Centrality

make some notes about the centrality indexes. answer the following questions:

1. what is that
2. what is the background/ideas/application of that
3. how to impletement it

contents

What is Centrality

It is the description about a element of a network and it describe how important the element is. And its specific definition is depended on the context.

It roots in the field of social network analysis, and it is used to describe how important a person is in the social network. Basically, the more central the person is in the network, the more important it is.

Basic example

Let's say the network is a friendship network, that is to say if two nodes are friends, then they share a edge. And we assume that the more friend a person has, the more import it is. For example, if someone is ill and lay in the hospital and its friend will visit it. A important person of course have many visitors. We can estimate the number of visitors by counting its degree. In formula, it is:

$C(v) = d(v)$

where *C* stands for the centrality, *d* for degree and *v* for vertex.

And this is called degree centrality, a most simple one. And it is one kind of **Structural Index** for the reason that it is just the same for the same vertex no matter how you draw the graph.

Centralities based on shortest path

Motivation

If it is only 5 minutes before the class start and you don't want to be late for the class, then you may probably choose a shortest route to attend the class in time.

It is often observed that the the shortest the path is, the more users go on it. And sometimes, things may only be loaded on the fastest/shortest way, for example, the router almost always put the datagram into the best way it knows in the route table which captures the shortest way.

It is reasonable to assume that, shortest path has something to do with the importance of its vertexes and edges.

Some Indexes

Those indexes is adaptable to non-shortest-path-based indexes, or we can say they are generic because they provide the ideas of measuring the centrality in difference context.

- Eccentricity

  The longest length of the shortest paths started with the vertex.
- Closeness Centrality

  The sum over the length of the shortest paths ended/started with the vertex.
- Centroid Value
- Stress Centrality

  The numbers of shortest paths passed over that vertex.
- Betweenness Centrality

  The sum over the fractions of the shortest paths passed over that vertex.
- Reach
- Traversal Set

Algorithm

Let's look at the betweenness centrality, its definition is:

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

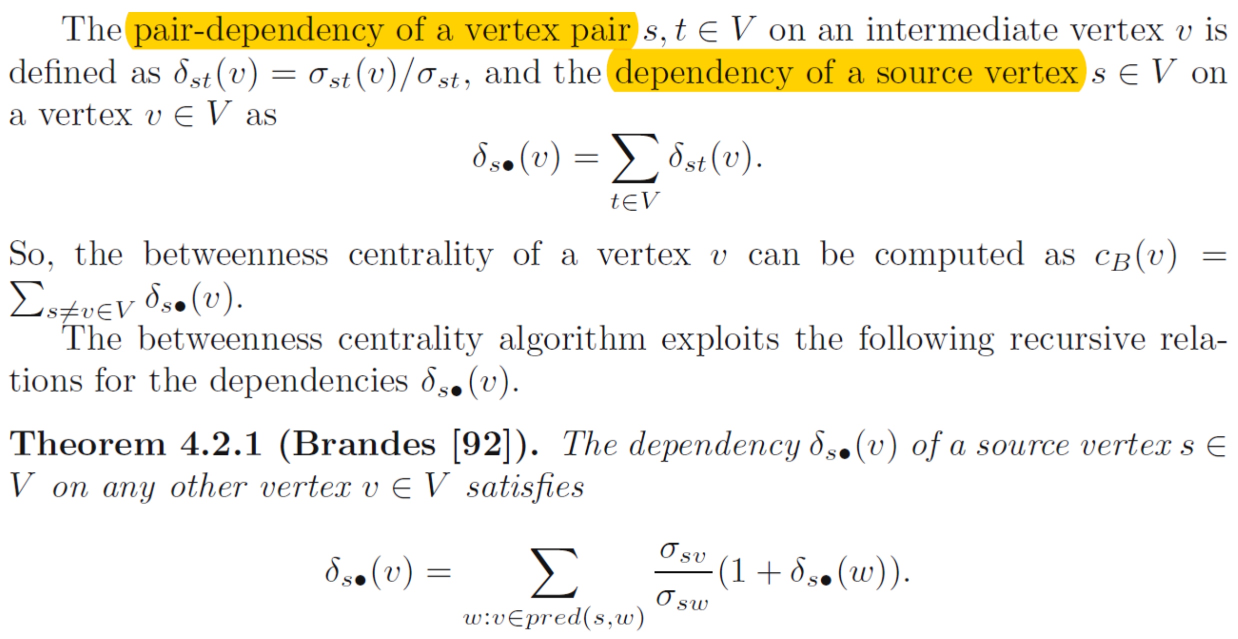
and σ_{st} stands for the number of shortest path between vertex *s* and *t*, and $\sigma_{st}(v)$ stands for the number of the shortest path that pass through the vertex *v*.

We have two ways to calculate this simple centrality:

- derive the centrality from its definition directly.
- derive the centrality by exploiting some properties of these centrality.

The first approach has two phases. First, Uses SSSP algorithm to calculate the shortest paths for every nodes and record the predecessor sets, then we can get the number of shortest paths between every pair of vertexes, and these can be done by modifying Dijkstra's algorithm ; Second, we can fix a pair of vertexes and find out that how many path passing a particular vertex by verifying the predecessor sets, and we sum over it, and these also can be done by scanning the predecessor table. And if you implement these naive approach, you will be grateful for Dijkstra, for his algorithm guarantee you a lot of conveniences.

The second approach exploits this discovery:



The pair-dependency of a vertex pair $s, t \in V$ on an intermediate vertex v is defined as $\delta_{st}(v) = \sigma_{st}(v)/\sigma_{st}$, and the dependency of a source vertex $s \in V$ on a vertex $v \in V$ as

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

So, the betweenness centrality of a vertex v can be computed as $c_B(v) = \sum_{s \neq v \in V} \delta_{s\bullet}(v)$.

The betweenness centrality algorithm exploits the following recursive relations for the dependencies $\delta_{s\bullet}(v)$.

Theorem 4.2.1 (Brandes [92]). *The dependency $\delta_{s\bullet}(v)$ of a source vertex $s \in V$ on any other vertex $v \in V$ satisfies*

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

Thus, we can calculate **the dependency of a source vertex** recursively, and we just need to modify the second step of the first naive approach into these recursive style. For the fact that, the dependency of a source vertex and be recorded and don't have to compute again, then the efficiency it better than the first approach, just like the time we save by storing the Fibonacci numbers into an array. For more optimizations, we use shortest path tree to guide the orders by which we calculate the dependency of a source vertex. We computer the dependency of a source vertex from the leaves of the shortest path tree backward to the root --- s, which has the value we exactly aim to get. The reason for such order hides in Theorem 4.2.1(especially, " $w : v \in pred(s, w)$"), and first kind of values we can get is the value of the leaves, which always yields an integer, and the second kind of values we can get the must be the value of a node that its out edge only connect to the leaves.

And what can we do with the dependency of a source vertex? Here is the useful equation:

$C_b(v) = \sum_{v \neq s \in V} \delta_{s\cdot}(v)$

To fasten these fastened algorithm, the idea of **random approximative algorithm** can be applied. It is that when we spanning a shortest path tree, instead of selection all the shortest path pairs, we select only a subset of it, and then calculate the betweenness centrality the same way. It is mathematically proved that it can reduce the time to call the SSSP form O(n) to O(log(n)) with an user-defined acceptable error rate.

Centralities based on Vitality

> It would be empty without me.

> --- *Without Me*, a famous rap song by Eninem

Motivation

"Is it vital?"

How do you answer this question?

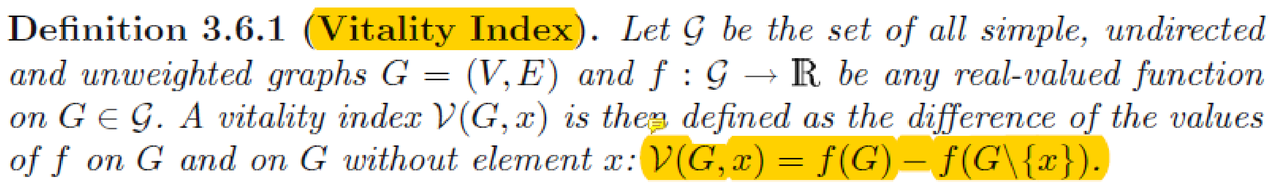
Or, let's ask, "Is drinking water vital to human being?"

A common answer would be "Of course it is vital, we would have already died deadly without water!"

Exactly, that is the way we put up with to measure the centrality of something. That is to say, Let's think about what would be like without such a element.

What would it be like without you? Well, you can image the lost, and discover that, only you are in some kind of social networks, then you would be of a little bit importance.

Definition



Definition 3.6.1 (Vitality Index). *Let $\mathcal{G}$ be the set of all simple, undirected and unweighted graphs $G = (V, E)$ and $f : \mathcal{G} \to \mathbb{R}$ be any real-valued function on $G \in \mathcal{G}$. A vitality index $\mathcal{V}(G, x)$ is then defined as the difference of the values of f on G and on G without element x:* $\mathcal{V}(G, x) = f(G) - f(G \backslash \{x\})$.

Some indexes

- shortcut value

  Maximum increase in distance between any two vertices if e = (u, v) is removed from the graph. It is not fit in the definition of vitality index, but the idea behind it is similar.

Algorithms

Let's look at shortcut value of all the edges.

There is also two approaches, the naive brute force one and the one that exploits some properties.

First, the naive method is pretty simple, just call the ASSP on every edge, and do a element-level subtraction and select the maximum difference. It takes m times ASSP.

The complex one has a similar idea to the one that calculates the betweenness centrality. It exploits a recursive property of the network. And I think the reason why these two algorithms are all recursive is **shortest paths recursion**, in fact, if we want to know the shortest distance from *s* to *t* in a recursive style, we need to recursive find out all the shortest distances from *s* to all the nodes that point to *t*. That is a recursive property of the shortest path.

The complex algorithm defines three variables in order to express the recursion we can exploit. Let's fix a vertex *u*, and we want to find out all the shortcut values of the edge out of *u*.

α_i : the distance of the plan A, which means if no edge is removed, the distance we need to travel to vertex *i* from *u*.

τ_i: the first vertex we would meet if we travel as plan A.

β_i: the plan B, which means if we cannot travel as plan A due to τ_i is removed and unreachable, and we need to find out another fastest plan to go to vertex *i*, and just as α_i, the β_i is the distance if we travel as plan B.

Notes that:

τ_i can be $\perp$, which means that in plan A, we can go the that way first or another way first, because they have the same distance and would travel as the same fast.

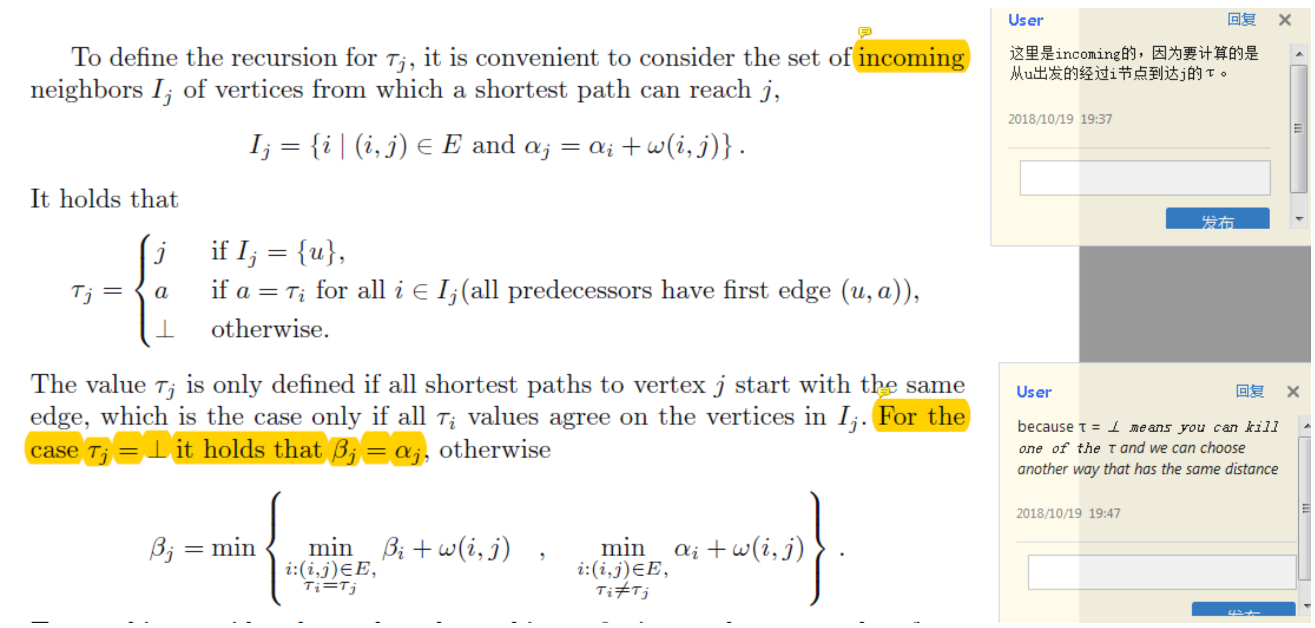
β_i can be ∞, because sometimes, there is no feasible plan B:

Then, here comes the recursion:

The Bottom Condition :

$\alpha_u = 0, \quad \tau_u = \emptyset, \quad \beta_u = \infty$

The Recursive Expression :



To define the recursion for τ_j, it is convenient to consider the set of incoming neighbors I_j of vertices from which a shortest path can reach j,

$$I_j = \{i \mid (i,j) \in E \text{ and } \alpha_j = \alpha_i + \omega(i,j)\}\,.$$

It holds that

$$\tau_j = \begin{cases} j & \text{if } I_j = \{u\}, \\ a & \text{if } a = \tau_i \text{ for all } i \in I_j (\text{all predecessors have first edge } (u,a)), \\ \perp & \text{otherwise.} \end{cases}$$

The value τ_j is only defined if all shortest paths to vertex j start with the same edge, which is the case only if all τ_i values agree on the vertices in I_j. For the case $\tau_j = \perp$ it holds that $\beta_j = \alpha_j$, otherwise

$$\beta_j = \min \left\{ \min_{\substack{i:(i,j)\in E,\\ \tau_i = \tau_j}} \beta_i + \omega(i,j) \quad , \quad \min_{\substack{i:(i,j)\in E,\\ \tau_i \neq \tau_j}} \alpha_i + \omega(i,j) \right\}.$$

User 回复 ✕

这里是incoming的，因为要计算的是从u出发的经过i节点到达j的τ。

2018/10/19 19:37

发布

User 回复 ✕

because τ = ⊥ means you can kill one of the τ and we can choose another way that has the same distance

2018/10/19 19:47

Here I don't want to write the equation myself, and my explanations are attached on the right to help understand better. The mathematical expressions are quite straightforward and need no explanation. And I think my analogue (β as plan B) helps a lot to understand the last expression, it is exactly the way you looking and the map and find out another fastest way, and this expression share the same idea as the shortest path recursion.

And we can see that the recursion expression of β is similar to the Dijkstra's execution. Let's imagine an animation that show the value of plan A and plan B are being returned back after the bottom condition is hit, and we will see a similar pattern as the Dijkstra's, that is, spanning out as a tree.

So the complexity of the algorithm is n times SSSP.