**Betweenness Centrality (Stress Centrality)**

*-Mrudula Ranganatha (1002032382)*

*-Sakshi (1001993702)*

**Introduction:**

**Betweenness centrality variation Stress centrality** measures the absolute number of shortest paths that pass through any given node in the graph. For every pair of vertices in a connected graph, there exists at least one shortest path between the vertices such that either the number of edges that the path passes through (for unweighted graphs) or the sum of weights of edges (for weighted graphs) is minimized.

It applies to a wide range of problems in network theory, including problems related to biology, scientific cooperation, networks and transport by representing the degree to which nodes stand in between each other, potentially forming a bridge between hubs in the network. Hence a node with higher betweenness centrality would have more control over the network as more information will pass through that node.

**Problem Statement:**

Implement Betweenness Centrality's variation known as the Stress Centrality on undirected graph, using brute-force approach and compare the accuracy and efficiency with existing algorithms using any graph analysis package.

**Implementation:**

**Approach-1:**

Breadth first search (BFS) is used to find all the shortest paths that exists between each pair of vertices in the graph. While calculating the shortest paths there are two options, to include source and target node in the shortest path or exclude them. In our implementation, we have excluded the source and target node. This approach is very time consuming as it takes multiple iterations find all the shortest paths between each pair of vertices.

Number of iterations = (N-1)\*N

Where, N is the number of vertices.

**Approach-2:**

BFS is used to find all the shortest paths that exists between each pair of vertices in the graph. But single traversal is used to find the shortest paths from the source to all other vertices. This approach avoids duplicate work by reducing the number of BFS traversals to N. Thereby reducing the execution time.

Memory is optimized by keeping track of predecessors instead of entire path.

**Approach-3:**

Implementation of Multiprocessing and multithreading is done on BFS traversal. We leveraged both lightweights’ threads and subprocesses. This led to improved performance.

We noticed that performance was better with subprocesses compared to all the approaches implemented. Multiprocessing makes more sense here since it is a CPU intensive task (not an IO task).

**Experimental Evaluation:**

**Data sets used:**

* We used the Stanford datasets repository to benchmark our algorithm.
* We also created a utility function to randomly generate a graph with a high density.
* The maximum graph size tested is 5000V, 104508E.

*Consider the following unweighted graph:*

A picture containing building

Description automatically generated

**Implementation of Approach-1:**

Approach-1 uses BFS to find all the shortest path between a source and target node. We store these paths in memory and then calculate the betweenness stress centrality by incrementing the betweenness centrality count of each vertex in the path discovered.

This approach makes use of multiple traversals between source and target with (N-1)(N-2)/2 iteration, where N is the number of vertices in graph.

With this approach we noticed the performance degrade when working with larger datasets.

*Table-1: Workflow of Approach-1:*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| From | To | Path | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 2 | (1,2) | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 3 | (1,3) | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 4 | (1,5,4) | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 5 | (1,5) | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 6 | (1,5,6) | 0 | 0 | 0 | 0 | 1 | 0 |
| 2 | 3 | (2,1,3)(2,5,3) | 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 4 | (2,5,4) | 0 | 0 | 0 | 0 | 1 | 0 |
| 2 | 5 | (2,5) | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 6 | (2,5,6) | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 4 | (3,5,4) | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 5 | (3,5) | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 6 | (3,5,6) | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 5 | (4,5) | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 6 | (4,5,6) | 0 | 0 | 0 | 0 | 1 | 0 |
| 5 | 6 | (5,6) | 0 | 0 | 0 | 0 | 0 | 0 |
|  |  | SUM | **1** | 0 | 0 | 0 | **8** | 0 |

**Optimization:**

Instead of doing multiple traversals, a single traversal is done, which will find all the shortest path from one source to all other vertices. This reduced the iteration by N times.

After comparing the results and execution time of initial BFS (multiple traversals) implementation to the new optimized BFS (single traversal) we noticed a significant drop-in execution time and the graphs that have high density can execute.

**Implementation of Approach-2:**

Instead of traversing through the graph multiple times to find the shortest paths, we keep track of only predecessors. This technique implemented on initial version of BFS algorithm provided results quicker for high density graphs.

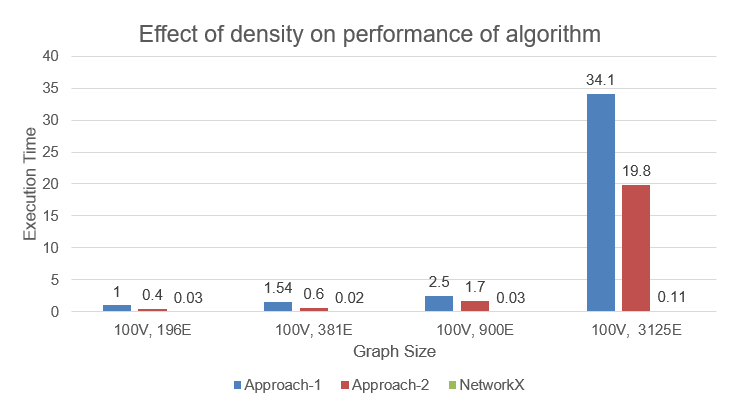
In addition, summation/counting of the occurrence of vertex v in all the possible paths between a source and target added latency. These computations are drastically reduced by accumulating partial sums of pair dependencies.

|  |  |  |
| --- | --- | --- |
| **Source** | **Predecessor list from source to target** | **Betweenness centrality** |
| 1 | P:{'1': [],  '3': ['1'] , '5': ['1'] , '2': ['1'] , '4': ['5'] , '6': ['5']} | {'1': 0.0,  '3': 0.0,  '5': 2.0,  '2': 0.0,  '4': 0.0,  '6': 0.0} |
| 3 | {'1': ['3'],  '3': [],  '5': ['3'],  '2': ['1', '5'],  '4': ['5'],  '6': ['5']} | {'1': 1.0,  '3': 0.0,  '5': 5.0,  '2': 0.0,  '4': 0.0,  '6': 0.0} |
| 5 | {'1': ['5'],  '3': ['5'],  '5': [],  '2': ['5'],  '4': ['5'],  '6': ['5']} | {'1': 1.0,  '3': 0.0,  '5': 5.0,  '2': 0.0,  '4': 0.0,  '6': 0.0} |
| 2 | {'1': ['2'],  '3': ['1', '5'],  '5': ['2'],  '2': [],  '4': ['5'],  '6': ['5']} | {'1': 1.0,  '3': 0.0,  '5': 7.0,  '2': 0.0,  '4': 0.0,  '6': 0.0} |
| 4 | {'1': ['5'],  '3': ['5'],  '5': ['4'],  '2': ['5'],  '4': [],  '6': ['5']} | {'1': 1.0,  '3': 0.0,  '5': 8.0,  '2': 0.0,  '4': 0.0,  '6': 0.0} |
| 6 | {'1': ['5'],  '3': ['5'],  '5': ['6'],  '2': ['5'],  '4': ['5'],  '6': []} | {'**1': 1.0,**  '3': 0.0,  '**5': 8.0,**  '2': 0.0,  '4': 0.0,  '6': 0.0} |

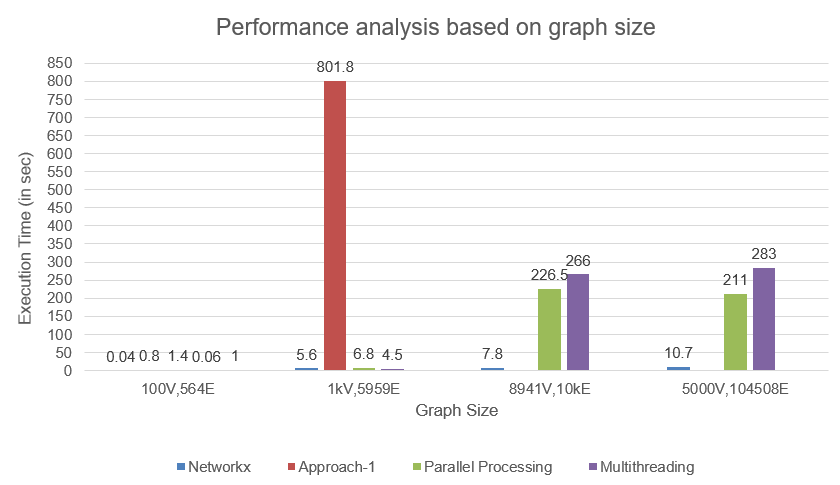
**Implementation of Approach-3:**

Further optimization was achieved to work on larger datasets by parallelizing the work to find the shortest paths and accumulating the betweenness within subprocesses. We experimented with different approaches – multi-threading vs multi-processing and tuning the number of workers.

**Experimental Results:**



* The above graph is plotted to analyse the effect of density of graph on the performance of our algorithm.
* As the density increases, the execution time also increases for all the algorithms.
* From the above graph, it is evident that Network-X outperforms Approach-1 and Approach-2.
* Due to reduced number of iterations, Approach-2 performs 40% better than Approach-1.



* The above graph is plotted to analyse the performance of each approach for various graph sizes.
* Approach-1 fails to execute for graph size 89941V, 10kE and above.
* Network-X outperforms all the implementations. But Multi-processor implementation performs better than all other implementations.
* For smaller graphs, multi-processing implementation takes more time as it requires initial setup.
* 6 node, 7 edge

Text

Description automatically generated

* 100 node and 194 edges  
  Text

  Description automatically generated
* 1000 node and 2021 edges

A picture containing text

Description automatically generated

**Benchmarking:**

Side by side execution time of our algorithm is compared with an existing Network-X library that uses the betweenness centrality method with the same graph input and realized Network-X produced better performance.

***betweenness\_centrality(graph, normalized = False, endpoints = False)***

**Outcome:**

* The betweenness centrality of nodes in the graph is calculated and compared with the outcome of Network-X to confirm the correctness.
* Graph density affected the performance of the algorithm greatly.
* Reducing the number of iterations improved the performance of the algorithm by 40% compared to initial version of our algorithm.
* Parallelization improved the performance of the algorithm greatly.

**Some real-world applications:**

* Finding most influential people in a social network (ex - Twitter).
* Finding most dominating accident locations w.r.t poor lighting conditions and bad roads.
* Finding highly popular/preferred co-actors for various genre combinations.

**Challenges Faced:**

* Deciding on the approach to improve the performance of initial version of our algorithm – We overcame this by going through multiple approaches and fixated on the approach that best suited our requirement.
* Finding the ground truth – As there are no existing library that calculate betweenness centrality using Stress Centrality method, we compared the top 20% of the vertices having highest betweenness centrality from our implementation and NetworkX outcome.

**References:**

* <https://en.wikipedia.org/wiki/Betweenness_centrality>
* <http://snap.stanford.edu/class/cs224w-readings/brandes01centrality.pdf>
* <https://networkx.org/documentation/stable/tutorial.html>