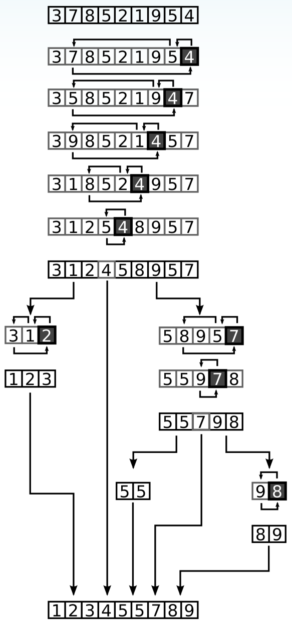
6.827:
Algorithm Engineering



A FASTER ALGORITHM FOR BETWEENNESS CENTRALITY

Julian Shun

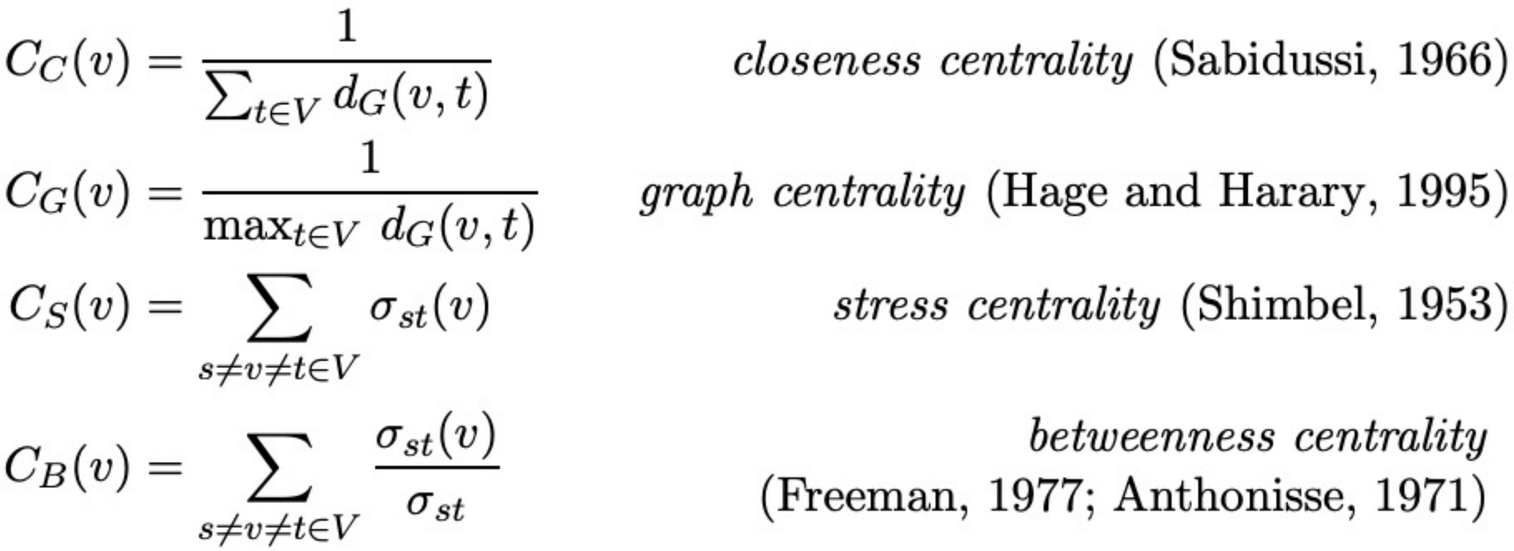
February 8, 2022

Material taken from “A Faster Algorithm for Betweeness Centrality” by Ulrik Brandes

Graph Centrality Indices

- Used to measure the important of vertices in a graph
- Applications
  - Influential actors in social networks
  - Key infrastructure nodes
  - Disease super-spreaders
  - Terrorism networks
  - Web page importance

Graph Centrality Indices



$$C_C(v) = \frac{1}{\sum_{t \in V} d_G(v,t)}$$

closeness centrality (Sabidussi, 1966)

$$C_G(v) = \frac{1}{\max_{t \in V}\ d_G(v,t)}$$

graph centrality (Hage and Harary, 1995)

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

stress centrality (Shimbel, 1953)

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

betweenness centrality
(Freeman, 1977; Anthonisse, 1971)

Betweenness Centrality

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

betweenness centrality
(Freeman, 1977; Anthonisse, 1971)

- **Pair dependency** of vertex v on vertices s and t is the ratio of shortest paths between s and t that go through v:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$$

σ_{st} is the number of shortest paths from s to t
$\sigma_{st}(v)$ is the number of shortest paths from s to t that go through v

$$\sigma_{st}(v) = \begin{cases} 0 & \text{if } d_G(s,t) < d_G(s,v) + d_G(v,t) \\ \sigma_{sv} \cdot \sigma_{vt} & \text{otherwise} \end{cases}$$

Betweenness Centrality

- Betweenness centrality of vertex v is computed as

$$C_B(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v).$$

- Traditional approach for computing betweenness centrality:
  - Compute all pairs shortest paths
  - Sum all pair dependencies
- This takes $\Theta(n^3)$ time and $\Theta(n^2)$ space

Goal of this paper

- We would like to reduce the time to $O(nm)$ for unweighted graphs and $O(nm + n^2 \log n)$ for weighted graphs, and reduce the space to $O(n + m)$

- For sparse graphs ($m << n^2$), this gives an improvement over the traditional approach

Shortest-Path Counting

- Define **predecessors** of v on shortest paths from s:

$$P_s(v) = \{u \in V \,:\, \{u,v\} \in E,\, d_G(s,v) = d_G(s,u) + \omega(u,v)\}\,.$$

Lemma 3 (Combinatorial shortest-path counting) *For* $s \neq v \in V$

$$\sigma_{sv} = \sum_{u \in P_s(v)} \sigma_{su}\,.$$

- Shortest paths can be counted using modification of Dijkstra’s algorithm (weighted) or breadth-first search (unweighted)
  - This takes O(nm) time for unweighted graphs and O(nm + n^2log n) time for weighted graphs

Summing Pair Dependencies

- The remaining bottleneck is to sum pair dependencies

$$C_B(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v).$$

- Naïve approach would take $\Theta(n^3)$ time since there are $\Theta(n^3)$ triples of vertices in the sum

Accumulation of Pair Dependencies

- Dependency of a vertex s on a vertex v is defined as

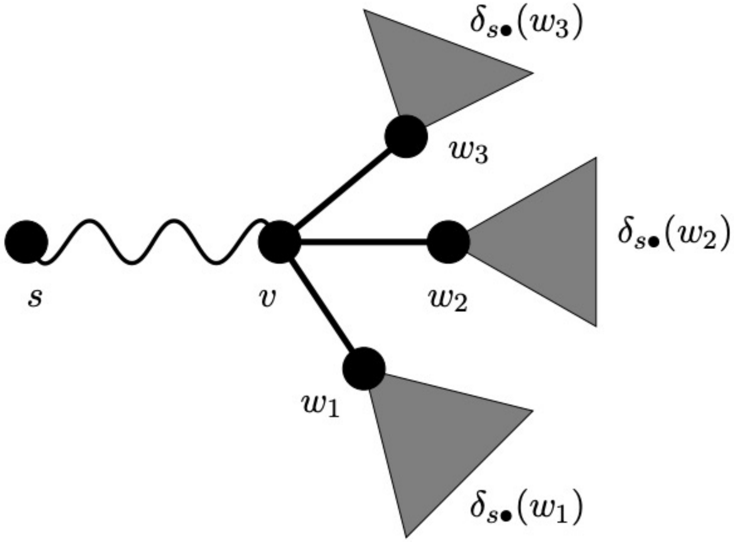
$$\delta_{s\bullet}(v) = \sum_{t \in V} \delta_{st}(v).$$

Accumulation of Pair Dependencies

- Special case for shortest paths from s that form a tree:

Lemma 5 *If there is exactly one shortest path from* $s \in V$ *to each* $t \in V$*, the dependency of* s *on any* $v \in V$ *obeys*

$$\delta_{s\bullet}(v) = \sum_{w\,:\,v \in P_s(w)} \left(1 + \delta_{s\bullet}(w)\right).$$

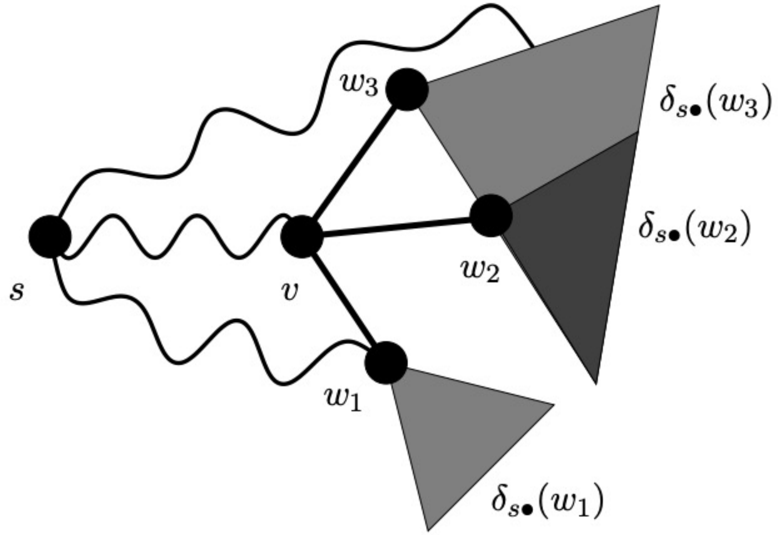


Accumulation of Pair Dependencies

- General case:

Theorem 6 *The dependency of $s \in V$ on any $v \in V$ obeys*

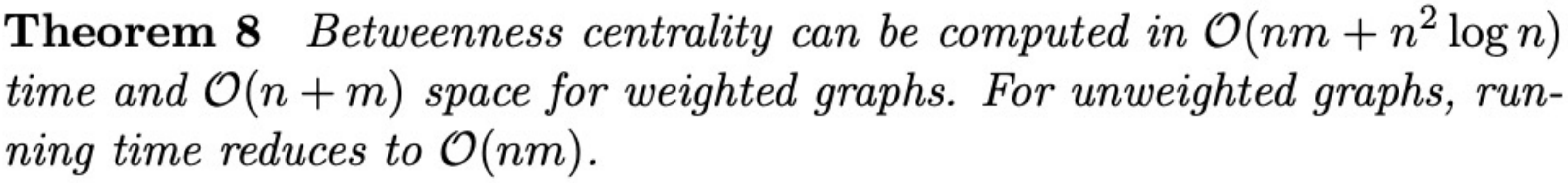
$$\delta_{s\bullet}(v) = \sum_{w\,:\,v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w))\,.$$



Accumulation of Pair Dependencies

- Dependencies of a vertex s on all other vertices can be computed using a modified single-source shortest paths algorithm
  - Traverse vertices in non-increasing order of distance from s and accumulate dependencies using previous formula

- Combined with the counting of shortest paths, we have:



Theorem 8 *Betweenness centrality can be computed in* $\mathcal{O}(nm + n^2 \log n)$ *time and* $\mathcal{O}(n + m)$ *space for weighted graphs. For unweighted graphs, running time reduces to* $\mathcal{O}(nm)$.

Experiments on Random Graphs

- Sun Ultra 10 SparcStation with 440 MHz clock speed

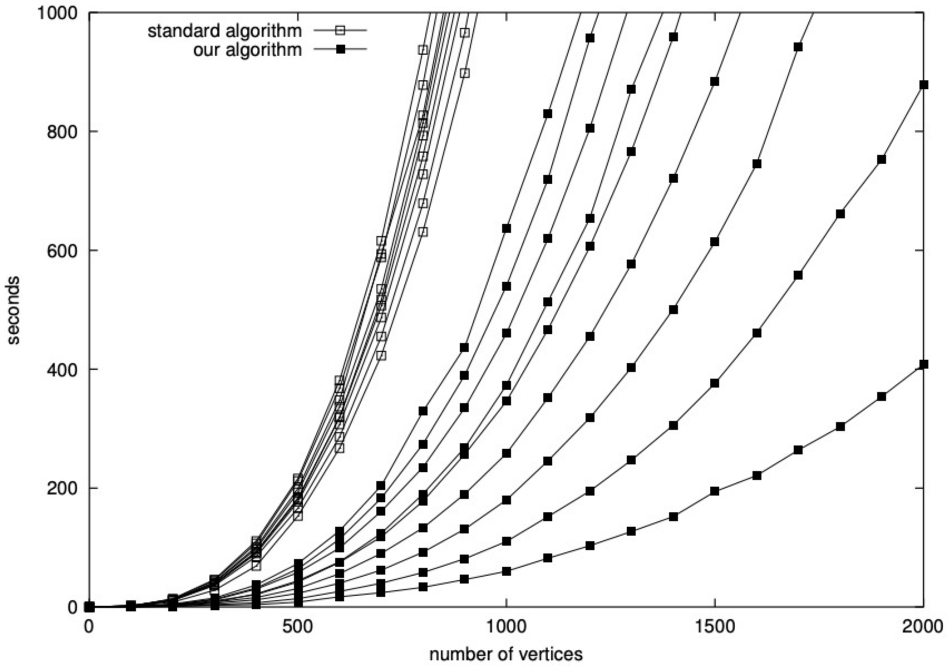


Figure 3: Seconds needed to the compute betweenness centrality index for random undirected, unweighted graphs with 100 to 2000 vertices and densities ranging from 10% to 90%

- The new algorithm significantly outperforms the standard algorithm, and its running time grows more slowly vs. data size

Experiments on Random Graphs

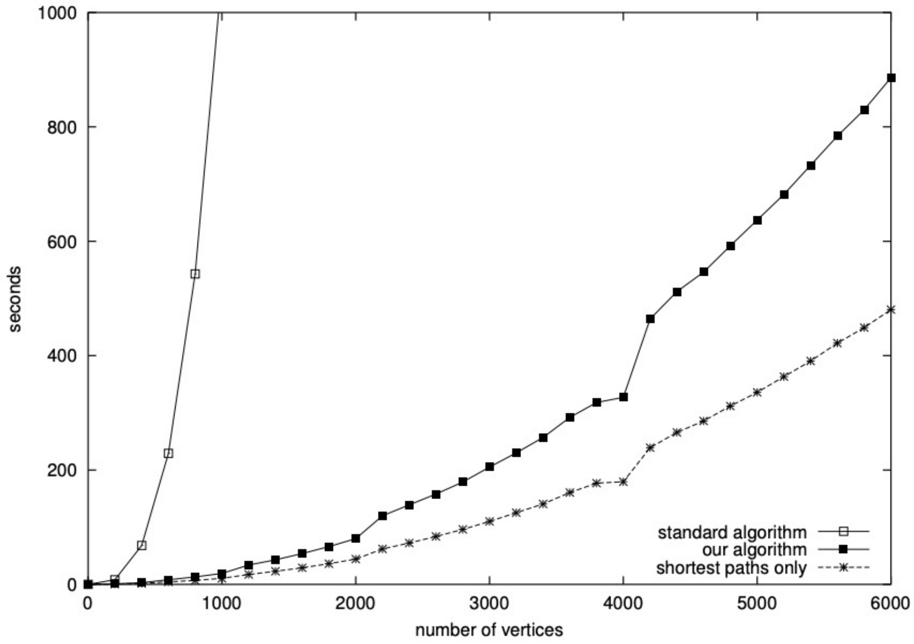


Figure 4: Seconds needed to the compute betweenness centrality index for random undirected, unweighted graphs with constant average degree 20. The funny jumps are attributed to LEDA internals

- Random graphs with fixed average degree of 20
- Again, the new algorithm is again faster than the standard algorithm

Strengths/Weaknesses

- Asymptotically faster algorithm with significant speedups in practice
- Applicable to other centrality measures
- Not much performance engineering discussed
    - No discussion of parallelism, locality, and constant-factor optimizations

Discussion

- Direction optimization is applicable here
  - However, early break doesn't work
- Parallelized version is similar to parallel BFS
  - Use fetch-and-add instead of compare-and-swap
- This algorithm still takes quadratic work, and cannot scale to the much larger graphs that we see today
  - To scale to large graphs, many approximate betweenness centrality algorithms have been proposed