Conall Moss

Implementation and Analysis of Incremental Betweenness Centrality on Large Graphs

Computer Science Tripos – Part II
Queens' College

May 10, 2024

Declaration

I, Conall Moss of Queens' 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. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: Conall Moss
Date: May 10, 2024

Acknowledgements

I would like to thank my supervisor, Timothy Griffin, for inspiring this project and helping me to develop my ideas. I would also like to thank my many friends who helped me so much by listening to my ideas and providing feedback on the typesetting of this dissertation. Finally, I would like to express my appreciation and gratitude for my brother, who has supported me on many occasions, both through the last three years and for everything else.

Proforma

Candidate number:	2420F
Project Title:	**Implementation and Analysis of Incremental Betweenness Centrality on Large Graphs**
Examination:	**Computer Science Tripos – Part II, 2024**
Word Count:	11931[1]
Code Line Count:	5520[2]
Project Originator:	The Candidate & Prof. Timothy Griffin
Project Supervisor:	Prof. Timothy Griffin

Original Aims of the Project

This project focuses on two recently published works for calculating incremental betweenness centrality: the iCentral paper [1] and Lee-BCC paper [2]. The iCentral paper also gives a brief comparison of the performance of both algorithms. Our aim was to implement these algorithms for ourselves in order to see the extent to which we can reproduce the results of this paper.

Work Completed

I created Python implementations of the two core algorithms, as well as a parallelised implementation of the iCentral paper's algorithm. I then compared the relative performance increases between my algorithms to those produced by the iCentral paper, which supported their claims. I also conducted a more in depth analysis of the results beyond the original paper's comparison, which yielded interesting new results. As a further extension, I investigated how the order edges are inserted affects the convergence of betweenness centrality, as well as how other graph factors correlate with betweenness centrality.

Special Difficulties

None.

[1]This word count was computed using `texcount`.
[2]This code line count was computed using `git ls-files | grep py | xargs cat | wc -l`

Contents

Chapter 1

Introduction

Graphs are important tools which can be used to model different types of relationships and processes, across a variety of disciplines. *Centrality measures* are often vital for the analysis of graphs, providing means to rank nodes based on chosen characteristics. A node's centrality is generally interpreted as its importance within a network. For example, a person with a high centrality value in a social network is typically considered to have more influence than others.

Betweenness centrality [3] is one of the most important centrality measures, measuring how frequently a node lies on shortest paths between other nodes [4]. This is illustrated by **Figure 1.1**, where central nodes that lie on many shortest paths have a high betweenness centrality (blue), and outer nodes that lie on few shortest paths have a low betweenness centrality (red). Betweenness centrality has been used for a variety of applications including community detection in social networks [5], urban road analysis in transport networks [6], and message routing in communication networks [7]. However, the usage in social network analysis is particularly challenging due to the size of the graphs involved.

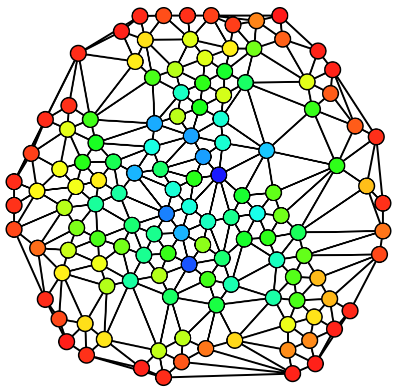


Figure 1.1: *A visual representation of betweenness-centrality. Each node is coloured based on its betweenness centrality value, from low (red) to high (blue). [8]*

Many modern graphs are inherently evolving. In a social network, people may join or leave the network or modify friends at any time, with each action affecting the graph's internal structure. Betweenness centrality is an expensive metric to compute, requiring $\mathcal{O}(|V||E|)$ time using Brandes' algorithm [9] for a graph with $|E|$ edges and $|V|$ nodes. For perspective, the Facebook social network graph has billions of nodes (users) and hundreds of billions of edges (connections) [10] and updates many times per second. It would be computationally impractical to recompute these values from scratch every time the graph changes.

Incremental betweenness centrality provides an efficient solution to this problem. By using information about the betweenness centrality of the graph before the change, we can reduce the required computation of the update.

In this dissertation we explore two recently published works on incremental betweenness centrality; the *iCentral paper* by Jamour et al. [1] and the *Lee-BCC paper* by Lee et al. [2].

We compare the methods and optimisations they use, contrast the ways they differ from each other, and analyse their performance on a range of real-world datasets. Extensionally, we also investigate factors that affect how betweenness centrality changes.

1.1 Project Summary

In our Preparation (**Chapter 2**) we formally introduce *betweenness centrality* and *incremental betweenness centrality*, and provide an overview of the two algorithms we explore in this dissertation (the iCentral algorithm and Lee-BCC algorithm).

Implementation (**Chapter 3**) **provides full explanations of how the two algorithms work**, beginning from the prerequisite knowledge they are based on. I then **provide my own analysis** of their methods, before describing how I **further developed my implementations through optimisation, parallelisation** and use of a **high-performance computing facility**. Finally, we introduce our **extensional work on factors that affect betweenness centrality**.

In our Evaluation (**Chapter 4**) we compare our implementation benchmarks for both time and space against those from the iCentral paper. The results we achieve **support the claims from the iCentral paper**, as well as **provide new observations not made by the paper**, which we explore and reason about. We also **evaluate the results from our investigation** into the factors that affect betweenness centrality, and **summarise our work against our success criteria**.

Chapter 2

Preparation

In this chapter, we discuss the background knowledge required for betweenness centrality and incremental betweenness centrality (**Section 2.1**). **Section 2.2** introduces the two algorithms we explore in this dissertation, providing an overview of what they contribute. We then outline the success criteria of the project as well as the software engineering techniques and methods used in **Section 2.3**.

2.1 Introduction to Betweenness Centrality

Graph theory is the mathematical study of graphs, while *network theory* (a subset of graph theory) instead focuses on the analysis of graphs which represent real-world systems. This is commonly done through *graph measures*: aggregations of useful graph information used to obtain meaningful results. Some examples of widely used graph measures are: *degree centrality*, which measures the number of edges connected to a node and *closeness centrality*, which measures how close a node is to all other nodes. This dissertation focuses on *betweenness centrality*.

***Betweenness centrality* measures the fraction of shortest paths between all pairs of nodes which pass through a given node.** This measure was first proposed by Anthonisse [11] and formalised by Freeman [3] as an indicator of how influential specific nodes are in a graph. For example, the highest betweenness centrality node in a communication network would likely see the greatest number of message transfers across it. Similarly, in a transportation network, the node with the highest betweenness centrality would have the highest number of passengers passing through it. In both cases, the removal of this node would cause significant disruption to message arrival times and journey delays respectively.

2.1.1 Formal Definition of Betweenness Centrality

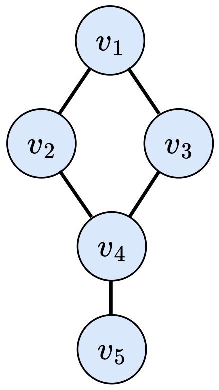
Let $G = (V, E)$ be a graph, where V is the set of nodes and $E \subseteq V \times V$ is the set of edges. Unless otherwise stated we will assume our graphs are *unweighted*, *undirected* and *connected.*

The *node betweenness centrality* ($BC_G(v)$) for a node $v \in V$ and the *edge betweenness centrality* ($BCE_G(e)$) for an edge are defined as follows:

$$BC_G(v) = \sum_{\substack{s,t \in V, \\ s \neq v \neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}} \qquad BCE_G(e) = \sum_{s,t \in V} \frac{\sigma_{st}(e)}{\sigma_{st}}$$

where:

σ_{st}	defines the number of shortest paths from node s to node t
$\sigma_{st}(v)$	defines the number of shortest paths from s to t that pass through node v
$\sigma_{st}(e)$	defines the number of shortest paths from s to t that use edge e



(s,t)	$\sigma_{s,t}$	Path	$\frac{\sigma_{s,t}(1)}{\sigma_{s,t}}$	$\frac{\sigma_{s,t}(2)}{\sigma_{s,t}}$	$\frac{\sigma_{s,t}(3)}{\sigma_{s,t}}$	$\frac{\sigma_{s,t}(4)}{\sigma_{s,t}}$	$\frac{\sigma_{s,t}(5)}{\sigma_{s,t}}$
(1,4)	2	1-2-4		0.5			
		1-3-4			0.5		
(1,5)	2	1-2-4-5		0.5		0.5	
		1-3-4-5			0.5	0.5	
(2,3)	2	2-1-3	0.5				
		2-4-3				0.5	
(2-5)	1	2-4-5			1		
(3-5)	1	3-4-5			1		
$BC_G(V) = \sum_{\substack{s,t \in V, \\ s \neq v \neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}}$:			0.5	1	1	3.5	0

Figure 2.1: *Betweenness centrality values for nodes in the above graph. 1-length paths have no contribution so have been ommitted for clarity.*

This metric is expensive to compute on large graphs. A naive implementation would find all single-source shortest paths between every pair of nodes, which would take $\mathcal{O}(|V|^3)$ time by performing a breadth-first search from every node. In Brandes' paper *A Faster Algorithm for Betweenness Centrality* [9], he was able to improve upon this bound, achieving a complexity of $\mathcal{O}(|V||E|)$. This approach utilises *node pair dependencies* and *source dependencies*– concepts which now underpin the majority of modern approaches.

Brandes' algorithm remains the fastest known algorithm for calculating *static* betweenness centrality (from scratch) without making any further assumptions about the graph's structure. However, while it provides a significant improvement upon previous methods, its computational cost can still grow too quickly for large graphs. As previously mentioned, social network graphs can contain billions of users, with multiple edges being added and removed every second. This makes real-time betweenness centrality calculations prohibitively expensive, even with modern day computation facilities.

2.1.2 Incremental Betweenness Centrality

Incremental computation is a technique used to improve the speed of repeated calculations by only recomputing outputs that depend on the changes to input data. This is particularly useful when the input is very large, but only changes by a small amount. This approach is used throughout computer science, with some common examples being: compiler design, software installation tools and spreadsheet calculators.

By applying incremental computation to the problem of efficiently computing betweenness centrality on evolving graphs, we get *incremental betweenness centrality.* "Efficient re-computation in dynamically changing networks" was described by Brandes' as a remaining challenge in the field [13].

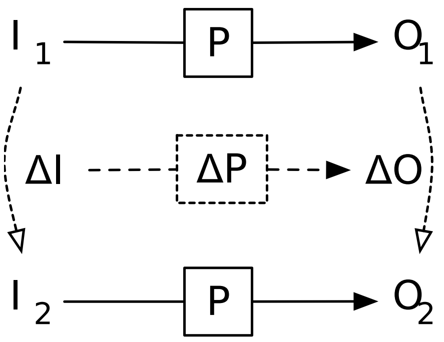


Figure 2.2: *The basic principle of incremental computing: given the previous output (O_1) and the change in the input (ΔI) we can skip an expensive full recalculation ($I_2 \to P \to O_2$) calculation by using our cheaper incremental operation ($\Delta I \to \Delta P \to \Delta O$) and applying this change to our previous state ($O_1 + \Delta O \to O_2$). [12]*

Initial approaches to this problem began by attempting to modify pre-existing incremental algorithms for all-pairs shortest path calculation, such as [14]. However, many of these algorithms assume uniqueness of shortest paths – an assumption which does not usually hold and can not be used for calculating betweenness centrality. More recent works maintain breadth-first search trees [15] or directed acyclic graphs [16] for each vertex in order to avoid the recomputation of these structures each update. However, these algorithms require a large amount of space to store these external structures between updates and incur additional costs to keep them maintained.

2.2 Algorithms Overview

This dissertation explores two modern approaches to incremental betweenness centrality which aim to minimise the update cost without maintaining any external structures.

This section introduces the two papers and provide an overview of their contributions to the field. Since the definitions and explanations of the algorithms themselves are so heavily intertwined with my own implementations, a full step-by-step explanation is detailed in **Chapter 3**.

2.2.1 The Lee-BCC Algorithm

The Lee-BCC algorithm was proposed by Lee et al. in their 2016 paper *Efficient algorithms for updating betweenness centrality in full dynamic graphs* [2]. This algorithm provides **two important novel contributions** to the field. **Firstly**, it introduced the uses of *biconnected component decomposition* to incremental betweenness centrality, as a way of minimising recomputation (explained further in **Subsection 3.1.2**). **Secondly**, it is the first work which is able to handle *fully dynamic* graphs, defined as graphs without limits on updates. This means all insertions and deletions of both edges and nodes are permitted.

The Lee-BCC algorithm is also unique in that it calculates *edge betweenness centrality* as opposed to *node betweenness centrality* (the more common and widely used variant). The motivation for this choice is that node betweenness centrality can be calculated from edge betweenness centrality (**Equation 2.1**), but the converse is non-trivial. Additionally, many community detection algorithms use edge betweenness centrality values. For example, the Newman-Girvan method [5] iteratively removes the highest betweenness centrality edge to form clusters of nodes. We can compute node betweenness centrality $BC_G(v)$ from the edge betweenness centrality $BCE_G(e)$ as follows:

$$BC_G(v) = \sum_{e \in \Gamma(v)} \frac{BCE_G(e)}{2} - (|V| - 1) \tag{2.1}$$

where $\Gamma(v)$ *is the set of edges incident on* v.

2.2.2 The iCentral Algorithm

The iCentral algorithm was created by Jamour et al. in their 2018 paper *Parallel Algorithm for Incremental Betweenness Centrality on Large Graphs* [1]. This paper contributes a novel method for efficiently identifying nodes whose shortest path trees do not change, as well as for incrementally updating the nodes whose path trees do change. The paper also addresses the modifications required to parallelise the algorithm in both shared and distributed memory systems.

However, a limitation of the iCentral algorithm is that, unlike the Lee-BCC algorithm, it is not able to handle vertex insertions or *bridge-edge* insertions (edges whose removal would increase the number of connected components). This means that the graph must be connected at all times.

Their evaluation compares their algorithm to a collection of similar algorithms for calculating incremental betweenness centrality. This included the Lee-BCC algorithm, which was one of the closest in performance. However, the results they produced showed a remarkably high comparative improvement, which I sought to verify externally. This served as the core inspiration for my dissertation: to see to what extent we can reproduce the results obtained by the paper.

2.3 Requirements Analysis

2.3.1 Success Criteria

There are six main objectives of my dissertation: the first three are implementation–based, and the last three are analysis–based.

1. **Implement the iCentral algorithm** according to its definition from the iCentral paper [1].
2. **Implement the Lee-BCC algorithm** according to its definition from the Lee-BCC paper [2].

3. **Implement the parallelisation modification** for the iCentral algorithm[1].
4. **Verify that my code is an accurate implementation** of the above algorithms, both logically and empirically:
    (a) **Logical Correctness** – ensuring my code *correctly reflects the pseudocode* defined by the papers.
    (b) **Empirical Correctness** – ensure my *algorithms work correctly* by testing against existing methods.
5. **Compute performance results** for a selection of real-world datasets.
6. **Compare my results** against those from the evaluation of the iCentral paper.

I also set out the following extensions for my project:

- **Extend the iCentral algorithm** to handle a greater variety of graphs and inputs by allowing it to handle vertex and bridge–edge insertions.
- Investigate the factors affecting the rate at which betweenness centrality values converge.
- Create benchmarks to show how representative betweenness centrality values are when only using a fraction of a dataset.
- Create a visual tool to better interpret and draw conclusions from betweenness centrality results.

2.3.2 Software Development Methodology

For the implementation of the two core algorithms, I chose to employ a waterfall [17] software engineering approach. This was well suited to my project as my requirements and functionality were well-defined from the beginning. As the waterfall model is poorly able to handle evolving requirements, I first made sure to fully understand the algorithms to determine the best way to implement them.

Once the two main algorithms were completed, I used an iterative approach to work on extending them. This included applying implementation optimisations to both algorithms, as well as the parallelisation modification of the iCentral algorithm. I found I was able to reuse my suite of *integration* and *regression* tests to quickly solve issues that arose.

2.3.3 Development Tools

After reviewing available languages and their libraries, I determined **Python** [18] would be the most appropriate programming language for this project. While the iCentral algorithm already has an existing C++ implementation, Python's readability and concision made it well–suited to the objectives of this dissertation.

[1]While the Lee-BCC algorithm could also similarly have been parallelised, this was out of the scope of my project as it was the less modern of the two approaches

Python also has an extensive collection of libraries; I used the **NetworkX** library [19] for graph creation and manipulation, and the **pytest** library [20] to create my test suite. For profiling, I used the **cProfile** library [21] to gain insight into per-function use times for optimisation, as well as the **time** and **memory_profiler** libraries [22] to compute time and memory performance results respectively for evaluation.

For parallelisation, I used the **multiprocessing** library [23]. Parallel processing works differently in Python than in many other languages due to the *Global Interpreter Lock (GIL)*: a mutex which coarsely enforces thread safety by only allowing one thread to control the Python interpreter at a time. The multiprocessing library alleviates this issue by creating entirely new processes, providing a way to achieve true process parallelism in Python.

Another tool I used extensively was the **Cambridge Research Services' High Performance Computing (HPC)** [24] facilities. This offered a way to run jobs with much higher resource limits for much longer than I could on my local machine. Execution on the HPC is handled by the **Slurm** workload manager, which allows Slurm scripts to be submitted to a job queue for execution.

Version control was done with **Git** [25], which I used to make frequent commits to store my source code. Git proved invaluable for development using the HPC. Since development was much easier on my local machine, and results collection had to be done on the remote server, I used Git extensively to transfer relevant files between the two. For extra resilience, all files on my local machine were additionally backed up with both automatic cloud syncing and frequent backups to an external drive.

2.4 Starting Point

Before starting this project, I had not worked with betweenness centrality or graph theory beyond university course material. I had used Python extensively but never any graphing, parallel computing or profiling libraries. I had also never used any high–performance computing facilities or job scheduling management tools.

In preparation for the project, I worked through the 1A Machine Learning and Real World Data topic on social networks (striked last year). I also familiarised myself with the area by reading through the Brandes' algorithm paper, iCentral paper and Lee-BCC paper, as well as some other approaches to incremental betweenness centrality.

Chapter 3

Implementation

This chapter is split into three main sections. In **Section 3.1** I provide explanations of how the two core algorithms work, beginning with my own description of their required prerequisite knowledge, and continuing onto my analysis of their methods. I then explore how I further developed my implementations through optimisation, parallelisation and high-performance computing. **Section 3.8** explores the extension work I completed on the factors which affect betweenness centrality.

Throughout this chapter I demonstrate that my Python code implementations reflect the pseudocode and concepts given by the papers, with supporting examples.

3.1 Implementation of the Core Algorithms

While iCentral and Lee-BCC employ similar principles, they are conceptualised quite differently. In my explanations I consolidate the different notation used, mostly employing the iCentral paper's notation.

For the purpose of this dissertation, we focus only on node and edge insertions on *undirected* and *unweighted* graphs. Though out of scope for this dissertation, all algorithms shown can be trivially extended to handle **deletions** (by reversing the insertion process), **weighted graphs** (using Dijkstra's algorithm instead of breadth-first search) and **directed graphs** (by following edge directions in our traversals).

3.1.1 Brandes' Algorithm

As discussed in **Section 2.1**, Brandes' algorithm is a cornerstone of betweenness centrality implementations, and remains the lowest complexity *static betweenness centrality* algorithm that relies on no further assumptions. Its concepts are used heavily in both papers.

Recall the definition of betweenness centrality from **Subsection 2.1.1**:

$$BC_G(v) = \sum_{\substack{s,t\in V,\\ s\neq v\neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}} \qquad BCE_G(e) = \sum_{s,t\in V} \frac{\sigma_{st}(e)}{\sigma_{st}} \tag{3.1}$$

Explanation: given our above definition for betweenness centrality, we can define the *pair dependency* between a source s and target t on a node v (denoted by $\delta_{st}(v)$) as a measure of the proportion of shortest paths from source node s to target node t to those which pass through v. We also define a single–sided *source dependency* of a source s on a node v, denoted by $\delta_{s\bullet}(v)$, as the sum of pair dependencies from one source to all targets on a given vertex:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} \quad \text{and} \quad \delta_{s\bullet}(v) = \sum_{\substack{t \in V, \\ t \neq v}} \delta_{st}(v) \tag{3.2}$$

This allows us to rewrite our definition of betweenness centrality into a more useful form:

$$BC_G(v) = \sum_{\substack{s \in V, \\ s \neq v}} \delta_{s\bullet}(v) \tag{3.3}$$

Additionally, for any v on a shortest path between s and t, we can define $\sigma_{st}(v)$ in terms of the paths on either side of v:

$$\sigma_{st}(v) = \sigma_{sv} \cdot \sigma_{vt} \tag{3.4}$$

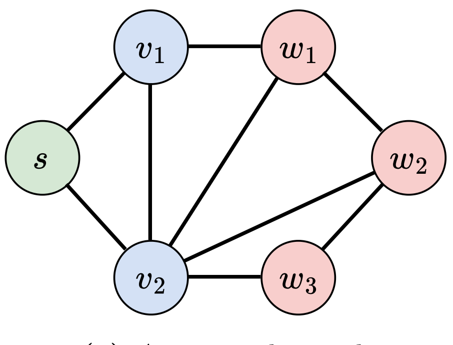
This led to Brandes' key observation: with respect to a source node s, the *source dependency* of v can be expressed in terms of the successors of v in the *breadth-first search (BFS) directed acyclic graph (DAG)* rooted at s, which represents the shortest paths from each node to s[1].

This allows us to rewrite our *source dependency* ($\delta_{s\bullet}(v)$) definition recursively:

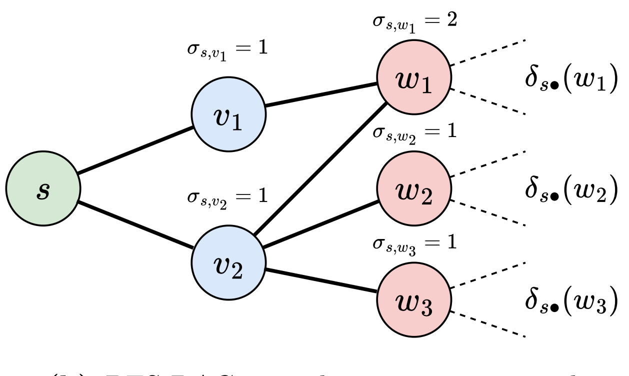
$$\delta_{s\bullet}(v) = \sum_{w:\ v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w)) \tag{3.5}$$

where $P_s(w)$ is the list of all parents (*predecessors*) of w in the *BFS DAG* out from s, such that $\{w : v \in P_s(w)\}$ is the set of successors of v.

Since the source dependency value for each node is **only dependent on its successors**, we can use a breadth-first search from s to calculate and store these successors, then reverse the breadth-first search order to apply this calculation to each node. Therefore, by only requiring a single breadth first search from each node in the graph, **we reduce our time complexity to $\mathcal{O}(|V||E|)$.**



(a) *An example graph.*



(b) *BFS DAG rooted on s in our example graph.*

Figure 3.1: *An example graph and the BFS DAG rooted at s that corresponds to it, showing σ and example δ values.*

[1] We omit a more in–depth explanation here as it is well covered by the Brandes' paper

Equation 3.6 demonstrates the source dependency calculations for $\delta_{s\bullet}(v_2)$ in **Figure 3.1**

$$\begin{aligned}\delta_{s\bullet}(v_2) &= \sum_{w\in\{w_1,w_2,w_3\}} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1+\delta_{s\bullet}(w)) \\ &= \frac{1}{2}\cdot(1+\delta_{s\bullet}(w_1)) + \frac{1}{1}\cdot(1+\delta_{s\bullet}(w_2)) + \frac{1}{1}\cdot(1+\delta_{s\bullet}(w_3)) \\ &= 2.5 + \frac{1}{2}\delta_{s\bullet}(w_1) + \delta_{s\bullet}(w_2) + \delta_{s\bullet}(w_3)\end{aligned} \tag{3.6}$$

```
def brandes(G: Graph) -> dict[Node, float]:
    BC: dict[Node, float] = defaultdict[3](float)
    for s in G.nodes: #Find and add source dependency for all nodes
        #Setup data structures required
        reverse_bfs_stack: Stack[Node] = Stack() #Stores reverse BFS ordering
        preds: dict[Node, list] = defaultdict(list) #Predecessors of each node (P)
        #Num. shortest paths from s to each node (σ)
        num_shortest_paths: dict[Node, int] = defaultdict(int)
        #Distances from s to each node
        bfs_dist: dict[Node, int] = defaultdict(lambda: -1)
        bfs_queue: Queue[Node] = Queue([s]) #main BFS queue
        source_dependency: dict[Node: float] = defaultdict(float) #Source
        ↪ dependencies (δ)
        #Initialise source node values
        num_shortest_paths[s] = 1
        bfs_dist[s] = 0

        #BFS for node information (predecessors, num. shortest paths)
        while not bfs_queue.empty():
            v = bfs_queue.dequeue() #Get next node
            reverse_bfs_stack.push(v)
            for w in G.neighbours(v):
                if bfs_dist[w] < 0: #Node found for first time?
                    bfs_queue.enqueue(w)
                    bfs_dist[w] = bfs_dist[v] + 1 #Set node distance
                #Does there exist a shortest path from s to w via v?
                if bfs_dist[w] == bfs_dist[v] + 1:
                    #Add new paths to w
                    num_shortest_paths[w] += num_shortest_paths[v]
                    preds[w].append(v) #Set as predecessor of w

        #Reverse bfs order to calculate source dependencies
        while not reverse_bfs_stack.empty():
            w = reverse_bfs_stack.pop() #Reverse bfs ordering
            for v in preds[w]: #Use Eq. 3.5
                source_dependency[v] += num_shortest_paths[v] / num_shortest_paths[w]
                ↪ * (1 + source_dependency[w])
            if w != s:
                BC[w] += source_dependency[w] / 2.0 #Summation of source dependencies
    return BC
```

Algorithm 1: Python code implementation of Brandes' algorithm. Variables have been renamed from Brandes' notation to aid readability.

[3] We use the `defaultdict` object from the `collections` module as a way to create dictionaries with default values, usually zero or an empty list. These could otherwise be implemented trivially.

The Python code shown in **Algorithm 1** provides a full implementation of Brandes' algorithm, demonstrating how we can use the recursive formula in a breadth-first search. We begin by defining our betweenness centrality dictionary (a mapping from nodes to floats) and begin iterating over all nodes to find their source dependencies. For each loop, we first create all our required structures with their initial values (Lines 5-15). For our initial breadth-first search, we find and store σ_{st} (`num_s_paths`) and $P_s(t)$ (`preds`) values (Lines 18-29). Then, by using our reverse BFS order (`reverse_bfs_stack`), we calculate source dependency values using our recursive formula (**Equation 3.5**) (Lines 32-37). From **Equation 3.3**, we can sum these source dependencies from every node (Line 35) to calculate our full betweenness centrality dictionary. By our chosen convention, we halve values to avoid counting the same path twice, since our input graph is undirected.

3.1.2 Biconnected Components Decomposition

While many previous works used *minimum union cycle* decomposition to limit the number of recomputations required each update, the papers we analyse are some of the first works to utilise *biconnected component composition.* This decomposition method is significantly quicker to compute, allowing the calculation to be done on the fly, and provides a finer decomposition. Methods using minimum union cycles had to precompute and maintain these cycles between runs as they were too expensive, adding extra overheads and requiring external structure storage.

A *biconnected graph* is a graph that will remain connected if any single vertex is removed. *Biconnected component decomposition* involves splitting a graph into a tree of maximally biconnected subgraphs called *biconnected components.* These are connected to each other by shared nodes known as *articulation points*, whose removal would increase the number of connected components of the graph.

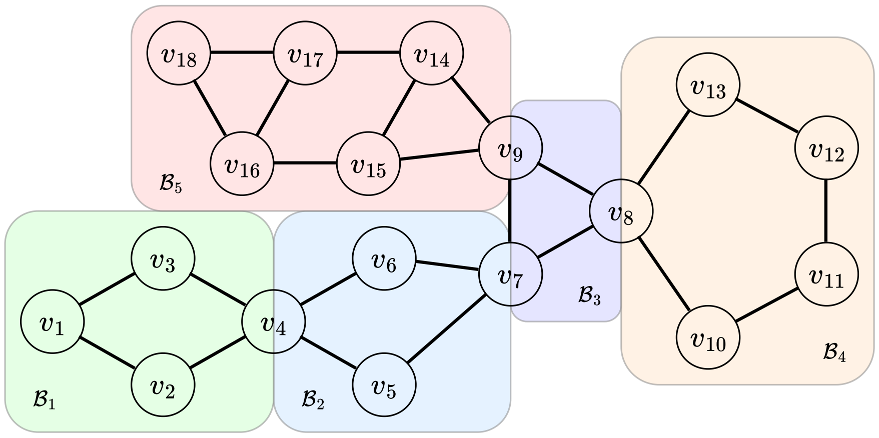


Figure 3.2: *Biconnected component decomposition of a graph.*

Figure 3.2 provides an example of this by showing a graph split into its tree of five separate biconnected components, with articulation points v_4, v_7, v_8, v_9 connecting them together.

Let us now update our graph G by inserting edge e to create G'. If both endpoints of e are in the same biconnected component in G, the decomposition of G' remains the same as G. However, if the edge connects two disjoint biconnected components $\mathcal{B}_i$ and $\mathcal{B}_j$, this will change the decomposition tree. A new biconnected component is formed by merging together $\mathcal{B}_i$, $\mathcal{B}_j$, and all components on the path between them. In either case, we refer to the (possibly newly created) biconnected component in G' that e resides in as our *affected component* $\mathcal{B}'_e$.

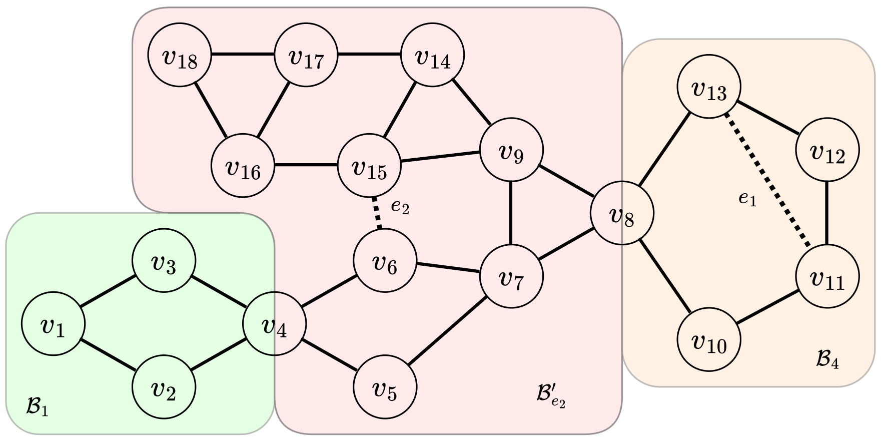


Figure 3.3: *Updated biconnected component decomposition of* **Figure 3.2** *when new edges* e_1 *and* e_2 *are added, displaying the two insertion cases.*

We can see the effect of this on **Figure 3.3**, which depicts the same graph as **Figure 3.2** but with two new edges. Since edge insertion $e_1 = (v_{11}, v_{13})$ occurs wholly within $\mathcal{B}_4$, this causes no changes to the decomposition. However, edge insertion $e_2 = (v_6, v_{15})$ joins components $\mathcal{B}_2$ and $\mathcal{B}_5$. Therefore, we join these two components together, as well as $\mathcal{B}_3$ as it lies on the path between the components, to form our new biconnected component $\mathcal{B}'_{e_2}$.

> **Theorem 1** *(from iCentral): Let G' be the new graph constructed by adding edge e to graph G. Let $\mathcal{B}'_e$ be the biconnected component of G' that edge e belongs to. For all $v \in G', v \notin \mathcal{B}'_e$ we have that $BC_{G'}(v) = BC_G(v)$.*

Theorem 1 states that betweenness centrality values do not change outside of our affected component $\mathcal{B}'_e$. This theorem is very useful; it enables us to limit the nodes that we need to calculate updates for to *at most* the nodes in the affected component We can leave the rest of the nodes as they were.

> **Lemma 1** *(from iCentral) Let G' be the new graph constructed by adding edge e to graph G. Let $\mathcal{B}'_e$ be the biconnected component of G' that edge e belongs to. For all nodes $v \in G'$, the pair dependency $\delta'_{st}(v)$ in G' either does not change or can be computed from within biconnected component $\mathcal{B}'_e$.*

Theorem 2 uses **Lemma 1** to allow us to split our edge updates down further into three types of contributions, each dependent on how many endpoints of a given path are in the affected component. Intuitively, these can be described as the *both in* contribution (A), the *one in, one out* contribution (B) and the *both out* contribution (C).

> **Theorem 2** *(adapted from iCentral paper): Let G' be the graph constructed by adding edge e to graph G, let B'_e be the biconnected component of G' that edge e belongs to (the affected component), let $\{a_1, \ldots, a_k\}$ be the set of articulation points*

of $\mathcal{B}'_e$, and let $G_1, \ldots, G_k$ be the subgraphs of G' connected to $\mathcal{B}'_e$ through each of $\{a_1, \ldots, a_k\}$ respectively. For all nodes $v \in \mathcal{B}'_e$ we have:

$$BC_G(v) = A(v) + B(v) + C(v)$$

where:

$$A(v) = \sum_{\substack{s,t \in \mathcal{B}'_e, \\ s \neq t \neq v}} \delta_{st}(v), \qquad B(v) = \sum_{\substack{s \in G_i, t \in \mathcal{B}'_e, \\ s \neq t \neq v, \\ i \in \{1,\ldots,k\}}} \delta_{st}(v), \qquad C(v) = \sum_{\substack{s \in G_i, t \in G_j, \\ s \neq t \neq v, \\ i,j \in \{1,\ldots,k\}, i \neq j}} \delta_{st}(v)$$

In this context:

Expression A represents the sum of contributions from all cases in which both nodes are in the affected component. This can be calculated using regular source dependency methods within the affected component, namely Brandes' BFS method.

Expression B represents the sum of contributions from all cases in which one endpoint (t) is in the affected component B'_e, and the other (s) is in a connected subgraph G_i connected to B'_e by a_i. Since paths from any node in G_i must pass through a_i to reach any node in B'_e, we can just use the pair dependency of a_i and t for all nodes in G_i. This allows us to aggregate each subgraph into a single equation, simplifying our equation to:

$$B(v) = \sum_{\substack{t \in \mathcal{B}'_e, \\ t \neq v, \\ i \in \{1,\ldots,k\}}} |V_{G_i}| \cdot \delta_{a_i t}(v)$$

Expression C represents the sum of contributions from all cases in which both endpoints are not in the affected component. Since all paths between any pair of nodes in G_i and G_j must pass through both a_i and a_j, we can use a similar logic to our simplification for B and deal with the entirety of both subgraphs at once. By using just the pair dependency of a_i and a_j for paths between any pair of nodes in G_i and G_j, we can aggregate both subgraphs into a single equation per pair of subgraphs, as shown:

$$C(v) = \sum_{\substack{i,j \in \{1,\ldots,k\} \\ i \neq j}} |V_{G_i}| \cdot |V_{G_j}| \cdot \delta_{a_i a_j}(v)$$

While the Type A and Type B contributions can be applied as source dependencies trivially, we break the Type C contribution down further to accurately account for all cases:

$$\delta_{G_i \bullet}(v) = \begin{cases} |V_{G_i}| \cdot |V_{G_v}| & \text{if } v \text{ is an articulation point} \\ \displaystyle\sum_{w : v \in P_{a_i}(w)} \frac{\sigma_{a_i v}}{\sigma_{a_i w}} \cdot \delta_{G_i \bullet}(w) & \text{otherwise} \end{cases}$$

where $\delta_{G_i \bullet}(v)$ represents the *external graph source dependency*: an extension of *source dependency* we use to specifically handle the aggregation of external subgraphs.

```
# Given input graph G and edge e:
all_bicons_nodes: list[set[Node]] = nx.biconnected_components(G)
affected_component: Graph = find_bicon_with_edge(all_bicons, e)

all_articulation_points: set[Node] = nx.articulation_points(G)
our_articulation_points: set[Node] =
↪  all_articulation_points.intersection(affected_component.nodes)

articulation_subgraph_size: dict[Node, int] = find_connected_subgraph_size(
                                G,
                                our_articulation_points,
                                affected_component
                            )
```

Algorithm 2: Python code for finding biconnected component decomposition information, using `NetworkX (nx)` library methods and utility functions `find_bicon_with_edge` and `find_connected_subgraph_size`.

The Python code in **Algorithm 2** shows the preparation and application steps required to extend Brandes' algorithm with the biconnected component optimisation we have described.

To calculate the required biconnected component decomposition information (**Algorithm 2**), we first use the NetworkX (`nx`) library's inbuilt `biconnected_components` function, which decomposes our graph into biconnected components (`bicons`) with Hopcroft and Tarjan's linear time algorithm [26]. We then identify our affected component with a linear search, checking for a component containing both endpoints. To get the articulation points of our component (`our_articulation_points`), we find the intersection of all the graph's articulation points, and our component's nodes. Finally, we get the size of each subgraph connected to each articulation point using a breadth-first traversal out from each articulation point, excluding edges in the affected component (`find_connected_subgraph_size`). This preparation is done before the main loop to find source dependencies, allowing us to replace the graph we use in the rest of the calculation with just our affected component graph.

The Python code in **Algorithm 3** is designed to replace the calculation of source dependencies from the Brandes' algorithm with the new optimisation (Lines 32-37 in **Algorithm 1**). We first calculate the required `source_dependency` and `external_dependency` values with our above formulas. Then, we apply each of them as required: Type A in all cases, and Type B and C only when s is an articulation point. We note that within our affected component, we perform a full recalculation of betweenness centrality to find and sum all source dependencies. We then replace the previous values with our newly calculated values.

The logic and theory described form the foundation of both algorithms. We now explore the intricacies of each algorithm that set them apart. The pseudocode given by each of the papers can be found in **Appendix A**[4].

[4]The Lee-BCC paper's pseudocode is split into two sections - the *UpdateBC* pseudocode and the *Update-Brandes* pseudocode

```python
#Extra datastructure required: External source graph dependency
external_dependency: dict[Node, float] = defaultdict(float)

while not reverse_bfs_stack.empty():
    w = reverse_bfs_stack.pop()

    #External dependencies case 1
    if (s in our_articulation_points) and (w in our_articulation_points):
        external_dependency[w] = articulation_subgraph_size[s] *
        ↪  articulation_subgraph_size[w]

    for v in preds[w]:
        #Source dependencies (as before)
        source_dependency[v] += num_shortest_paths[v] / num_shortest_paths[w]
                                * (1 + source_dependency[w])
        if (s in our_articulation_points):
            #External dependencies case 2
            external_dependency[v] += external_dependency[w]
                                * num_shortest_paths[v] / num_shortest_paths[w]

    if w != s:
        #Apply Type A contributions
        BC[w] += source_dependency[w] / 2.0

    if (s in our_articulation_points):
        #Apply Type B contributions
        BC[w] += source_dependency[w] * articulation_subgraph_size[s]
        #Apply Type C contributions
        BC[w] += external_dependency[w] / 2.0
```

Algorithm 3: Python code for calculating source dependencies using biconnected components. Replaces the source dependency summation step from the Brandes' algorithm.

3.1.3 Lee-BCC Algorithm

The most noticeable difference between the Lee-BCC algorithm compared to most other algorithms is that it calculates edge betweenness centrality instead of node betweenness centrality. Because of this, we first extend our definition of source dependency to include edges, and replace our summation of source dependencies with the following formula:

$$\begin{aligned}\delta_{s,\bullet}(e(v_1,v_2)) &= \sum_{t\in V}\delta_{st}(e(v_1,v_2))\\ &= \frac{\sigma_{s,v_1}}{\sigma_{s,v_2}}\cdot(1+\delta_{s,\bullet}(v_2))\end{aligned}$$

The Lee-BCC algorithm is also unique in that it is designed for fully dynamic graphs. This introduces two new considerations that we need to account for. The first is that the algorithm can take as input any number of nodes and edges, instead of just a single edge. As this functionality is not shared by the iCentral algorithm, I chose to keep both their interfaces consistent by modifying the Lee-BCC algorithm to only accept single edge inputs (the same as iCentral). We do still provide a method of node insertion since if either endpoint of the inserted edge does not already exist, we add it to the graph.

The second consideration we account for is that it allows bridge edge insertions, letting us handle a broader selection of inputs and graphs. However, as our previous methods of edge insertion rely on the graph being connected, we must handle this case separately. This is because bridge-edge insertions introduce entirely new paths between nodes that did not exist before, instead of just altering the pre–existing paths.

```
# Given input graph G and edge e:
for v in e: #For both edges
    if v not in G.nodes: #Check if exists in graph
        G.add_node(v) #Adding if it does not already exist

#Check if endpoints are in different subgraphs, returning those subgraphs or None
subgraphs: tuple[Graph, Graph] | None = find_bridge_subgraphs(G, e)
G.add_edge(e)
```

Algorithm 4: Python code for detecting if an inserted edge is a bridge edge, storing either the subgraphs it bridges or `None` in `subgraphs`.

As illustrated by the Python code in **Algorithm 4**, to handle these cases we first check to see if either endpoint does not already exist in the graph, adding it if not (node insertion). We then check if the inserted edge is a bridge edge by seeing if the two endpoints are not in the same connected component (with `find_bridge_subgraphs`), storing these components if so, before adding the new edge itself to the graph. We note that a newly added node will inherently mean the edge is a bridge edge, since that node must not be connected to anything else. This Python code represents lines 8–9 of the *UpdateBC* pseudocode given by the Lee-BCC algorithm.

Lemma 2 *Let $G_s = (V_s, E_s)$ and $G_t = (V_t, E_t)$ be two disconnected subgraphs of graph G with $v_s \in V_s$ and $v_t \in V_t$, such that G_s and G_t are each connected themselves. Let G' be the graph constructed by adding edge $e = (v_s, v_t)$ to G. For all nodes $v_i \in V_s$ and $v_j \in V_t$, the any path between v_i and v_j in G'_e must pass through both v_s and v_t.*

If e is a bridge edge, then by **Lemma 2** all paths between the two subgraphs that e connects must pass through both endpoints of e. This allows us to use a similar logic to our B-type and C-type equation simplifications from **Subsection 3.1.2** to aggregate the source dependencies from all nodes in a subgraph into a single equation, shown below for G_s:

$$\forall e_s \in E_s : \mathrm{BCE}_G^{\mathrm{incr}}(e) = |V_t| \cdot \delta_{s\bullet}(e_s)$$

where $\mathrm{BCE}_G^{\mathrm{incr}}$ represents the increment to edge betweenness centrality we add to its previous value.

Algorithm 5 demonstrates this process, reflecting lines 9–12 of the Lee-BCC *UpdateBC* pseudocode (**Figure A.2**). We first find the (edge) source dependencies of each endpoint of e on its respective subgraph ($\delta_{v_s\bullet}(v_i)$ and $\delta_{v_t\bullet}(v_j)$), using a Brandes' BFS modified for edges (Lines 3-4). Then, we iterate over the edges of both subgraphs, applying the new source dependency on each edge from the entirety of the other subgraph (Lines 5-8). Finally, since all paths from all node pairs between the two subgraphs must all pass through e itself, we set our new edge's centrality to the product of the subgraph sizes (Line 9).

When e is not a bridge edge, we therefore also have no node insertions. In these cases, we use the biconnected component decomposition optimisation, which combines our Python code from **Algorithm 1** and **Algorithm 3**. This reflects the *Update-Brandes* pseudocode (**Figure A.3**) from the Lee-BCC paper.

```python
if (subgraphs is not None):
    v_s, v_t = e #Splitting edge into endpoint nodes
    subgraph_s, subgraph_t = subgraphs #Splitting subgraphs out

    #Find dependencies of endpoints onto the entire subgraph
    edge_source_dependency_s = find_edge_source_dependencies(subgraph_s, v_s)
    edge_source_dependency_t = find_edge_source_dependencies(subgraph_t, v_t)
    #Apply new source dependencies from all new paths formed to edges
    for e_s in subgraph_s.edges:
        #Increment by the size of other subgraph times dependency
        BCe[e_s] += subgraph_t.num_nodes * edge_source_dependency_s[e_s]
    for e_t in subgraph_t.edges:
        BCe[e_t] += subgraph_s.num_nodes * edge_source_dependency_t[e_t]
    #Set BC of our inserted edge, since all paths pass through it
    BCe[e] = len(subgraph_s) * len(subgraph_t)
```

Algorithm 5: Python code for handling bridge edge insertion by applying dependencies from all new paths formed at once per edge.

3.1.4 iCentral Algorithm

The novelty of the iCentral algorithm is based on the following observation:

We can define betweenness centrality for our initial graph G and its updated version G' in terms of the sums of their source dependencies ($\delta_{s\bullet}(v)$ and $\delta'_{s\bullet}(v)$ respectively) like so:

$$BC_G(v) = \sum_{s \in V, s \neq v} \delta_{s\bullet}(v) \qquad BC_{G'}(v) = \sum_{s \in V, s \neq v} \delta'_{s\bullet}(v)$$

If the value of $\delta_{s\bullet}(v)$ is unaffected by our edge insertion, then $\delta_{s\bullet}(v) = \delta'_{s\bullet}(v)$. Furthermore, if $\forall v \in V : \delta_{s\bullet}(v) = \delta'_{s\bullet}(v)$, then we don't need to recalculate the source dependency $\delta'_{s\bullet}$ at all as it will have retained its previous contribution. Using this, we aim to find some $Q \subseteq V$ such that $s \in Q \implies \exists v \in Q : \delta_{s\bullet}(v) \neq \delta'_{s\bullet}(v)$. This gives us the set of nodes we need to recalculate source dependencies for: our *recalculation set*. We can then rewrite betweenness centrality in an incremental form, utilising our previous betweenness centrality values from before the edge update:

$$BC_{G'}(v) = BC_G(v) - \sum_{s \in Q, s \neq v} \delta_{s\bullet}(v) + \sum_{s \in Q, s \neq v} \delta'_{s\bullet}(v) \tag{3.7}$$

The set Q can be found by taking a breadth-first search out from both endpoints of e in graph G (before the new edge is inserted). For all nodes in V, we use our BFS to check if each node is equidistant from our two endpoints, adding them to Q if the distances are unequal, as shown below:

For graph G and insertion edge $e = (v_1, v_2)$:

$$Q = \{s \in V : d_{s,v_1} \neq d_{s,v_2}\}$$

where $d_{s,t}$ denotes the distance between s and t

If our node is equidistant from both endpoints, then the introduction of e would not cause any change to any shortest paths from that node since $d_{v_1,s} = d_{v_1,s} \implies d_{v_1,s} + 1 > d_{v_2,s}$. However, if our node is not equidistant (assuming w.l.o.g. that v_1 is further than v_2), we introduce a new path from v_1 to s through v_2 utilising our new edge. This path would be shorter or equal to our previous path v_1 to s since $d_{v_1,v_2} = 1 \land d_{v_1,s} > d_{v_2,s} \implies d_{v_1,s} \geq d_{v_2,s} + d_{v_1,v_2}$ which would change the source dependency of s and therefore requiring recalculation.

By combining this with our biconnected component decomposition optimisation, we hope to reduce the set of nodes we need to recalculate source dependencies for even further. However, offsetting this optimisation is the fact that we are now required to calculate both $\delta_{s\bullet}(v)$ and $\delta'_{s\bullet}(v)$ for every node. These can each be calculated using a Brandes' breadth-first search on G and G' respectively. These calculations must be done separately, therefore requiring two separate iterations of the expensive Brandes' breadth-first search per node.

```python
# Rename affected_component to affected_component_new
affected_component_old = affected_component_new.copy()
affected_component_old.remove_edge(e)

v1, v2 = e #Split edge into nodes
#BFS for distances to each node in the affected component without edge
distances_v1: dict[Node, int] = bfs_distances(affected_component_old, v1)
distances_v2: dict[Node, int] = bfs_distances(affected_component_old, v2)
#Initialise recalculation set Q
recalculation_set: set[Node] = set()
for s in affected_component.nodes:
    if distances[v1] != distances[v2]:
        recalculation_set.add(s)
```

Algorithm 6: Python code for creating optimised recalculation set.

Our Python code in **Algorithm 6** shows how we create our recalculation set. We create dictionaries mapping each node to its distance from each endpoint, using two breadth-first traversals. We then iterate through all nodes, adding them to our set if the distances are unequal. This step is applied after our biconnected decomposition stage (**Algorithm 2**). Then, instead of iterating over all nodes in the affected component, we now use our recalculation set (though the graph used is still the affected component). This Python code reflects lines 3–9 of the pseudocode presented by the iCentral paper (**Figure A.1**).

To calculate the $\delta_{s\bullet}(v)$ and $\delta'_{s\bullet}(v)$ values for every node, we perform a Brandes' BFS accumulation of source dependencies twice for every node. In the first search, we calculate contributions using the initial graph (`affected_component_old`) as input, subtracting them from our betweenness centrality dictionary. Conversely, in the second search, we calculate contributions using the updated graph (`affected_component_new`) as input, adding them to our betweenness centrality dictionary. In the iCentral's pseudocode, this is done by lines 11–25 and 26–40 for each, reflected by our Python code in **Algorithm 1** and **Algorithm 3**.

Bridge-edge Insertions in iCentral [Extension]

Once I had fully implemented and understood both algorithms, I decided to extend my iCentral implementation with the ability to handle bridge-edge insertions and node insertions. This was done by adapting the logic used by the Lee-BCC algorithm and applying it to the iCentral algorithm, converting the edge-centrality calculations to node-centrality calculations.

3.2 Comparison of Methods

Overall, the optimisation used by the iCentral algorithm causes it to function differently to the Lee-BCC algorithm. While the Lee-BCC algorithm uses biconnected component decomposition to allow for incremental computation of betweenness centrality over the entire graph structure, it must still conduct a full static recalculation over the affected component. On the other hand, iCentral extends this further to perform *incremental computation within the affected component* by finding the overall change to the component as a result of the edge update.

Interestingly, neither algorithm uses previous betweenness centrality values during their computations, only updating the mapping in some way. This means that if we provide as an input to each algorithm a dictionary that maps every node to zero, as opposed to to their current betweenness centrality value, the calculations done by the algorithms will remain the same. In the Lee-BCC algorithm, this will return the new centrality values to update our mapping with, and in the iCentral algorithm, this will return the relative change to add on to each value. This is very useful for calculating performance results on large graphs, as we do not need to calculate the previous state of betweenness centrality before each trial update.

3.2.1 Complexity Analysis

Let V and E be the sets of nodes and edges in the input graph G. Excluding Lee-BCC's bridge edge insertion case, both algorithms consist of a preparation stage and a calculation stage[5].

The preparation stage of both algorithms consists of first decomposing the graph into its biconnected components and then calculating the size of the subgraphs connected by each articulation point of the affected component. These can be done using Hopcroft and Tarjan's linear time algorithm and simple graph traversals respectively, therefore requiring $\mathcal{O}(|V|+|E|)$ time and space. iCentral also conducts further breadth-first traversals within the affected component in order to find the recalculation set, but this does not change the bounding complexity.

In the calculation stage, we then iterate over all nodes in the recalculation set, which has size bounded by $|V|$. Each iteration per node is dominated by the (Brandes) breadth first traversal over our nodes, requiring $\mathcal{O}(|V| + |E|)$ time and space, as the rest of each iteration is just applying these values to our betweenness centrality dictionary. This means the complexity of the calculation stage is $\mathcal{O}(|V||E|)$ time and $\mathcal{O}(|V| + |E|)$ space (since the space can be freed after each iteration).

In summary, both algorithms therefore have the same asymptotic complexity of $\mathcal{O}(|V||E|)$ time and $\mathcal{O}(|V| + |E|)$ space, which is the same as Brandes' algorithm. However, while asymptotic worst–case complexity analysis is useful for finding the dominant complexity terms, it hides constants and obscures the practical optimisations we would see with real-world applications. We therefore re–evaluate our algorithms using *practical time complexities* that try to use closer bounds on the data processed at each stage, and the steps taken by the algorithms.

While the practical complexity of the preparation stage of both algorithms remains the same, the complexity of the calculation stages can be more closely defined. Let $\mathcal{B}'_e = (V_{\mathcal{B}'_e}, E_{\mathcal{B}'_e})$

[5]iCentral duplicates this calculation stage: once for decrements using the old affected component, then again for increments using the new affected component

be the biconnected component in G' that edge e belongs to, and Q be the recalculation set from iCentral. In the Lee-BCC and iCentral algorithms respectively, the calculation stage's complexity consists of $|V_{\mathcal{B}_e}|$ (the affected component) and $|Q|$ iterations (the recalculation set) of a $\mathcal{O}(|E_{\mathcal{B}_e}|)$ complexity loop (the BFS). Since $|Q| \leq |V_{\mathcal{B}_e}| \leq |V|$ and $|E_{\mathcal{B}_e}| \leq |E|$, we can achieve practical time complexities of $\mathcal{O}(|V_{\mathcal{B}_e}||E_{\mathcal{B}_e}| + |V| + |E|$ and $\mathcal{O}(2 \cdot |Q||E_{\mathcal{B}_e}| + |V| + |E|)$ for the Lee-BCC and iCentral algorithms each. We include the constant multiple of 2 in the iCentral's analysis to represent the fact that we duplicate the calculation stage per node relative to the Lee-BCC algorithm. Since the space complexity of both algorithms is derived from the preparation stage, this remains the same as before.

In practice, we hope the size of our practical bounds is much smaller than the theoretical bounds, but this is dependent on the graph structure and the update edge. For example, looking at the *email-EuAll* dataset[6], we see its largest biconnected component has less than 50% of the nodes and 20% of the edges of its largest connected component. This means that for edge insertions in the largest biconnected component, we could expect to see a $10x$ speedup with the Lee-BCC algorithm compared to our worst case analysis.

For parallelisation of the iCentral algorithm, we see that each of our $|Q|$ iterations are independent of one another. With T processing units, we can therefore achieve an asymptotic complexity of $\mathcal{O}(\frac{|V|}{|T|}|E| + |V| + |E|)$, and a practical complexity of $\mathcal{O}(2 \cdot \frac{|Q|}{|T|}|E_{\mathcal{B}_e}| + |V| + |E|)$, at the cost of $\mathcal{O}(T(|V| + |E|))$ space as each unit requires its own storage space.

3.3 Optimisations

Upon initially writing my implementations, I wanted to make sure that I had taken all reasonable precautions to ensure my algorithms were not slowed down by any unfair implementation factors, and that a running time comparison between them would be objective and unbiased. I also realised that my implementations were several orders of magnitude slower than the results obtained by the paper – higher than the difference I was expecting from Python versus C++. These led me to start optimising both of my implementations to ensure the only timing constraints were from Python and the algorithm itself.

The first step was to refine my data structures to best represent their use cases. An example of this was swapping lists for dedicated *double-ended queues* when they were used as stack and queue objects, as these have $\mathcal{O}(1)$ insert and access at both ends (unlike Python lists).

As this didn't make a significant difference, I used a profiler to gain insight into how the execution time was being spent at a function level. The `cProfile` Python library allowed me to view timing information about every function called over a given block of code including the number of calls and time spent in each function. This showed me where the largest amounts of time were being spent, allowing me to eliminate them.

Upon analysing these results, the first thing I noticed was that accessing and iterating over the neighbours of a node in a graph was more expensive than I had expected, especially when using views of graphs. To solve this, I instead manually stored the dictionary-of-dictionaries adjacency matrices of the required graphs instead of the graphs themselves, as they incurred much lower overhead. This change had a large performance increase on both algorithms.

[6]Full information on datasets in **Section 4.2**

Another optimisation issue I had was that since the Lee-BCC algorithm has many dictionaries keyed by undirected edges, we had to ensure the structure treated edges (a, b) and (b, a) the same. Initially, I tried more elegant solutions to normalise edges, like subclassing a dictionary to overwrite the `get` and `set` methods, in order to enforce a total order of $a < b$ on every edge. I also tried creating a custom data class to enforce normalisation, but since indexing by edge was such a frequent operation, these solutions added surprisingly high amounts of overhead. This is because Python has a relatively high function call overhead. The most efficient method was to simply inline our function into every use case:

```python
# Before - (1,2) != (2,1):
BCe[(v_s,v_t)] += edge_increase
#Tried but slow:
BCe[normalise(e)] += edge_increase
# After - quicker:
BCe[(v_s,v_t) if v_s < v_t else (v_t, v_s)] += edge_increase
```

Another avenue I explored was to use an alternate implementation of Python itself. Python is a language with many features that make it great for readability and extensibility but also runs much slower than many other languages (100–1000x slower than C++, the language used by the iCentral paper). While the standard reference implementation, *CPython*, directly compiles Python code to bytecode to be interpreted, *PyPy* [27] is a drop-in replaceable alternative which uses a *just-in-time* compiler to trace code during execution and optimise *hot-code* to machine code dynamically to boost performance at runtime. This alone was able to reduce execution time by x2-3 compared to CPython, likely due to the majority of runtime coming from a collection of relatively short loops that it was able to optimise well.

The overall benefits of the optimisation steps I took are two-fold. I was able to increase the performance of both algorithms by a factor of around x20. The limiting factor of both algorithms is now majoritively due to standard Python object methods of that can not be reduced further. I can also say that I have taken all reasonable steps to ensure the comparison between my algorithms implementations is fair, and there are no additional implementation overheads which disproportionately affect either algorithm.

3.4 Multiprocessing

As described in **Section 3.2**, the expensive part of the iCentral algorithm (the *calculation stage*) consists of many independent calculations of source dependencies, which therefore can efficiently be executed in parallel. In their paper, they describe some of the modifications necessary to run the iCentral algorithm on a shared memory system, which I used to create a parallelised implementation of the algorithm.

I began by experimenting with some of the useful structures the `multiprocessing` module provides. The first option I used was to create a *Pool* object, which controls a collection of *workers* (processes for executing tasks in parallel). I then mapped the source dependency function over all recalculation nodes. This technique assigns each call of the function to an available thread in the pool, returning the final object on completion. I could then iterate through this to sum the dependencies.

However, the result of the above approach is an iterable storing each return value, which can only be accessed once all work in the pool has finished being processed. Since some executions

```python
process_recalc_nodes: list[set[Nodes]] = distribute(recalculation_set, num_procs)
#multiprocessing library alias to mp
result_queue: mp.Queue = mp.Queue()
resources = (...) # Tuple of required resources for each function. call

workers = [] #Store workers
for recalc_nodes in process_recalc_nodes: #Iterate through node allocations
    #Create processes to execute worker function on node allocation
    p: mp.Process = mp.Process(
                    target=worker,
                    args=(recalc_nodes, result_queue, resources)
                    )
    p.start() #Start parallel process
    workers.append(p)

for _ in range(num_processes):
    #Expect a result from all processes
    bc_update = result_queue.get(block=True) #Wait for results
    for node, val in bc_update.items():
        #Add results to the main BC dictionary
        BC[node] += v

return BC
#mutliprocessing handles the automatic joining of processes
```

```python
#Worker function to handle local aggregation of results
def worker(recalc_nodes: list[Node], res_queue: mp.Queue, resources) -> None:
    bc_update = defaultdict(float)
    for s in recalc_nodes:
        #Each call adds results to local dictionary
        bc_upd = calculate_source_dependencies(s, bc_upd, resources)
    #Return to queue when all nodes processes
    result_queue.put(bc_update)
```

Algorithm 7: Python code showing parallelisation of iCentral algorithm, using `multiprocessing` processes.

were mapped over hundreds of thousands of nodes, and each returned value was a dictionary of similar size, this quickly became prohibitively memory–consuming.

To solve this issue, I instead tried giving each thread direct access to the dictionary to apply its results. I did this by using a `multiprocessing.manager`, which stores the dictionary in shared memory and handles access using proxies. However, this was not thread-safe and since threads were trying to perform increment operations on it, extra mutexes were required for safe access. These mutexes, as well as access through the manager itself, added considerable overheads, making this solution impractical.

My next solution was to use memory safe queues and handle the processes myself. The main thread would spawn in `|cores|-1` processes, which would each takes nodes from a shared input queue, calculate the source dependency for it, and place its return value onto an output queue. The main thread would then aggregate these results while waiting for other threads to finish. While this worked significantly better than previous methods, large overheads in the `get` method of shared memory queues made this process slow, and led me to my final architecture.

I instead chose to pre–allocate each process an equal share of the nodes, which they would process to update their own local dictionary, only returning this dictionary to the main thread once their share of nodes had been processed. Across the methods tried, this proved by far to incur the least additional overheads, with my Python code demonstrating the process in **Algorithm 7**.

3.5 High Performance Computing Facility

In order to get results for the algorithms on large graphs, I used the *Cambridge Research Centre's High Performance Computing facility* (HPC). This allowed me to run my CPU intensive tasks for much longer than I could on my local machine, with significantly more resources available. While my local machine has 16GB of RAM and 16 cores, the HPC facility provided me with 76 cores and 512GB of RAM per job, and allowed me to submit multiple jobs to run simultaneously.

To compute performance results across all datasets for the three algorithms, I created a slurm script for each dataset and algorithm. Each script submits a job to execute my `run_dataset.py` file with the required arguments for the given trial. This setup provided a unified way of ensuring each trial was conducted in the same manner, making it very easy to alter all tests at once for each new trial.

3.6 Testing

To ensure my implementations were empirically correct, I used the `pytest` framework to build a suite of tests. I primarily tested for holistic correctness using the `NetworkX` library's `betweenness_centrality` function as my ground truth. I did this by generating a collection of example graphs, each progressively larger and some specifically chosen to test specific graph features and edge cases. Once my implementations were complete and working, I extended this suite with regression testing for useful functions, which allowed me to optimise and parallelise my code more easily since I could quickly locate where bugs were introduced.

3.7 Repository Overview

The main algorithms of the project were set up as a Python module, allowing anyone to easily run and use my code. The followingtree provides a structural overview of my repository:

```
Incremental-Betweenness-Centrality/
├── src/
│   ├── utils/
│   │   ├── bfs_utils.py
│   │   ├── component.utils.py
│   │   ├── dependency_utils.py
│   │   └── general_utils.py
│   ├── iCentral.py
│   ├── iCentral_p.py
│   └── LeeBCC.py
├── tests/
│   └── ...
├── datasets/
│   └── ...
├── hpc/
│   ├── result_data/
│   │   └── ...
│   ├── slurm/
│   │   ├── slurm_results/
│   │   └── slurm_scripts/
│   └── run_dataset.py
├── extension/
│   └── ...
└── dev/
    └── ...
```

`src/` stores the main implementation code, primarily within the three `iCentral.py`, `LeeBCC.py` and `iCentral_p.py` files.

`src/utils/` stores the auxiliary files containing the utility function definitions used by the main implementations.

`tests/` stores the code used for testing my implementations, as described in **Section 3.6**.

`datasets/` stores all the datasets used to run the algorithms. These were downloaded from KONECT [28]. Further details are in **Section 4.2**.

`hpc/` stores all files related to using the RCS HPC facilities. The `slurm/slurm_scripts/` subdirectory stores all of the bash scripts that are used to run each trial, storing results in the `slurm/slurm_results/` subdirectory. I would then copy those out to `result_data/`, where I could analyse the data.

`extension/` stores all files related to my extension work on the factors that affect the convergence of betweenness centrality.

`dev/` stores all old and miscellaneous files which I used during the development of the project.

3.8 Factors Affecting the Convergence of Betweenness Centrality [Extension]

Beyond the scope of my main project, I also investigated how the order in which edges are inserted into a graph affects the convergence of betweenness centrality. We measure this by comparing the betweenness centrality of all nodes after each edge insertion to their final values once the graph is complete. I also explore how other graph properties correlate with betweenness centrality, which is further detailed in **Subsection 4.4.1**.

Our motivation for investigating this is to try to discover useful indicators that can be used to tell us which edges will likely have a large effect on the betweenness centrality of a graph. This information would prove useful if we have some control over the next edge(s) to insert, or if we wanted to try to get the most representative centrality values with the least edge insertions.

To explore this, we investigate different metrics to order the insertion of edges, to create edge–lists. We then add the edges in our chosen order to the graph, recording the betweenness centrality values after each insertion. We then compare how these values differ from the final graph's betweenness centrality to generate a *loss* value after each insertion, which we can plot to visually see each edge–list's *convergence*, quick we define as how the loss tends to its final state. **Algorithm 8** demonstrates the Python code used to test this.

```python
def get_edgelist_loss(G_final: Graph, edgelist: List[Edge]) -> dict:
    final_bc = get_betweenness(G_final)
    G: Graph = nx.Graph() #Start with empty graph

    #Tracked values:
    losses = []

    for e in edgelist:
        G.add_edge(e) #Insert edge
        bc: dict[Node, float] = get_betweenness(G) #Get current graph betweenness
        ↪  centrality
        loss: float = get_dict_loss(bc, final_bc) #Get loss against final centrality
        losses.append(loss)

    return losses
```

Algorithm 8: Python code showing our process of calculating our loss values per edge of an inputted edge–list.

While computing the betweenness centralities for a given edge list was easy to implement, I quickly found that formalising a definition of *loss* between two betweenness centrality dictionaries was difficult. This is because betweenness centrality is a mapping from nodes to values, and quantifying how close one mapping is to another is non-trivial. I concluded the best way to approach this would be to take the sum of point-wise differences for each node, defaulting values to zero if the node entry doesn't exist (as shown in **Algorithm 9**). I experimented with other loss functions but these added extra undesirable behaviour. For example, using quadratic loss penalised an overshooting loss value much more than an undershooting loss value. In the end, I concluded absolute difference to be the most representative measure.

Another issue I found was in normalising betweenness centrality dictionaries. Since betweenness centrality values scale quadratically with the number of nodes, if normalisation was not applied

```python

def get_dict_loss(d1: Dict[Node, float], d2: Dict[Node, float]) -> float:
loss: float = 0.0
nodes: set[Node] = set(d1.keys()).union(set(d2.keys())) #Get all nodes from either
↪   input dictionary
for node in nodes:
    loss += loss_fn(d1.get(node, 0), d2.get(node, 0)) #Accumulate loss values
    #dict.get(node, 0) gets the dictionary's value of the node, or 0 if it does not
    ↪   exist
return loss

def loss_fn(v1: float, v2: float) -> float:
    return abs(v1-v2) #Chosen loss function: absolute difference
```

Algorithm 9: Python code representing chosen method to calculate the loss between two dictionaries of betweenness centrality.

then our initial values would be insignificantly small against the final values, and the 'best' order would always be the one that added new nodes the fastest. This is not ideal as it introduces other biases to our data. After trialling a few alternatives, I found the best approach was to normalise each value by the number of edges in the graph (**Algorithm 10**). This provides a fair scaling since each iteration uniformly increases the edge count.

```python
def get_betweenness(G: Graph) -> dict[Node, float]:
    bc = dict(nx.betweenness_centrality(G, normalised=False)) #By default NetworkX
    ↪   applies its own normalisation
    edge_count = max(G.number_of_edges(), 1) #to avoid division by zero
    for node, val in bc.items():
        bc[k] = v / edge_count
    return bc
```

Algorithm 10: Python code representing chosen method of normalising each betweenness centrality dictionary.

As this investigation was data and results driven, we explore our results in **Subsection 4.4.1**.

Chapter 4

Evaluation

In our evaluation, we compare our implementation benchmarks (for time and space) against those from the iCentral paper. We observe new results not mentioned in their paper, which we provide reasoning and supporting evidence for. We also show the results of our analysis into the factors that affect betweenness centrality. We then evaluate our work against the original success criteria.

4.1 Evaluation Methods

To calculate the performance results for my implementations, we used a variety of real–world datasets to analyse how the algorithms perform on different graph structures and sizes. Full details of the graphs used are provided in **Section 4.2**.

For each dataset, we convert it into a *simple, unweighted, undirected* graph by removing weight and direction information from edges, as well as removing any self–loops should they exist. For consistency, we only will use the largest connected component of each graph for our tests.

To conduct each test, we randomly pick and remove one existing edge from a copy of our graph, ensuring it remains connected. We then pass this modified graph and our chosen edge into the algorithm we are testing, and let it calculate the incremental betweenness centrality, recording our statistics for each run.

To conduct the tests, I used the Cambridge Research Computing Service's High Performance Computing facility, previously explained in **Section 3.5**. As I was constrained to a maximum of 12 hours of runtime per job by the HPC, the number of data points that could be collected varied. For our small graphs, we collected data for the first 100 edge insertions for the three algorithms. For our large graphs (where edge insertions took significantly longer) we collected as many data points as possible over 24 hours (across two runs) in order to get a representative sample of data points. We ensured the same edge insertions and number of data points were analysed for each algorithm.

Our test differs from that in the iCentral paper for two main reasons. Firstly, we record maximum memory requirements for each run, which the paper did not do for their iCentral vs. Lee-BCC comparison test, to see if there are any significant differences. Additionally, since they did not have access to an implementation of the Lee-BCC algorithm, they limited their tests to only within the largest biconnected component. We expand this to the largest connected component (it would not make sense to expand this to the full graph as betweenness centrality values are calculated independently across disconnected subgraphs).

4.2 Datasets

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V_{\mathcal{B}}\rvert$	$\lvert E_{\mathcal{B}}\rvert$	$\lvert V_{\mathcal{B}}\rvert/\lvert V\rvert$	**Deg.**	**Dpts**
email-EuAll	224,832	339,925	36,106	151,162	16.1%	2.8	14
linux	30,817	213,208	27,876	210,197	90.5%	13.9	19
topology	34,761	107,720	23,592	96,497	67.9%	6.2	46
sx-mathoverflow	24,668	187,939	18,627	181,884	75.5%	16.1	34
slashdot-threads	51,083	116,573	19,506	84,950	38.2%	4.6	39
chess	7,115	55,779	6,250	54,884	87.8%	15.3	100
elec	7,066	100,667	4,786	98,387	67.7%	28.3	100
wikispeedia	4,179	50,500	4,105	50,426	98.2%	24.2	100

Table 4.1: *Graph Datasets used in our experiments.*

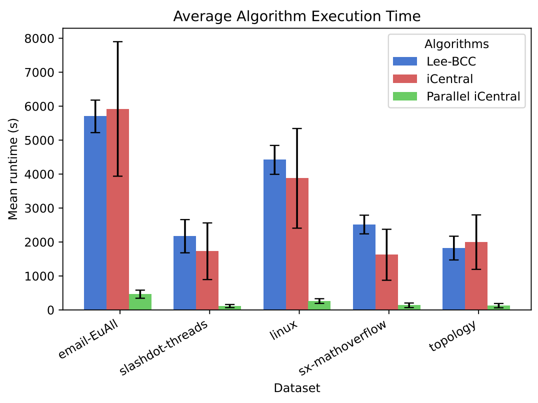
The datasets we used (downloaded from KONECT [28]) are shown in **Table 4.1**. For each graph we show:

- $|V|$: the number of nodes in the graph.
- $|E|$: the number of edges in the graph.
- $|V_{\mathcal{B}}|$: the number of nodes in the largest biconnected component.
- $|E_{\mathcal{B}}|$: the number of edges in the largest biconnected component.
- $|V_{\mathcal{B}}|/|V|$: the percentage of nodes in the biconnected component.
- Average Degree (Deg.): the average degree of nodes in the graph.
- Datapoints (Dpts): the number of datapoints collected for analysis.

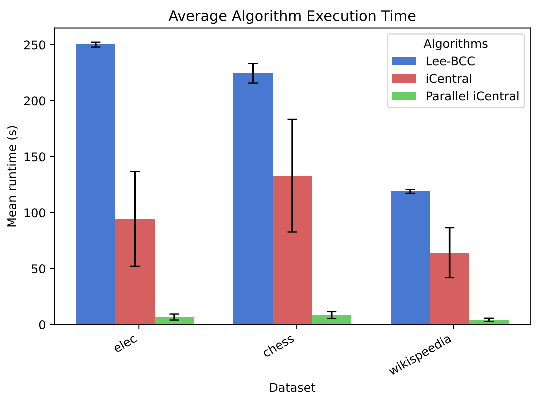
We also provide a brief description of each dataset below:

- **email-EuAll**: email communication network between an EU institution's researchers.
- **linux**: Linux source code file inclusions network
- **topology**: connection network between autonomous systems of the internet.
- **sx-mathoverflow**: user interactions from StackExchange site *MathOverflow*.
- **chess**: Chess game network between players.
- **elec**: Vote network of English Wikipedia users in admin elections.
- **slashdot-threads**: Reply network of technology website *Slashdot*.
- **wikispeedia**: Paths from *Wikispeedia* game (users tasked to get from one Wikipedia article to another using only click links).

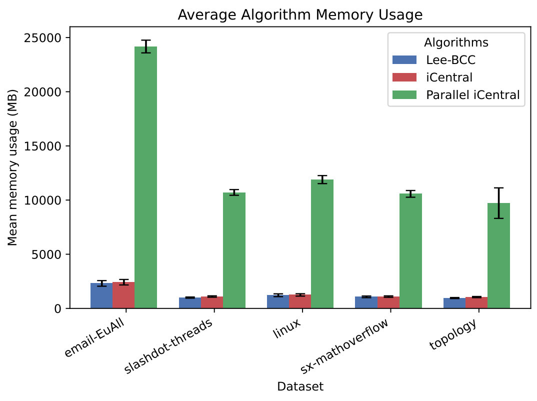
We also split our graphs into the categories of *large graphs* (email-EuAll, linux, topology, sx-mathoverflow) and *small graphs* (chess, elec, wikispeedia) depending on if they have more or less than 10,000 nodes.



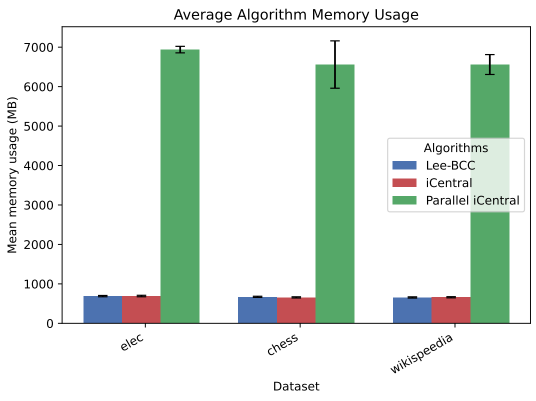
(a) *Average algorithm running times on the large graphs.*



(b) *Average algorithm running times on the small graphs.*



(c) *Average algorithm memory usages on the large graphs.*



(d) *Average algorithm memory usages on the small graphs.*

Figure 4.1: *Raw results for both running time and memory usage for all algorithms on all datasets.*

4.3 Results

Here we analyse how the iCentral algorithm compares to the Lee-BCC algorithm in **Subsection 4.3.1**, as well as how well our algorithm parallelises in **Subsection 4.3.2**. For some of these comparisons, we scale our results to show how the algorithms perform relative to each other. We additionally present the raw results for both run time and memory usage in **Figure 4.1**[1]. Since our implementations are written in Python, and the implementations used in the iCentral paper are written in C++, it would be biased to compare our results directly to the paper. Therefore, we only concentrate on the *relative* performance differences between the algorithms.

[1]Error bars shown in diagrams represent standard deviation of results.

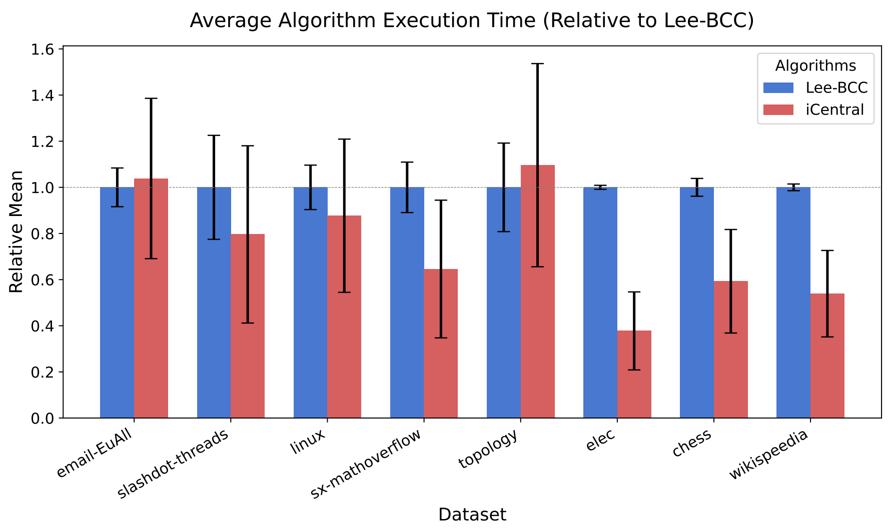


Figure 4.2: *Average Lee-BCC vs. iCentral run times, relative to Lee-BCC.*

4.3.1 iCentral vs. Lee-BCC Comparison

Figure 4.2 shows the relative performance of the iCentral algorithm versus the Lee-BCC algorithm. On average across the datasets, the iCentral algorithm is **1.5x** times faster than the Lee-BCC algorithm. While this is not as much as the 2.21x speed-up the iCentral paper claims, we do reach a maximum of **2.6x** faster on the *elec* dataset. Overall, we provide supporting evidence for the paper's original claim that the **iCentral algorithm has significant performance improvements over the Lee-BCC algorithm.**

An interesting observation however is that while the iCentral algorithm performs better than the Lee-BCC algorithm in general, **Figure 4.2** shows that it performs slightly worse on two of our datasets (*email-EuAll* and *topology*). We also see it has a much higher standard deviation than the Lee-BCC algorithm's results. This is a new observation that we make as standard deviation was not mentioned or analysed by the iCentral paper.

Investigating this further, we notice that these deviations are directly caused by variations in the recalculation set size, as this determines how many iterations are required. Across trials on the same graph, we see that the size of the recalculation set is strongly proportional ($r = 0.999$) to the time taken for the algorithm to finish, which is exemplified by **Figure 4.3**, showing a strong linear correlation between running time and recalculation size. This makes logical sense from our understanding of the algorithm.

Since the Lee-BCC algorithm always includes all the nodes in the affected component as its recalculation set, all edges in the same biconnected component will have the same recalculation set, resulting in a more consistent running time. On the other hand, the iCentral algorithm defines its recalculation set as the set of nodes non-equidistant from the two endpoints of the inserted edge, which greats varies depending on the edge. Because of this optimisation, the iCentral algorithm must perform our expensive breadth first search twice per node in the set, meaning it will perform twice as slow as Lee-BCC when their recalculation sets are the same – the whole affected component. However, when the recalculation set is small, significantly less work is required. This supports our practical complexity analysis from **Subsection 3.2.1**.

This variability causes the iCentral algorithm to have a much higher standard deviation

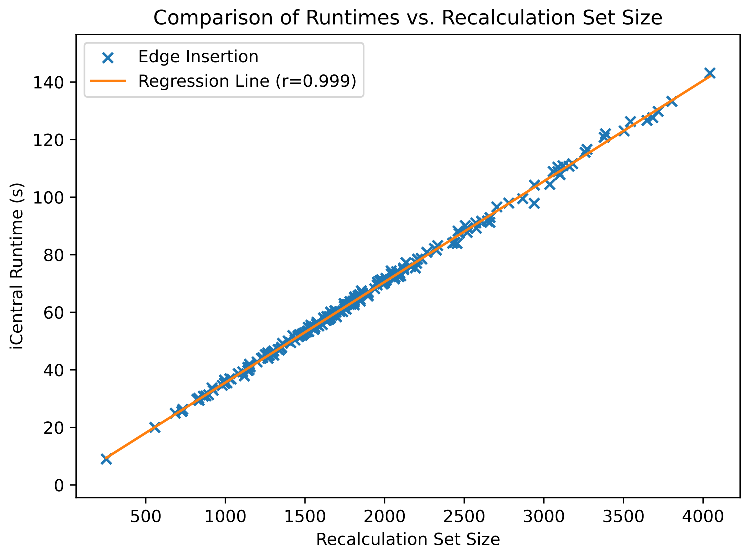


Figure 4.3: *iCentral run time vs. recalculation set size, per edge insertion on the wikispeedia dataset.*

than the Lee-BCC algorithm. **This makes the iCentral algorithm's running time less predictable, and highly dependent on the inserted edge, while the Lee-BCC algorithm is slower but more consistent overall.**

It is this variability that causes the significantly higher standard deviation in the running time, thus meaning that while the iCentral algorithm performs much better than the Lee-BCC algorithm **it is much more sensitive to the input edge used.**

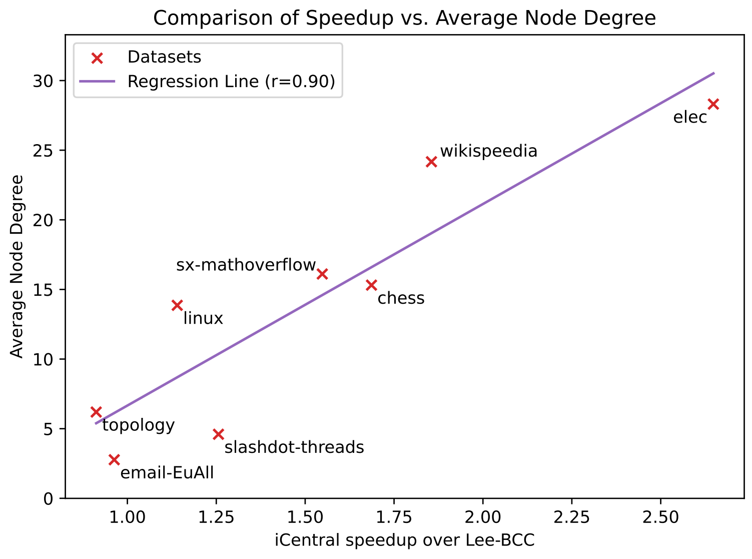


Figure 4.4: *Relative speedup of iCental vs. Lee-BCC against average node degree, per dataset.*

Another interesting observation we make is how the relative performance between the iCentral and Lee-BCC algorithms change as the average node degree increases (as depicted in **Figure 4.4**). We theorise that a higher average node degree means more paths are less likely to be affected by our inserted edge. This causes the set of non-equidistant nodes to be smaller, generating a smaller recalculation set with our iCentral optimisation, leading to faster execution times.

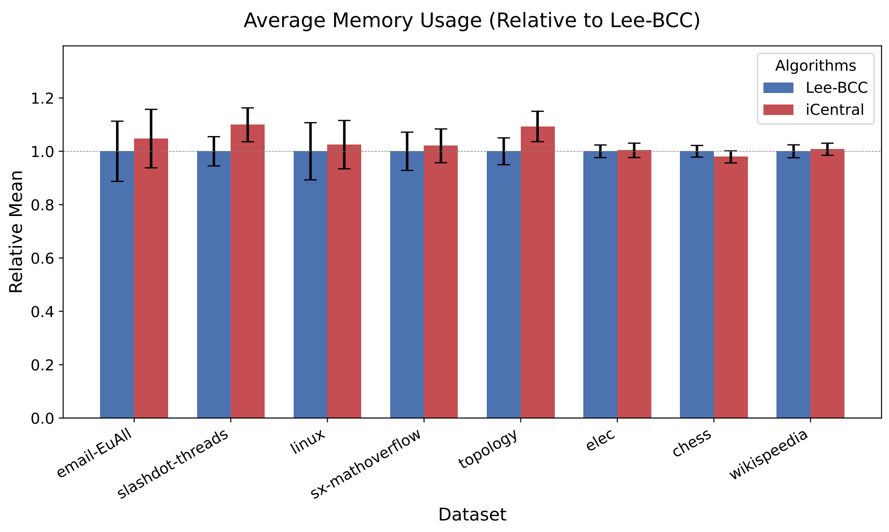


Figure 4.5: *Average Lee-BCC vs. iCentral memory usages, relative to Lee-BCC.*

This could be used to determine cases where the Lee-BCC algorithm should be used in favour of the iCentral algorithm; for example, a low average node degree in a graph would suggest a much slower runtime of iCentral. Since this metric is usually dependent on the underlying data a given network represents and is very quick to calculate, this could be used as a good indicator of when the Lee-BCC algorithm may be a better option to use. Further work could explore the usage of this observation as a heuristic; if this observation holds across a larger sample of inputs, a hybrid algorithm could dynamically choose which algorithm to use depending on the average node degree.

Memory Requirements. We additionally look at the memory requirements of the two algorithms (depicted in **Figure 4.5**). While the Lee-BCC algorithm performs marginally better across almost all of the datasets, the difference is very small and not significantly consequential. This is likely caused by the extra preparation steps iCentral takes in order to generate its recalculation set. These results support our space complexity analysis of the algorithms.

4.3.2 iCentral vs. Parallel iCentral

In **Figure 4.6**, we compare our sequential implementation of the iCentral algorithm with our parallelised implementation (running on 20 cores). On average across our datasets, our parallelised approach shows a 14x speedup. This supports the iCentral paper, which achieved a 10x speedup in their similar experiment, showing that the algorithm can be effectively parallelised for significant performance gains. This is reinforced in **Figure 4.7** where we see the algorithm scales well over a variety of numbers of cores. For each case, we record the relative speedup against our base case run time of 2 cores (one main process and one worker process). This shows that the scalability of our algorithm extends beyond our previous 20 core case.

Memory Requirements. **Figure 4.8** shows that the parallelised implementation of the iCentral algorithm (with 20 cores) uses approximately 10x the memory of the sequential version on average. This supports our space complexity analysis, as a lot of the memory requirement for the algorithms comes from the preparation step which doesn't require extra memory for parallelisation. However, due to the way Python handles multiprocessing with separate Python processes, our implementation choice to use Python has a larger effect on our parallelised comparison results than our single–threaded comparison, so these results may vary by implementation–specific details.

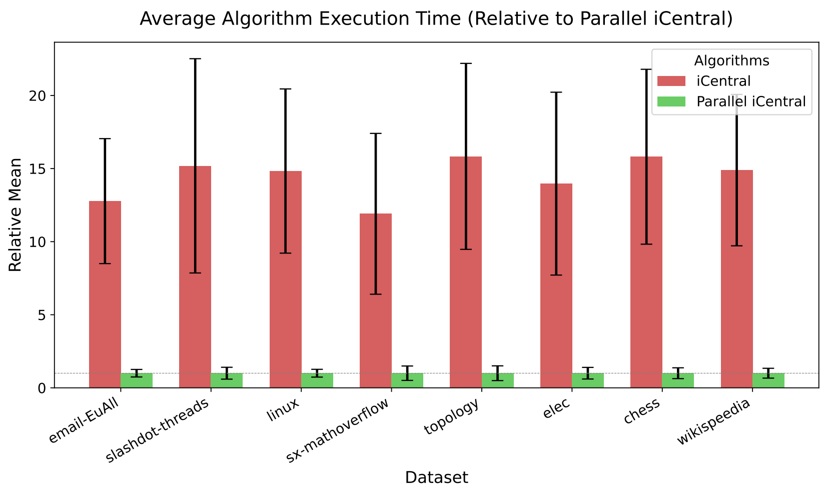


Figure 4.6: *Average iCentral vs. Parallelised iCentral run times (running on 20 cores), relative to parallelised iCentral (bar heights show how much slower serial iCentral is compared to parallel).*

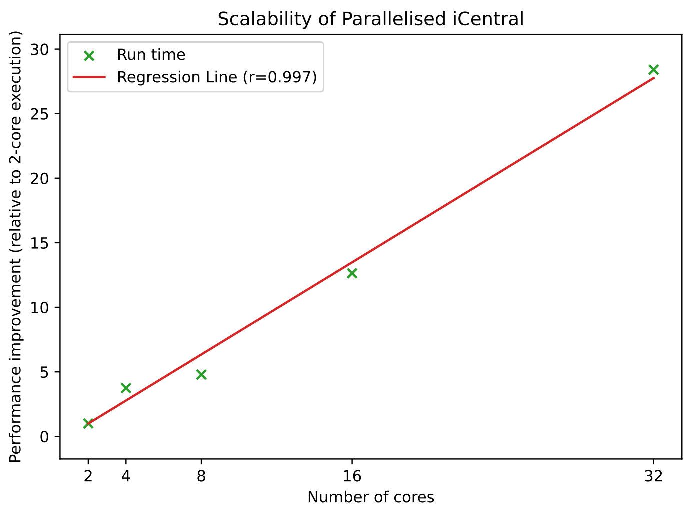


Figure 4.7: *Scalability of Parallelised iCentral algorithm on elec dataset.*

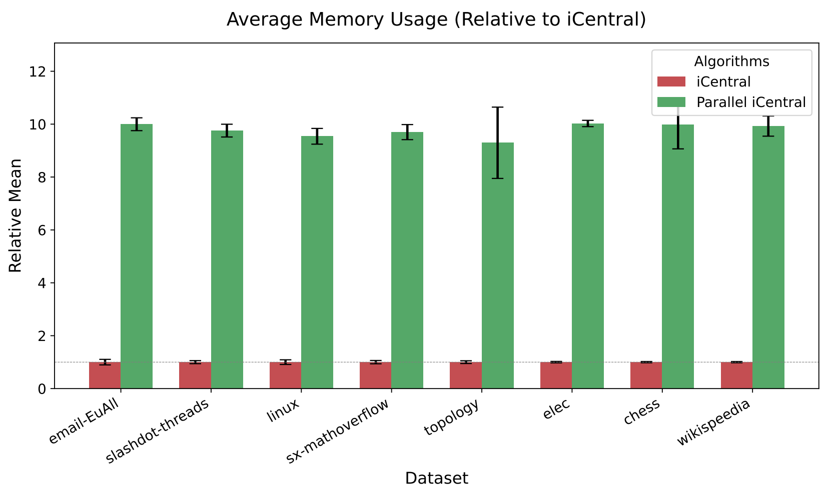


Figure 4.8: *Average iCentral vs. Parallelised iCentral memory usages (running on 20 cores), relative to serial iCentral.*

4.4 Factors Affecting the Convergence of Betweenness Centrality [Extension]

In this section, we further analyse the methods and discuss the results of our investigation into the factors that affect betweenness centrality, as described in **Section 3.8**. We recall our definition of *loss* as a measure of how close our current betweenness centrality is compared to the final centrality mapping, and *convergence* describing how this loss tends to its final state.

We begin by creating some orders of edges for us to trial. These are defined as follows:

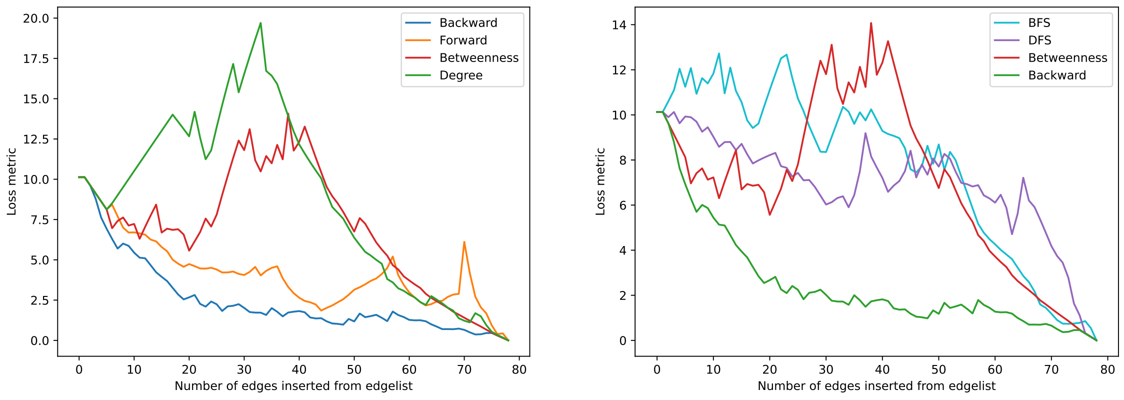
- 'Betweenness' edge–list: Edges are inserted based on their edge betweenness centrality value in the final state of the graph, highest first.
- 'Degree' edge–list: Edges are inserted based on the degree centrality (number of neighbours) of their endpoints, highest first.
- 'BFS' edge–list: Edges are inserted in a breadth-first search order (such that all edges are traversed).
- 'DFS' edge–list: Edges are inserted in a depth-first search order (such that all edges are traversed).
- 'Forward' edge–list: Edges chosen by greedily picking whichever edge results in the smallest loss value after it is inserted (starting from an empty graph, filling in edges).
- 'Backward' edge–list: Edges chosen by greedily picking whichever edge results in the smallest loss value *after it is removed*, working back from a full graph. We then reverse this to get an edge *insertion* order.

Starting this investigation, I had three main hypotheses:

1. The 'betweenness' edge–list ordering would cause the loss to converge very quickly to its final value.
2. The 'degree' edge–list ordering would cause the loss to converge quickly to its final value.
3. Prioritising edges which do not increase the number of connected components will allow the loss to converge faster (but have less effect than the above two methods).

We therefore use the 'betweenness' and 'degree' orderings to test the first two of our above hypothesis. The 'BFS' and 'DFS' orderings are standard graph traversal methods that we use for comparison and also follow the third hypothesis. For comparison to these, we use the 'forward' and 'backward' edge–lists as indicators of the fastest possible convergences we can achieve given our measurement system defined in **Section 3.8**. However, since these are calculated by greedily minimising over all possible edges at each step, they are not useful by themselves.

For the following example graphs, we show results using the *karate club* graph [29], consisting of 34 nodes and 78 edges. However, having experimented with other similar social network graphs,



(a) *Losses for Backward, Forward, Betweenness and Degree edge–lists.*

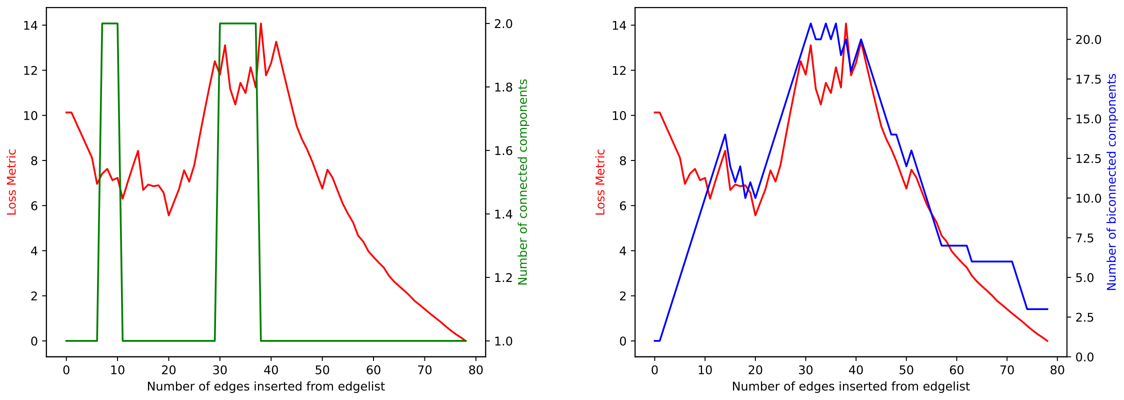
(b) *Losses for BFS, DFS, Betweenness and Backward edge–lists.*

Figure 4.9: *Loss metric vs. Edges inserted, for each of our edge list orderings.*

our results seem to generalise. We use much smaller graphs than our previous experiments here, as otherwise, each investigation would take too long.

From **Figure 4.9a** we see that our baseline edge–lists perform well, with the 'backward' edge–list trending very quickly to a low loss value, as expected. We also see 'betweenness' performing better than 'degree', which again follows our expectations. Interestingly however, in **Figure 4.9b** we observe that the 'BFS' and 'DFS' orderings perform very similarly to 'betweenness' – a surprising result as I had expected 'betweenness' to perform significantly better. This provided strong evidence for my third hypothesis, which led me to shift my focus and now investigate the **graph properties which affect and correlate with changes in betweenness centrality**.

We do this similarly to before, but instead we track two graph properties along with our loss values to see how they change with each insertion. Namely, we track the number of connected components and the number of biconnected components.



(a) *Correlation of loss and number of connected components.*

(b) *Correlation of loss and number of biconnected components.*

Figure 4.10: *Plots showing correlation between graph properties and our loss metric over edge insertions using the 'betweenness' edge–list.*

Figure 4.10 shows an example of our results, where we track the above factors against loss for the 'betweenness' edge–list. We use this as one of the clearest examples, though our results generalise across the other edge–lists and datasets. We explain our findings here:

- **Number of connected components** (**Figure 4.10a**). Across the results, we could observe that whenever the number of connected components increased the loss increased sharply as well, and vice versa. However, since the number of connected components tended to not vary much, mostly staying around 1 or 2, it was hard to draw further results from this.
- **Number of biconnected components** (**Figure 4.10b**). Across the results, we observed a strong correlation between the number of biconnected components and betweenness centrality, showing a correlation co–efficient of 0.77 in our example. They both fluctuate heavily, with their peaks and troughs closely aligning.

4.4.1 Summary of Investigation

To conclude the results of our experiment, we refer back to our initial hypotheses. We found that while our 'betweenness' and 'degree' orderings performed reasonably well at causing fast convergence, simpler methods like 'BFS' and 'DFS' performed similarly and sometimes better. This is surprising, as I had not expected a depth first search to give a better edge insertion order for converging betweenness centrality, than the betweenness centrality of the edges themselves.

Also, while I had initially hypothesised that keeping the number of connected components low will correlate with faster convergence, a more useful and significant result was the correlation with the number of biconnected components. I believe this is because of how betweenness centrality becomes concentrated at articulation points, therefore the creation and removal of these nodes has a large effect on the loss value.

Interpreting our above results with respect to our initial motivation, we suggest that if we can control the next nodes/edges to explore, we should try to follow a standard traversal pattern, preferably a depth-first search, to best represent betweenness centrality values. Additionally, since changes in the number of biconnected (and also connected) components and can be found with much lower complexity than betweenness centrality, they serve as good indicators of whether an inserted edge may have a large effect on our betweenness centrality values. Practically, this could be useful if we wanted to batch up multiple edges together into a single insertion to save time (as supported by the Lee-BCC algorithm), but only when we think this will significantly change betweenness centrality.

4.5 Review of Success Criteria

Here I review my work and evaluate of results against my initial success criteria, detailing my key contributions:

1. **Implement the iCentral algorithm** according to its definition from the iCentral paper.
2. **Implement the Lee-BCC algorithm** according to its definition from the Lee-BCC paper.

3. **Implement the parallelisation modification** for the iCentral algorithm.
4. **Verify my code is an accurate implementation** of the above algorithms, both logically and empirically:
    (a) **Logical Correctness** – ensuring my code *correctly reflects the pseudocode* defined by the papers.
    (b) **Empirical Correctness** – ensure my *algorithms work correctly* by testing against existing methods.

As demonstrated throughout my implementation section, **I have created full working Python implementations for the iCentral algorithm (Subsection 3.1.4), Lee-BCC algorithm (Subsection 3.1.3) and the parallelised modification of the iCental algorithm (Section 3.4)**. This involved gaining a thorough understanding of all the required theory that defines how and why both algorithms work.

Throughout these sections, I have shown how the **my Python code correctly reflects the pseudocode** and theory defined by the paper. I have also **conducted tests of my algorithm** against the pre-existing NetworkX library's static betweenness centrality function to **provide empirical evidence that my algorithms work correctly.**

Additionally, I have **created readable Python implementations** that are structured to **accurately reflect the pseudocode** given by the papers. This makes it very easy to understand how they work, as well as providing clear implementation definitions for notational concepts from the papers. In contrast, the C++ code presented by the iCentral algorithm is difficult to reconcile with their pseudocode, and there exists **no prior implementations of the Lee-BCC algorithm that are publicly available**. I have **released my code as open source** to encourage further development into the field.

5. **Compute performance results** for a selection of real–world datasets.
6. **Compare my statistics** against those from the iCentral paper.

From computing performance results of my Python implementations for the three algorithms, I have been able to support and expand upon the comparison results from the iCentral paper, as evidenced throughout **Section 4.3**.

Our evaluation yielded the following conclusions:

- We **provide evidence to support the iCentral paper's claim** that the iCentral algorithm performs better than the Lee-BCC algorithm, **showing a significant improvement of 1.5x**.
- We **contribute our own analysis of the high variance seen in the results of the iCentral algorithm** compared to the Lee-BCC algorithm. **We hypothesise the reason why this occurs, justified by evidence** from our results.
- We **investigate the effects of the graph's structure on the relative performance of the two algorithms**. We present evidence to show that the relative performance

improvement of the iCentral algorithm over the Lee-BCC algorithm is **highly correlated with the average node degree of the graph**, and describe how this observation could be used for further optimisation.

- We provide **evidence to support the iCentral paper's claims about the scalability of the iCentral algorithm,** showing substantial performance improvements over its serial implementation.
- The results we have obtained **provide further empirical evidence for our complexity analysis**, with all timing and memory results scaling as we would expect.

I also completed the following extensions of my initial project:

- I have **extended the iCentral algorithm to accept a larger range of inputs** I could not previously handle by supporting the bridge-edge insertions and node insertions. This used and applied the theory I learnt from studying the Lee-BCC algorithm.
- I have **investigated potential factors which affect the convergence of betweenness centrality,** with the results summarised in **Subsection 4.4.1**. Specifically, this included:
  - Analysis of how the order in which edges are inserted into a graph affects the convergence of betweenness centrality. We conclude *graphs traversals such as breadth–first search and depth–first search provide an edge ordering that causes betweenness centrality to converge quickly.*
  - Analysis into factors which affect and correlate with changes in betweenness centrality. We conclude *biconnected components are a strong indicator of large changes in betweenness centrality.*

Chapter 5

Conclusions

In conclusion, **this project was a resounding success**. I completed **all core success criteria**, in addition to **two significant extensions**. I describe my project's achievements in further detail in **Section 4.5**.

5.1 Lessons learnt

Through this project, I learned various industry-standard systems and tools with which I had no prior experience. I applied a profiler to investigate hidden time costs; integrated the Python multiprocessing module to perform parallel computation; and used a high-performance computing cluster to schedule jobs on large datasets, exceeding the capabilities of my local machine.

A considerable amount of time was spent understanding these above tools since I was not previously familiar with any of them. This made the development and debugging process slow. In hindsight, I should have accounted for this by re-evaluating my timeline and allocating more time to understanding these tools.

On a more positive note, I was able to learn a lot of useful knowledge and feel confident in applying this to other projects. Additionally, I now have a greater understanding of graph theory and the techniques we can use to solve problems relating to it. yeah

5.2 Future Work

The primary avenue for future work would be to investigate further optimisations, extending those presented by the papers. In particular, one may be able to utilise additional assumptions about the graph structure to further reduce the work required. Additionally, we could even target optimisations for live systems in which new edges may be inserted during runtime.

Another direction would be to analyse how other graph properties, not just those considered in my extension, affect betweenness centrality. Furthermore, I would have liked to use graph properties to create a more formal set of heuristics to indicate which node/edges are best to explore in a graph. This could provide a more representative approximation of final betweenness centrality with the fewest edge insertions.

Finally, while this dissertation's work has focused on *incremental betweenness centrality*, many other centrality measures exist and are used frequently in analysis, such as eigenvector and PageRank centrality. I would have liked to explore the algorithms behind these centralities, and in particular, evaluate their susceptibility to the optimisations shown to be successful in this project.

Bibliography

[1] Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. “Parallel algorithm for incremental betweenness centrality on large graphs”. In: *IEEE Transactions on Parallel and Distributed Systems* 29.3 (2017), pp. 659–672.

[2] Min-Joong Lee, Sunghee Choi, and Chin-Wan Chung. “Efficient algorithms for updating betweenness centrality in fully dynamic graphs”. In: *Information Sciences* 326 (2016), pp. 278–296. ISSN: 0020-0255. DOI: `https://doi.org/10.1016/j.ins.2015.07.053`. URL: `https://www.sciencedirect.com/science/article/pii/S0020025515005617`.

[3] Linton C Freeman. “A set of measures of centrality based on betweenness”. In: *Sociometry* (1977), pp. 35–41.

[4] Mark Newman. *Networks: An Introduction*. Oxford University Press, Mar. 2010. ISBN: 9780199206650. DOI: `10.1093/acprof:oso/9780199206650.001.0001`. URL: `https://doi.org/10.1093/acprof:oso/9780199206650.001.0001`.

[5] Mark EJ Newman and Michelle Girvan. “Finding and evaluating community structure in networks”. In: *Physical review E* 69.2 (2004), p. 026113.

[6] Stefan Lämmer, Björn Gehlsen, and Dirk Helbing. “Scaling laws in the spatial structure of urban road networks”. In: *Physica A: Statistical Mechanics and its Applications* 363.1 (2006), pp. 89–95.

[7] Elizabeth M Daly and Mads Haahr. “Social network analysis for routing in disconnected delay-tolerant manets”. In: *Proceedings of the 8th ACM international symposium on Mobile ad hoc networking and computing*. 2007, pp. 32–40.

[8] Claudio Rocchini. URL: `https://commons.wikimedia.org/w/index.php?curid=1988980`.

[9] Ulrik Brandes. “A faster algorithm for betweenness centrality”. In: *Journal of mathematical sociology* 25.2 (2001), pp. 163–177.

[10] *A comparison of state-of-the-art graph processing systems*. URL: `https://engineering.fb.com/2016/10/19/core-infra/a-comparison-of-state-of-the-art-graph-processing-systems/`.

[11] Jac M Anthonisse. *The rush in a directed graph*. Stichting Mathematisch Centrum. Mathematische Besliskunde, 1971.

[12] Evithyan. URL: `https://commons.wikimedia.org/w/index.php?curid=52107996`.

[13] Ulrik Brandes. “On variants of shortest-path betweenness centrality and their generic computation”. In: *Social networks* 30.2 (2008), pp. 136–145.

[14] Camil Demetrescu and Giuseppe F Italiano. “A new approach to dynamic all pairs shortest paths”. In: *Journal of the ACM (JACM)* 51.6 (2004), pp. 968–992.

[15] Oded Green, Robert McColl, and David A Bader. “A fast algorithm for streaming betweenness centrality”. In: *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*. IEEE. 2012, pp. 11–20.

[16] Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. “Betweenness centrality–incremental and faster”. In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2014, pp. 577–588.

[17] Winston W Royce. "Managing the development of large software systems (1970)". In: (2021).

[18] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual.* Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[19] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX.* Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[20] Holger Krekel et al. *pytest 8.0.0.* 2004. URL: `https://github.com/pytest-dev/pytest`.

[21] *cProfile.* URL: `https://docs.python.org/3/library/profile.html`.

[22] Gervais Pedregosa. *memory_profiler.* URL: `https://pypi.org/project/memory-profiler/`.

[23] *multiprocessing.* URL: `https://docs.python.org/3/library/multiprocessing.html`.

[24] *Cambridge RCS HPC.* URL: `https://www.hpc.cam.ac.uk/`.

[25] Scott Chacon and Ben Straub. *Pro git.* Apress, 2014.

[26] John Hopcroft and Robert Tarjan. "Algorithm 447: efficient algorithms for graph manipulation". In: *Communications of the ACM* 16.6 (1973), pp. 372–378.

[27] *PyPy Python Implementation.* URL: `https://www.pypy.org/`.

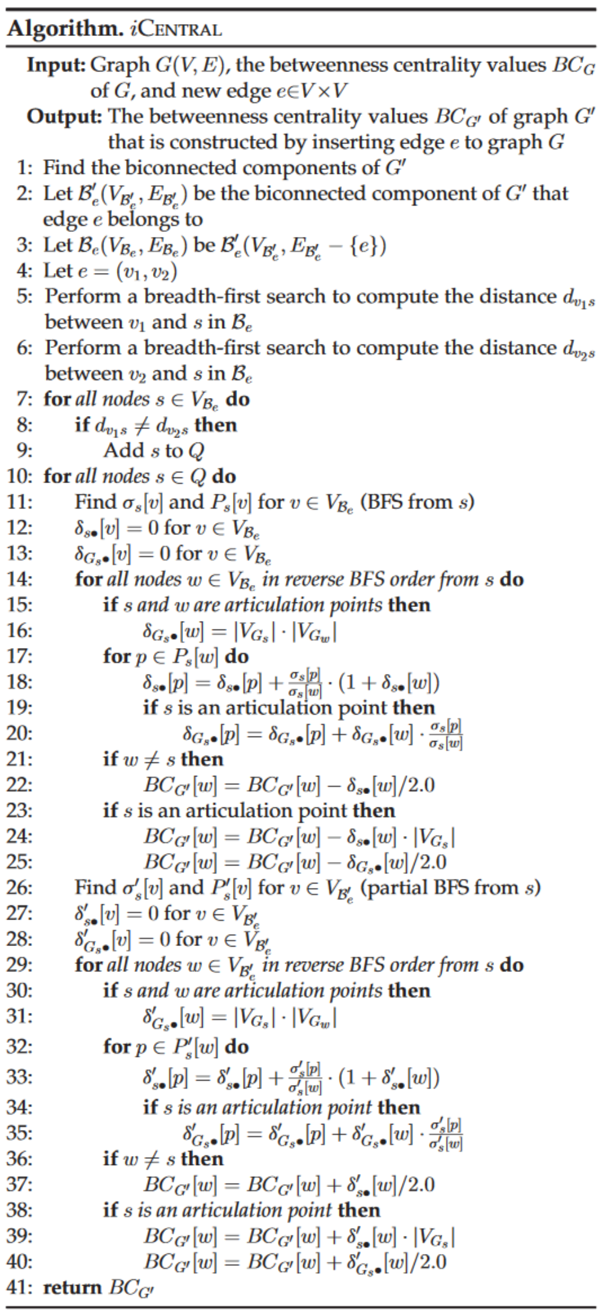
[28] Jérôme Kunegis. "KONECT – The Koblenz Network Collection". In: *Proc. Int. Conf. on World Wide Web Companion.* 2013, pp. 1343–1350. URL: `http://dl.acm.org/citation.cfm?id=2488173`.

[29] Wayne W Zachary. "An information flow model for conflict and fission in small groups". In: *Journal of anthropological research* 33.4 (1977), pp. 452–473.

Appendix A

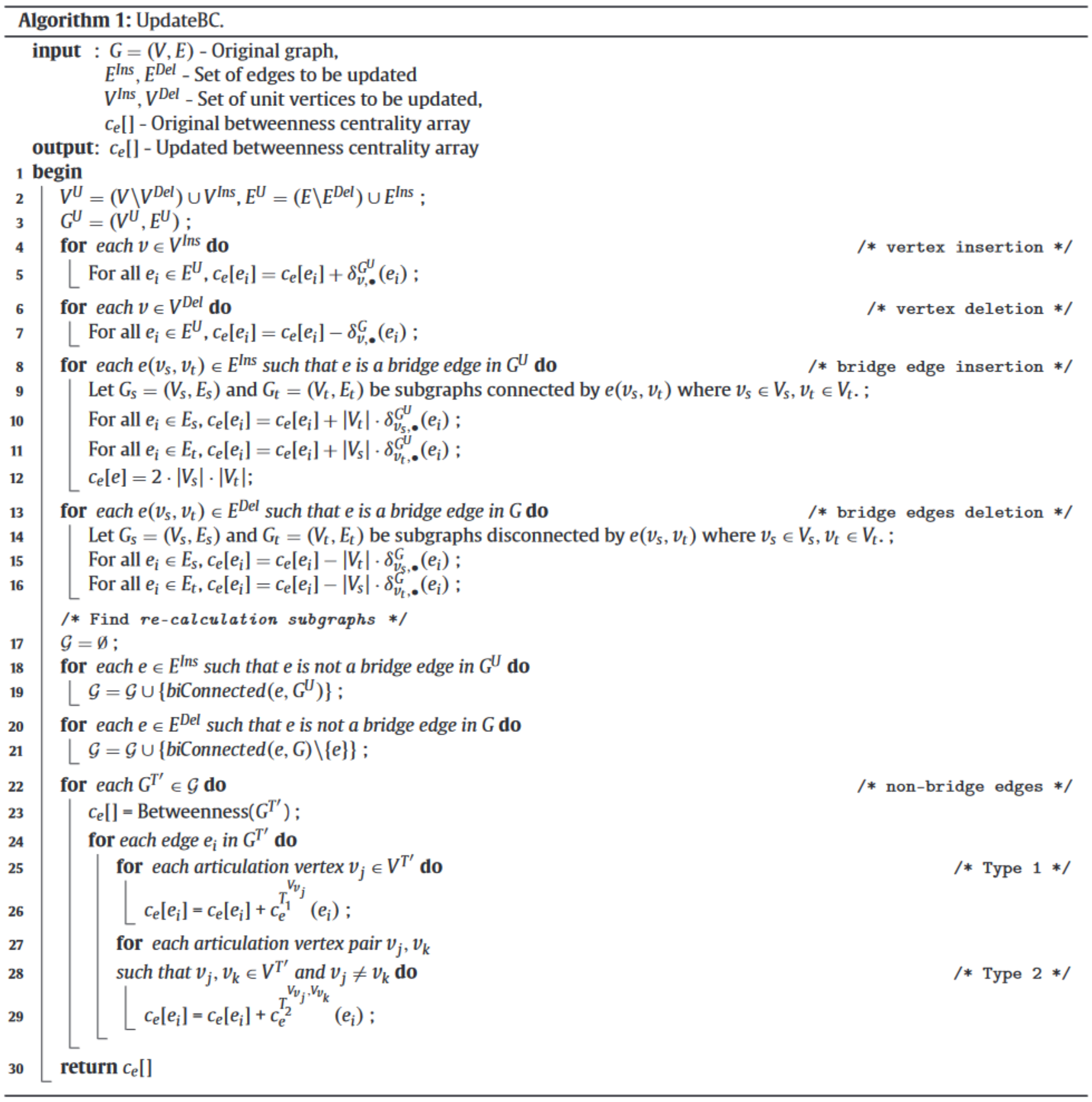
Pseudocode from Papers

Here we show the pseudocode from the two papers.



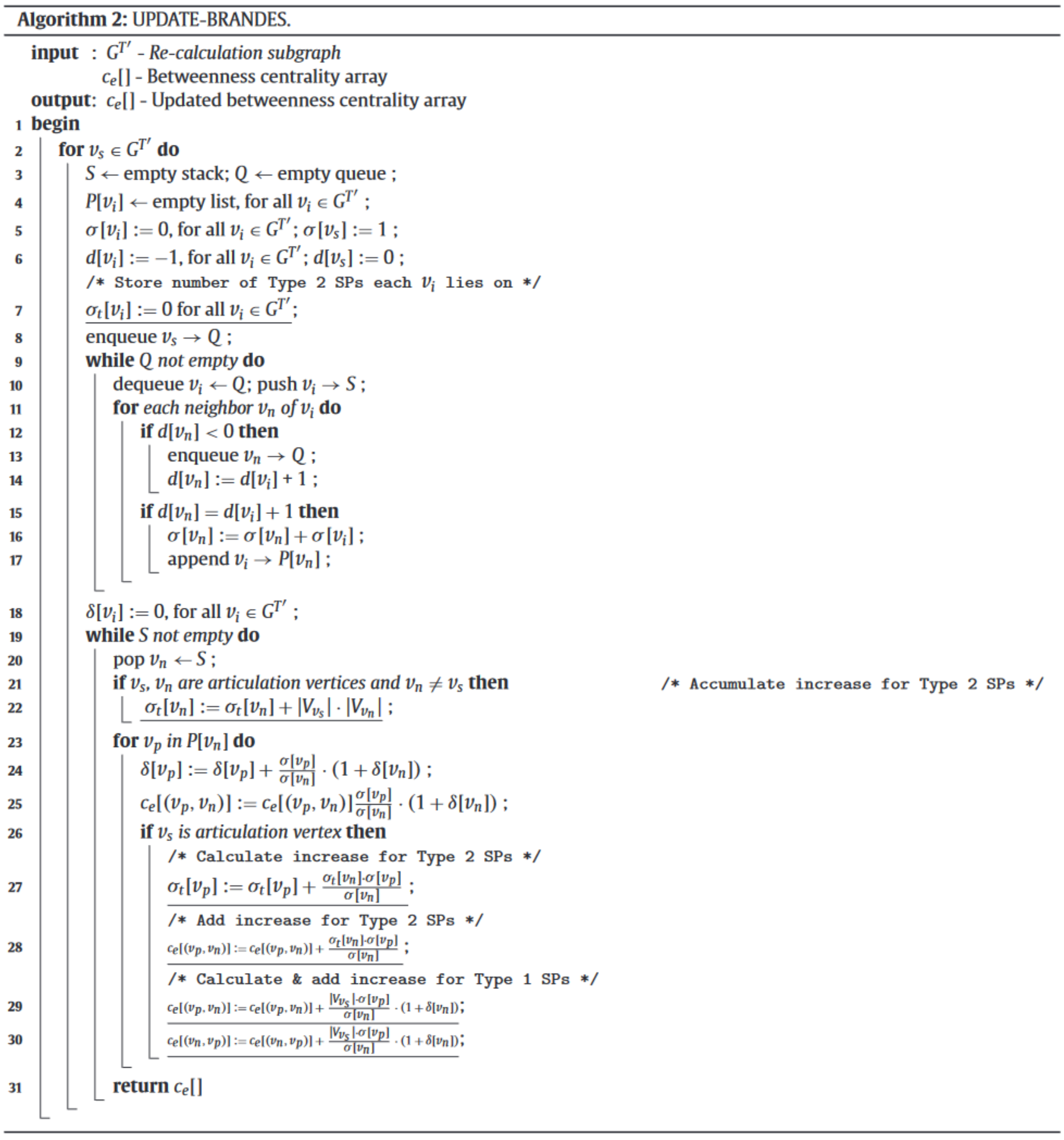
```
Algorithm. iCentral
Input: Graph $G(V, E)$, the betweenness centrality values $BC_G$
        of $G$, and new edge $e \in V \times V$
Output: The betweenness centrality values $BC_{G'}$ of graph $G'$
        that is constructed by inserting edge $e$ to graph $G$
 1: Find the biconnected components of $G'$
 2: Let $\mathcal{B}'_e(V_{\mathcal{B}'_e}, E_{\mathcal{B}'_e})$ be the biconnected component of $G'$ that
    edge $e$ belongs to
 3: Let $\mathcal{B}_e(V_{\mathcal{B}_e}, E_{\mathcal{B}_e})$ be $\mathcal{B}'_e(V_{\mathcal{B}'_e}, E_{\mathcal{B}'_e} - \{e\})$
 4: Let $e = (v_1, v_2)$
 5: Perform a breadth-first search to compute the distance $d_{v_1 s}$
    between $v_1$ and $s$ in $\mathcal{B}_e$
 6: Perform a breadth-first search to compute the distance $d_{v_2 s}$
    between $v_2$ and $s$ in $\mathcal{B}_e$
 7: for all nodes $s \in V_{\mathcal{B}_e}$ do
 8:     if $d_{v_1 s} \neq d_{v_2 s}$ then
 9:         Add $s$ to $Q$
10: for all nodes $s \in Q$ do
11:     Find $\sigma_s[v]$ and $P_s[v]$ for $v \in V_{\mathcal{B}_e}$ (BFS from $s$)
12:     $\delta_{s\bullet}[v] = 0$ for $v \in V_{\mathcal{B}_e}$
13:     $\delta_{G_s\bullet}[v] = 0$ for $v \in V_{\mathcal{B}_e}$
14:     for all nodes $w \in V_{\mathcal{B}_e}$ in reverse BFS order from $s$ do
15:         if $s$ and $w$ are articulation points then
16:             $\delta_{G_s\bullet}[w] = |V_{G_s}| \cdot |V_{G_w}|$
17:         for $p \in P_s[w]$ do
18:             $\delta_{s\bullet}[p] = \delta_{s\bullet}[p] + \frac{\sigma_s[p]}{\sigma_s[w]} \cdot (1 + \delta_{s\bullet}[w])$
19:             if $s$ is an articulation point then
20:                 $\delta_{G_s\bullet}[p] = \delta_{G_s\bullet}[p] + \delta_{G_s\bullet}[w] \cdot \frac{\sigma_s[p]}{\sigma_s[w]}$
21:         if $w \neq s$ then
22:             $BC_{G'}[w] = BC_{G'}[w] - \delta_{s\bullet}[w]/2.0$
23:         if $s$ is an articulation point then
24:             $BC_{G'}[w] = BC_{G'}[w] - \delta_{s\bullet}[w] \cdot |V_{G_s}|$
25:             $BC_{G'}[w] = BC_{G'}[w] - \delta_{G_s\bullet}[w]/2.0$
26:     Find $\sigma'_s[v]$ and $P'_s[v]$ for $v \in V_{\mathcal{B}'_e}$ (partial BFS from $s$)
27:     $\delta'_{s\bullet}[v] = 0$ for $v \in V_{\mathcal{B}'_e}$
28:     $\delta'_{G_s\bullet}[v] = 0$ for $v \in V_{\mathcal{B}'_e}$
29:     for all nodes $w \in V_{\mathcal{B}'_e}$ in reverse BFS order from $s$ do
30:         if $s$ and $w$ are articulation points then
31:             $\delta'_{G_s\bullet}[w] = |V_{G_s}| \cdot |V_{G_w}|$
32:         for $p \in P'_s[w]$ do
33:             $\delta'_{s\bullet}[p] = \delta'_{s\bullet}[p] + \frac{\sigma'_s[p]}{\sigma'_s[w]} \cdot (1 + \delta'_{s\bullet}[w])$
34:             if $s$ is an articulation point then
35:                 $\delta'_{G_s\bullet}[p] = \delta'_{G_s\bullet}[p] + \delta'_{G_s\bullet}[w] \cdot \frac{\sigma'_s[p]}{\sigma'_s[w]}$
36:         if $w \neq s$ then
37:             $BC_{G'}[w] = BC_{G'}[w] + \delta'_{s\bullet}[w]/2.0$
38:         if $s$ is an articulation point then
39:             $BC_{G'}[w] = BC_{G'}[w] + \delta'_{s\bullet}[w] \cdot |V_{G_s}|$
40:             $BC_{G'}[w] = BC_{G'}[w] + \delta'_{G_s\bullet}[w]/2.0$
41: return $BC_{G'}$
```

Figure A.1: *Pseudocode displaying the iCentral algorithm.*



```
Algorithm 1: UpdateBC.
input  : $G = (V, E)$ - Original graph,
         $E^{Ins}, E^{Del}$ - Set of edges to updated
         $V^{Ins}, V^{Del}$ - Set of unit vertices to be updated,
         $c_e[]$ - Original betweenness centrality array
output: $c_e[]$ - Updated betweenness centrality array
begin
    $V^U = (V \backslash V^{Del}) \cup V^{Ins}, E^U = (E \backslash E^{Del}) \cup E^{Ins}$ ;
    $G^U = (V^U, E^U)$ ;
    for each $v \in V^{Ins}$ do                                        /* vertex insertion */
        For all $e_i \in E^U, c_e[e_i] = c_e[e_i] + \delta^{G^U}_{v,\bullet}(e_i)$ ;
    for each $v \in V^{Del}$ do                                         /* vertex deletion */
        For all $e_i \in E^U, c_e[e_i] = c_e[e_i] - \delta^{G}_{v,\bullet}(e_i)$ ;
    for each $e(v_s, v_t) \in E^{Ins}$ such that $e$ is a bridge edge in $G^U$ do     /* bridge edge insertion */
        Let $G_s = (V_s, E_s)$ and $G_t = (V_t, E_t)$ be subgraphs connected by $e(v_s, v_t)$ where $v_s \in V_s, v_t \in V_t$. ;
        For all $e_i \in E_s, c_e[e_i] = c_e[e_i] + |V_t| \cdot \delta^{G^U}_{v_s,\bullet}(e_i)$ ;
        For all $e_i \in E_t, c_e[e_i] = c_e[e_i] + |V_s| \cdot \delta^{G^U}_{v_t,\bullet}(e_i)$ ;
        $c_e[e] = 2 \cdot |V_s| \cdot |V_t|$;
    for each $e(v_s, v_t) \in E^{Del}$ such that $e$ is a bridge edge in $G$ do       /* bridge edges deletion */
        Let $G_s = (V_s, E_s)$ and $G_t = (V_t, E_t)$ be subgraphs disconnected by $e(v_s, v_t)$ where $v_s \in V_s, v_t \in V_t$. ;
        For all $e_i \in E_s, c_e[e_i] = c_e[e_i] - |V_t| \cdot \delta^{G}_{v_s,\bullet}(e_i)$ ;
        For all $e_i \in E_t, c_e[e_i] = c_e[e_i] - |V_s| \cdot \delta^{G}_{v_t,\bullet}(e_i)$ ;
    /* Find re-calculation subgraphs */
    $\mathcal{G} = \emptyset$ ;
    for each $e \in E^{Ins}$ such that $e$ is not a bridge edge in $G^U$ do
        $\mathcal{G} = \mathcal{G} \cup \{biConnected(e, G^U)\}$ ;
    for each $e \in E^{Del}$ such that $e$ is not a bridge edge in $G$ do
        $\mathcal{G} = \mathcal{G} \cup \{biConnected(e, G) \backslash \{e\}\}$ ;
    for each $G^{T'} \in \mathcal{G}$ do                                     /* non-bridge edges */
        $c_e[]$ = Betweenness($G^{T'}$) ;
        for each edge $e_i$ in $G^{T'}$ do
            for each articulation vertex $v_j \in V^{T'}$ do                         /* Type 1 */
                $c_e[e_i] = c_e[e_i] + c_e^{T_1^{V_{v_j}}}(e_i)$ ;
            for each articulation vertex pair $v_j, v_k$
            such that $v_j, v_k \in V^{T'}$ and $v_j \neq v_k$ do                     /* Type 2 */
                $c_e[e_i] = c_e[e_i] + c_e^{T_2^{V_{v_j},V_{v_k}}}(e_i)$ ;
    return $c_e[]$
```

Figure A.2: *UpdateBC pseudocode from the Lee-BCC paper.*



Algorithm 2: UPDATE-BRANDES.

```
input  : $G^{T'}$ - Re-calculation subgraph
         $c_e[]$ - Betweenness centrality array
output : $c_e[]$ - Updated betweenness centrality array
begin
    for $v_s \in G^{T'}$ do
        $S \leftarrow$ empty stack; $Q \leftarrow$ empty queue ;
        $P[v_i] \leftarrow$ empty list, for all $v_i \in G^{T'}$ ;
        $\sigma[v_i] := 0$, for all $v_i \in G^{T'}$; $\sigma[v_s] := 1$ ;
        $d[v_i] := -1$, for all $v_i \in G^{T'}$; $d[v_s] := 0$ ;
        /* Store number of Type 2 SPs each $v_i$ lies on */
        $\sigma_t[v_i] := 0$ for all $v_i \in G^{T'}$;
        enqueue $v_s \rightarrow Q$ ;
        while $Q$ not empty do
            dequeue $v_i \leftarrow Q$; push $v_i \rightarrow S$ ;
            for each neighbor $v_n$ of $v_i$ do
                if $d[v_n] < 0$ then
                    enqueue $v_n \rightarrow Q$ ;
                    $d[v_n] := d[v_i] + 1$ ;
                if $d[v_n] = d[v_i] + 1$ then
                    $\sigma[v_n] := \sigma[v_n] + \sigma[v_i]$ ;
                    append $v_i \rightarrow P[v_n]$ ;
        $\delta[v_i] := 0$, for all $v_i \in G^{T'}$ ;
        while $S$ not empty do
            pop $v_n \leftarrow S$ ;
            if $v_s$, $v_n$ are articulation vertices and $v_n \neq v_s$ then      /* Accumulate increase for Type 2 SPs */
                $\sigma_t[v_n] := \sigma_t[v_n] + |V_{v_s}| \cdot |V_{v_n}|$ ;
            for $v_p$ in $P[v_n]$ do
                $\delta[v_p] := \delta[v_p] + \frac{\sigma[v_p]}{\sigma[v_n]} \cdot (1 + \delta[v_n])$ ;
                $c_e[(v_p, v_n)] := c_e[(v_p, v_n)] \frac{\sigma[v_p]}{\sigma[v_n]} \cdot (1 + \delta[v_n])$ ;
                if $v_s$ is articulation vertex then
                    /* Calculate increase for Type 2 SPs */
                    $\sigma_t[v_p] := \sigma_t[v_p] + \frac{\sigma_t[v_n] \cdot \sigma[v_p]}{\sigma[v_n]}$ ;
                    /* Add increase for Type 2 SPs */
                    $c_e[(v_p, v_n)] := c_e[(v_p, v_n)] + \frac{\sigma_t[v_n] \cdot \sigma[v_p]}{\sigma[v_n]}$ ;
                    /* Calculate & add increase for Type 1 SPs */
                    $c_e[(v_p, v_n)] := c_e[(v_p, v_n)] + \frac{|V_{v_s}| \cdot \sigma[v_p]}{\sigma[v_n]} \cdot (1 + \delta[v_n])$;
                    $c_e[(v_n, v_p)] := c_e[(v_n, v_p)] + \frac{|V_{v_s}| \cdot \sigma[v_p]}{\sigma[v_n]} \cdot (1 + \delta[v_n])$;
            return $c_e[]$
```

Figure A.3: *Update-Brandes pseudocode from the Lee-BCC paper.*

Appendix B

Project Proposal

The project proposal is shown on the following pages.

1 Introduction

Over the last decade, there has been an increase in the usage of large graph statistics. One of the most popular and useful measures is the *Betweenness Centrality* (BC) [1]. This measure provides a value of the relative importance of every node within a graph, defined as the relative number of shortest paths that rely on the node. The betweenness centrality of a node v is calculated as the fraction of shortest paths between a given pair of nodes that passes through v, summed across all possible pairs of nodes. It is a highly important measure, widely used in analysis of large graphs for a variety of applications such as network analysis, community detection and social influence.

Definition 1. *Let $G = (V, E)$ be a graph, where V is the set of nodes and $E \subseteq V \times V$ is the set of edges. The Betweenness Centrality for node $v \in V$ is defined as:*

$$BC_G(v) = \sum_{\substack{s,t \in V, \\ s \neq v \neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Where σ_{st} is the number of shortest paths from node s to node t, and $\sigma_{st}(v)$ is the number of shortest paths from s to t that pass through v.

This metric is often used on very large graphs and is very expensive to compute. For unweighted graphs the calculation of this metric naively requires $O(|V|^3)$ time, by performing breadth-first search on every node to find their shortest paths to all other nodes. This was improved on by Brandes [2] to create an $O(|V| \cdot |E|)$ algorithm, where he introduced concepts that underpin most modern BC algorithms.

Another area of recent research has been on incremental algorithms. Modern graphs that we often want to derive statistics from are usually both very large and constantly changing (e.g. users joining/leaving a social network, changes in friendships between users). To recalculate an expensive metric for the whole graph for every single change would be prohibitively computationally expensive, so we may instead try to identify what has changed within each update, and recompute only the affected values. This allows us to avoid unnecessary calculations and greatly increase performance.

In this project I will aim to study and implement the iCentral algorithm, proposed by Jamour et al. in 2017 [3] (which I later refer to simply as "the paper") for calculating incremental betweenness centrality, which provides an efficient solution to the above problems. This paper uses the Lee-BCC algorithm [4] as one of the baselines it compares its new algorithm to, and also includes details to allow for an *embarrassingly parallel* optimisation. I will try to implement both of these which will allow me to corroborate the comparative results achieved by Jamour et al.'s paper by testing their execution performance on the same datasets. As the project will requrie a lot of data-processing, I aim to write my implementation in Python, and due to the intensive resource requirements for large datasets I will likely use the Cambridge High Performance Computing system.

Extensionally, I would like to further the range of datasets[1,2] experimented on to a wider variety than those used to further reinforce their results, as well as find other similar solutions

[1]SNAP dataset: `http://snap.stanford.edu/data/`

[2]KONECT dataset: `http://konect.cc/networks/`

and implementations online and see how they compare. I would also like to study the factors that affect the rate at which each increment converges onto the true value of BC, for example the order that data is incrementally provided to the algorithm, or the value(s) of other graph properties. Additionally, I would like to try to generate heuristics for optimising convergence rate as well as confidence grades for BC accuracy given a fraction of the graph. Finally, I would also like to experiment with creating visualisations for our metric in different scenarios to provide high-level and useful information.

2 Structure of the Project

The main components to my project are:

- **Understanding**: Reading through the algorithms to fully understand how and why they work, as well as trying to create a more intuitional reading of them.
- **Implementation**: Writing an implementation of both of these algorithms in Python myself. This can be tested against other existing algorithms online (e.g. NetworkX centrality Python library[3]).
- **Data testing**: Testing the performance comparison of my algorithms versus those from the paper on their chosen datasets.

Possible extensions to my project are:

- **Expanding comparisons**: Testing on a wider selection of data sets to see if similar results are obtained. Also comparing against other available similar solutions.
- **Analysis**: There are three main topics I would like to focus on investigating:
  - Factors that affect the rate at which the betweenness centrality values converge to their true value
  - How well we can generate confidence grades for the accuracy of the BC values for a given fraction of the data
  - Factors that affect the betweenness centrality values of nodes throughout the graph
- **Visualisation**: Create a high level and useful way to view betweenness centrality for common uses.

3 Evaluation

3.1 Datasets

- The main dataset used to compare the algorithms is the Twitter-Munmun dataset (460K nodes, 833K edges).

[3] `https://networkx.org/documentation/stable/reference/algorithms/centrality.html`

- Other large datasets[4] used in the comparisons include:
  - epinions (119K nodes, 700K)
  - email-EuAll (224K nodes, 340K edges)
  - com-dblp (317K nodes, 1M edges)
  - web-NotreDame (325K nodes, 1.1M edges)

3.2 Data ordering

- Most datasets do not have any ordering or timestamp data inherent in them.
- For these cases, I will use randomised algorithms to simulate incoming data, which may include:
  - choosing nodes completely randomly
  - expanding along a frontier
  - exploring via random walks

3.3 Implementation Correctness

- There are two main ways to test the correctness of an implementation: logically and empirically.
- To test correctness logically, I will compare my implementations against their associated proofs to ensure they align correctly.
- To test correctness empirically, I will first test each section of the algorithms with unit testing. I will then use existing implementations (including both incremental and the more widely available non-incremental) solutions to test my algorithms on toy datasets. This can include both synthetic datasets (e.g. Erdős-Réyni models [5]) or small real-world datasets.

3.4 Metrics

- The core measures I will be taking to compare the two algorithms are a) the total runtime of execution and b) the total memory consumption, as these are the measurements from the paper.
- I may also extensionally compare variances within the execution times per increment, e.g. one algorithm may usually be fast but infrequently require an expensive computation, while another is slower but more consistent.

[4]all graphs mentioned are available from the SNAP and KONECT collections

4 Success Criteria

For the project to be deemed a success, the following must be successfully completed:

1. Implement the iCentral algorithm according to its definition from the paper [3]
2. Implement the Lee-BCC algorithm according to its definition from the paper [4]
3. Implement the parallelisation extension for the iCentral algorithm
4. Verify the correctness of the above algorithms using the methods defined in Evaluation: Implementation Correctness
5. Compute performance results for my implementations for the chosen datasets
6. Compare my statistics against those produced by the paper

5 Timetable and Milestones

Weeks 1 to 2 (16 Oct 23 – 29 Oct 23)

- Preparatory work
- Read research papers and relevant related material

Weeks 3 to 4 (30 Oct 23 – 12 Nov 23)

- Finish reading through papers and solutions
- Get clear understanding of how each algorithm works
- Find and familiarise myself with tools and libraries to use, including making toy programs
- E.g. Python libraries, Git workflow,
- **Milestone: Have working development setup and workflow**

Weeks 5 to 6 (13 Nov 23 – 26 Nov 23)

- Start writing of background draft for dissertation

Weeks 7 to 8 (27 Nov 23 – 10 Dec 23)

- Finish writing of background draft for dissertation
- Start algorithm implementation
- **Milestone: Have draft of background section written**

9 to 14 (11 Dec 23 – 7 Jan 24) - *Winter Break*

- Finish writing of implementations for algorithms
- Experiment with High Performance Computing systems
- Take time off for Christmas
- **Milestone: Have working implementation of algorithms**

Weeks 15 to 16 (8 Jan 24 – 21 Jan 24)

- Write progress report
- Trial algorithm on large datasets
- **Milestone: Submit progress report (Friday 2nd Feb)**

Weeks 17 to 18 (22 Jan 24 – 4 Feb 24)

- Run algorithms on large datasets
- Perform algorithm comparisons
- Start writing of implementation draft for disser
- **Milestone: Have algorithm comparison results**

Weeks 19 to 20 (5 Feb 24 – 18 Feb 24)

- Continue writing of implementation draft for dissertation
- Start work on extensions

Weeks 21 to 22 (19 Feb 24 – 3 Mar 24)

- Continue writing of implementation & evaluation drafts for dissertation
- Continue work on extensions
- **Milestone: Have draft of implementation section written**

Weeks 23 to 24 (4 Mar 24 – 17 Mar 24)

- Continue writing of evaluation drafts for dissertation
- Finish work on extensions
- **Milestone: all project work completed**

Weeks 25 to 26 (18 Mar 24 – 30 Mar 24)

- Finish writing of evaluation drafts for dissertation

Weeks 27 to 28 (1 Apr 24 – 14 Apr 24) - *Easter Break*

- Start/Finish writing of conclusions drafts for dissertation
- Polish draft and submit to supervisor, project checkers and DoS
- **Milestone: Full project draft to be submitted to Supervisor, Project Checkers and DoS for feedback**

Weeks 29 to 30 (15 Apr 24 – 28 Apr 24)

- Address all comments on drafts
- Iterative feedback loop for new edits/changes

Weeks 30 to 31 (29 Apr – 10 May) - *Final Submission Deadline*

- Finish work on final dissertation
- **Milestone: Submit Dissertation (preferably by 3rd May)**

6 Resource Declaration

For the implementation of the algorithms, I will be using my personal PC (CPU: i7-10700: 8-core @ 2.9 GHz, 16GB RAM, 1TB SSD, 2TB HDD), with a backup secondary laptop (Dell XPS 15 9530). For the execution of my algorithms on small datasets, my machine should suffice, but for large datasets I will use the Cambridge High Performance Computing system, for which I have been approved access to the SL2 & SL3 clusters with Timothy Griffin as my Principle Investigator.

I will use Git version control for frequent backups of my code and dissertation. I will also use OneDrive automatic cloud backups as well as taking periodic snapshots to save to my hard drive.

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

References

[1] Linton C Freeman. “A set of measures of centrality based on betweenness”. In: *Sociometry* (1977), pp. 35–41.

[2] Ulrik Brandes. “A faster algorithm for betweenness centrality”. In: *Journal of mathematical sociology* 25.2 (2001), pp. 163–177.

[3] Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. “Parallel algorithm for incremental betweenness centrality on large graphs”. In: *IEEE Transactions on Parallel and Distributed Systems* 29.3 (2017), pp. 659–672.

[4] Min-Joong Lee, Sunghee Choi, and Chin-Wan Chung. “Efficient algorithms for updating betweenness centrality in fully dynamic graphs”. In: *Information Sciences* 326 (2016), pp. 278–296.

[5] P Erdős and A Rényi. “On random graphs I”. In: *Publ. math. debrecen* 6.290-297 (1959), p. 18.