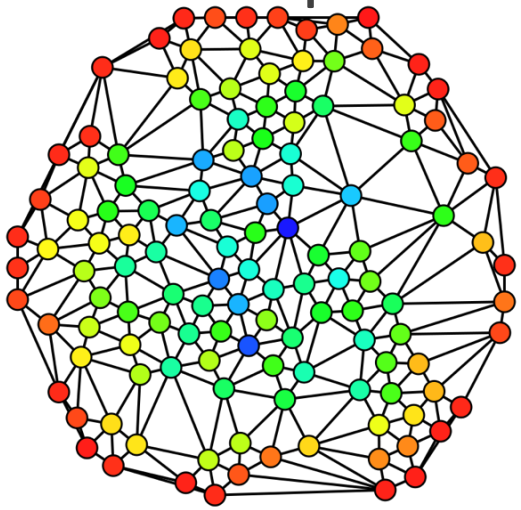
Betweness Centrality

ENDRIAS KAHSSAY

What is it?

- The centrality of a vertex v is the fraction of the shortest paths that go through v.
- A measure how important a vertex is in a Graph.



An undirected graph colored based on the betweenness centrality of each vertex from least (red) to greatest (blue).

Applications

- Finding nodes where most of the information flows through
- - Identifying hotspots in Network

- Identifying key actors in a social network
- - Terrorist network

- Biology
- - Identifying proteins that are good targets for medicine

Definition

- Let G = (V, E, ω) be a graph with node set V = V (G), edge set E = E(G) where w > 0
- σ_{st} denote the number of shortest path from s and t.
- $\sigma_{st}(v)$ denotes the number of shortest paths going through v from s and t

Let $C_B(v)$ denote the betweeness centrality of v.

More Definition

- Let d(s,v) be the weight of shortest path from s to v.

- Define Ps(v) to be the vertices that lie on the shortest path from s to v and have a direct Edge to v.

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

Computing Betweness Centrality

Basic algorithm to compute $C_B(v)$ for all V:

1) compute the length and number of shortest paths between all pairs.

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

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

can be calculated through modified SSSP Dijkstra algorithm which runs

$O(nm + n^2 \log n)$.

Computing Betweness Centrality

2) Computing CB(v) can be done by summing the ratio of shortest paths going throuhg v.

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

This is Expensive! There are O(n) s,t,v, so this takes $O(n^3)$

Not feasible for large graphs

BA Algorithm For Betweness Centrality

Achieves Θ(n(m+n log n)) for weighted graphs which is the cost of computing single-source shortest paths (SSSPs).

let P(v) = $\{v : (v, w) \in E \wedge d(s, w) = d(s, v) + \omega(v, w)\}$

P(v) represents the vertices who have outgoing edges to v, and lie on the shortest path from S to v.

$$\sigma_{sw} \text{ as } \sum_{v \in P_s(w)} \sigma_{sv}.$$

Dynamic Betweness Centrality

Real world graph topologies change often.

In the social network example:

New people join, people leave, add other people as friends etc.

The update might only affect small parts of the graph.

Recomputing even with BA is quadratic!

Ibet – a new dynamic algorithm

- It takes inspiration from BA which is the fastest algorithm for computing Betweness Centrality on a static graph.

- Also from KWCC, KDB which are dynamic centrality algorithms.

- Primary Contribution:

  A faster algorithm for updating pairwise distances

  A faster algorithm for updating the centrality of nodes

Ibet

Algorithm handles edge additions or weight changes.

Vertex addition can be handled by adding the edges of the vertex one by one.

Ibet Pseudo Code

Let the edge added be (u,v).

1) Identify the set of source vertices that are affected by (u,v).

$$\{s \in V : d(s,t) > d'(s,t) \vee \sigma_{st} \neq \sigma'_{st}\}$$

2) Identify the target vertices that are affected by (u,v).

$$s \in V \text{ as } T(s) := \{t \in V : d(s,t) > d'(s,t) \vee \sigma_{st} \neq \sigma'_{st}\}$$

Ibet

3) Update the distances and number of shortest paths efficiently.

→ This will only change for vertices that are part of the SSSP dags rooted in the affected vertices.

→ There is a high overlap between the SSSP's of the affected vertices.

→ The algorithm efficiently avoids recomputing overlaps using a BFS like algorithm.

Ibet Pseudo code

4) Change the betweness centrality of affected nodes by accumulating contributed of each affected node once.

→ Identify the old contribution of nodes whose old shortest paths from s went through v, but which have been affected by the edge insertion.

→Identify the new contribute of those nodes.

→ Subtract old contribution and add new contribution.

RunTime

Step 1 and 2 (augmenting the ASSP results) :

$$\Theta(||S(v)|| + ||T(u)|| + \sum_{y \in T(u)} |S(p(y))|),$$

Step 3 and 4:

$$\Theta(||\tau(s)|| + |\tau(s)| \log |\tau(s)|)$$

Comparison of Ibet with KDB and KWCC

KDB only works on unweighted graphs.

KWCC goes through the same edges of affected targets multiple times even though only going through one is sufficient which leads to a worst case O(VE) in the accumulation phase.

KDB looks at more nodes in updating APSP data structures as it doesn’t isolate the nodes for which SSSP that have changed.

Experiments

Implemented BA, KDB, KWCC, IBET in C++.

All code was sequential.

Graphs are undirected and unweighted.

Multiple real world networks considered of size < 26000.

Performance

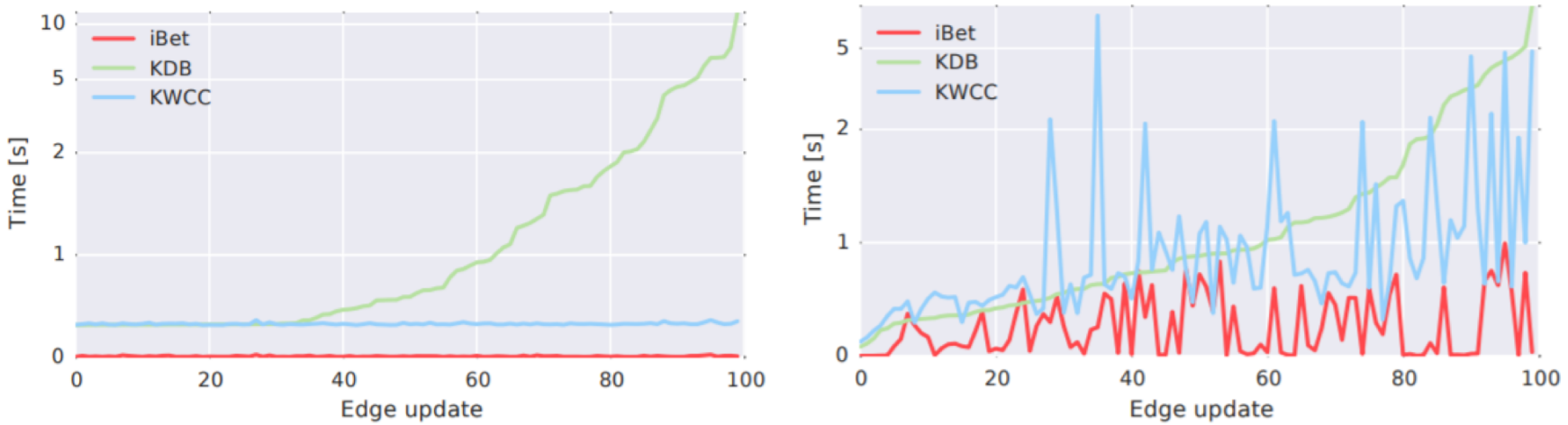


Figure 3 Running times of iBet, KDB and KWCC for 100 edge updates on `oregon1-010526`. Left: imes for the APSP update step. Right: times for the dependency update step.

Ibet outperformed BA by 179x, KDB by and KWCC of 13 and 22.9 respectively.

Limitations:

- In the worst case, a full recomputation is needed.
- High memory overhead of $O(n^2)$ makes it infeasible for large graphs (has to store d(s,t) for all s,t.
- Doesn't handle negative edge weights, edge deletions.
- Test data only shows the algorithm for relatively small graphs that are not weighted.

Future Work:

- Parallelizing the algorithm.
- Improving the memory footprint.
- Batching edge updates.

Related work:

Approximation algorithms for dynamic graphs

Only computing the K highest betweness scores in a dynamic graph.

Thanks for listening!