CS552 Assignment: Minimizing Maximum Betweenness Centrality and Stress Centrality via Link Addition

April 7, 2025

Abstract

This report investigates methods to minimize the maximum betweenness centrality and stress centrality in networks through strategic link addition. We present our methodology for computing centrality measures, identifying optimal links, and optimizing the implementation. The report includes algorithm complexity analysis and experimental results comparing our approach with brute-force methods.

1 Introduction

Betweenness centrality and stress centrality are key metrics in network analysis that measure a node's influence based on its position in shortest paths. This report examines the problem of minimizing the maximum centrality values in a network through strategic link addition, which can improve network robustness and efficiency.

2 Methodology

2.1 Computing Centrality Measures

We used modified version of Brandes' algorithm to allow efficient computation of both betweenness centrality and stress centrality in single traversal per source node.

Definitions

Let $G = (V, E)$ be an unweighted, undirected graph where V is the set of nodes and E is the set of edges.

- Let σ_{st} denote the number of shortest paths from node s to node t.
- Let $\sigma_{st}(v)$ be the number of those paths that pass through node v.
- The **betweenness centrality** of node v is defined as:

$$C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

- The **stress centrality** of node v is the total number of shortest paths passing through v:

$$C_S(v) = \sum_{s \neq v \neq t} \sigma_{st}(v)$$

Algorithm Overview

The algorithm operates in two main phases:

1. **Forward Phase (BFS):** Computes shortest path counts σ, distances, and predecessor lists for each source node s.
2. **Backward Phase (Dependency Accumulation):** Accumulates dependencies in reverse topological order to compute centrality contributions.

- **Forward Phase (BFS):** For a given source node s, perform a Breadth-First Search (BFS) to compute:
  - $\sigma(v)$: The number of shortest paths from node s to node v.
  - $\text{dist}(v)$: The shortest path distance from s to v.
  - $\text{pred}(v)$: The list of predecessor nodes of v on shortest paths from s.
- **Backward Phase (Dependency Accumulation):** Traverse the BFS tree in reverse topological order to compute dependencies:
  - $\delta_1(v)$: Standard Brandes' dependency score, representing the fraction of shortest paths that pass through node v. It contributes to **betweenness centrality**.

$$\delta_1(v) = \sum_{w : v \in \text{pred}(w)} \frac{\sigma(v)}{\sigma(w)} (1 + \delta_1(w))$$

  - $\delta_2(v)$: Dependency count defined as the number of shortest paths that originate at v and end at nodes deeper in the BFS tree (i.e., at a greater distance from s). This is used for stress centrality computation.
- **Stress Centrality Contribution:** For each node v (excluding source s):

$$\text{stress}(v) += \sigma(v) \cdot \delta_2(v)$$

  This captures the total number of shortest paths passing through v, weighted by the number of paths that originate at v and reach further nodes.
- **Note:** *Betweenness centrality* is accumulated using δ_1, while *stress centrality* is accumulated using the product $\sigma(v) \cdot \delta_2(v)$. Both values are halved after all iterations to account for double-counting in undirected graphs.

  Why it works: The total number of shortest paths passing through a node v can be understood combinatorially. Each such path can be viewed as a concatenation of:
  - A shortest path from the source s to v, counted by $\sigma(v)$, and

– A continuation from v to deeper nodes in the BFS tree, counted by $\delta_2(v)$.

Hence, the total number of full paths going through v is the product $\sigma(v) \cdot \delta_2(v)$, which represents all ways in which a path reaching v can be extended further down the tree.

The following pseudocode summarizes the approach:

Algorithm 1: Compute Betweenness and Stress Centrality (`g.compute()`)

Input: Adjacency list `adj` representing graph G
Output: Maximum betweenness and stress centrality values

foreach *source node* $s \in V$ **do**
  Initialize $\texttt{dist}[v] \leftarrow -1, \texttt{sigma}[v] \leftarrow 0, \texttt{pred}[v] \leftarrow []$ for all $v \in V$
  Set $\texttt{dist}[s] \leftarrow 0, \texttt{sigma}[s] \leftarrow 1$
  Initialize empty queue Q, stack S
  Enqueue s into Q
  while Q *not empty* **do**
    Dequeue $v \leftarrow Q$, push v into S
    foreach *neighbor* $w \in \texttt{adj}[v]$ **do**
      if $\texttt{dist}[w] < 0$ **then**
        $\texttt{dist}[w] \leftarrow \texttt{dist}[v] + 1$
        Enqueue w
      if $\texttt{dist}[w] = \texttt{dist}[v] + 1$ **then**
        $\texttt{sigma}[w] += \texttt{sigma}[v]$
        Append v to $\texttt{pred}[w]$
  Initialize $\delta[v] \leftarrow 0.0, \delta_2[v] \leftarrow 0$ for all $v \in V$
  while S *not empty* **do**
    Pop $w \leftarrow S$
    foreach *predecessor* $v \in \texttt{pred}[w]$ **do**
      $\texttt{contrib} \leftarrow \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
      $\delta[v] += \texttt{contrib}$
      $\delta_2[v] += \delta_2[w] + 1$
    if $w \neq s$ **then**
      $\texttt{betweenness}[w] += \delta[w]$
      $\texttt{stress}[w] += \delta_2[w] \cdot \sigma[w]$
Divide all $\texttt{betweenness}[v]$ and $\texttt{stress}[v]$ values by 2 (for undirected graph symmetry)
Return maximum values of betweenness and stress arrays

2.2 Identifying Optimal Missing Links

- **Baseline Centrality Computation:** Initial values of betweenness centrality and stress centrality are computed for the graph.
- **Identification of Candidate Edges:** All missing edges (i.e., pairs of nodes not currently connected) are identified as potential additions to the network.
- **Iterative Edge Evaluation:** For each edge candidate,

  - Missing edge is temporarily added to the graph.
  - Centrality measures are recomputed with the edge included.
  - If a new minimum for either betweenness or stress centrality is found, it is recorded along with the corresponding edge.
  - The edge is then removed before evaluating the next candidate.
- **Final Selection:** After evaluating all missing edges, the edges that result in the lowest maximum betweenness and stress centrality are reported.

3 Algorithm Complexity Analysis

3.1 Time Complexity

Let V be the number of vertices and E be the number of existing edges in the input graph. Let M be the number of missing edges, i.e., $M = \binom{V}{2} - E$.

- **Single Centrality Computation (`g.compute()`)** The function is based on Brandes' algorithm and computes both betweenness and stress centrality. For an unweighted graph:

$$\text{Time complexity} = \mathcal{O}(V \cdot (V + E))$$

- **Missing Edge Enumeration (`g.findMissingEdges()`)** All unordered pairs of nodes not connected by an edge are collected:

$$\text{Time complexity} = \mathcal{O}(V^2)$$

  (assuming adjacency matrix check or linear scan per pair).

- **Main Evaluation Loop** For each missing edge:
  - Temporarily add the edge.
  - Run `g.compute()`.
  - Remove the edge.

  There are M such iterations. Each iteration costs $\mathcal{O}(V \cdot (V + E))$. Hence, total time complexity:

$$\mathcal{O}(M \cdot V \cdot (V + E)) = \mathcal{O}((V^2 - E) \cdot V \cdot (V + E))$$

- **Overall Time Complexity**

$$\mathcal{O}(V^3 \cdot (V + E) - V \cdot E \cdot (V + E))$$

  In the worst case (sparse graph), this simplifies to:

$$\mathcal{O}(V^4)$$

Conclusion: The time complexity of our algorithm is better than brute force $\mathcal{O}(V^5)$

4 Experimental Results

4.1 Testing and Validation

To validate the correctness of the implementation, Barabási-Albert graphs of varying sizes were generated. The results produced by the algorithm were cross-verified against those obtained from a simple brute-force approach.

4.2 Result Summary

Table 1: Centrality Minimization Results

Graph	**Min Betweenness**	**Edge (Betweenness)**	**Min Stress**	**Edge (Stress)**
Graph 1	3154.06	(2, 168)	12580	(12, 168)
Graph 2	1537.55	(8, 138)	7619	(37, 79)
Graph 3	2765.33	(8, 180)	14048	(8, 93)
Graph 4	2097.19	(17, 35)	8529	(17, 35)
Graph 5	2442.20	(0, 167)	12171	(12, 158)
Graph 6	1034.77	(9, 113)	5578	(4, 90)

4.3 Performance

Graph	**Time taken - parallel (sec)**	**Time taken - sequential(sec)**
Graph 1	33.98	108.68
Graph 2	11.92	39.31
Graph 3	41.97	133.80
Graph 4	9.79	46.36
Graph 5	28.64	126.17
Graph 6	10.56	42.41