**Reconstructing collapsed complex network to find most influential nodes**

Fengkuangtian Zhu

September 13, 2017

Solution of Datacastle Master Competition

Team ID: zhfkt

**Abstract**

This solution paper is to demonstrate the solution of algorithm in identifying vital nodes in complex networks for Datacastle Master Competition . Compared with traditional algorithms including Betweenness[bet], Closeness[clo], Degree[ciheap], PageRank[page] and Collective Influence[ci] [ciheap], the new proposed algorithm of reconstructing collapsed complex network to find most influential nodes is able to achieve the better performance on the 8 competition datasets in terms of robustness[rb] and speed. The new implementation by c++ of the popular algorithm Collective Influence (CI) is introduced as well. All code in the paper can be found at the link <https://github.com/zhfkt/ComplexCi> [doicomp]

**Competition background[match]**

Disparate networks, including social networks, communication networks and biological networks, are playing an increasingly important role on natural and social-economic systems. A core problem, therein, is to measure the significance of individual nodes. For instance, a super spreader in HongKong triggered transmission of SARS to a significantly greater number of other people than 100 normal infected persons; a rumor re-tweeted by a celebrity may spread much broader than that by an obscure person.

Therefore it is necessary to develop a method to identify the virulence genes in large-scale gene regulatory networks, to find the super-spreaders in large-scale social networks, and to detect the key enterprises with serious systematic financial risk in large-scale financial networks.

Those tasks could be formalized as a generic challenge that is identifying vital nodes in networks that are important for sustaining connectivity. This challenge, aka optimal percolation, is a well-documented issue in network science. With great anticipation of making big progress on this problem, we successfully invited some experts and hope the great participants will create novel and effective solutions.

The competition provides 4 real networks from different fields, including autonomous system network, Internet network, road network and social network, and 4 classical artificial networks as well (total 8 datasets). The numbers of nodes of these networks range from 0.7 million to 2 million. All of them are considered as undirected networks. <TABLE 0> shows the network name and corresponding numbers of nodes

<TABLE 0>

**Introduction**

Overall, the target of finding most influential nodes algorithm is to give a ranking list of nodes according to their importance. The top-ranked nodes will have more importance. We can remove the nodes from the top-ranked ones in the ranking list generated by algorithm and calculate the size of giant component after each removal, which will break down the network into many disconnected pieces. The ratio of giant component will reach zero with the one-by-one removal operation finally. Therefore, the better algorithm, the sooner the network will collapse to the zero giant component with smaller count of provided nodes. Robustness[rb] is introduced in the Competition to quantify the performance of ranking nodes methods.

There are several traditional algorithms such as Betweenness[bet], Closeness[clo], Degree[ciheap], PageRank[page] and Collective Influence[ciheap] for the target. In this paper, the new algorithm is proposed for reconstructing collapsed complex network in order to find most influential nodes. Compared with the traditional algorithms, the new method is able to achieve the better performance in terms of robustness and speed.

This solution paper will be organized as follows. First, several traditional algorithms will be used to investigate the performance of 8 competition datasets as comparative algorithms. Next, the new reconstructing algorithm will be introduced and verified on datesets in performance. The verification will also include the process of searching parameters for the best result. Lastly, the conclusion will be given. All code in this paper can be found at the link <https://github.com/zhfkt/ComplexCi> [doicomp]

**Experiments on the current algorithms**

In this section, tradition algorithms of Betweenness[bet], Closeness[clo], Degree[ciheap], PageRank[page] and Collective Influence[ciheap] will be verified on 8 Datacastle datasets .

**Betweenness**

Betweenness is a centrality measure of a vertex within a graph . Betweenness centrality quantifies the number of times a node acts as a bridge along the shortest path between two other nodes[wikibet][bet] . Here Betweenness is used to find the most influential nodes in the complex networks. Robustness value after applying Betweenness on 8 competition datasets is shown in the <TABLE 1>

<TABLE 1>

8 datasets are all verified in parallel supported by GraphTools [gt] on the 4-core CPU machine (Intel Xeon E5-2667v4 Broadwell 3.2 GHz).

Although Betweenness algorithm is supported to be executed in utilizing all CPU cores in parallel by GraphTools [gt] to boost , it is still obvious that Betweenness spends nearly 24 days/ 1 month in completing the verification of 8 datasets. Considered the time and effort , robustness value of Betweenness is not very well. Betweenness fails to process the networks of million scale to find most influential nodes.

**Closeness**

In a connected graph, the Closeness centrality of a node is the average length of the shortest path between the node and all other nodes in the graph. Thus the more central a node is, the closer it is to all other nodes [wikiclo][clo] . Here Closeness is also used to find the most influential nodes in the complex networks. Robustness value after applying Closeness on 8 competition datasets is shown in the <TABLE 2>

<TABLE 2>

8 datasets are all verified in parallel supported by GraphTools [gt] on the <Tencent>. Pls notice that CPU cores in verifying Closeness are 8 cores, double than above CPU cores of machine verifying Betweenness

Similar with Betweenness , although Closeness algorithm is supported to be executed in utilizing all CPU cores in parallel by GraphTools [gt] to boost , it is still obvious that Closeness spends nearly 7 days/ 1 week in completing the verification of 8 datasets. Considered the time and effort , robustness value of Closeness is also not very well. Closeness fails to process the networks of million scale to find most influential nodes as Betweenness .

**PageRank**

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites [wikipage][page]. Here PageRank is also used to find the most influential nodes in the complex networks. Robustness value after applying PageRank on 8 competition datasets is shown in the <TABLE 3>

<TABLE 3>

8 datasets are all verified in single thread supported by GraphTools [gt] on the CPU (Intel Xeon E5-2667v4 Broadwell 3.2 GHz).

Compared with Betweenness and Closeness , algorithm PageRank gets much better result in robustness and speed. Especially , PageRank just run in single thread and uses less resources than Betweenness and Closeness. Robustness value of PageRank is 1.436 and also better than Betweenness and Closeness

**Degree**

Historically first and conceptually simplest is degree centrality, which is defined as the number of links incident upon a node . i.e. the number of ties that a node has.[degreewiki]. The nodes are ranked by degree, and sequentially removed starting from the node of highest degree. The concept of High Degree Adaptive (HDA) is purposed in the [ci] as a better strategy which is a slightly different from the original Degree algorithm. Degree of the remaining nodes in the adaptive version will be recomputed after each node removal. HDA is used here to find the most influential nodes in the complex networks. Robustness value after applying HDA on 8 competition datasets is shown in the <TABLE 4>

<TABLE 4>

8 datasets are all verified in concurrent on the 4-core CPU machine (Intel Xeon E5-2667v4 Broadwell 3.2 GHz). The time doesn't cover IO read/write from/to disk. Betweenness and Closeness are using multiple CPU cores for one dataset in parallel . However , experiments of HDA including below Collective Influence are using one CPU per each dataset and just start at the same time in concurrent.

We can see that the simple greedy algorithm of HDA performs well and effectively in getting million-scale networks using less than 30 seconds.

**Collective Influence (CI)**

Collective Influence (CI) algorithm using optimal percolation for localizing the minimal number of influential nodes is introduced in [ci][ciheap]. The problem of finding the minimal set of influencers can be mapped to the optimal percolation. As mentioned in [lv], CI will take effect for detecting most influential nodes guaranteeing the global connection of the network in terms of Robustness, which is also mainly suggested by the official challenge tips. It accepts “ball radius” as its input parameters, and the higher radius, the better result but more time and effort will be spent. Especially, if radius is set to zero, the Collective Influence will degenerate to HDA algorithm described above.

In [ci][ciheap], they also develop the reinsertion step, which is the post-process and refinement of CI algorithm. After the networks break down into many pieces through removing nodes using CI , reinsertion process will be called in the following steps from the original paper[ciheap] :

“*Reinsertion adds back one of the removed nodes, which is chosen such that, if once reinserted, it joins the smallest number of clusters. Reinsertion algorithm does not require that the reinserted node joins the clusters of smallest sizes, but only the minimum number of clusters, independently from their sizes. When the node is reinserted reinsertion also restores the edges with its neighbors which are in the network (but not the ones with neighbors not yet reinserted, if any). The procedure is repeated until all the nodes are back in the network. When implementing the reinsertion, Reinsertion add back a finite fraction of nodes at each step. ”* In their simulations they reinserted 0.2% of nodes at each step and a fraction smaller than 0.2% does not change the results.

In order to verify the CI on competition dataset results, I utilize 2 implementations of the algorithm. One is provided by the original paper written in c language [ccode\_ci], named CI\_HEAP. The other is newly developed in c++ implementation by myself [doicomp], named ComplexCi .

CI\_HEAP and ComplexCi share the same following internal parameters:

1. The start points of reinsertion in CI\_HEAP and ComplexCi are the same. Both start to reinsert the node when the size of giant component collapses to 1% in the whole network.
2. The finite fraction of nodes at each reinserted step in CI\_HEAP and ComplexCi are the same and both reinsert 0.1% for each step.

There are several following differences between CI\_HEAP and ComplexCi in implementing algorithm.

1. Compared with the initial CI purposed in [ci], CI\_HEAP boosts the algorithm by utilizing max-heap data structure for processing very efficiently the CI values. The computational complexity of CI will be O(N log N) when removing nodes one-by-one, made possible through an appropriate data structure to process CI.   
    My application ComplexCi uses red-black tree with STL (Standard Template Library ) SET as different data structure to store and update CI values. In the field of C++ programming , SET and MAP container in STL are usually implemented as red-black tree, which is a kind of self-balancing binary search tree. Average computational complexity of red-black tree in Searching, Inserting and Deleting are all O(log N). Compared with max-heap, though red-black tree doesn’t overcome performance in deleting and updating, red-black tree is still able to achieve O(N log N) in the overall computational complexity
2. When processing the reinsertion algorithm, CI\_HEAP uses basic statistic method to label the graph connected component indices, which is very time-consuming. Considered that the problem of deciding which node will be reinserted is invoked in several times in the reinsertion algorithm, we can reserve the information for each reinsertion and prepare it for the next decision, other than being forced to label the graph connected component indices of the reconstructing complex network halfway from the beginning.  
    ComplexCi uses disjoint-set data structure to store the graph connected component indices as the new reinsertion algorithm. There are 2 operations involved in the disjoint-set data structure , Find and Union. For the Find operation, we can use it to locate which connected component indices the node belongs to. For the Union operation, when the nodes are reinserted into the graph, it can help us to merge the arbitrary nodes into one connected component efficiently based on the previous reconstructed graph. The overall flow of the new reinsertion algorithm is that:   
   1. We have the initial collapsed complex network and build the corresponding disjoint-set data structure
   2. Then choose the left nodes to reconstruct the network.
      1. For each left node i, if once reinserted, “Find” operation will be used to select the connected component indices of its neighborhoods nodes one by one in the disjoint-set data structure. We mark the unique number of the connected component as Ni for this node i.
      2. Find the smallest Nx value among all left nodes Ni, we reinsert the corresponding node x into the network by the “Union” operation. It will update the connected component indices information in the disjoint-set data structure as well.
      3. Repeat to find the next reinsert node in step b until all left nodes are consumed.

Path Compression in Find and Union By Rank are usually the techniques to optimize the performance of disjoint-set data structure, which are also used here.

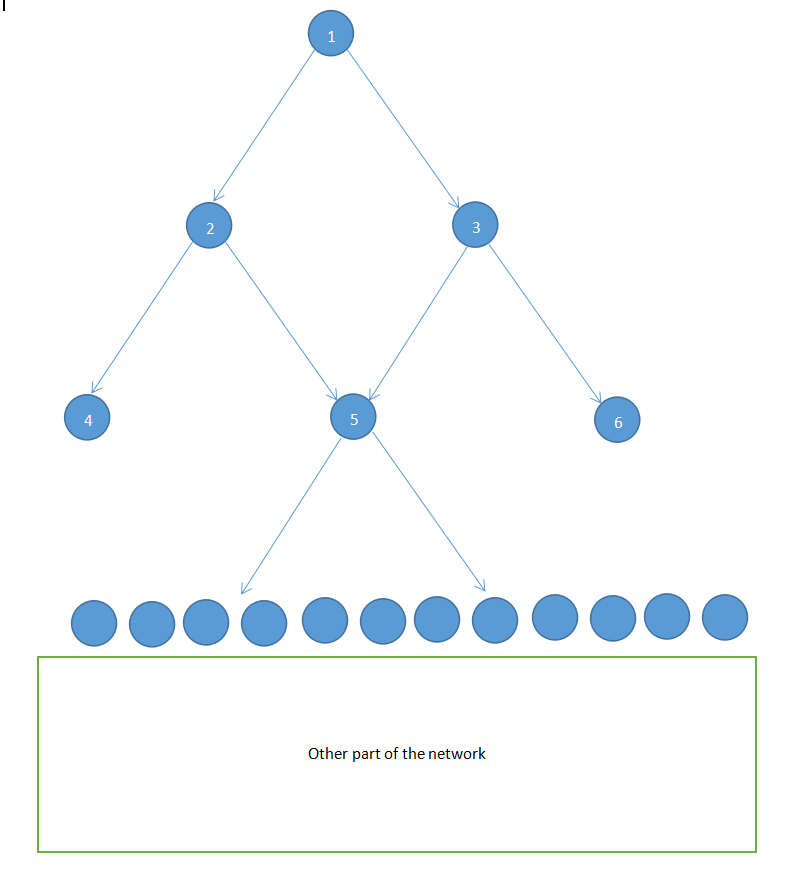
From the point of computational complexity, the original reinsertion in CI\_HEAP is (n+n)\*n=2n2=n2, which means (label graph connected component indices in n nodes network + find which node is suitable to be reinserted among n nodes) \* (repeat n times until all left nodes are consumed). New reinsertion algorithm in ComplexCi is (1+n\*reverse\_aka(n)+reverse\_aka(n)) \*n , which means (no need to label again + find which node is suitable to be reinserted among n nodes\* use FIND operation of disjoint-set pre node+UNION the final node and reinsert it into the graph) \* (repeat n times until all left nodes are consumed). Using both path compression and union by rank ensures that the amortized time per Find and Union operation is only reverse\_aka(n)[disjointanalysis1,2], which is optimal, where reverse\_aka(n) represents the inverse Ackermann function. This function has a value reverse\_aka(n)<5 for any very large value of n that can be written in this physical universe, so the disjoint-set operations take place in essentially constant time[disjointwiki]. The complexity (1+n\*reverse\_aka(n)+reverse\_aka(n)) \*n can be simplified to (1+n\*1+1)\*n=n2. Compared with the original reinsertion in CI\_HEAP , though the complexity of disjoint-set reinsertion is the same , the longest time-consuming time of labeling graph connected component indices is eliminated. The new algorithm just introduces the inverse Ackermann function, which is nearly constant time in exchange. There is another advantage of disjoint-set reinsertion that we can know the number of nodes in the arbitrary connected component in real-time because disjoint-set data structure supports to record the RANK value of each connected component. In the below experiment analysis, we can see that the disjoint-set strategy in ComplexCi performs more efficiently than original reinsertion algorithm in CI\_HEAP.

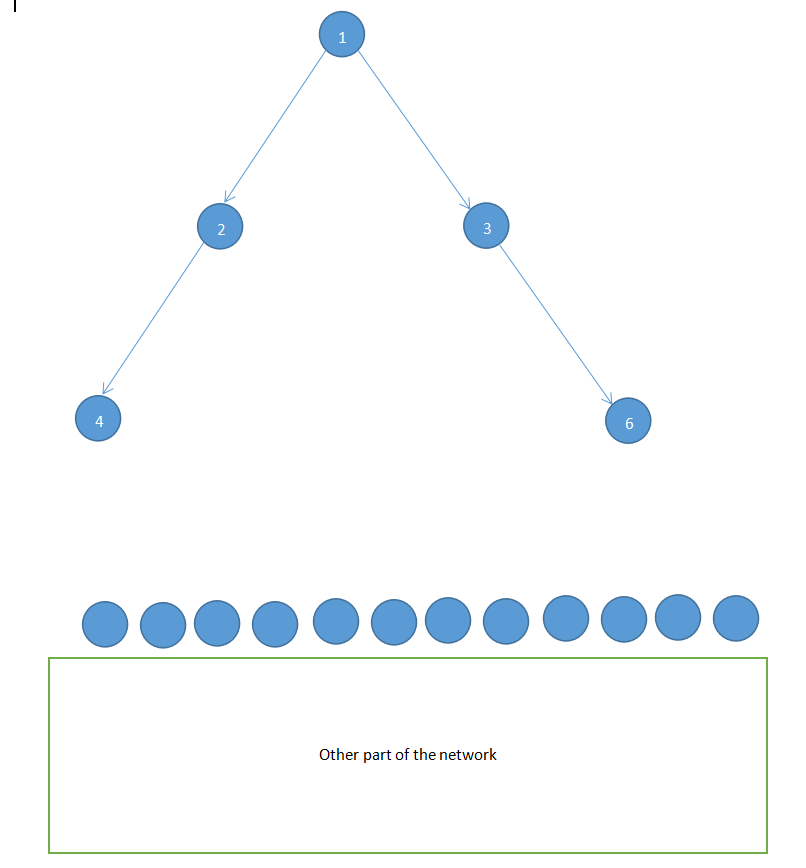
1. In the real , reinsert Batch . use nth\_element instead of sort

In addition, here I also would like to correct one saying in the [ci\_heap].

*The CI values of nodes on the farthest layer at ℓ + 1 are easy to recompute. Indeed, let us consider one of this node and let us call k its degree. After the removal of the central node its CI value decreases simply by the amount k − 1.*

Let’s give an example:





For the above network, I assume that ball radius is 1 and the next candidate removed node is node 5. Before the removal, CI value of node 1 is ((3-1)+(3-1))\*(2-1)=4. After removal, it will be changed to ((2-1)+(2-1))\*(2-1)=2. However, if we follow the saying of decreasing simply by the amount k − 1 for the ball radius l+1 (here is 2), CI value of node 1 will be 4-(k-1)=4-(2-1)=3 . I think [ci\_heap] made this mistake because they might assume there is only one shortest path from l+1 node to the removed node. In fact, from the example we can see that such assumption is not correct, there are 2 shortest paths from node 1 to node 5. Fortunately, after scanning their provided code, I found that they didn’t use this concept in their implementation and still calculated the CI value of l+1 node in the original formula.

This section describes the experiment on the ComplexCi and CI\_HEAP. Radius of 0,1,2 as input parameters is used to verify the performance on 8 competition datasets both for my c++ implementation and original c code. Datasets are all verified in concurrent on the 4-core CPU machine (Intel Xeon E5-2667v4 Broadwell 3.2 GHz). The time doesn't cover IO read/write from/to disk. The “minPoint” and “algoEndsPoint ” row are also involved into the result to evaluate the performance as brief reference. minPoint Pointsmin means/ Pointsmin2 means

<TABLE 5>

<TABLE 6>

<TABLE 7>

<TABLE 8>

<TABLE 9>

<TABLE 10>

8

It is also curious about the roles of reinsertion plays in the overall performance. What performance of result will be if we remove the reinsertion in the algorithm of CI ? I also try to verify the case for disabling reinsertion . Here are the results without reinsertion

<TABLE 11>

<TABLE 12>

<TABLE 13>

<TABLE 14>

<TABLE 15>

<TABLE 16>

From the experiments, we can observe that:

1. Consuming time is increasing according to the higher ball radius in both ComplexCi and CI\_HEAP. After debugging and profiling, I find that most time-consuming section is to find the nodes in the radius by using breadth-first-search (BFS) algorithm. We can not ignore BFS complexity if ball radius is high.
2. Use the same parameter and nearly the same result

Both can

* 1. Slight different maybe the delete history
  2. Slight different maybe the reinsert history

1. ComplexCI is better in speed without reinsertion in high radius
   1. Not sure the reason but maybe the red-black tree behaviour different with heap
   2. Maybe my bfs is better
2. ComplexCI is better in speed with reinsertion