CS 598: Communication Cost Analysis of Algorithms

Lecture 14: Betweenness centrality, sample sort, and the mergesort algorithms of Cole and Goodrich

Edgar Solomonik

University of Illinois at Urbana-Champaign

October 10, 2016

Graph centrality

The problem of betweenness centrality is a close derivative of APSP

- for each vertex, a centrality score gives the number of shortest paths that go through it
- if the number of shortest paths between vertices s, t is $\sigma(s, t)$ and the number of these that go through vertex v is $\sigma_v(s, t)$, the betweenness centrality score is

$$\lambda(v) = \sum_{s,t \in V} \sigma_v(s, t)/\sigma(s, t)$$

- note that $\sigma_v(s, t) = \sigma(s, v) \cdot \sigma(v, t)$ if $d(s, t) = d(s, v) \cdot d(v, t)$
- this problem is important in analysis of biology, transport, and social network graphs
- it also has some interesting algorithmic solutions...

Computing betweenness centrality (BC)

There are two major alternatives for computing BC

- we can modify BFS/Dijkstra/Bellman-Ford/Floyd-Warshall/path-doubling to keep track of path multiplicities
- whenever we take a minimum of weights of different paths, we want to add multiplicities if their weight is the same
- algebraically, this can be interpreted as a 'geodetic' semiring, where the set of values is a 'geodesic' - a tuple containing the weight and the multiplicity
- the simplest algorithm is then to compute APSP along with $\sigma(s,t)$ and then compute

$$\lambda(v) = \sum_{s,t\in V, d(s,v)\cdot d(v,t)=d(s,t)} \sigma(s,v)\cdot\sigma(v,t)/\sigma(s,t)$$

  which is no harder than matrix multiplication, for a cost no greater than APSP

Brandes' algorithm for BC

However, [Brandes 2001] proposed a method that forgoes computing a dense distance matrix

- his method was based on SSSP, utilizing BFS or Dijkstra
- compute $d(s,:)$ and $\sigma(s,:)$ for a given s
- then consider partial centrality factors $\zeta(s,v)$ such that

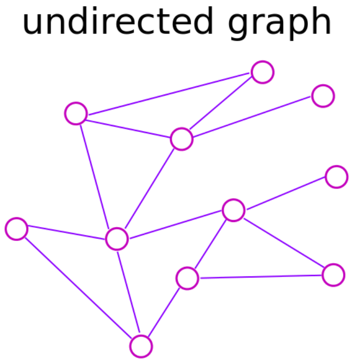
$$\zeta(s,v) = \sum_{t \in V, d(s,v) \cdot d(v,t) = d(s,t)} \sigma(v,t)/\sigma(s,t)$$

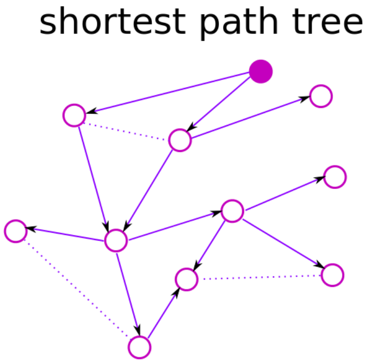
- from $\zeta(s,v)$ we can construct the centrality scores via

$$\lambda(v) = \sum_{s} \sigma(s,v) \cdot \zeta(s,v)$$

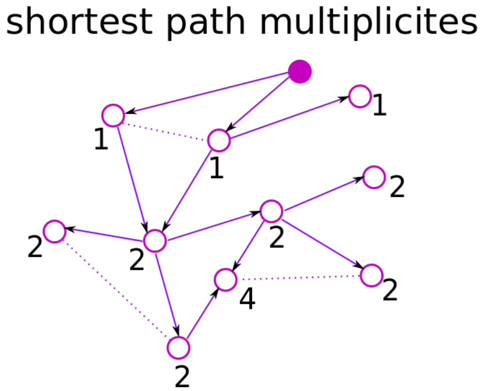
- however, we need some knowledge of $\sigma(v,t)$ and for all v such that $d(s,v) \cdot d(v,t) = d(s,t)$

Shortest path tree





Shortest path tree multiplicities



Partial centrality factors

So, what does $\zeta(s,v) = \sum_{t \in V, d(s,v)\cdot d(v,t) = d(s,t)} \sigma(v,t)/\sigma(s,t)$ represent?

- the shortest path tree from s, is induced by a partial ordering $<_s$ on V, where $v <_s t$ if $d(s,v) \cdot d(v,t) = d(s,t)$
- so for any $v \in V$, the set of vertices t such that one of the shortest paths from s to t passes through v is

$$\Pi_s(v) = \{t : t \in V, v <_s t\} = \{t : t \in V, d(s,v) \cdot d(v,t) = d(s,t)\}$$

- we can now write $\zeta(s,v) = \sum_{t \in \Pi_s(v)} \sigma(v,t)/\sigma(s,t)$
- the subset of vertices in $\Pi_s(v)$ connected to v by one edge is

$$\begin{aligned} \pi_s(v) &= \{u : u \in V, v <_s t, \nexists z, v <_s z <_s t\} \\ &= \{u \in V, w((s,v)) \cdot d(v,u) = d(s,u)\} \end{aligned}$$

- graphically, we can represent the shortest path tree as $T_s = (V, E_s)$ where $(v,u) \in E_s$ if $u \in \pi_s(v)$

Integrating partial centrality factors

So, how can we compute $\zeta(s,v) = \sum_{t \in \Pi_s(v)} \sigma(v,t)/\sigma(s,t)$?

- Q: for any leaf $l \in V$ in the shortest path tree T_s what is $\Pi_s(l)$?
- A: $\Pi_s(l) = \emptyset$, and furthermore $\zeta(s,v) = 0$
- now, for any node x whose children are all leaves ($\pi_s(x) = \Pi_s(x)$),

$$\zeta(s,x) = \sum_{t \in \pi_s(x)} 1/\sigma(s,t)$$

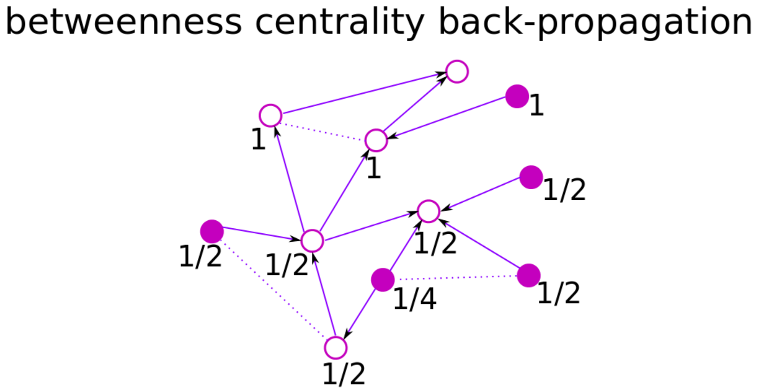
- more generally, for any node v and $t \in \Pi_s(v) \setminus \pi_s(v)$ each shortest paths from v to t must go through some node in $\pi_s(v)$, so

$$\sigma(v,t) = \sum_{p \in \pi_s(v)} \sigma(p,t)$$

- furthermore, we have $\sigma(v,t)/\sigma(s,t) = \sum_{p \in \pi_s(v)} \sigma(p,t)/\sigma(s,t)$
- and then it follows

$$\zeta(s,v) = \sum_{p \in \pi(s,v)} \left(\frac{1}{\sigma(s,p)} + \zeta(s,p) \right)$$

Centrality factors in shortest path tree



Brandes' algorithm for BC

The given relationship allows partial centrality factors to be collected by a scheme that looks like reverse BFS

- once all children have their partial centrality factors, the parent can compute theirs
- the leaves of the shortest path tree immediately know their centrality factors $\zeta(s, v) = 0$
- the parents collect contributions from all of the shortest paths that go through them to other nodes
- this can again be written as sparse-matrix sparse-vector multiplication
- since we know the shortest path tree, we need to 'relax' each edge no more than once
- therefore, for each starting vertex, computing centrality scores from multiplicities has the same bandwidth and computation cost as BFS
- however, it can have a somewhat greater latency cost, because the number of SpMSpVs depends on the depth of the shortest path tree, which can be greater than the graph diameter

Communication-efficient BC

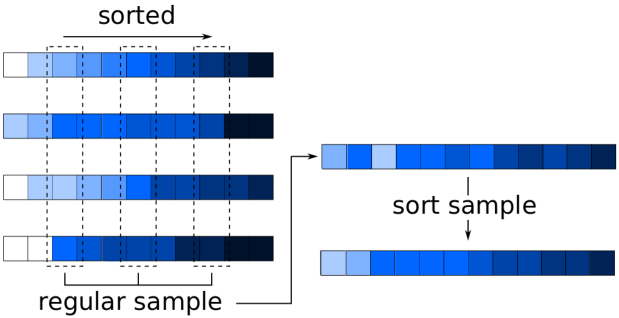
We can perform Brandes' algorithm communication-efficiently by doing many SSSPs at a time

- given $M = O(n^2/P)$ memory, we should just do APSP with an efficient algorithm
- generally we can expect to have $M = O(c|E|/P)$ memory, so we can store $c \geq 1$ copies of the graph
- assuming that $c|E| \leq n^2$, we can perform $O(c|E|/n)$ SSSPs at a time
- each SSSP is an independent BFS or Bellman-Ford execution
- instead of SpMSpV, we will now have SpMSpM, which is more communication-efficient
- for instance, we can make c copies of the graph, and perform an SSSP (SpMVs) with each set of P/c processors
- for undirected graphs, obtain an interprocessor bandwidth cost of

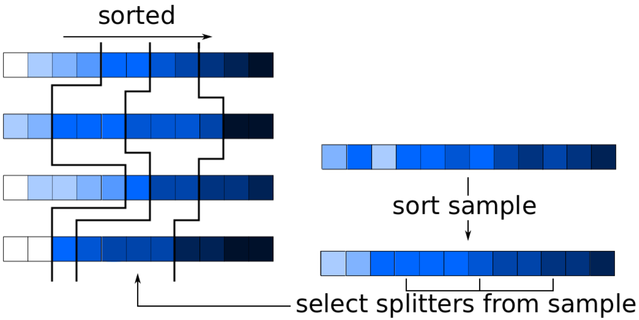
$$O\left(\frac{n^2}{\sqrt{cP}} + \frac{n\sqrt{|E|}}{P^{2/3}}\right)$$

Short pause

Parallel sorting with regular sampling



Parallel sorting with regular sampling



Why is regular sampling guaranteed to work?

Given a sample of size $P(P-1)$, no part will contain more than $2n/P$ elements

- we select $P-1$ splitters from sorted sample of size $P(P-1)$
- n/P^2 elements are between two subsequent sample elements from any processor
- if there are k_j elements from processor j between splitter i and $i+1$, it has at most $(k_j+1)n/P^2$ elements between these splitters
  - if distribution on each processor roughly the same $k_j \approx 1$
  - if distribution is uneven, we can have $k_j \gg 1$, but can bound total $\sum_{j=1}^{p} k_j = P$
  - therefore, the total number of elements in each interval is bounded by $\sum_{j=1}^{p}(k_j+1)n/P^2 \leq 2n/P$

Communication-cost of sample sort

Lets consider the communication cost of sample sort in more detail

- if the sample is of size less than the size of the partition we need, we can just sort it sequentially then partition
- for a BSP algorithm we want $s = P$ partitions, for a cache-efficient sort we want $s = n/H$
- s-way partitioning is load balanced given a sample of size s^2
- so, if $s^2 = O(n/s)$, i.e. $s = O(n^{1/3})$, naive sample sort works fine, because its cheaper to sort the sample than a partition
- if $n < s^3$, sorting the sample sequentially might be too expensive
- we can sort the sample recursively, unless $n < s^2$, when the sample is larger than the original problem

Parallel cache-oblivious sample sort

Sample sort can be adapted to be parallel and cache oblivious [Blelloch, Gibbons, Simdhavi 2010]

- sort $n^{1/3}$ subsequences of $n^{2/3}$ elements recursively in parallel
- collect regular sample of size $n^{1/3}$ from each subsequence, a total size of $n^{2/3}$ and sort it
- select $n^{1/3}$ splitters from the sample
- merge splitters with subsequences to compute offsets (e.g. by cache-oblivious bitonic merge)
- reorder data by cache-oblivious transpose on $n^{1/3} \times n^{1/3}$ matrix of subsequences (each of length $n^{1/3}$ on average)
- sort $n^{1/3}$ subsequences of $O(n^{2/3})$ elements recursively in parallel