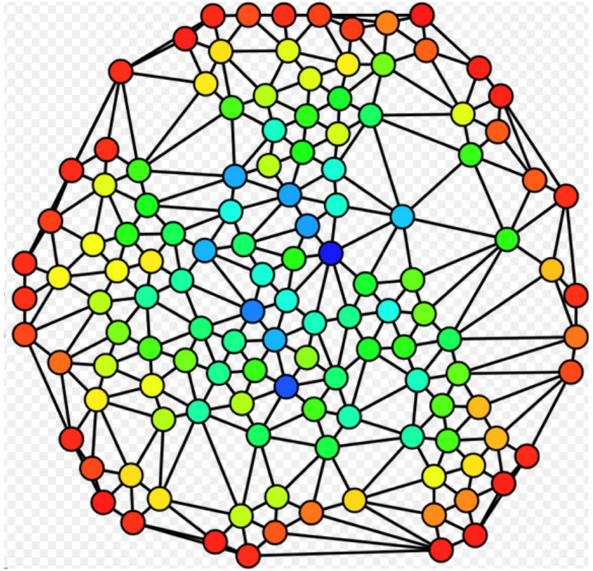
A Faster Algorithm For Betweenness Centrality

Paper by Ulrik Brandes, University of Konstanz
Brief 6.886 Presentation by Taylor Andrews
Tuesday, February 12th 2019

Agenda

- Background and Problem Motivation (3 min)
  - Social network scope
- The Brandes Algorithm (22 min)
  - **Unweighted graphs in O(nm) runtime and O(n + m) space**
  - Weighted graphs in O(nm + n^2logn) runtime and O(n + m) space
- Experimental Results (1 min)
  - Processing Synthetic and Real-World Datasets
- Questions and Discussion (4 min)

What is "Centrality" ?



- In graphs, **centrality** can imply:
  - Importance
  - Influence
  - General well-connectedness
- "Betweenness" is a measure of **centrality**
  - based on shortest paths
- First formalized by a Sociologist (Freeman 1977)

Claudio Rocchini https://commons.wikimedia.org/wiki/File:Graph_betweenness.svg

Extreme Centrality in Social networks

- Celebrities = Social Outliers

Centrality Of Most People (Vertices)

- **Adult mean of 338, median of 200**
- “**2.32 billion monthly active users** on Facebook as of December 31, 2018” [Facebook.com]
  - Only 15% have more than 500 friends
- Notice social networks are still sparse

(Pew Research Center)

http://www.pewresearch.org/fact-tank/2014/02/03/what-people-like-dislike-about-facebook/

Previous "Betweenness" Centrality Challenges

- Well-known $\Omega(n^3)$ bottleneck to compute betweenness centrality (Freeman 1977; Anthonisse, 1971)
- Prohibitive when V > few hundred :(
- Linkage-based approximation (Everett et al., 1999)
- Brandes saw an opportunity, and for exact calculations

- Leverage social sparsity (from the common case)

Introducing The Betweenness Centrality Algorithm

- Betweenness centrality values shortest path influence [Freeman, 1977; Anthonisse 1971]
- Leverage traversal algorithms (BFS)
- Two Major Steps
  - First major step in algorithm figures out all shortest paths
  - Second final step of using them to accumulate “dependency”

The Betweenness Centrality Algorithm: Only 2 Steps

- The first major step is calculating all shortest paths for all vertices, introducing our first construct:

  lowercase sigma σ

- The second step is calculating all “pair-dependencies” introducing our second construct:

  lowercase delta δ

The Algorithm In Code

(We will return to this)

STEP 1: Shortest Path Accumulation

σ

STEP 2: Pairwise Dependency Accumulation

δ

Algorithm 1: Betweenness centrality in unweighted graphs

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V;   σ[s] ← 1;
    d[t] ← −1, t ∈ V;   d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + σ[v]/σ[w] · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

The Betweenness Centrality Algorithm Definition

(draw example 1)

- Betweenness Centrality definition: for a vertex v, for each starting and final vertex (s and t), sum the following important measurement: the number of shortest paths between each s and t that cross through an intermediary v, divided by the number of total shortest paths between each s and t
- This important measurement is called a “pair-dependency”, and betweenness centrality requires calculating it for all pairs of other vertices

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

betweenness centrality
(Freeman, 1977; Anthonisse, 1971)

The Betweenness Centrality Algorithm: Constructs

- Accumulate pair-dependencies to get ans:

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

betweenness centrality
(Freeman, 1977; Anthonisse, 1971)

- For top half, “number of shortest paths between s and t passing through v”

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

- Pair-dependency also gets its own construct:

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

The Algorithm In Code: Step 1

(We will return to this)

STEP 1: Shortest Path Accumulation

σ

STEP 2: Pairwise Dependency Accumulation

δ

Algorithm 1: Betweenness centrality in unweighted graphs

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V;   σ[s] ← 1;
    d[t] ← −1, t ∈ V;   d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + σ[v]/σ[w] · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

First Step: Counting Number of Shortest Paths

- Recall first construct, lowercase sigma σ
- Originally accomplished with algebraic path counting
  - Calculates number of paths of length shorter than network diameter
  - Matrix multiplications were dominating factor
  - Excess calculations; for betweenness centrality we only need number of shortest paths between each pair of vertices
- Recall opportunity in sparsity of social networks
  - Count shortest paths with traversal: BFS (or Dijkstra if weighed)

First Step: Formalizing Predecessors (draw example 2)

- Create formal definition of a predecessor:

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

- Read (informally) as the set of vertices where one last edge (each {u, v}) connects each predecessor (each u) with the successor v, where the weight of the last edge added to the distance from s to each predecessor equals the total distance from s to the successor.

First Step: Predecessor Relationship

- Predecessors allow us to work toward recursive solution thanks to separating out a successor and final edge

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

- Examine number of shortest paths to predecessors, and notice it is the same as num shortest paths to successor
- Remember, lowercase sigma σ is number of shortest paths

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

when

$$d_G(s, u) < d_G(s, v)$$

First Step Final Thoughts Before Step 2

- Corollary 4 provides bounds on finding shortest paths for all vertices using BFS and Dijkstra's as **O(nm)** and O(nm + n^2logn)
  - Run traversal n times (from n vertices)

- Now we will see Brandes' innovation for step 2

The Algorithm In Code: Step 2

(We will return to this)

STEP 1: Shortest Path Accumulation

σ

STEP 2: Pairwise Dependency Accumulation

δ

Algorithm 1: Betweenness centrality in unweighted graphs

$C_B[v] \leftarrow 0, v \in V$;
for $s \in V$ **do**
  $S \leftarrow$ empty stack;
  $P[w] \leftarrow$ empty list, $w \in V$;
  $\sigma[t] \leftarrow 0, t \in V$; $\sigma[s] \leftarrow 1$;
  $d[t] \leftarrow -1, t \in V$; $d[s] \leftarrow 0$;
  $Q \leftarrow$ empty queue;
  enqueue $s \rightarrow Q$;
  while *Q not empty* **do**
    dequeue $v \leftarrow Q$;
    push $v \rightarrow S$;
    foreach *neighbor w of v* **do**
      // w found for the first time?
      if $d[w] < 0$ **then**
        enqueue $w \rightarrow Q$;
        $d[w] \leftarrow d[v] + 1$;
      end
      // shortest path to w via v?
      if $d[w] = d[v] + 1$ **then**
        $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$;
        append $v \rightarrow P[w]$;
      end
    end
  end
  $\delta[v] \leftarrow 0, v \in V$;
  // S returns vertices in order of non-increasing distance from s
  while *S not empty* **do**
    pop $w \leftarrow S$;
    for $v \in P[w]$ **do** $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$;
    if $w \neq s$ **then** $C_B[w] \leftarrow C_B[w] + \delta[w]$;
  end
end

Second Step: Sum All Pair-Dependencies

- Recall the second step of accumulating pair-dependencies gives us

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

betweenness centrality
(Freeman, 1977; Anthonisse, 1971)

- Top half: "number of shortest paths between s and t passing through v"

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

- Single pair-dependency shortened to: $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$

Need accumulation for all shortest paths for each v

Second Step: Sum Pair-Dependencies...

- Introduce new notion of **“dependency”** to simplify all pairwise-dependencies of vertex s to vertex t each v
- Recall that second construct (pair-dependency delta δ) is created from first construct (number of shortest path counting sigma σ)

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} \quad \Longrightarrow \quad \delta_{s\bullet}(v) = \sum_{t \in V} \delta_{st}(v)$$

Second Step: Sum Pair-Dependencies…Recursively!

- This is the novel predecessor relationship leveraged while performing BFS / Dijkstra on these sparse graphs
- The general case is covered by Theorem 6

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

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

Proving Important Theorem 6 (Briefly)

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

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

- Extend pair-dependency to include an intermediary edge e {v,w} (from predecessor v to successor w) as well as vertex v

$$\delta_{s\bullet}(v) = \sum_{t \in V} \delta_{st}(v) = \sum_{t \in V} \sum_{w : v \in P_s(w)} \delta_{st}^{(e)}(v, \{v, w\}) = \sum_{w : v \in P_s(w)} \sum_{t \in V} \delta_{st}^{(e)}(v, \{v, w\})$$

- Let w be any vertex with v as a predecessor. Of the shortest paths from s to w, many first go from s to v and then use {v, w}. The ratio times the shortest paths from s to t that go through successor w equal the number of shortest paths that contain {v, w} and v

Proving Theorem 6 (Briefly) Using Predecessor Edge

- Theorem 6 Proof Continued

$$\delta_{st}(v,\overset{(e)}{\{v,w\}}) = \begin{cases} \frac{\sigma_{sv}}{\sigma_{sw}} & \text{if } t = w \\ \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \frac{\sigma_{st}(w)}{\sigma_{st}} & \text{if } t \neq w \end{cases}$$

Inserting this into $\delta_{st}(v,\overset{(e)}{\{v,w\}}) = \sum_{w\,:\,v\in P_s(w)} \sum_{t\in V} \delta_{st}(v,\overset{(e)}{\{v,w\}})$

$$\sum_{w\,:\,v\in P_s(w)} \sum_{t\in V} \delta_{st}(v,\overset{(e)}{\{v,w\}}) = \sum_{w\,:\,v\in P_s(w)} \left(\frac{\sigma_{sv}}{\sigma_{sw}} + \sum_{t\in V\setminus\{w\}} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \frac{\sigma_{st}(w)}{\sigma_{st}} \right)$$

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

The Algorithm In Code: The Final Answer

(Final Time)

for $v \in P[w]$ **do** $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$;
if $w \neq s$ **then** $C_B[w] \leftarrow C_B[w] + \delta[w]$;

Theorem 6 updates dependency for predecessors by examining successor, and accumulates B.C.

STEP 1: Shortest Path Accumulation

STEP 2: Pairwise Dependency Accumulation

Algorithm 1: Betweenness centrality in unweighted graphs

$C_B[v] \leftarrow 0, v \in V$;
for $s \in V$ **do**
  $S \leftarrow$ empty stack;
  $P[w] \leftarrow$ empty list, $w \in V$;
  $\sigma[t] \leftarrow 0, t \in V$; $\sigma[s] \leftarrow 1$;
  $d[t] \leftarrow -1, t \in V$; $d[s] \leftarrow 0$;
  $Q \leftarrow$ empty queue;
  enqueue $s \rightarrow Q$;
  while *Q not empty* **do**
    dequeue $v \leftarrow Q$;
    push $v \rightarrow S$;
    foreach *neighbor w of v* **do**
      // w found for the first time?
      if $d[w] < 0$ **then**
        enqueue $w \rightarrow Q$;
        $d[w] \leftarrow d[v] + 1$;
      end
      // shortest path to w via v?
      if $d[w] = d[v] + 1$ **then**
        $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$;
        append $v \rightarrow P[w]$;
      end
    end
  end
  $\delta[v] \leftarrow 0, v \in V$;
  // S returns vertices in order of non-increasing distance from s
  while *S not empty* **do**
    pop $w \leftarrow S$;
    for $v \in P[w]$ **do** $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$;
    if $w \neq s$ **then** $C_B[w] \leftarrow C_B[w] + \delta[w]$;
  end
end

Synthetic and Real Dataset Experimental results

tions currently in use. The experiment was performed on a Sun Ultra 10 SparcStation with 440 MHz clock speed and 256 MBytes main memory.

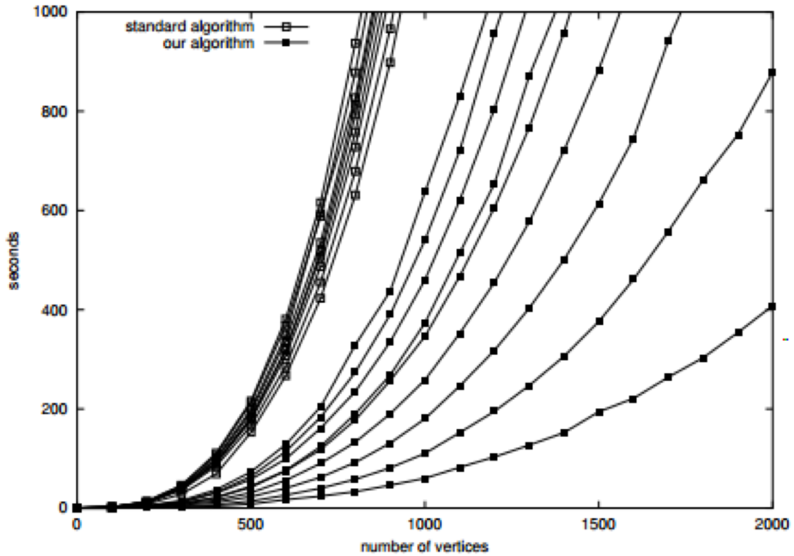


Figure 3: Seconds needed to the compute betweenness centrality index for random undirected, unweighted graphs with 100 to 2000 vertices and densities ranging from 10% to 90%

The speed-up was also validated in practice, by analysis of an instance of 4,259 intravenous drug users with 61,693 directed weighted links, originating from 197,216 unique contacts.[3] Not only because of running time, but also because of the memory required to store the distance and shortest-paths count matrices, betweenness centrality could not be evaluated for this network to date. The largest subnetwork previously analyzed had 494 actors with 1,774 links (taking 25 minutes on a 200 MHz Pentium Pro PC). Our implementation determined the betweenness centrality index of the whole network in 448 seconds, using less than 8 MBytes of memory.

- Computed previously uncomputable dataset in 448 seconds, using less than 8MB of memory

Basic Unweighted Algorithm Extension

- Corollary 7 shows DAG extension (but not as relevant to social network scope)
- Theorem 8 extends Theorem 6 for weighted graphs (Dijkstra)
- Undirected graphs require Betweenness Centrality scores divided by two
    - Why?
    - Shortest paths are considered twice

Supports Other Centrality Measures (Limited Explanation)

- Claims single-source shortest-path traversal supports "easy computation" of other shortest path centrality measures
- But only a one sentence explanation

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

(Valente and Foreman, 1998)

$$C_R(v) = \frac{\sum_{t \in V} (D(G) + 1 - d_G(v,t))}{(n-1) \cdot D(G)}$$

Questions and Discussion

- Some discussion questions:

- Results from Sparc 440 Mhz/256MB, 200MHz Pentium Pro, 450MHz Pentium III
  - What could be further optimized given modern hardware?

- Paper claims all “standard centrality indices based on shortest paths can … be evaluated simultaneously.”
  - What parts of the code would be strategically reused?
  - What would be the challenges?

- Paper algorithm code does not explicitly show shortest path calculations in parallel.
  - Do we think it is possible? Why or why not?

Thank you For Listening and Participating

- Paper "A Faster Algorithm for Betweenness Centrality" by Ulrik Brandes (University of Konstanz)

- Presented by Taylor Andrews
    - tandrews@mit.edu