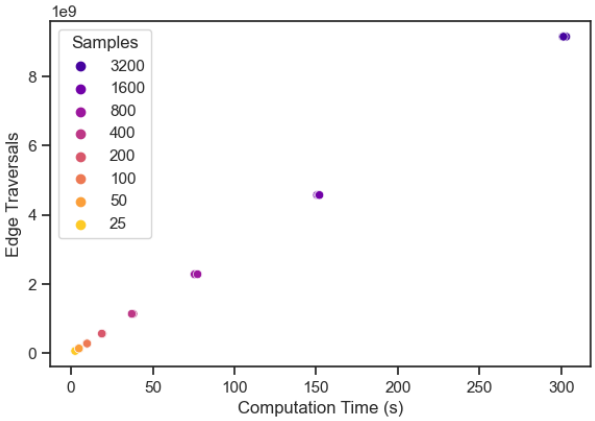

Figure 4.1: `BP2007` run on `com-amazon` graph, Computation Time(s) vs Edge Traversals

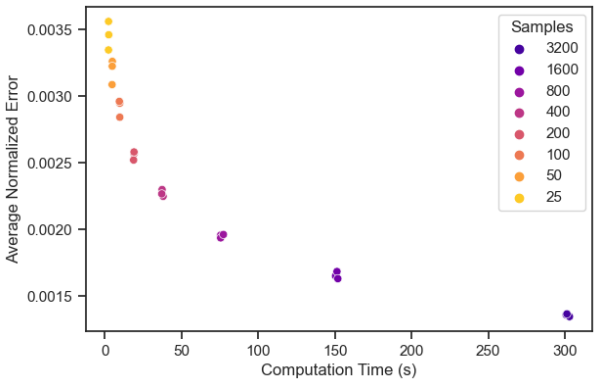

Figure 4.2: `BP2007` run on `com-amazon`, Computation Time(s) vs Average Normalized Error

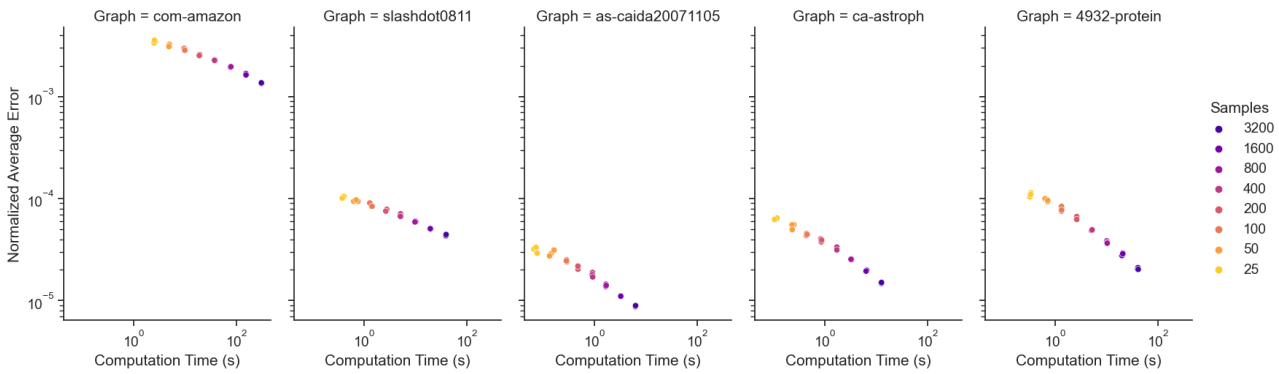

Figure 4.3: BP2007 run on all graphs, Time(s) vs Average Normalized Error (log-log)

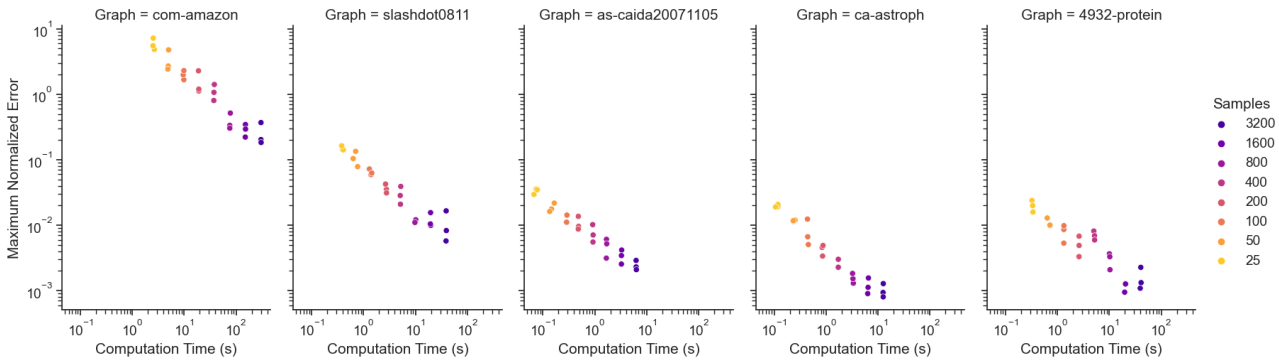

Figure 4.4: BP2007 run on all graphs, Time(s) vs Maximum Normalized Error (log-log)

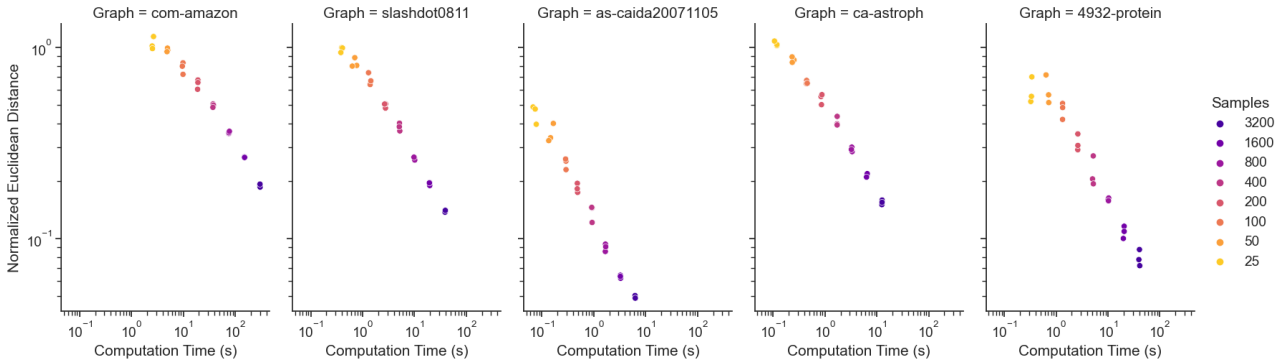

Figure 4.5: BP2007 run on all graphs, Time(s) vs Normalized Euclidean Distance (log-log)

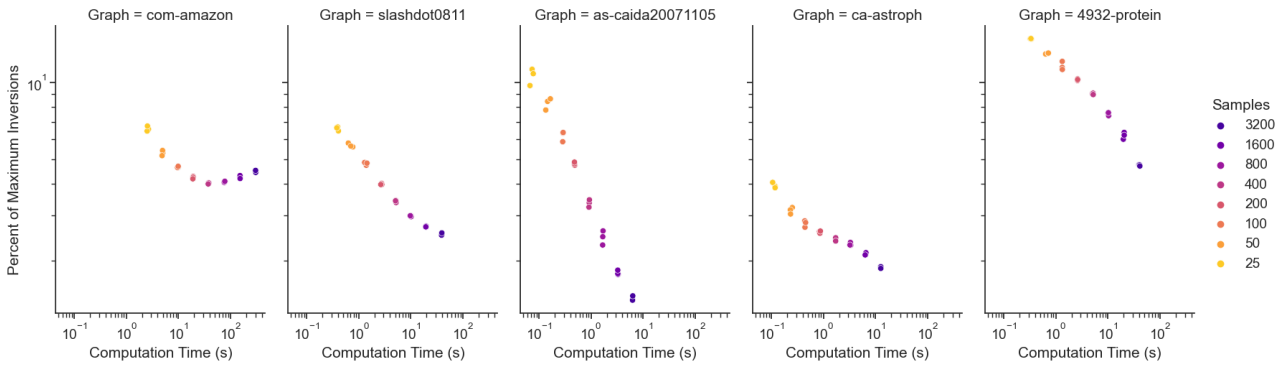

Figure 4.6: BP2007 run on all graphs, Time(s) vs Percent of Maximum Inversions (log-log)

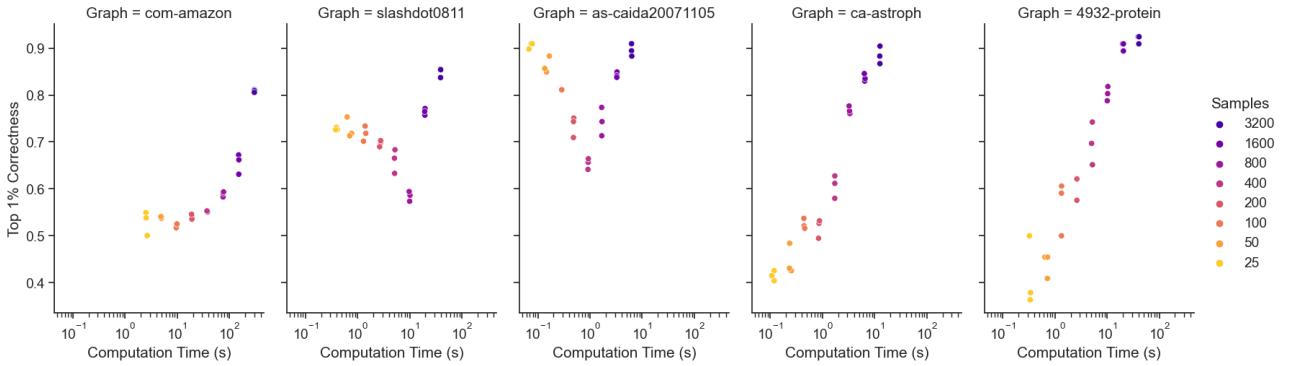

Figure 4.7: `BP2007` run on all graphs, Time(s) vs Top 1% Correctness (log-lin)

Almost all of these results are as expected, with measures of error decreasing inversely proportional to computation time. There are two major exceptions. In Figure 4.6, the number of inversions increases when going above 400 samples on `com-amazon`. Geisberger et al. note a similar phenomenon occur in their tests and hypothesize that this is due to unimportant nodes getting a large boost to their centrality from being near a chosen pivot [5].

In Figure 4.7, the top 1% correctness is high at 25 samples on `slashdot0811` and `as-caida20071105` and decreases until 200 samples. One possible explanation for this is on these networks, running `BFS` from any node will pass through a high centrality node, giving it a non-zero centrality. When the number of samples increases, the number of non-zero centralities increases, competing for space in the top 1%.

4.3.6 Geisberger Linear and Bisection

The algorithms by Geisberger et al. are all modifications of `BP2007`, making it useful to compare them to each other. In Figure 4.8, I compare the Euclidean distance for `BP2007`, `Geisberger Linear`, and `Geisberger Bisection`. I use Euclidean distance since it follows the same trends as average and maximum error, but scales more uniformly between different graphs. I also evaluate the inversion percent and top 1% correctness in Figure 4.9 and Figure 4.10 respectively. Since `Geisberger Bisection Sampling` takes an additional parameter, I evaluate it in Section 4.3.7.

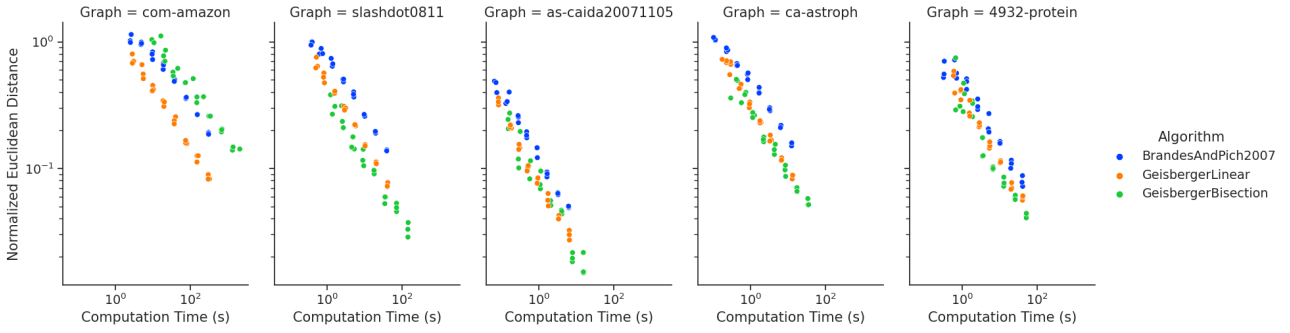

Figure 4.8: `BP2007`, `Geisberger Linear`, and `Geisberger Bisection`, Computation Time(s) vs Normalized Euclidean Distance (log-log)

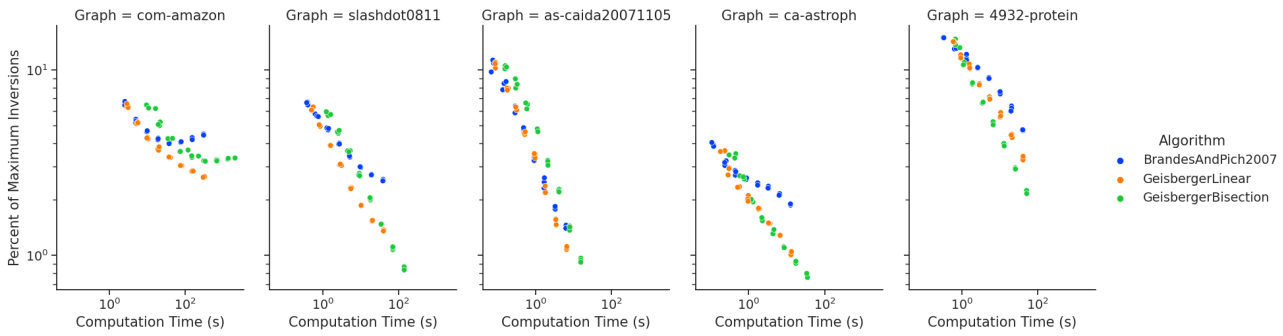

Figure 4.9: `BP2007`, `Geisberger Linear`, and `Geisberger Bisection`, Computation Time(s) vs Percent of Maximum Inversions (log-log)

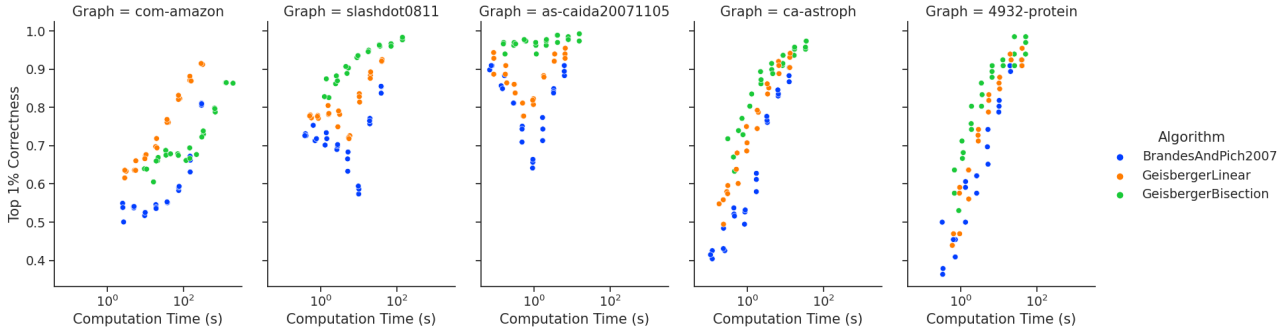

Figure 4.10: `BP2007`, `Geisberger Linear`, and `Geisberger Bisection`, Computation Time(s) vs Top 1% Correctness (log-lin)

In Figure 4.8, we can see that although `Geisberger Bisection` takes more time per sample than either of the other two algorithms, it achieves the lowest Euclidean distance per second of computation. However, on `com-amazon`, the performance is substantially worse, and `Geisberger Linear` performs best.

In Figure 4.9, we can see `Geisberger Linear` and `Geisberger Bisection` fix the issue that `BP2007` has with inversions on `com-amazon` (with `Geisberger Linear` having notably better performance). Additionally, they perform much better than `BP2007` on `slashdot0811` and `ca-astroph`.

In Figure 4.10, we can see that `Geisberger Bisection` solves the problem that `BP2007` (and to a lesser extent, `Geisberger Linear`) have with top 1% correctness on `slashdot-0811` and `as-caida20071105`. However, it has a similar dip at around 200 samples on `com-amazon`.

4.3.7 Geisberger Bisection Sampling

In addition to the number of pivots, `Geisberger Bisection Sampling` takes an additional parameter: the number of samples when accumulating centralities. Here, I compare `BP2007`, `Geisberger Linear`, `Geisberger Bisection`, and `Geisberger Bisection Sampling` with five different values for the number of accumulation samples. I have plotted `com-amazon` and `slashdot0811` as these two are the largest graphs I used. The 95% confidence interval for each metric is shaded.

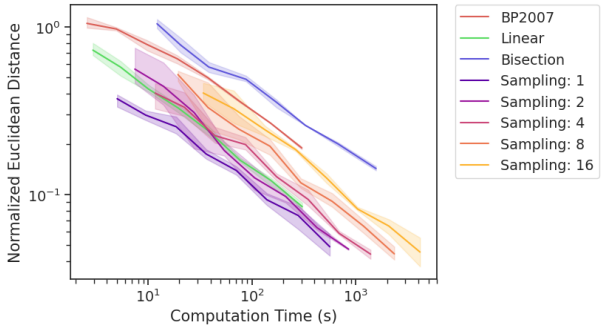

Figure 4.11: `BP2007` and `Geisberger` family run on `com-amazon`, Computation Time(s) vs Normalized Euclidean Distance (log-log). 95% confidence intervals shaded.

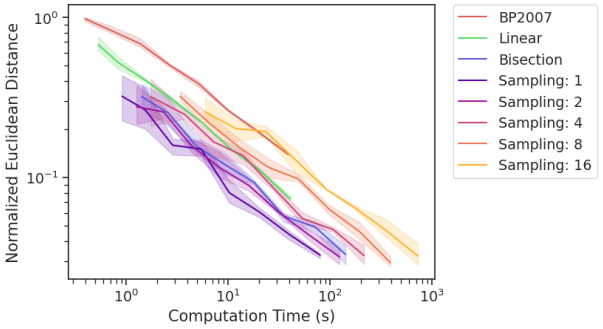

Figure 4.12: `BP2007` and `Geisberger` family run on `slashdot0811`, Computation Time(s) vs Normalized Euclidean Distance (log-log). 95% confidence intervals shaded.

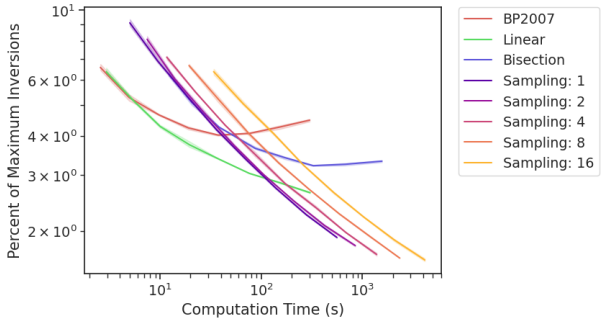

Figure 4.13: `BP2007` and `Geisberger` family run on `com-amazon`, Computation Time(s) vs Percent of Maximum Inversions (log-log). 95% confidence intervals shaded.

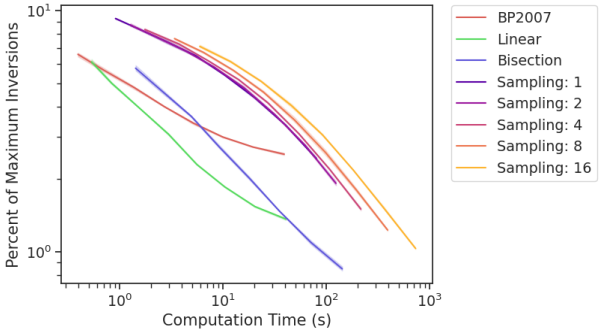

Figure 4.14: `BP2007` and `Geisberger` family run on `slashdot0811`, Computation Time(s) vs Percent of Maximum Inversions (log-log). 95% confidence intervals shaded.

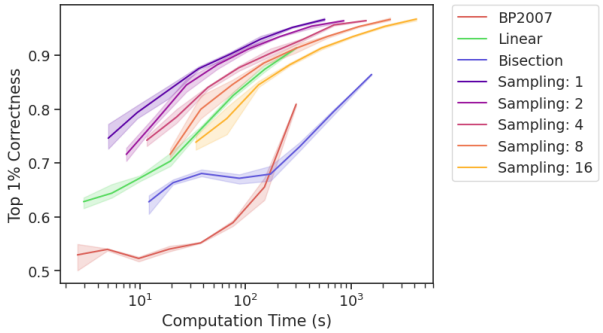

Figure 4.15: `BP2007` and `Geisberger` family run on `com-amazon`, Computation Time(s) vs Top 1% Correctness (log-lin).
95% confidence intervals shaded.

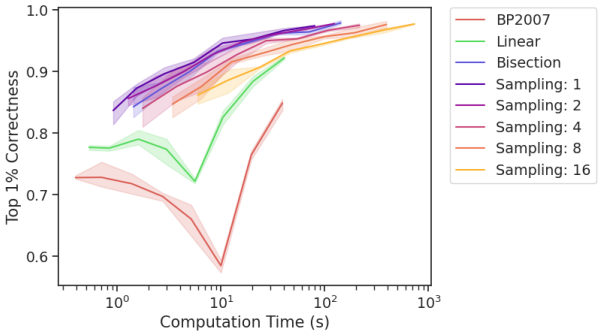

Figure 4.16: `BP2007` and `Geisberger` family run on `slashdot0811`, Computation Time(s) vs Top 1% Correctness (log-lin).
95% confidence intervals shaded.

As we can see, `Geisberger Bisection Sampling` achieves a lower euclidean distance and significantly higher top 1% correctness than the other methods. In fact, using just one sample gives the highest accuracy per second of computation. This a remarkable result and runs contrary to those found by Geisberger et al. They found that using more samples allows for higher accuracy per second of computation [5].

Interestingly, the number of inversions (Figure 4.14) behaves differently to the rest of the statistics, and `Geisberger Bisection Sampling` does a much worse job of estimating it on `slashdot0811`. Determining the cause of this behavior could be the basis of an interesting project.

4.3.8 Bader

The algorithm by Bader et al. performs the same operations as `BP2007`, but has an adaptive termination condition. This means that each execution will fall somewhere on the curves in Section 4.3.5, with the termination conditioning determining where. Thus, it is useful to understand the effects that differing parameter values have on the termination time. `Bader` takes two parameters, a vertex v and a value alpha. Bader et al. state that v must be a high centrality vertex, and test with vertices in the top 1% of centralities [8]. They require that alpha must be ≥ 2.

I vary the percentile that I select nodes from — ranging from the top 0.1% of nodes to the top 20% of nodes — to test how high the centrality needs to be. This reflects the real-world scenario where we do not know exact centralities, but may be able to guess which nodes have high centralities. I use two different values of alpha, 2 and 5. Because of the high variability of results, I run each experiment ten times.

Running the `Bader` experiments on `com-amazon` would have approximately 42 hours, so was not done.

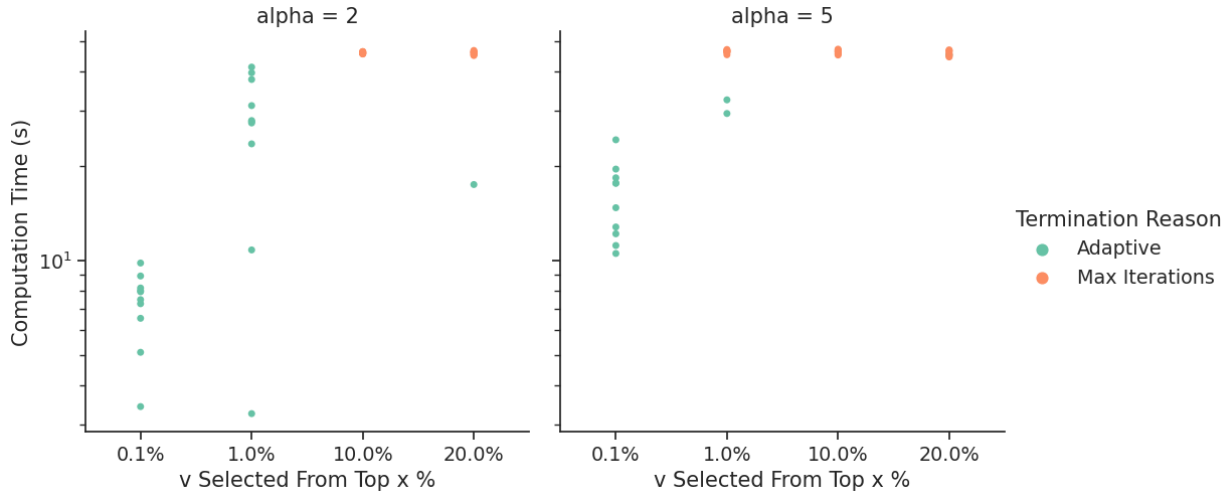

Figure 4.17: `Bader` run on `slashdot0811`, Computation Time (s) vs Top Percent of Nodes that v is Selected From (log scale)

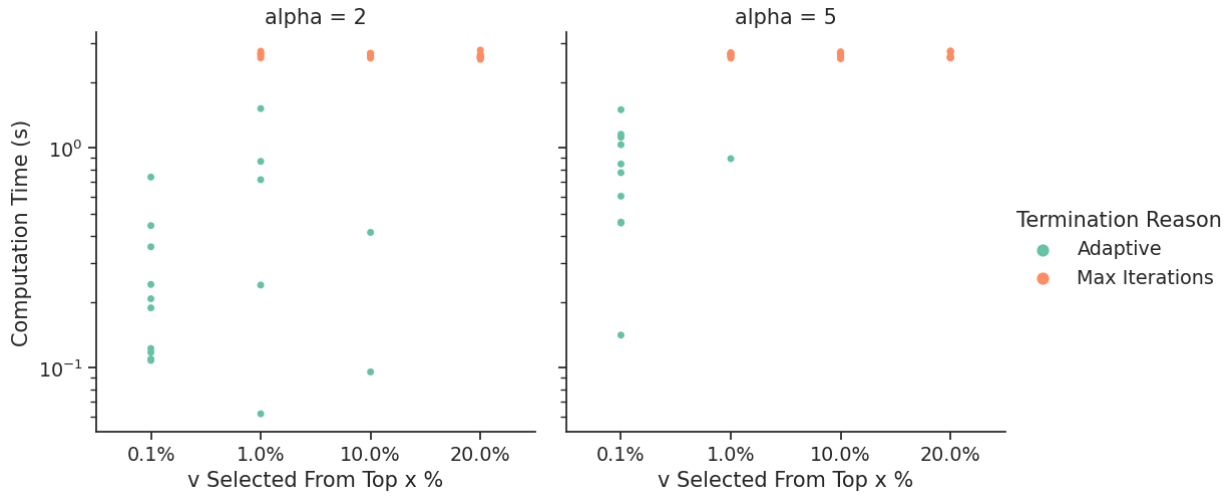

Figure 4.18: `Bader` run on `as-caida20071105`, Computation Time (s) vs Top Percent of Nodes that v is Selected From (log scale)

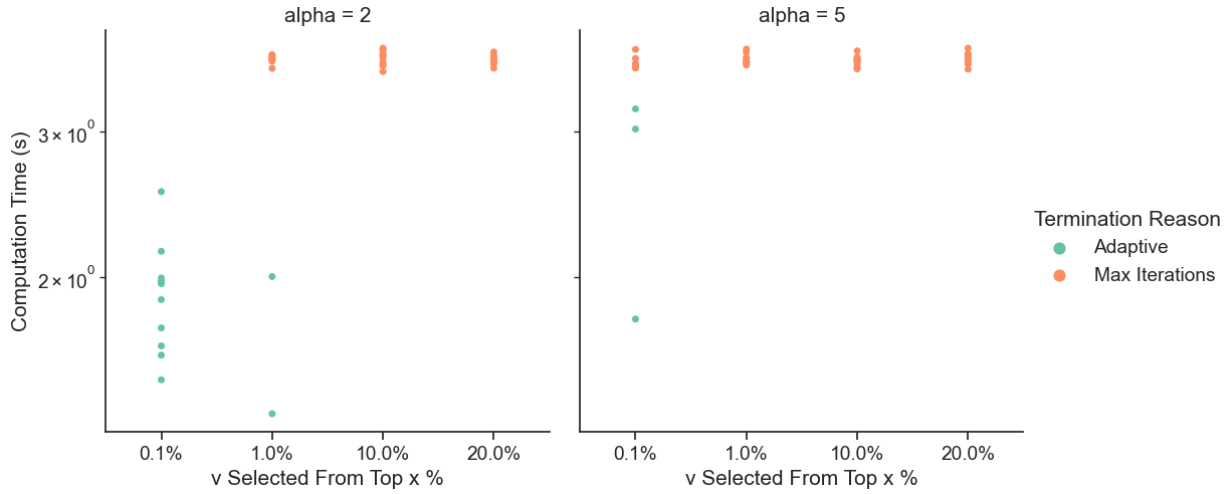

Figure 4.19: `Bader` run on `ca-astroph`, Computation Time (s) vs Top Percent of Nodes that v is Selected From (log scale)

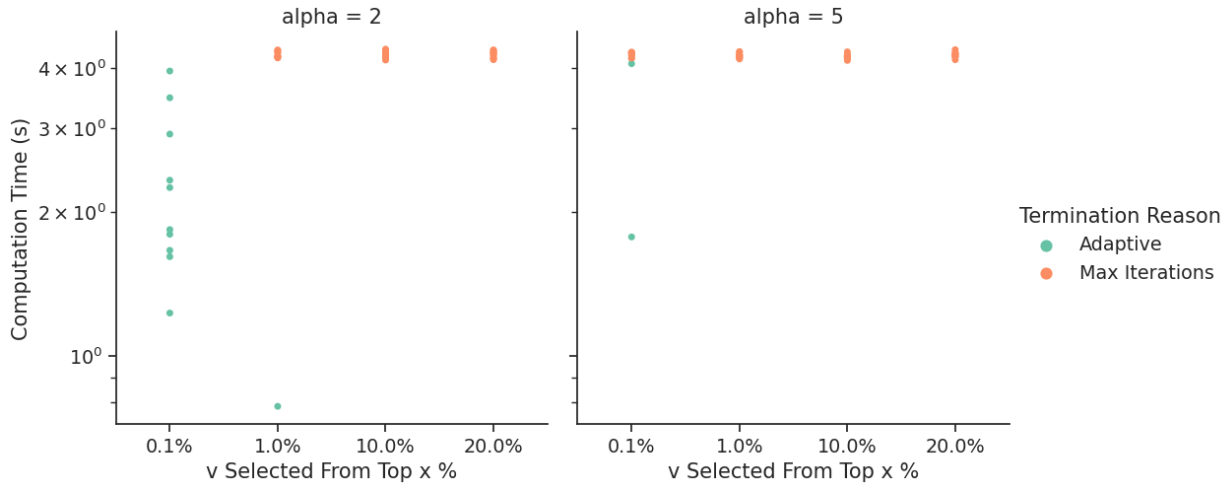

Figure 4.20: `Bader` run on `4932-protein`, Computation Time (s) vs Top Percent of Nodes that v is Selected From (log scale)

As we can see, the runtime of `Bader` depends a great deal on both alpha and the centrality of the node that we choose. For alpha = 5, we have to pick a node in the top 0.1% to avoid simply running into the cutoff of $n/20$ samples, for a graph with n nodes. If alpha = 2, both the quality of node we select and the graph affect the runtime. If the node has a higher centrality, the runtime decreases, as Bader et al. predict. However, on `ca-astroph` and `4932-protein`, even a node from the top 1% of centralities causes `Bader` to hit the cutoff.

Thus, in order to use `Bader` as an adaptive algorithm, one needs to supply it with a node in the top 0.1%, and use alpha = 2. Even then, there is a great deal of variation in the number of samples used (and thus the overall runtime). This makes `Bader` hard to use as an adaptive algorithm, as it requires a very good estimate of centralities and has a highly variable runtime.

4.3.9 KADABRA

`KADABRA` utilizes a different calculation technique to all other algorithms discussed thus far and is adaptive. Just as in `Bader`, it is useful to determine the effect that the parameter lambda has on performance. I illustrate this in Figures 4.21 and 4.22. These figures illustrate that both accuracy and computation time rapidly increase with decreasing values of lambda.

However, since `KADABRA` uses a different framework to the other algorithms, the most relevant evaluation of it is simply a direct comparison to the other algorithms. I do this in Section 4.3.10.

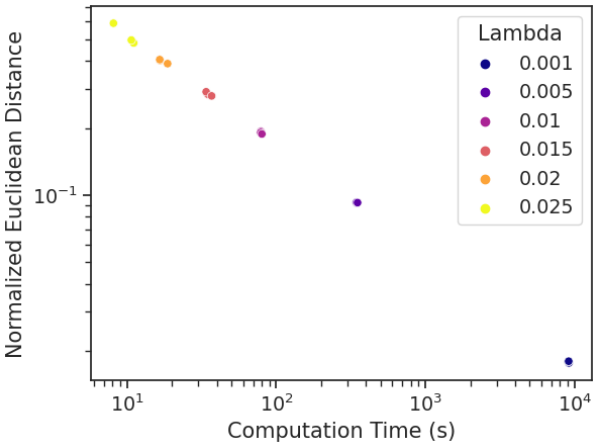

Figure 4.21: `KADABRA` run on `com-amazon`, Computation Time(s) vs Normalized Euclidean Distance (log-log).
95% confidence intervals shaded.

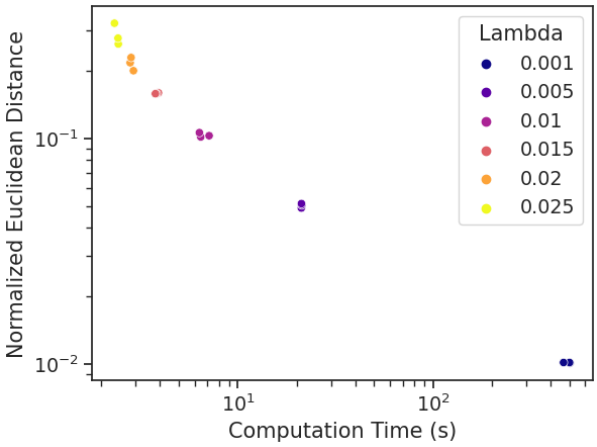

Figure 4.22: `KADABRA` run on `slashdot0811`, Computation Time(s) vs Normalized Euclidean Distance (log-log).
95% confidence intervals shaded.

4.3.10 Overall Algorithm Comparison

Having examined the individual performance of algorithms that estimate betweenness centrality (`BrandesSubset` and `Brandes++` compute target set betweenness centrality), we can examine their relative performance. Since the accuracy-per-second of `Bader` is identical to `BP2007`, it is excluded from our comparison. I have plotted all five statistics for all graphs, with 95% confidence intervals shaded.

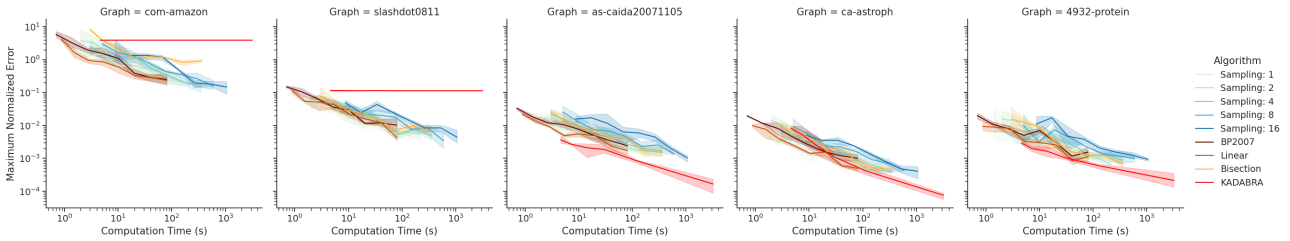

Figure 4.23: All algorithms, Computation Time(s) vs Maximum Normalized Error (log-log)

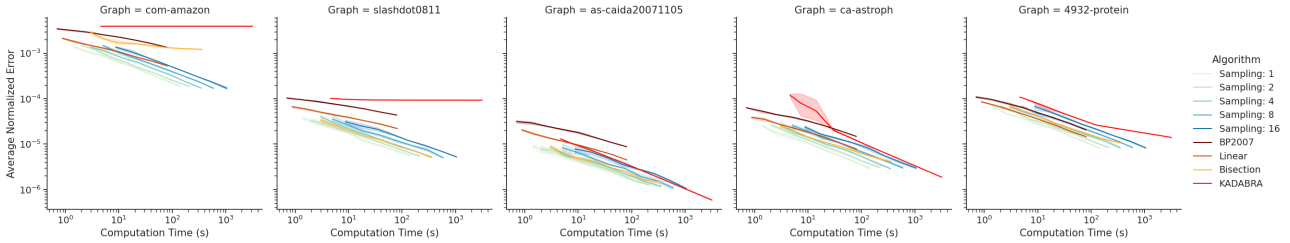

Figure 4.24: All algorithms, Computation Time(s) vs Average Normalized Error (log-log)

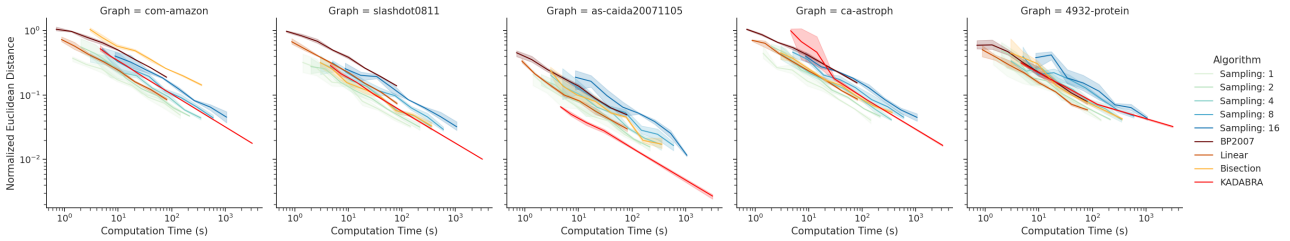

Figure 4.25: All algorithms, Computation Time(s) vs Normalized Euclidean Distance (log-log)

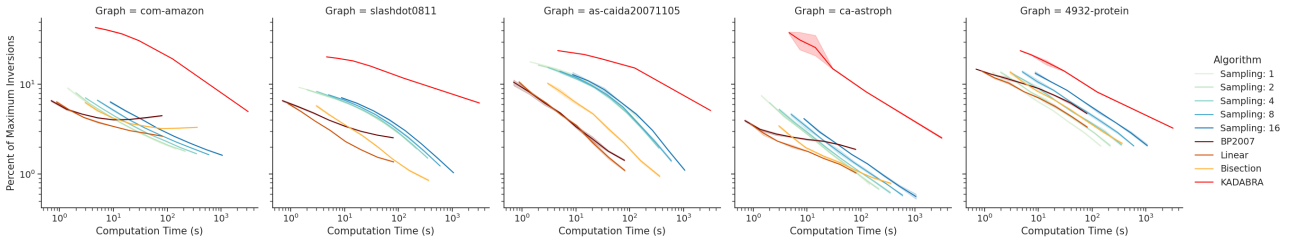

Figure 4.26: All algorithms, Computation Time(s) vs Percent of Maximum Inversions (log-log)

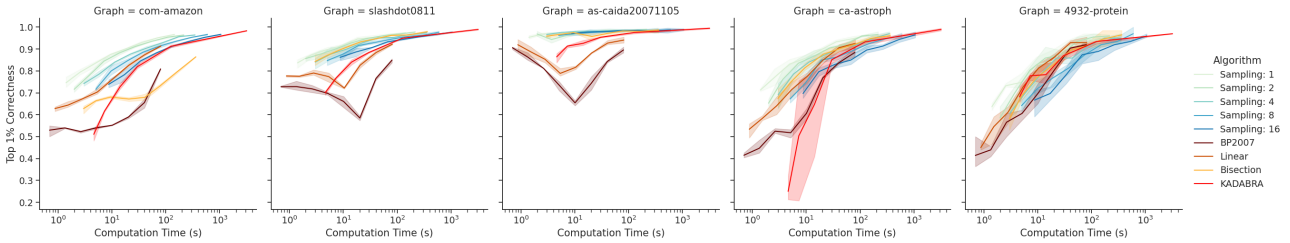

Figure 4.27: All algorithms run on all graphs, Computation Time(s) vs Top 1% Correctness (log-lin)

Strikingly, the maximum and average error of KADABRA on com-amazon and slashdot is nearly constant. In fact, the average error on com-amazon decreases by about 1.46×10^{-7} as lambda decreases from 0.025 to 0.001. It is possible this is due to a bug, but KADABRA behaves as expected on all other graphs, and on all other statistics.

It is clear from these results that the 'best' algorithm depends a great deal on what metric of "accuracy" is desired. For applications where only the maximum error matters, KADABRA will either give exceptionally good results or fail entirely. Besides KADABRA, Geisberger Linear tends to have the highest accuracy, though as all of the confidence intervals overlap, it is impossible to give a complete assessment.

Average error gives a much clearer ranking. The Geisberger Bisection Sampling algorithm with one accumulation sample consistently gives the lowest average error.

The same results hold for euclidean distance, though Geisberger Linear also gives good results on 4932-protein, and KADABRA performs exceptionally well on as-caida20071105.

Applications that require a ranking of betweenness centralities face a much trickier decision. On com-amazon, ca-astroph, and 4932-protein, Geisberger Bisection Sampling eventually outperforms the other algorithms. However, it performs very poorly on slashdot0811 and as-caida20071105. Geisberger Linear consistently performs well on all graphs.

When computing the top 1% of centralities, Geisberger Bisection Sampling again performs very well. Geisberger Linear has a dip in accuracy on two graphs, Geisberger Bisection does not converge quickly on com-amazon, and BP2007 has poor performance overall. KADABRA has low accuracy with high values of lambda, but quickly converges with the rest of the algorithms.

4.3.11 Brandes Subset and Brandes++

Of the algorithms I evaluated, only Brandes Subset and Brandes++ operate on a subset of nodes. They are both exact algorithms, so none of the accuracy statistics are relevant, only runtime and the number of edge traversals.

Both algorithms take a target set of nodes to compute centralities relative to. In each iteration, I used the same target set for Brandes Subset and all six partition settings in Brandes++. Below, I have plotted the time per node in the target set, as well as the number of edge traversals per node in the target set. I have broken down the time for Brandes++ into four sections, though in most graphs only three are visible:

1. Time to partition the graph
2. Time to construct SKELETON — this takes significantly less time than the other components, so is not visible in any figure
3. Time to run Brandes Skeleton
4. Time to run the Centrality sub-algorithm. This dominates the runtime of Brandes++

In each chart, the error bar represents the one standard deviation. I was unable to run Brandes++ on com-amazon or slashdot0811 because storing the distance from every node in the target set to every frontier node (required for the Centralities sub-algorithm) overflowed the Java heap. I thus show charts for as-caida20071105 and ca-astroph as they are the largest remaining graphs.

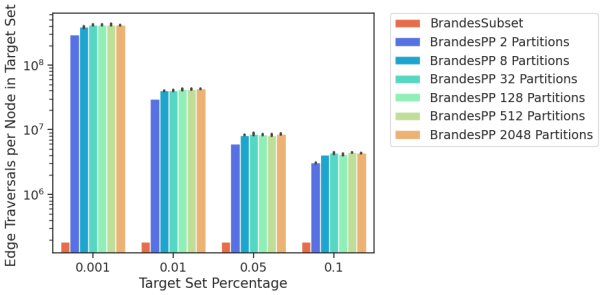

Figure 4.28: `Brandes Subset` and `Brandes++` run on `as-caida20071105`, Portion of Graph in Target Set vs Edge Traversals per Node in Target Set(s) (log scale).

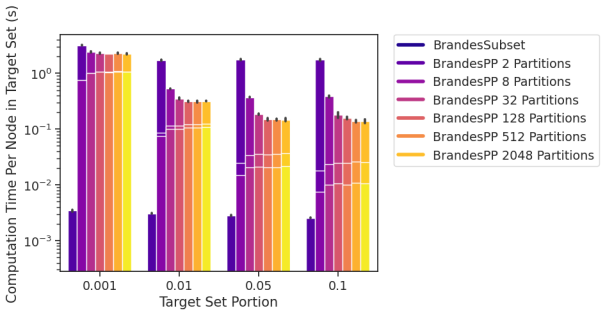

Figure 4.29: `Brandes Subset` and `Brandes++` run on `as-caida20071105`, Portion of Graph in Target Set vs Computation Time per Node in Target Set(s) (log scale).

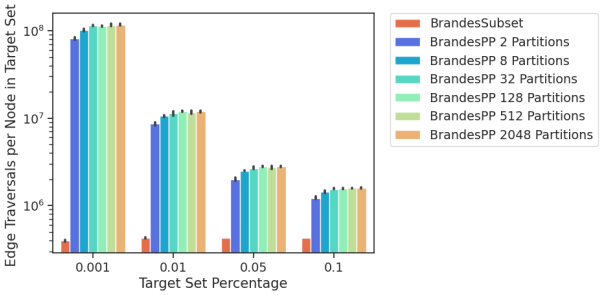

Figure 4.30: `Brandes Subset` and `Brandes++` run on `ca-astroph`, Portion of Graph in Target Set vs Edge Traversals per Node in Target Set(s) (log scale).

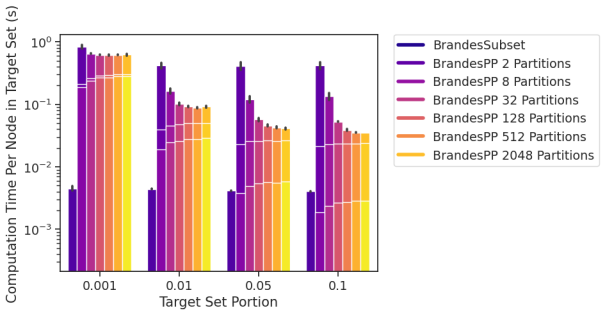

Figure 4.31: `Brandes Subset` and `Brandes++` run on `ca-astroph`, Portion of Graph in Target Set vs Computation Time per Node in Target Set(s) (log scale).

As we can see, both the number of edge traversals and the total runtime grow slower than the size of the target set. Additionally, the total runtime can be decreased by increasing the number of partitions, though these gains rapidly drop away.

However, note that `Brandes++` is several orders of magnitude slower than `Brandes Subset` (the y-axes are on a log scale). This is due to two factors. First, time to partition the graph is actually greater than the runtime of `Brandes Subset`. Second, even if the graph partition were instant, the `Centrality` algorithm scales very poorly. It requires storing the distance from every node in the target set to every frontier node in the graph, taking $\mathcal{O}(p \cdot n)$ memory. Additionally, it requires an $\mathcal{O}(p \cdot n^2)$ operation to find the centralities of non-frontier nodes. This causes the `Centrality` algorithm to dominate the runtime of `Brandes++`.

These experimental results directly contradict the conclusions by Erdős et al. [10], and raise questions as to where this discrepancy comes from. Their implementation of `Centrality` follows exactly the same steps as mine, making this discrepancy unlikely to be from implementation details. It is possible that their `SSSP` implementation is slow, so the speedup from running `Brandes Skeleton` on the smaller SKELETON graph substantially improves their runtime. Definitively determining the cause of this discrepancy requires substantially more investigation.

Chapter 5

Conclusions

In this project, I have compared the speed and accuracy of nine algorithms for computing betweenness centrality, a computationally intensive metric widely used in graph analysis. I created a framework for implementing and testing these algorithms, and found that all of my implementations apart from `KADABRA` are nearly as fast as some of the fastest known implementations.

By testing these algorithms on real world datasets, I was able to determine which algorithm performs best on each error metric, across a variety of datasets. While evaluating the performance of these algorithms, I found surprising trends in several algorithms. This includes evidence that runs contrary to conclusions drawn by Erdős et al. and Geisberger et al.

There is a great deal of room for future work to determine exactly what causes these surprising trends. I will list some possible lines of investigation:

1. Why does `KADABRA` perform so poorly on percent of maximum inversions?
2. What causes the dip in top 1% correctness for `BP2007` and `Geisberger Linear`?
3. Why do `BP2007` and `Geisberger Bisection` perform so poorly on `com-amazon`?
4. What causes the discrepancy between my results and those by Erdős et al. and Geisberger et al.?
5. What properties of the graph affect the performance of each algorithm?

These questions can only be answered with thorough analysis and a large volume of tests, but may provide valuable insights into which algorithms to use, and how to create better ones.

Over the course of this project, I have learned a great deal about writing efficient code, understanding scientific papers, creating large pieces of literature, and project management. By implementing nine algorithms, I have become very familiar with how to write efficient Java code. Additionally, having to thoroughly understand many scientific papers has given me skills in parsing scientific literature. Writing this dissertation has helped me develop invaluable skills in explaining large projects. Finally, I have learned lessons about project management. My original aims were overly ambitious for the time I had for the project, and while I managed to finish all of my core requirements, I had to use all of the time I set apart as a buffer.

If I were to re-do the project, I would spend more time analyzing how difficult each algorithm is to implement, and omit the `Brandes++` algorithm, as implementing it took far longer than any other algorithm.

I hope that my code and experimental results can be of use to the scientific community, and appreciate the window into academia that this project has provided.

Bibliography

[1] Navavat Pipatsart et al. "Network Based Model of Infectious Disease Transmission in Macroalgae". In: *International Journal of Simulation: Systems, Science and Technology* 19 (2018), pp. 11.1–11.8. DOI: 10.5013/IJSSST.a.19.05.11.

[2] Evelien Otte and Ronald Rousseau. "Social network analysis: a powerful strategy, also for the information sciences". In: *Journal of Information Science* 28.6 (2002). _eprint: https://doi.org/10.1177/016555150202800601, pp. 441–453. DOI: 10.1177/016555150202800601.

[3] Philip Nuss et al. "Mapping supply chain risk by network analysis of product platforms". In: *Sustainable Materials and Technologies* 10 (2016), pp. 14 –22. ISSN: 2214-9937. DOI: https://doi.org/10.1016/j.susmat.2016.10.002.

[4] Gil Amitai et al. "Network Analysis of Protein Structures Identifies Functional Residues". In: *Journal of Molecular Biology* 344.4 (2004), pp. 1135 –1146. ISSN: 0022-2836. DOI: https://doi.org/10.1016/j.jmb.2004.10.055.

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

[6] Ulrik Brandes. "A faster algorithm for betweenness centrality". In: *The Journal of Mathematical Sociology* 25.2 (2001). Publisher: Routledge _eprint: https://doi.org/10.1080/0022250X.2001.9990249, pp. 163–177. DOI: 10.1080/0022250X.2001.9990249.

[7] Linton C. Freeman. "A Set of Measures of Centrality Based on Betweenness". In: *Sociometry* 40.1 (1977). Publisher: [American Sociological Association, Sage Publications, Inc.], pp. 35–41. ISSN: 00380431. URL: http://www.jstor.org/stable/3033543.

[8] David A. Bader et al. "Approximating Betweenness Centrality". In: *Algorithms and Models for the Web-Graph*. Ed. by Anthony Bonato and Fan R. K. Chung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 124–137. ISBN: 978-3-540-77004-6.

[9] Michele Borassi and Emanuele Natale. "KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation". In: *CoRR* abs/1604.08553 (2016). _eprint: 1604.08553. URL: http://arxiv.org/abs/1604.08553.

[10] Dóra Erdős et al. "A Divide-and-Conquer Algorithm for Betweenness Centrality". en. In: *Proceedings of the 2015 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, June 2015, pp. 433–441. ISBN: 978-1-61197-401-0. DOI: 10.1137/1.9781611974010.49.

[11] Ziyad AlGhamdi et al. "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. event-place: Chicago, IL, USA. New York, NY, USA: Association for Computing Machinery, 2017. ISBN: 978-1-4503-5282-6. DOI: 10.1145/3085504.3085510.

[12] Eugenio Angriman, Alexander van der Grinten, and Henning Meyerhenke. *NetworKit*. Publication Title: NetworKit. Apr. 2021. URL: https://networkit.github.io/index.html.

[13] John Matta, Gunes Ercal, and Koushik Sinha. "Comparing the speed and accuracy of approaches to betweenness centrality approximation". In: *Computational Social Networks* 6.1 (Feb. 2019), p. 2. ISSN: 2197-4314. DOI: 10.1186/s40649-019-0062-5.

[14] Ziyad AlGhamdi et al. *A Benchmark for Betweenness Centrality Approximation Algorithms on Large Graphs*. 2017. URL: https://ecrc.github.io/BeBeCA/.

[15] natema. *natema/kadabra*. original-date: 2016-04-27T17:14:43Z. July 2020. URL: https://github.com/natema/kadabra (visited on 04/23/2021).

[16] Ulrik Brandes and Christian Pich. "CENTRALITY ESTIMATION IN LARGE NETWORKS". In: *International Journal of Bifurcation and Chaos* 17.07 (2007). _eprint: https://doi.org/10.1142/S0218127407018403, pp. 2303–2318. DOI: 10.1142/S0218127407018403.

[17] Michele Borassi, Pierluigi Crescenzi, and Michel Habib. "Into the Square - On the Complexity of Quadratic-Time Solvable Problems". In: *arXiv:1407.4972 [cs]* (July 2014). URL: http://arxiv.org/abs/1407.4972 (visited on 02/13/2021).

[18] Joshua Brakensiek. *Lecture 17: The Strong Exponential Time Hypothesis*. URL: http://web.stanford.edu/class/cs354/scribe/lecture17.pdf.

[19] Maarten Löffler and Jeff M. Phillips. "Shape Fitting on Point Sets with Probability Distributions". In: *Lecture Notes in Computer Science Algorithms - ESA 2009* (2009), pp. 313–324. DOI: 10.1007/978-3-642-04128-0_29.

[20] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20.1 (Dec. 1998). Place: USA Publisher: Society for Industrial and Applied Mathematics, pp. 359–392. ISSN: 1064-8275.

[21] Dora Erdos. *Dora Erdos Homepage*. URL: https://cs-people.bu.edu/edori/code.html#Betweenness_centrality (visited on 04/22/2021).

[22] Dora Erdős. *A question about "A Divide-and-Conquer Algorithm for Betweenness Centrality"*. Feb. 2021.

[23] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. June 2014. URL: http://snap.stanford.edu/data.

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

[25] Peer Bork, Lars Juhl Jensen, and Christian von Mering. *Welcome to STRING*. URL: https://string-db.org/.

[26] George Karypis. *METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering*. Mar. 2013. URL: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview (visited on 05/05/2021).

[27] Zardosht Kasheff. *Partial Parallelization of Graph Partitioning Algorithm METIS Term Project*. en. 2004. URL: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-895-theory-of-parallel-systems-sma-5509-fall-2003/projects/kasheff.pdf (visited on 04/25/2021).

[28] Michael L. Fredman et al. "The pairing heap: A new form of self-adjusting heap". en. In: *Algorithmica* 1.1-4 (Nov. 1986), pp. 111–129. ISSN: 0178-4617, 1432-0541. DOI: 10.1007/BF01840439.

[29] Keith Schwarz. *Archive of Interesting Code - FibonacciHeap.java*. URL: https://keithschwarz.com/interesting/code/?dir=fibonacci-heap (visited on 04/25/2021).

[30] Eric D. Friedman, Rob Eden, and Jeff Randal. *GNU Trove*. URL: http://trove4j.sourceforge.net/html/overview.html (visited on 05/05/2021).

[31] Marek Kirejczyk. *Counting inversions in an array*. 2011. URL: https://stackoverflow.com/a/6424847.

[32] Dimitrios Michail et al. "JGraphT—A Java Library for Graph Data Structures and Algorithms". en. In: *ACM Transactions on Mathematical Software* 46.2 (June 2020), pp. 1–29. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3381449.

Appendix A

Parallelizability

Parallelizing the betweenness centrality algorithms was a proposed extension that I decided not to do. However, it is relevant to note on the parallelizability of the algorithms I evaluate. It is actually possible to effectively parallelize all of the algorithms I have discussed.

Apart from `Brandes++`, all of the algorithms spend the vast majority of their runtime in a loop that is essentially the following

Algorithm 3: Main loop of all `Betweenness Centrality Algorithms`

```
while ¬ termination_condition do
    x ← get_next_pivot()
    temp_centralities ← calculate(x)
    centralities ← centralities + temp_centralities
```

In all algorithms, get_next_pivot() and calculate(x) have no dependencies between iterations of the loop, and only `Bader` and `KADABRA` have a dependence in computing termination_condition. Thus, it is trivial to parallelize all of the algorithms besides `Brandes++`, `Bader`, and `KADABRA` by parallelizing iterations of this loop.

In `Bader` and `KADABRA`, the termination condition depends on the total accumulated betweenness centralities, which complicates the parallelization. However, if one can efficiently check this end condition (which does not need to be checked every iteration), 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, as well as 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 Skeleton` can be parallelized in a similar manner to `Brandes`.

Project Proposal