Counting Triangles And Connected Components In Dynamic Graphs

A Project Report

submitted by

HEMANT KARASALA

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

April 2016

THESIS CERTIFICATE

This is to certify that the thesis titled **Counting Triangles And Connected Components In Dynamic Graphs**, submitted by **Hemant Karasala**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor Of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof.Rupesh Nasre
Research Guide
Professor
Dept. of Computer Science
IIT-Madras, 600 036

Place: Chennai

Date: 27th April 2016

ABSTRACT

Counting the number of triangles in a graph has many applications in graph network analysis. Calculating the connected components in graphs also have their applications in various types of problem solving, like the 2-satisfiability problem and the Dulmage-Mendelsohn decomposition. This project deals with the computation of counting triangles and connected components in dynamic graphs, i.e graphs that change over time.

Since it is unfeasible to re-calculate the metrics for a graph every time edits are made, ways must be found to make the process less expensive. The way this can be done depends on the algorithm and implementation. This project is built upon Ligra, a lightweight shared memory parallel graph processing framework.

As edits, which are either removal or addition of edges happen, the new triangle count, and in the other case - the new component labels are found with minimized computation and the results are contrasted with the case where the entire graph is re-computed.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The next chapter discusses the background and related work of the problem. The two chapters after that briefly discuss counting triangles and connected components. The section following that contains the results. In this chapter we will go through the background of the problem.

1.1 Problem Background

The project is about Counting Triangles and Labelling Connected Components in a dynamically changing graph with editing edges in an efficient manner that takes lesser time than recomputing for the entire graph.

1.2 Formulation

The project is implemented upon the ligra framework. The framework is static and works on fixed graphs. The file input takes inputs in a very specific format and creates a graph data structure to be used. So, an implementation to take and apply edits to the graph structure is done as part of the project.

1.3 Solution Summary

The algorithm is run for a smaller ligra working list consisting of vertices that are part of the edits. This information is used to compute the new changed graph metrics without having to recompute for the entire graph.

1.4 Outline of Thesis

Counting Triangles and Labelling Connected Components in a dynamically changing graph with editing edges can be calculated in an efficient manner that takes lesser time than recomputing for the entire graph, simply by alterring the working lists to make them more local, because of the localized nature of the problems.

CHAPTER 2

Background and Related Work

2.1 Graph Algorithms

Many data can be formulated into the form of graphs. Graph is a data structure where items are represented as vertices and their relationships are represented as edges. The relationship can be one-way in a directed graph, or two way in a undirected graph. Graph algorithms deal with working on these graphs, whether it is for search, or calculating graph measures, classification etc. Graph algorithms include graph traversal algorithms like breadth first search, depth first search, Depth bound breadth first search etc. There exist heuristics that help guide graph search. Algorithms under this include A* algorithm and the Triple-S* algorithm. Graphs can be weighted, where the edges have a value associated with them. This can be considered as a degree of the relationship among items.

2.2 Parallel Graph algorithms

Graphs can be worked on in a parallel manner in convenient places. Portions of the graph algorithms can be parallelized to decrease time. Like when data stored in arrays needs to worked on with independent operations. Or when the tasks on the graph are localized, multiple threads can work on different portions of a graph, in a distributed memory manner. Or in a shared memory manner.

2.3 Ligra

Ligra is a lightweight graph framework with shared memory. It is suitable for implementing parallel graph traversal algorithms where subsets of vertices are processed in each iteration. The largest publicly available real world graphs all fit in shared memoy. This brings the need for an open source project which can give the benefits of shared memory graph processing which gives improvements of many orders of magnitude as compared to distributed memory graph frameworks.

2.4 Dynamic graph algorithms

In many of the applications of graph algorithms like network analysis, graphics, VLSI designs, there is a need for small changes over time like addition or removal of edges. This has recently increased in importance, and there is a whole field of algorithms and data structures specifically for such dynamic graphs. The goal of dynamic graph

algorithms is to efficiently update the required graph metric/measure after each change without having to recompute the whole graph from scratch each and every time.

CHAPTER 3

Parallel Dynamic Triangle Counting

3.1 Graph representation

The graph format consists of a file with the number of vertices and edges in the first two lines, followed by the position in the file of where the out neighbour list of each vertex begins, followed by the out neighbours of each vertex(or can be called the out edges of each vertex), each written one per line. This representation is known as the pbbs Adjacency graph format. Used in the Problem based benchmark suite.

http://www.cs.cmu.edu/ pbbs/benchmarks/graphIO.html

The datasets which are taken from the stanford public graph data website are in another representation, namely one where each line in the file is an edge with both vertices of the edge written. This input is converted as part of the project. A huge array of linked lists, where each represents the out neighbours list, or edge list of each vertex takes in the edges and adds them into the linked lists. Then, the file in the aforementioned pbbs Adjacency graph format is outputed.

3.2 Description

Counting Triangles is about counting the number of directed triangles formed by the edges in a graph. The number of triangles is a computationally expensive graph statistic which is used in complex network analysis, more specifically in calculating transitivity and clustering coefficients. There are also real world applications like spam detection and uncovering hidden thematic structures in the web.

Graphs are ubiquitous in this age of huge data collection, including but not limited to the internet, the world wide web, protein interaction networks and many other complicated structures. Large scale graphs are also found in social networks. A key feature of these graphs is that they keep changing over time. New vertices to the graph keep appearing as users increase in a social network. Even more importantly, the edges keep changing as people add new friends, prune their friend lists etc. This project is mainly focused on graphs with changing edges.

3.3 Algorithm

The basic algorithm for triangle counting is the expensive adjacency graph multiplication algorithm. The elements on the diagonal on the product of the adjacency graph with itself n times gives the number of walks of length n. As the only possible shape for a walk of length 3 is a triangle, the sum of the diagonal elements gives us the number of triangles with repeated countings. Dividing by 6, gives us the exact count. This algorithm costs as much as matrix multiplication which is in the order of n cube, which is brought further down by using Strassen's or Coppersmith-Winograd's algorithms.

3.4 Optimizations

However, the format of adjacency matrices is not particularly useful in building graph frameworks. This format isn't too handy for working on smaller localized portions of the graph either. The Ligra framework has implementation for working with subsets of the graph's vertices making it easier to work on shortening the processes.

In the counting triangles problem, when an edge is added or removed, the changes in the count happen with this edge itself. So, the triangles from the vertices of the edge can be computed before and after the administration of the change to the graph and the difference can be added to the earlier count of triangles. When an edge is removed, we are essentially removing the triangles that this edge contributed to the graph. When an edge is added, we are adding the new triangles contributed. Each vertex stores the triangles it has contributed to the graph in its structure.

Because of the technicality with the values of labels, some among the total number of triangles of the vertices of an edge can be with their neighbours. So, in the re-computation phase, the neighbours are added to the working list as well, along with the two vertices of the edge. This is a necessity because, the alternative of having the counts at all vertices would mean that there would be some redundant computation that would have to be done. By allowing the vertices to keep repeated(redundant) counts of the triangles, this necessity can be remvoed. The total count can be divided by 3 to get the count of triangles. In this system, we need only add the two vertices of the edge in the re-compuation phase. This further reduces our computation time. But at the cost of increasing the usual computation.

This technicality in implementation of Ligra Counting Triangles code proves costly in very dense graphs where adding the neighbours might lead to adding of most of the vertices present. But for the purposes of the graphs Ligra is intended to deal with, ergo the public domain graphs which are usually sparse, this algorithm suffices.

The code can be found at https://github.com/hemantkarasala/ligradyn. The folder 'apps' has the implementations of various algorithms. The Triangle.c has been edited for the purpose of the project.

3.5 Implementation

The counting triangles algorithm here in ligra goes through the vertices in the work list and calls a subroutine which checks the list of neighbours of each vertex to see if a triangle is formed. The vertex label values are expected in a specific order to prevent re-counting.

The implementation iterates through all the vertices and only sends case where the source edge is greater in its label value than the destination edge to the counting triangle subroutine. Here, the the vertex of the in-edge coming to the source is expected to be smaller than the source vertex. Without this condition, there will be two triangles counted for the same triangle. Because, this triangle would be counted anyway, and the vertex with the in-edge here is larger than the source, ergo it will be computed when that vertex is reached in the iterations of the implementation. With this condition, in case of the vertex of the in-edge being smaller, we only have one count in this subroutine, while in the case of it being larger, we only have one count in the subroutine called from that vertex.

CHAPTER 4

Parallel Dynamic Connected Components

4.1 Graph Representation

The graph format consists of a file with the number of vertices and edges in the first two lines, followed by the position in the file of where the out neighbour list of each vertex begins, followed by the out neighbours of each vertex(or can be called the out edges of each vertex), each written one per line. This representation is known as the pbbs Adjacency graph format. Used in the Problem based benchmark suite.

http://www.cs.cmu.edu/ pbbs/benchmarks/graphIO.html

The datasets which are taken from the stanford public graph data website are in another representation, namely one where each line in the file is an edge with both vertices of the edge written. This input is converted as part of the project. A huge array of linked lists, where each represents the out neighbours list, or edge list of each vertex takes in the edges and adds them into the linked lists. Then, the file in the aforementioned pbbs Adjacency graph format is outputed.

4.2 Description

An undirected graph can be split into many subgraphs among which any two vertices are connected by a path. This is called, splitting the graph into connected components. A vertex without any edge, for example is a connected component by itself. A graph which is connected consists of only one connected component which is the entire graph itself.

Relaxing the definition of connection with the edge weights being above/below a certain threshold to imply an edge, the problem of splitting a graph into connected components finds applications in various problems like image segmentation, edge detection, blob extraction, blob discovery, region extraction and region labelling etc.

4.3 Algorithm

The algorithm in ligra does the job in a less efficient manner owing to using it's framework. It initially assigns unique component labels to all the vertices, then iterates through the working list of all vertices and changes the label of a vertex to its neighbour's value if the neighbour's value is lower. Two lists of labels are maintained and this process is repeated until the lists converge. In each iteration, the vertices whose labels don't change are pruned out.

4.4 Optimizations

The algorithm to compute connected components in a more efficient manner, i.e in linear time, is fairly straightforward. We start off with a breadth first search or a depth first search including all the vertices reached as part of the current component, this will find the entire component the said vertex belongs to. We loop through all the vertices this way starting a breadth first search if and only if the vertex is not part of any component already.

When an edge is added in the graph of the connected components problem, the two components become one. It simply suffices to go through the list of labels and change the labels of all the vertices with the same label as the two vertices of the edge to the same label. If the two vertices which are being joined are of the same component, nothing need be done. When an edge is removed, there is computation that needs to be done. The vertices with the same labels as the two vertices of the edge are given new unique labels and placed in a working list for recalculating their labels. They could still be in the same component or they could split into two.

The code can be found at https://github.com/hemantkarasala/ligradyn. The folder 'apps' has the implementations of various algorithms. The Connected.c has been edited for the purpose of this implementation.

4.5 Implementation

One method of computing the connected components of a graph is to maintain an array IDs of size |V | initialized such that IDs[i] = i, and iteratively have every vertex update its IDs entry to be the minimum IDs entry of all of its neighbors in G. The total number of operations performed by this algorithm is O(d(|V |+|E|)) where d is the diameter of G. For high-diameter graphs, this algorithm can perform much worse than standard edge-based algorithms which require only O(|V |+|E|), but for low-diameter graphs it runs reasonably well.

CHAPTER 5

Experiments and Results

5.1 Experimental Setup

For testing the correctness, we need small homemade graphs and a host of edits that would alter the result. For testing the performance, we need large graphs meant for Ligra like the public graphs available on the web like the social network graphs and p2p network graphs. This is because the Ligra framework is meant precisely to tackle such graphs. These usually start off at the size of over five thousand nodes and over twenty thousand edges and go to much larger sizes.

We expect the run-times of the optimized implementation to be much lesser than the run-times of the re-computation code for the performance testing graphs in case of the counting triangles. This is because, the example graphs we have have an average degree which is a single digit number. For example, one of the performance testing p2p network graph has over 10000 nodes with under 40000 edges giving it an average degree of less than 4. These are sufficiently sparse and won't lead the computation to be nearly the same which would happen in the case of a very densely connected graph. As mentioned in the previous section, this is a product of the implementation. The implementation could be altered by allowing repeated computations to store the counted triangles in each vertex structure to prevent the necessity of adding the neighbouring edges to the working list, but this would make the usual counting operations more computationally costly. In the case of the public network graphs that Ligra is supposed to deal with, the graphs are sparse anyway, so it doesn't make sense to compromise on the computation cost to make the dynamic computation costs lesser. That can be done in cases of very dense graphs.

In case of the Connected Components, there is a difference between the incremental and decremental changes to the graph. Incremental is very simple, as it only involves altering the list to combine the labels of the two components the new edge is joining. In case of the decremental, removal of an edge can lead to the recomputation of the entire graph. Whether the optimized implementation on the decremental edits can give a good performace or not doesn't depend on sparsity or density. It depends more on the size of the components. If the graph has many small components, it would be much less costly. But in these graph networks we took as data, i.e the snapshots of a peer to peer network as we will see in the next section, there usually aren't a large number of components. So, the computation is nearly as costly as the re-computation of the entire graph. Although the time is lesser than the re-compuation it isn't as vastly significant like the gap between the run-times of the implementation and the re-computation in the incremental changes on the graph.

5.2 Inputs

Gnutella is a large peer to peer network. It was the fist decentralized system of connections and led to others adapting its model.

Peer-to-peer (P2P) computing or networking is a distributed application architecture that partitions tasks or work loads between peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes.

Peers make a portion of their resources, such as processing power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts.[1] Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model in which the consumption and supply of resources is divided. Emerging collaborative P2P systems are going beyond the era of peers doing similar things while sharing resources, and are looking for diverse peers that can bring in unique resources and capabilities to a virtual community thereby empowering it to engage in greater tasks beyond those that can be accomplished by individual peers, yet that are beneficial to all the peers.[2]

While P2P systems had previously been used in many application domains,[3] the architecture was popularized by the file sharing system Napster, originally released in 1999. The concept has inspired new structures and philosophies in many areas of human interaction. In such social contexts, peer-to-peer as a meme refers to the egalitarian social networking that has emerged throughout society, enabled by Internet technologies in general.

Owing to its decentralized nature, Gnutella's network is all interconnected and doesn't have a huge number of small connected components. These public domain test cases have been taken from: http://snap.stanford.edu/data/index.html

5.3 Overall results

The tables 5.1 and 5.2 are the results of the counting triangles while the other two are the results of the connected components. The R in the bracket signifies that the results are of the recomputation of the entire graph while the other column without an R in the bracket are the results of the optimized implementation.

The results do reflect the expected results. In the connected components, the decremental is very expensive computation and is nearly as much as the recomputation itself. While the incremental graph changes starkly contrasts with very very less computation time.

In the counting traingles, both the decremental and the incremental changes are computed with lesser time in the implementation than the recomputation.

Table 5.1: Counting Triangles, Incremental

Number of Nodes	Number of Edges	Incremental(R)	Incremental
5	6	0.000258	0.0000269
10	11	0.000204	0.0000410
15	16	0.000148	0.0000350
6301	20778	0.00162	0.000178
8114	26014	0.00155	0.000129
8846	31839	0.00314	0.000103
10879	39995	0.00279	0.000139

Table 5.2: Couting Triangles, Decremental

Number of Nodes	Number of Edges	Decremental(R)	Decremental
5	6	0.000270	0.0000210
10	11	0.000220	0.0000390
15	16	0.000170	0.0000219
6301	20778	0.00137	0.000103
8114	26014	0.00126	0.000110
8846	31839	0.00277	0.000122
10879	39995	0.00196	0.000150

Table 5.3: Connected Components, Decremental

Number of Nodes	Number of Edges	Decremental(R)	Decremental
5	6	0.000159	0.000104
10	11	0.000213	0.000158
15	16	0.000290	0.000234
6301	20778	0.00167	0.00145
8114	26014	0.00200	0.00151
8846	31839	0.00266	0.00178
10879	39995	0.00139	0.00115

Table 5.4: Connected Components, Incremental

Number of Nodes	Number of Edges	Incremental(R)	Incremental
5	6	0.0000811	0.000032
10	11	0.000216	0.000155
15	16	0.000226	0.000197
6301	20778	0.00163	0.000011
8114	26014	0.00270	0.000019
8846	31839	0.00321	0.000013
10879	39995	0.00268	0.000019

5.4 Effects of Various parameters

The number of edges is the key parameter the explanation of the results hinges upon. The graphs can be seen to be sparse based on the number of edges which is the reason for choosing the implementation the way we did in the counting triangles.

In the lower number of vertices and edges, the other computations involved in the code overshadow the actual graph computation times and the results are not very clear, although the total time of the implementation is lesser than the recomputation. That's alright because the purpose of those homemade graphs was to check the correctness of the code. The performance testing p2p network graphs show us the results we expected.

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

The aim of the project to efficiently compute the Number of Triangles and the labelling of the connected components as the graph is edited dynamically has been achieved. The run-times of the implementation are consistently lower than those of the re-computations.

As expected, there is a vast difference in the run-times in the connected components in the incremental graph edits. And there is a small margin of difference in the connected components in the decremental graph edits. The counting triangles has similar performance for both incremental and decremental.

6.2 Future Work

The alternate implementation for counting triangles for denser graphs can be tested with randomly generated dense graphs to see the performance change in the dynamic part and the actual computation part. These two implementations could work complementarily based on the graph in question.

Instead of running after each edit, the code could run after a few edits. Testing could be done to see how many edits makes the process the most optimal.