Computer Science Tripos - Part II - Project Proposal

Evaluating Betweenness Centrality Algorithms for Real World Datasets

Albert Kutsyy ak2149 Trinity College

Originator: Dr Timothy Griffin

March 5, 2021

Project Supervisor: Dr. Timothy Griffin

Directors of Studies: Dr. Sean Holden, Dr. Neel Krishnaswami, Prof. Frank Stajano

Project Overseers: Prof. Marcelo Fiore, Dr. Robert Mullins

1 Introduction and Description

Graphs represent entities and the relationships between them. They are used in a wide variety of applications, including biology, business, social media, and web pages [9]. Analyzing these graphs is useful for scientific purposes (disease spread, web growth) as well as for policy decisions (product recommendations, routing control). Betweeness centrality is one measure of the importance of a node in a graph and is commonly used in applications ranging from optimizing internet routing to finding potential leaders in terrorist networks [1]. It is also computationally intense to calculate — the most commonly used algorithm (`Brandes`) takes $O(nm + n^2 \log n)$ time for a weighted graph of n nodes and m edges [6].

Numerous algorithms to speed up this computation have been proposed since Brandes published his algorithm in 2001, including several for approximating rather than directly calculating betweeness centrality [5]. Given the scale of "interesting" graphs (the `wiki-meta` graph of Wikipedia edit history has 5.8 million nodes) and diversity of algorithms, it is important to use the fastest betweeness centrality algorithm which will satisfy accuracy constraints [9].

To this end, I will implement five algorithms (see Selected Algorithms) for calculating betweeness centrality, three of which are approximate algorithms. I will instrument each of these algorithms in order to obtain useful performance metrics (including time per node, total time, memory usage, and number of graph reads). For each algorithm, I will run one of the fastest published implementation on a large graph (1,000+ nodes). I will then run my own implementation on the same graph and compare the total run-times. If the times are within the same order of magnitude and the outputs are identical (or of similar accuracy for the approximate algorithms), I can conclude that my implementation is correct and has included all important optimizations. As all five algorithms will be

implemented within the same framework, their relative performance will be an accurate comparison of the algorithms themselves, not just their implementation.

After verifying my implementation of each of the algorithms, I will run them on at least five non-temporal graphs from the Stanford SNAP dataset collection, Network Repository, and STRING. Stanford SNAP is a high-quality collection of graphs from a range of fields, including online communities, location services, citations, and email communication [9]. Network Repository is a very large collection of graphs from a wider range of domains, including biology and transportation [11]. STRING is a large collection of protein networks [4].

After running each algorithm on each dataset (multiple times for the approximation algorithms as they are non-deterministic), I will evaluate their relative performance and accuracy. I plan to use the output of the `Brandes` algorithm as a ground truth for evaluating the accuracy of the approximate algorithms.

Table 1: Selected Algorithms

Algorithm	Type	Description and Reason for Including
`Brandes` [5]	Exact	`Brandes` is the most commonly used betweeness centrality algorithm [6] and has $O(nm + n^2 \log n)$ runtime. It is used as a baseline to represent the current state of the world.
`Brandes++` [6]	Exact	Erdos et al. claims that `Brandes++` achieves significantly better (up to 75x) performance than `Brandes` while still giving exact results [6]. Its performance relative to approximate algorithms has never been studied.
Geisberger et al. [8]	Approximate	The algorithm by Geisberger et al. is based off of randomly selecting source nodes ("pivots") to calculate betweeness centrality [8]. It was the most accurate approximate algorithm tested by Matta et al. [10].
Bader et al. [2]	Approximate	This algorithm uses a different sampling strategy. On a supercomputer, it had significantly better performance than any other tested algorithm [1]
`KADABRA` [3]	Approximate	This algorithm, developed by Borassi and Natale, selects random shortest paths in a complicated fashion in order to guarantee bounds on accuracy [3]. It was the fastest algorithm in tests by Matta et al. [10].

2 Starting Point

Each of the five algorithms for betweeness centrality have a published implementation. However, the implementations use different frameworks for reading the graph, and in

many cases, are in entirely separate languages. Further, most implementations of these algorithms are completely uninstrumented.

Comparisons between these algorithms are relatively rare. Bader et al. does not compare the performance of their algorithm to any other [2]. Borassi and Natale [3] compare the performance of their algorithm to the `RK`, `ABRA-Aut`, and `ABRA-1.2` algorithms. Geisberger et al. [8] and Erdős et al. [6] compare their algorithms with `Brandes`. Brandes [5], in turn, compares his algorithm to the `Floyd-Warshall` algorithm.

`Floyd-Warshall` [7] is an all-shortest-paths algorithm published in 1962 which can be modified to calculate betweeness centrality. I will not be examining it as it has $O(n^3)$ runtime for n nodes and as such cannot be run on as large graphs as the other algorithms [5]. Further, it has fallen out of use since `Brandes` was published in 2001.

Two major studies have attempted to benchmark betweeness centrality algorithms. Al-Ghamdi et al. [1] uses a supercomputer to benchmark a large number of algorithms, including those by Brandes, Bader et al. and Geisberger et al. Their work focuses primarily on the process of benchmarking rather than the results they obtain.

The other study by Matta et al. [10] attempts to evaluate algorithms in "real world" scenarios, benchmarking simply by evaluating the speed and accuracy of algorithms at performing two tasks (clustering and immunization). The algorithms they compare include those by Borassi and Natale, Brandes, and Geisberger et al.

From the Computer Science Tripos, I have knowledge and experience in Java, algorithms, and data science. I have used high-power computing servers before in my internship, but never the ones operated by the Systems Research Group. I will need to do significant reading about graphs, graph metrics, probabilistic algorithms, and code instrumentation.

3 Work to be Done

I will use the Waterfall software development model while undertaking this project. The work to be done in each stage follows.

3.1 Requirements

The requirements stage has already been completed, as it consisted of understanding the requirements and scope of the Part II project, and doing research into previous projects.

3.2 Design

The Design stage is the project proposal. Tasks to be completed are in the Implementation section and the schedule can be found in the Plan of Work section.

3.3 Implementation

Tasks

1. Verify availability of all resources.
2. Determine how to use high power computing facilities.
   Dependent on: 1.
3. Install one of the fastest known implementations for each algorithm.
4. Through experimentation, determine the maximum size of graphs I can use.
   Dependent on: 2. and one algorithm of 3.
5. Choose and download all selected graphs.
   Dependent on: 4.
6. Implement testing framework.
7. Implement all five algorithms with instrumentation.
8. Integrate algorithms into testing framework.
   Dependent on: 6. and 7.
9. Compare the performance my implementations to the fast implementation.
   Dependent on: 3. and 8.
10. Run each algorithm on each selected graph.
    Dependent on: 5. and 9.

Schedule

The schedule is outlined in the Plan of Work section.

Technical Details

I will use Java to implement the five algorithms described in Selected Algorithms and instrument them to record (at least) the total time, time per node, memory usage, and number of graph reads. Java was chosen because it is highly portable, reasonably efficient, and high level. The portability will allow me to use my laptop for testing and a high power server for actual experiments. Reasonable efficiency is desirable to reduce the time that experiments take, and the high-level features should make it easier to implement the more complicated algorithms.

Each implementation will be single threaded and run on a single core. This is for two reasons. First, some algorithms are inherently more parallelizable than others, so by using multiple cores, we no longer have a level comparison. This came up in Matta et al. where the relative ranking of `KADABRA` and `McLaughlin2014` depended on how good of a GPU they used [10]. Second, implementing these already complicated algorithms with parallelism is beyond the scope of a Part II project and better suited as an extension.

I will use the four performance metrics described previously to evaluate the performance of each algorithm. For approximation algorithms, I will also evaluate the maximum and average error in betweeness centralities across all nodes, as well as the accuracy of the overall and top 1% node rankings. I will use the results from `Brandes` as the ground truth for these measures.

3.4 Verification

I will evaluate that I have met the criteria described in the Success section. Following this, I will write my dissertation.

Further, I will evaluate the correctness of each algorithm I implement by running both my implementation and one of the fastest known implementations on the same large graph. I will also use this to evaluate that I have achieved reasonable efficiency in my implementations and have not missed major optimizations. I will do this evaluation for each algorithm within a few weeks of implementing it, before I can consider the algorithm "done".

3.5 Maintenance

Maintenance is beyond the scope of the Part II project.

4 Success

4.1 Success Criterion

This project will be successful if the following criteria are met:

- All five algorithms described in Selected Algorithms have been implemented and each meets the following criteria:
  - Is instrumented with all selected performance measures.
    - This includes, but is not limited to, time per node, total time, memory usage, and number of graph reads.
  - Gives verifiably correct results on test graphs (for exact algorithms) or within expected accuracy (for approximation algorithms).
- Each algorithm has been run on each of the five selected graphs and the instrumentation results have been collected.

4.2 Possible Extensions

Parallelization

Several of the algorithms are amenable to parallelization. One possible extension would be to modify my testing framework and implementations of the algorithms to exploit hardware parallelism. I would then be able to compare the relative performance of the algorithms in multicore environments.

Characterize relationship between graph statistics and algorithm performance

Another possible extension would be to compute various graph statistics on each of the graphs I use for testing, and attempt to determine if there is some relation between the statistics and which algorithm will perform best. Using this, I would attempt to create a method to predict which algorithm would perform best on a new graph based on a collection of graph statistics.

5 Plan of Work

Michaelmas Term

24/10 – 06/11

Verify that all resources are available. Install all necessary software and packages.

Read about instrumenting algorithms.

Continue to do background reading on analyzing graphs, specifically looking for how to reason about graph algorithms.

Implement the testing framework. This involves parsing the graph and hooks for instrumentation.

*Milestone **6 November:** decide on a set of instrumentation metrics.*
*Milestone **6 November:** Confirm access to, and log into, the high power computing server.*

07/11 – 20/11

Implement the `Brandes` algorithm. Test on a small graph, comparing output to `JGraphT`.

Learn how to run code on the high power computing server.

*Milestone **20 November:** Run my instrumented* `Brandes` implementation *on small graphs.*

21/11 – 04/12

Run my implementation of `Brandes`, as well as one of the fastest known implementation, on a large graph.

Run `Brandes` on increasingly large graphs to determine an upper bound on graph size for testing.

*Milestone **4 December:** Determined maximum size of graphs.*
*Milestone **4 December:** Completed and verified my implementation of* `Brandes`.

Christmas Vacation

04/12 – 18/12

Implement the `Brandes++` algorithm. Test on a small graph, comparing output to `JGraphT`.

Run my implementation of `Brandes++`, as well as one of the fastest known implementation, on a large graph.

*Milestone **18 December:** Completed and verified my implementation of* `Brandes++`.

08/01 – 20/12

Implement the Geisberger et al. algorithm. Test on a small graph, comparing output to `JGraphT`.

Run my implementation of `Brandes++`, as well as one of the fastest known implementation, on a large graph.

*Milestone **20 January:** Completed and verified my implementation of* Geisberger et al.

Lent Term

21/01 – 05/02

Implement the Bader et al. algorithm. Test on a small graph, comparing output to `JGraphT`.

Run my implementation of Bader et al., as well as one of the fastest known implementation, on a large graph.

Write progress report

*Milestone **04 February:** Progress report complete.*
*Milestone **05 February:** Completed and verified my implementation of* Bader et al.

06/02 – 19/02

Implement the `KADABRA` algorithm. Test on a small graph, comparing output to `JGraphT`.

Run my implementation of `KADABRA`, as well as one of the fastest known implementation, on a large graph.

Present project report to overseers.

*Milestone **16 February:** Progress report presentation given.*
*Milestone **19 February:** Completed and verified my implementation of* `KADABRA`.

20/02 – 05/03

Select 5 graphs

Run all algorithms on all five graphs, multiple times for the approximate algorithms.

*Milestone **05 March:** Have results for all executions.*

06/03 – 19/03

Evaluate results of executions.

Begin writing the dissertation.

Easter Vacation

20/03 – 27/04

Finish skeleton draft.

Incorporate feedback from supervisor.

Finish writing dissertation and proof-read.

*Milestone **25 March:** Skeleton draft submitted to supervisor.*
*Milestone **21 April:** Final dissertation draft submitted to supervisor and Director of Studies.*

Easter Term

28/04 – 14/05

Incorporate feedback from supervisor and Director of Studies.

*Milestone **14 May:** Dissertation submitted.*

---

6 Resource Declaration

Personal Resources

I will primarily use my personal laptop for writing the dissertation and development. For development, I will use an IDE such as IntelliJ IDEA. I will maintain a git repository with both the written work and source code, and will back it up to GitHub daily. I will also have OneDrive continuously back up the git repository. In case of failure I will simply switch to the MCS.

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

Computing Resources

Betweeness centrality is very computationally expensive. Matta et al. used a computer somewhat more powerful than my laptop and found that `Brandes` took over 48 hours to execute their clustering test on a 100,000 node graph [10]. If we take this as an upper bound on the time to process one graph, it could take up to 50 days to run all 25 combinations of algorithms and graphs. I have been granted permission for the following resource, which I will use to significantly speed up the computation time.

> Resource: access to the compute servers *rio*, *yellow*, and *nile*.
> Institution: Systems Research Group (`https://www.cl.cam.ac.uk/research/srg/`)
> Sponsor: Dr. Andrew Moore (`andrew.moore@cl.cam.ac.uk`)
>
> This requires setting up a Computer Laboratory account.

In case there is some problem with this resource, I will either use Amazon Web Services or Google Cloud with free credits for students, and/or restrict myself to relatively small graphs.

Datasets

I will use five different graphs from a combination of Stanford SNAP [9], Network Repository [11], and STRING [4]. In the case that one or both sites become unavailable, I will use a different network dataset collection, which there are dozens of.

References

[1] Ziyad AlGhamdi, Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. A benchmark for betweenness centrality approximation algorithms on large graphs. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, New York, NY, USA, 2017. Association for Computing Machinery.

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

[3] Michele Borassi and Emanuele Natale. KADABRA is an adaptive algorithm for betweenness via random approximation. *CoRR*, abs/1604.08553, 2016.

[4] Peer Bork, Lars Juhl Jensen, and Christian von Mering. Welcome to string.

[5] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[6] Dora Erdos, Vatche Ishakian, Azer Bestavros, and Evimaria Terzi. A divide-and-conquer algorithm for betweenness centrality, 2015.

[7] R.W. FLOYD. Algorithm 97–shortest path. *Communications of ACM*, 5:345.

[8] Robert Geisberger, Peter Sanders, and Dominik Schultes. *Better Approximation of Betweenness Centrality*, pages 90–100.

[9] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

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

[11] Ryan A. Rossi and Nesreen K. Ahmed. Network data repository: The first interactive network data repository, 2020.