KU LEUVEN kulak

Computer Science
Data Structures and Algorithms
jeroen.vanderdonckt@student.kuleuven.be

Report: Collective Influence Algorithm

Academic year 2016-2017

Jeroen Van Der Donckt

Introduction

The main goal of this project is to understand and implement the *Collective Influence Algorithm* by reading and understanding 2 papers. The first paper [1] introduces the problem and gives an idea for a solution, which is minimizing the largest eigenvalue of the non-backtracking matrix of the network. (Note: In this report 'the eigenvalue' references this largest eigenvalue of the non-backtracking matrix of the network). This is equivalent to reducing the giant component of the network by removing one by one the node with the largest CI value.

The second paper [2] gives an in-depth explanation of the linear time implementation. So the complexity of CI is $O(Nlog(N))$ with N the number of nodes. This requires an appropriate data structure to process the CI values, which shall be a simple max-heap.

1 Design

There are three classes used for the implementation (in C#) of this project, as you can see in the UML diagram in the appendix of this report. The first class is the class `Node`, this class is quite trivial. The constructor of this class will take 2 parameters; the nodeID of the node and the nodeID of one of its neighbors. The node object can be further initialized by adding its other neighbors with the *addNeighbor* function. This class also stores the CI value, the degree and the maxheap index of the node.

The second class is the class `MaxHeap`, this class has only 1 attribute: a List in which all the nodeID's are stored. In this class the methods *heapify* and *buildMaxHeap* are implemented as we have seen in the course. The *removeRoot* function removes the root, puts the rightmost element at the root and restore the max-heap property by calling heapify on the new root.

The last class is the class `Network`, this is a static class, so there is no constructor. In the next section there is a more in depth explanation of this class. The class contains 4 static properties. The first one is an array of Node objects in which the array-index of each node object is the nodeID of the node. This ensures that, once the network has been initialized, each node can be easily accessed by their nodeID in $O(1)$ time. The second static attribute is a variable referencing the total degree of all nodes in the network (when there aren't any nodes removed yet). The third static variable keeps track of the total sum of CI values from all the nodes in the network. By changing this value when a node is removed or updated, we can save a lot of computing time. Otherwise you had to iterate over the whole network to get the sum of CI values. This is required after each deletion, with the corresponding updates of the nodes, so we can check if the eigenvalue is still larger than 1. The final static property references the MaxHeap object of the network.

2 Network class

In this class are all the important methods for the functionality of the algorithm implemented, so it deserves some more explanation. Each important method will be discussed in the following subsections.

executeAlgorithm This method executes the whole Collective Influence algorithm. The user needs to pass 2 parameters through to this function. The first one is the file path of the network and the second one is the depth at which you want to search. The next methods, which are called by this function, will guide you through the execution of the algorithm.

readTextFile First of all the function *readTextFile* reads in the text-file and creates for each different nodeID a unique Node object. Every Node object will be added to the array (in Network) at the index in accordance with their nodeID. This method will also initialize the total degree and the total sum of CI values of the network. There is just one restriction for this method: the string that you give as parameter to the executeAlgorithm function must correspond to one of the 13 given databases. Otherwise the size of the nodeNetwork array will not be correct, because a switch case selects the correct size for the given file. The corresponding database is recognized by the name of the file.

breadthFirstSearchDerivateBall This function is another important part of the Network class, it is implemented by using a Queue and a HashSet to save the visited nodes. The result of this method is a Queue of nodeIDs from nodes at a distance l from the initial node. This method is used the most during the execution of the algorithm. Because of that it is after even plenty of optimizations still the bottleneck of my code.

updateCIvalues This method updates the CI value of each node in the Ball(i, l+1), with i the nodeID of the removed node. The subtrees of the updated node are heapified after each update. Actually this isn't that simple at all since this happens in parallel, but more on that in the parallelization section.

validEigenvalue Finally there is the method *validEigenvalue* which checks if the eigenvalue will be larger than 1 or not. Therefore we just need to test if the division of the total sum of CI values by the total degree at begin is larger than 1. Since the (positive) power of a number larger than 1 will always be larger than 1. The total degree at the begin is equal to $N < k >$ with $< k >$ the average degree and N the number of nodes in the network. So this method checks if the eigenvalue is valid (> 1) after every update described by the function above. The giant component of the network is destroyed when the eigenvalue is smaller than 1, so we stop searching for influencers in that case. Then the algorithm terminates and all the influencers found, with some info about the duration of the steps in the algorithm, are written to a text file by using the *writeToFile* method.

3 Remarks

I have two remarks about the implementation of the algorithm. The first one is about a section in the second paper [2], the following text is cited from the explanation of step 4 (updating the CI values of the algorithm).

> *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.*

This is only true if at most one edge connects the node at distance l + 1 with any node at distance l. When the node is connected with multiple nodes on the edge at distance l, the CI value of the node needs to decrease more than k - 1. More precisely if we call m the number of nodes at distance l the node at distance l + 1 is connected to, than we should decrease the node at distance l + 1 its CI value by $m * (k - 1)$. It is hard to keep track of this so a better solution might be to recalculate the CI value of nodes at a distance l + 1 as well.
When I compared the method in the paper with the adapted version explained above, I noticed a small improvement in precision (less nodes removed) but a big loss in speed. Anyway since the whole algorithm is actually an approximation for finding the optimal set of influencers (NP-hard) and the difference in removed nodes between the more correct version and the one in the paper is not that big, I used the paper its method. In my opinion the small improvement in precision does not pay off to the big loss in speed.
All credits for this remark should go to Alexander Braekevelt since he discovered it by himself and discussed this afterwards with me.

The second remark is about something I noticed while testing the implementation on the given files. My initial thought about the CI value was that it could only decrease but during testing the code I remarked that sometimes the CI value of a node increases. While looking more deeper into this, it became clear that the increase of the CI value was due to an increase of the number of nodes on the edge. So suddenly after you remove a node, there may be more nodes at a distance l than there were before. First of all this seemed wrong, but since all the test were corrects there may be a case in which this happens. The figure underneath 1 shows an example of a case in which this happens, the node we want to calculate the CI from is the green node. When the depth is 3 there aren't any nodes found by the breadth first search method in the left figure. But after removal of the red node in the right figure, all the orange nodes are at a distance 3 from the green node. This will result in an increase of the CI value for the green node. Later on it became clear that this subtlety is avoided in the second paper [2] by stating that the eigenvalue decreases in *tree-like* networks.

> *... it is easy to show that, for tree-like random graphs,* $\lambda(l;0) = k - 1$*, where* $k = < k2 > / < k >$*. Removing nodes decreases the eigenvalue* $\lambda(l;q)$*, and ...*

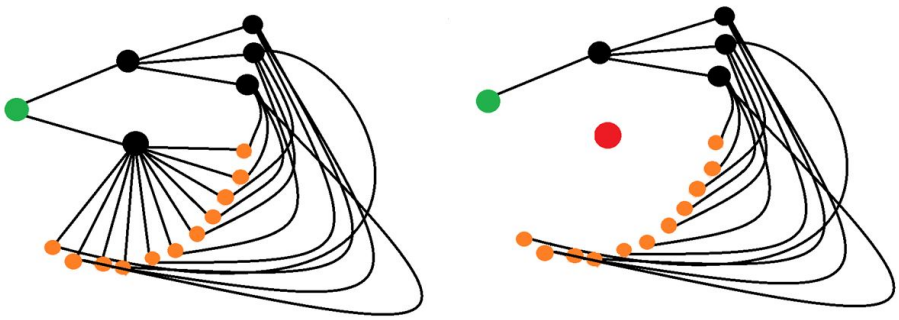

Figure 1: Example where CI value increases

4 Results

By looking at the results in table table 1 you can see I managed to run the algorithm on all the networks except for the last one, which had a disk size of more than 30 GB. Because the whole text file needed to be read in and all the different nodes saved into objects, it is kinda impossible to do this for 30 GB when you only have 12 GB RAM. The LiveJournal social network has a size of 1 GB, for which the algorithm found the influencers (depth 1) in a total time less than 19 minutes.
I noticed that generally the initial eigenvalue decreases while the time needed increases for each extra depth you search in a network. The number of influencers found decreases also as you search deeper in the network. I also ran the algorithm at depth 0, which is just the High Degree algorithm. As expected I noticed this approach to the optimal influencers problem works a lot faster, but it finds way more influencers. For example in the table with the results of the Email-Enron network you can easily see these conclusions. Another thing I noticed was that my implementation works way faster for networks with a relative smaller amount of edges compared to their number of nodes. A good example is the CA-roadnetwork where I could easily search at greater depths.
For three social networks I could search so deep that the initial eigenvalue, when no nodes are removed yet, is smaller than one. This occurred for CA-GrQC at depth 13, p2p-Gnutella08 at depth 7 and for higgs-reply-network at depth 19. This is also mentioned in the second paper [2].

Filename	Nodes	Edges	Max depth
CA-GrQc	5 242	14 496	13
p2p-Gnutella08	6 301	20 777	7
p2p-Gnutella30	36 682	88 328	4
Email-Enron	36 692	183 831	4
higgs-reply-network	38 918	32 523	19
higgs-mention-network	116 408	150 818	5
higgs-retweet-network	256 491	328 132	3
higgs-social-network	456 626	14 855 842	1
CA-roadnetwork	1 965 206	2 766 607	17
soc-pokec	1 632 803	30 622 564	1
cit-Patents	3 774 768	16 518 948	2
soc-LiveJournal1	4 847 571	68 993 773	1
com-Friendster	65 608 366	1 806 067 135	/

Table 1: Overview results

Depth	0	1	2	3	4
Time	0 min 0.622 s	0 min 2.625 s	0 min 48.392 s	25 min 37.164 s	175 min 15.508 s
Eigenvalue	138.158	112.229	48.844	18.613	8.078
Influencers	4095	3281	3139	3016	3003

Table 2: Email-Enron network

5 Optimizations

5.1 Parallelization

The major optimization of the implementation was by using parallel for loops. This is used for the initialization at the begin and the update after each removal of the CI values. This made the algorithm work twice as fast as before. But because the update from the CI values is in parallel, it is necessary to store the updated CI value in a variable. After all the updated CI value are calculated, the CI value is set equal to the updated CI value and the heap is heapified from the node its heap index. Because in parallel this causes problems with the max-heap property, we are forced to calculate to do this for each updated node.

5.2 Diagnostic tools

I analysed my program a lot by using the build-in Visual-Studio Diagnostic Tools. This showed the allocated process memory and the CPU usage of the running program. It was also possible to check which function used the most CPU time. This tool helped me a lot with improving the efficiency of my program by showing me the bottlenecks of the implementation.

5.3 Script

Visual Studio generates an executable file (.exe) when you build your program. Writing a batch-file that executes this script with high priority on the CPU, made the program execute around 46% faster.

5.4 Test Class

Because I wasn't so sure that my implementation was correct I wrote a test class. This class tests all the important functions from the Network class described above. This helped me to debug some minor mistakes.

5.5 Visualization

For the presentation there was a visualization of the algorithm implemented. The *QuickGraph* package was used for this, first there are dot files rendered and then by using a script PNG-images are created. This showed that weak nodes are removed and also helped me with writing my tests based on the visualization of the network. Because this report is already quite extended, I won't go in any further details about this.

5.6 Other possible optimizations

Other optimizations may be running the program on a GPU or a Beowulf-cluster instead of a regular CPU. Perhaps using for example an SQL database efficiently might be more useful for larger networks than just do everything on the saved objects in your RAM.

Conclusion

The major goals of this projects are accomplished: I understand the paper quite well and the results of the implementation were also satisfying. The topic really interests me and the fact that it is such a recent subject makes it even better.

References

[1] Flaviano Morone & Hernán A. Makse, *Influence maximization in complex networks through optimal percolation*, published online, 01 July 2015, `http://www.nature.com/nature/journal/v524/n7563/abs/nature14604.html`, Nature **524**, 65–68 (06 August 2015)

[2] Flaviano Morone, Byungjoon Min, Lin Bo, Romain Mari & Hernán A. Makse, *Collective Influence Algorithm to find influencers via optimal percolation in massively large social media*, published online, 26 July 2016, `http://www.nature.com/articles/srep30062`, Nature **Scientific Reports 6**, Article number: 30062

Model::CollectiveInfluenceAlgorithm