應用於線上社群網路分析之
雲端運算平行分群演算法

研究生: 謝子強　　　　　　指導教授: 高榮鴻

國立交通大學電信工程研究所碩士班

摘要

隨著社群網路服務的興盛，營運商對於社群網路的資料分析十分感興趣。然而，在大量資料的限制之下，傳統運行在單一電腦上的演算法已經無法承受使用者所產生的海量資料。因此，如何設計雲端的平行演算法便成為一項重要的課題。本文關注於設計高凝聚性子圖 $(k - truss)$ 之雲端平行分群演算法，我們提出了兩個演算法來降低計算的時間成本。以上的演算法分別應用於兩種不同的情況，第一個演算法適用於靜態網路，意即對於社群網路的一個時間點做分析；第二個演算法適用於動態網路，對於一個社群網路選擇兩個較近的時間點，我們的演算法可以利用前一個時間點的分析結果來加速當前的運算。在驗證的部分，我們分別測試了人工合成的資料和實際的資料，包含了電子郵件網路、線上社群網路等。以真實的資料測試靜態網路分群演算法，在特定的情況可以達到加速的效果，若是處於效率較差的情形，演算法也可以自動切換成原始的法子避免計算成本的增加。以人工合成的資料測試動態網路分群演算法，在網路變化不大的情況之下可降低執行時間。

關鍵詞：社群網路服務、社群網路分析、雲端計算

Truss-Based Clustering Algorithms in MapReduce for Analyzing Massive Online Social Networks

Student : Tzu-Chiang Hsieh　　　Advisor : Rung-Hung Gau

Institute of Communications Engineering
National Chiao Tung University

Abstract

With the rise of social networking service, operators are very interested in the data analysis of social network. However, due to the large data size, traditional algorithms running on single computer cannot deal with it. Thus, how to design an algorithm in MapReduce is an important issue. The thesis focuses on designing algorithms for finding cohesive subgraph ($k - truss$) in MapReduce. We propose two algorithms applied to two situations. The first one is designed for static social networks. In other words, it can analyze a social network at a snapshot. The second one is designed for dynamic social networks. Given two close snapshots of a social network, we use the analytical results of the first one to speed up the analysis of the second one. We test the proposed algorithms by both synthetic data and real world data including email communication network, online social network and so on. The first algorithm is faster than the original one when applied to real world data. In case the proposed algorithm does not improve the performance, our algorithm can automatically switch to the original algorithm. We verify the second algorithm using synthetic data. It is faster than the original one if the change of networks is below a threshold.

keywords: social networking service, social network analysis, MapReduce

Contents

List of Figures

Chapter 1

Introduction

In recent years, social network analysis (SNA) becomes more and more important because of the emergence of large scale social networking service (SNS) such as Facebook, Twitter, LinkedIn and so on. According to statistic of SocialBakers [1], there are close to 1 billion people in the world using Facebook, and about 13 million users in Taiwan. To analyze the huge network, one possible way is using MapReduce technology [2][3][4][5], and we will discuss it in section 2.3. By the way, there have been many works [6][7][8][9] using MapReduce to deal with big data.

Given a social network, many researchers are interested in identifying groups of closely bound people, or better known as cohesive subgroups. Typical examples of such groups in practice are formed by families and friends. Furthermore, some researchers use it to identify terrorists or criminal organization. The most well-known cohesive graph is clique [10][11], which is a complete graph. However, there are two problems for clique: first, the definition is too strict (in real world, not all members have relationship with each other in many groups); Second, the time complexity of computing clique is NP-hard. A relaxation version is k-clique [11][12], which is a maximal subgraph G' of G such that the distance in G of any two vertices in G' is equal to or smaller than k. The disadvantages of k-clique are: ① The diameter of a k-clique may be larger than k. ② A k-clique may be disconnected.

Thus, k-clan and k-club [11][12] were proposed in 1979. A k-clan of G is a k-clique G' of G such that the distance in G' of any two vertices in G' is equal to or smaller than k. The other is k-club, which is a maximal subgraph G' of G such that the diameter of G' is equal to or smaller than k. However, because they are the modified version of clique, the problem of computational difficulties in large scale networks is not solved. Another cohesive subgraph is k-plex [13]. A k-plex with c vertices is a subgraph whose degree of each vertex is between $(c-1)$ to $(c-k)$. Unfortunately, it is also NP-hard for computation.

To overcome the computational difficulties, Seidman proposed k-core [14] in 1983. A k-core is a maximal subgraph whose degree of each vertex is equal to or larger than k. By the way, a k order clique is also a (k-1)-core with k vertex. Although k-core can be computed in polynomial time, the found components are too numerous. Thus, Seidman described it as a "seedbed" which can precipitate other cohesive graphs such as clique.

In 2008, Cohen [15] sought a substructure which can be computed in polynomial time and is not overly-numerous. He created a new cohesive graph called k-truss, in which a tie between two actors A and B is legitimate only if it supported by at least k−2 other actors excluding A and B. A maximal k-truss of graph $G = (V, E)$ could be computed in polynomial time and the size of the subgraph is between clique and k-core. Thus, we regard it as a good cohesive graph and we will detailed introduce it in section 2.2. To the best of our knowledge, there are only three algorithms for computing k-truss [15][16][17]. In addition, only [16] is a MapReduce algorithm. We propose an improved MapReduce algorithm for finding truss in chapter 3. We summarize the above cohesive graphs in table 1.1.

Table 1.1: The analysis of cohesive graphs

Item	**Advantage**	**Drawback**
Clique	Very tight	Too strict, NP-hard
K-clique	Tight	Not intuitive, probably disconnected, NP-hard
K-clan	Tight	NP-hard
K-club	Tight	NP-hard
K-plex	Tight	NP-hard
K-core	Polynomial time	Loose, numerous
K-truss	Tight, polynomial time	

Another topic in our thesis is about dynamic network. Most of the social networks in reality change over time. The change rate of dynamic social networks is very quick, but the magnitude of the changes is slight. Because the above characteristics, many researchers study it from different perspectives. Some of them [18][19] focus on identifying the communities and tracking their evolutions in dynamic networks. The basic idea is using some information retrieved from a series of communities rather than using only one snapshot. The information includes the changes of communities such as birth, death, merging, splitting, expansion and contraction. [20] focuses on designing an adaptive algorithm to reduce the run-time. Whenever the network updates, they use the results of the previous snapshot to speed up the computational time of current snapshot instead of re-calculating the network.

Our work is different from the above. First, we want to find k-trusses instead of the communities without formal definition. Second, we focus on the speed of the algorithm instead of comparing the results with ground-truth because our algorithm is designed for finding particular structure. Third, our algorithm can deal with big data but the above works cannot.

The rest of the thesis is organized as follows. In chapter 2, we define notations frequently used, introduce the k-truss, and elaborate on MapReduce. In chapter 3, we propose an improved algorithm for truss finding with MapReduce. We first show the original algorithm and then illustrate our algorithm. In chapter 4, we design an adaptive algorithm for truss finding with MapReduce. In chapter 5, we evaluate our algorithm and compare it with the original algorithm. The data set includes artificial and real world data.

Chapter 2

Notations and Backgrounds

2.1 Notations

We first define some notations to analyze the problem. The notations are described in table 2.1 and it will be used throughout the thesis.

Table 2.1: Notations

Notation	**Description**
$G = (V, E)$	An undirected, unweighted simple graph G
$V(G)$	The vertex set of G
$E(G)$	The edge set of G
$\|V\|$	The number of vertices in G
$\|E\|$	The number of edges in G
$Tri(G)$	The triangle set including all triangles of G
$T_k(G)$	The edge set composed of all maximal k-trusses of G
G_n	The network observed at snapshot n
G_0	The initial network
ΔG	The change between G_n and G_{n-1}. ΔG is composed of ΔG_{add} and ΔG_{remove}
ΔG_{add}	The maximal edge set appearing in G_n, but not in G_{n-1}
ΔG_{remove}	The maximal edge set appearing in G_{n-1}, but not in G_n
$\mathbf{G} = \{G_0, G_1, ..., G_n\}$	A dynamic social network including $n + 1$ snapshots

2.2 The Definition of Truss

The concept of truss is originally proposed by Cohen [15] in 2008. The basic idea of truss is as follow: if two actors are strongly tied in the community structure, they are likely to share common actors. He created a new cohesive graph called k-truss, in which a tie between two actors A and B is legitimate only if it supported by at least k–2 other actors who are tied to both A and B. Here are the formal definitions about k-truss:

Definition 1 *A k-truss is a connected graph in which each edge is supported by at least k–2 triangles including the edge. ($k > 2$)*

Definition 2 *A maximal k-truss of graph G is a k-truss T that is not contained in a larger k-truss of G.*

Definition 3 *Given a parameter k, the truss finding problem is to identify all maximal k-trusses of the graph.*

K-truss also have many intuitive physical characteristics and the nature of the graph theory [15][17]. We introduce some important properties of them here: ① Truss can be considered as the relaxation version of clique. The reason is that for each edge in clique, it must be supported by the other nodes in the clique. However, for each edge in k-truss, it is only supported by $k-2$ nodes. ② Each k-truss T of graph G is a subgraph of a (k-1)-core of G. The reason is that each node in T has degree more than $k-1$, so T is also a (k-1)-core. ③ Truss finding problem can be computed in polynomial time [15]. After realizing the properties of truss, let us visualize it. Figure 2.1 on the next page is an example for $T_3(G)$ in the network G. Colored nodes represent 3-trusses, and different colors represent different maximal trusses.

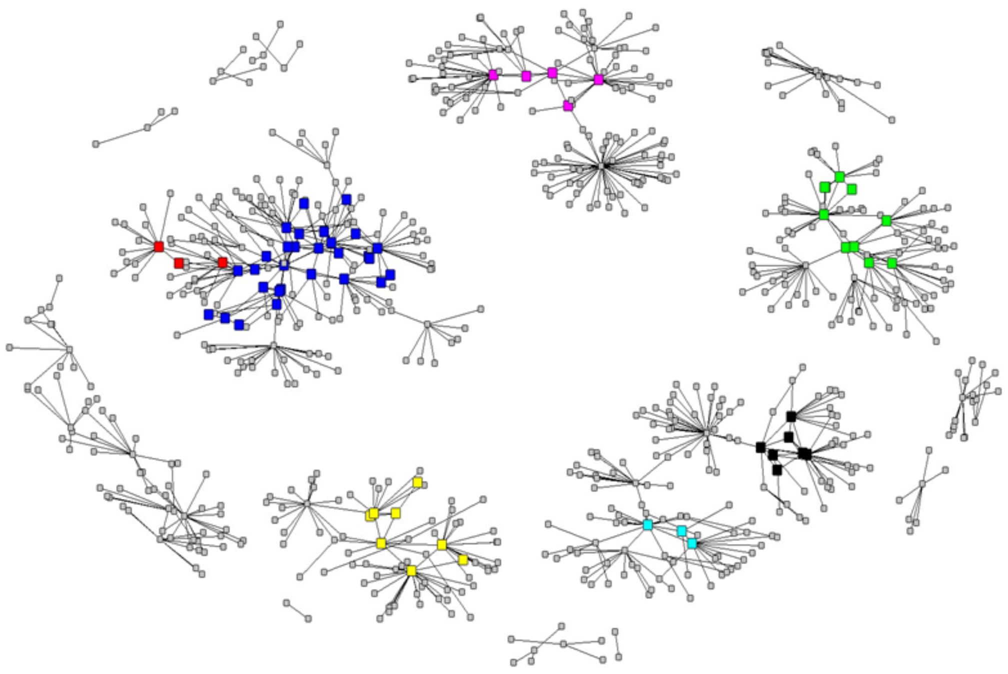

Figure 2.1: Maximal trusses in the network ($k = 3$)

2.3 MapReduce Overview

MapReduce is a software framework for processing huge datasets in-parallel on a large number of nodes; If all nodes are on the same local network and use similar hardware, it is referred to a cluster; If the nodes are shared across geographically distributed systems and use more heterogeneous hardware, it is referred to a grid. Computational processing can occur on data stored either in unstructured storage assets such as file systems or in structured storage assets such as databases. MapReduce can process data on the storage assets or near it to decrease transmission of data.

The framework of MapReduce is shown in figure 2.2 on the next page. The execution flow is as follow: ① When user program is executed, the program will be cloned to the master and each worker. ② The master assigns which workers are mappers and which workers are reducers. ③ The data blocks in distributed file system are assigned to mappers for Map. ④ The results

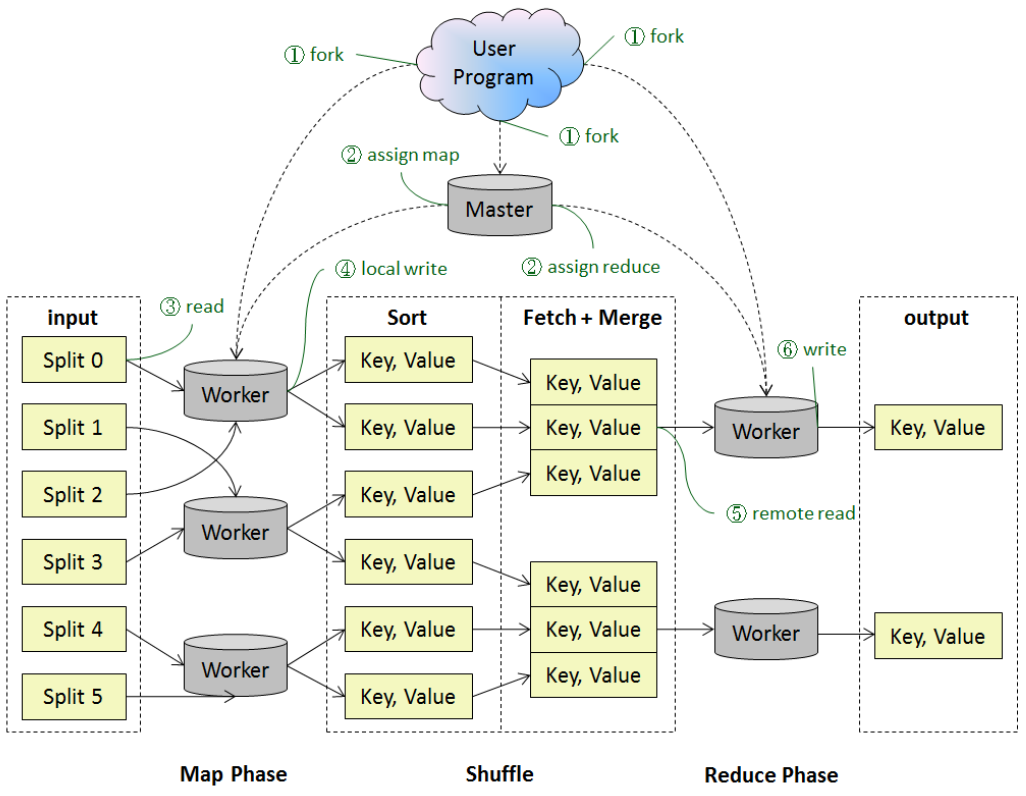

Figure 2.2: MapReduce execution framework

of Map are stored into the local disk of workers. ⑤ Remote read the result which is sorted and merged from Map, and then execute Reduce on workers. ⑥ Write the outputs which user demand.

Here, we introduce ”Map” and ”Reduce” step in detail. In ”Map” step: the master takes the input file, divides it into many smaller sub-problems, and assigns them to the mappers. Each mapper may do this again in turn, leading to a multi-level tree structure. The mappers process the smaller sub-problems and pass the answers back to their master node. In ”Reduce” step: the master node gets the original answers from the reducers which combine all the answer of related sub-problems. Note, the answers of the sub-problems have been computed by the mappers.

Chapter 3

Improve Truss Finding in MapReduce

3.1 The Existing Algorithm

We outline the truss finding algorithm proposed by Cohen [16] in algorithm 1. The basic idea of step 1 is to reduce the computational complexity of step 2. Let us explain it in detail: step 2 includes two phase; Phase 1 gets triads (unclosed triangle), and phase 2 combines triads with edges to get triangles. For each vertex, the time and space complexity in phase 2 is proportional to square of the degree. Thus, if we choose the vertex with lower degree as center node, the computational complexity can be reduced.

Algorithm 1: The original version of truss finding in MapReduce

```
   Input: $G$
   Output: $T_k(G)$
1  Augment the edges with vertex valences.
2  Enumerate triangles.
3  For each edge, record the number of triangles containing that edge.
4  Keep only the edges with sufficient support.
5  If step 4 dropped any edges, return to step 1.
6  Find the remaining graph's components; each is a truss.
```

Recall that for a k-truss, each edge is contained in at least k–2 triangles. Thus, the goal of Step 3-4 is to discard edges without sufficient support. The above instructions are very intuitive, but in fact they abandon some useful information at the same time. We will show that in section 3.2. Step 5 makes sure that each edge has sufficient support (if some edges is dropped in step 4, the edges adjacent to them may has insufficient support). When no more edge is dropped in step 4, we find the edge set $T_k(G)$. However, we cannot identify them because we don't have their group identity. Thus, step 6 is designed for traversing each component to solve the problem.

3.2 The Improved Algorithm

In the previous section, we have mentioned that the original algorithm discards some useful information. Thus, we propose an improved algorithm to deal with the problem. We will use the graph in figure 3.1 to illustrate our algorithm. In the example, we want to find $T_4(G) = \{(A,B),(A,C),(A,D),(B,C),(B,D),(C,D)\}$. In algorithm 1, step 3-4 abandon edges without sufficient support and go back to step 1-2 to recalculate triangles (as long as there are some discarded edges). If we can reserve some information in step 3-4, we are able to avoid recalculating triangles by updating triangles and supports.

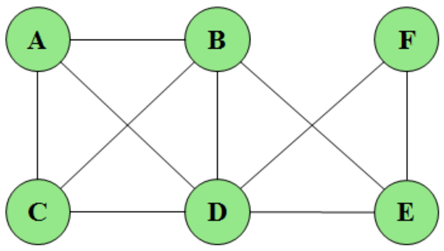

Figure 3.1: The graph for illustrating the improved algorithm

We write down our algorithm in pseudo codes and illustrate them later.

Algorithm 2: Prepocessing (MapReduce A)

```
   Input: <recordId, triangle>
   Output: <edge, support & triangle>
 1 function map(recordId, triangle)
 2 for $i = 1$ to $3$ do
 3     edge[i] ← getEdge(triangle, i)
 4     EMIT(edge[i], triangle)
 5 end
 6 end function
 7 function reduce(edge, triangle)
 8 support ← the total number of edge
 9 for each record do
10     EMIT(edge, support & triangle)
11 end
12 end function
```

Algorithm 3: Triangle-based filter (MapReduce B)

```
   Input: <recordId, edge & support & triangle>
   Output: <edge, support & triangle> or <edge, support-1>
 1 function map(recordId, edge & support & triangle)
 2 for each record do
 3     EMIT(triangle, edge & support)
 4 end
 5 end function
 6 function reduce(triangle, edge & support)
 7 if the supports of all edges eqaul to ot larger than k-2 then
 8     for each record do
 9         EMIT(edge, support & triangle)
10     end
11 end
12 else
13     for each record does not have enough support do
14         EMIT(edge, support-1)
15     end
16 end
17 end function
```

Algorithm 4: Update support (MapReduce C)

```
Input: <recordId, edge & support & triangle> or <recordId, edge &
       support-1>
Output: <edge, support & triangle>
1  function map(recordId, edge & support & triangle)
2  for each record do
3      EMIT(edge, support & triangle)
4  end
5  end function
6  function reduce(edge, two type records)
7  decreaseSupport ← the total number of <edge, support-1>
8  for each <edge, support and triangle> record do
9      EMIT(edge, (support-decreaseSupport) & triangle)
10 end
11 end function
```

Algorithm 5: Merge Duplicate Edges (MapReduce D)

```
Input: <recordId, edge & support & triangle>
Output: <edge, an empty string>
1  function map(recordId, edge & support & triangle)
2  for each record do
3      EMIT(edge, support & triangle)
4  end
5  end function
6  function reduce(edge, support & triangle)
7  EMIT(edge, an empty string)
8  end function
```

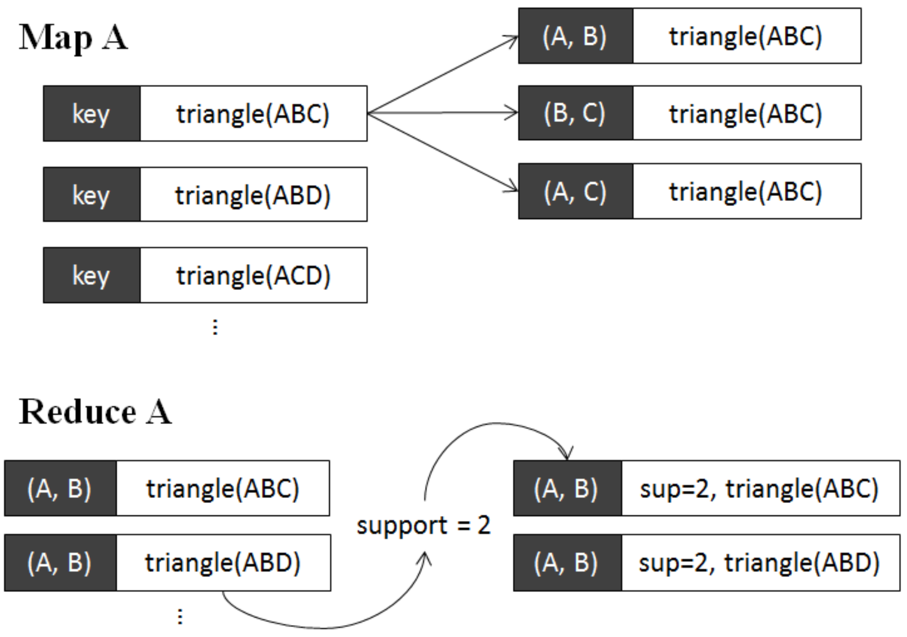

Figure 3.2: Preprocessing (phase 1)

When triangles have been calculated at least once, we want to retain the edges with sufficient support. Our goal is to avoid recalculating triangles but update them. A key point is to reserve triangle identities for each edge, and we will explain how it works in the subsequent phase.

Figure 3.2 illustrates how phase 1 realizes the key point. The phase is similar to step 3 in the original algorithm, but we reserve the triangle identity for each edge in map and also reserve edges without sufficient support in reduce. For each triangle record, the mapper emits three records. For each record, its key is the edge and the value is the original triangle identity. Then, the reducer calculates the support for each edge.

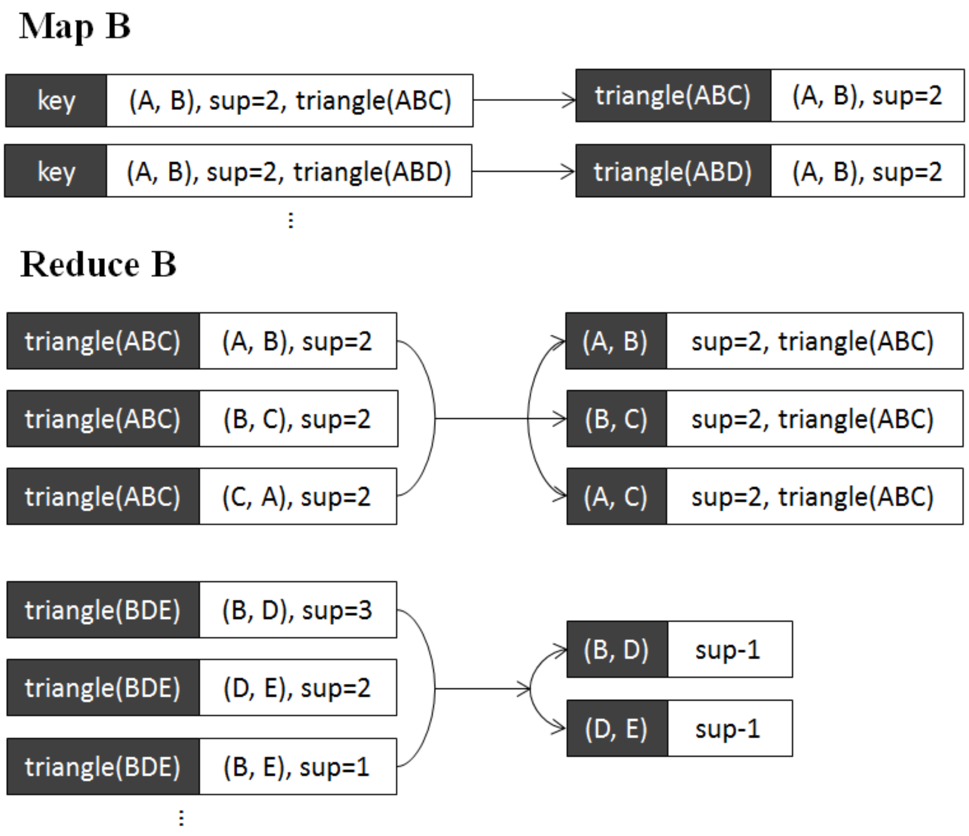

Figure 3.3: Triangle-based filter (phase 2)

In phase 2, we design a triangle-based filter to percolate edges with sufficient support and legitimate triangles. Given the supports of edges for each triangle, we can differentiate which triangle is legitimate. Then, we can process them according to their situations.

In figure 3.3, the mapper changes the order of the text to prepare for the next stage. The reducer operates according to four situations. For each triangle, if all edges are legitimate, we retain all of them. If one or two edge(s) do not have sufficient support, the reducer emits record $sup-1$ for the other edge(s). It is for updating the support of the retained edge in the next phase. If all of edges do not have sufficient supports, the reducer doesn't emits any record. Note that an edge may be included in several triangles.

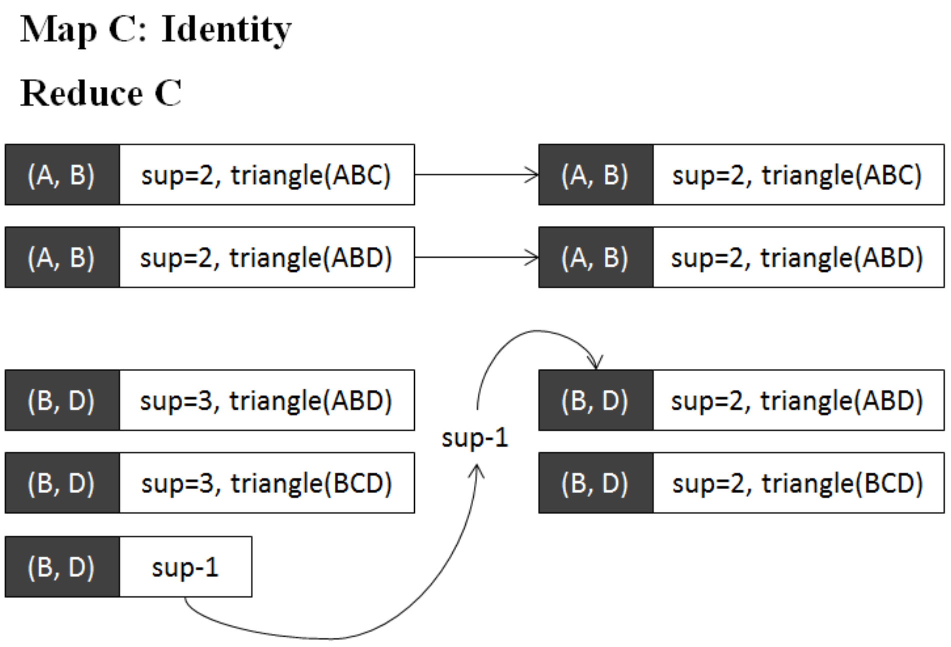

Figure 3.4: Update support (phase 3)

In the third phase, we update the support of corresponding edges. Moreover, the formation of reducer's output coincides with the input of Map A. The example is shown in figure 3.4. An mapper is simply an identity mapping. The reducer is composed of two sub-phases; First it accumulates all records with $sup - 1$, and then updating the support of the other type of records. Note that the output format of Reduce C is the same with the input format of Map B. We can use the property to update triangle and support iteratively.

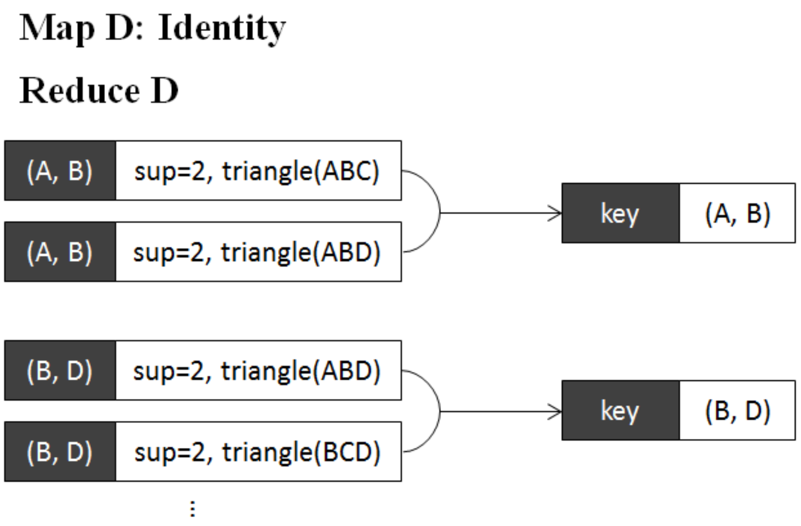

Figure 3.5: Merge records which have the same edge (phase 4)

The phase 2-3 in the previous paragraph can form a structure to update triangle and support iteratively. Until there is no record with $sup - 1$ in phase 3, the while loop terminates. Next, we have to merge records which have the same edge for eliminating redundant messages (phase 4). The detail is illustrated in figure 3.5. The mapper emits records by themselves, and the reducer merges records which have the same edge but in different triangles.

Algorithm 6: The improved version of truss finding in MapReduce

```
  Input: $G$
  Output: $T_k(G)$
1 Augment the edges with vertex valences.
2 Enumerate triangles for remaining edge.
3 Preprocessing. (MapReduce A)
4 Triangle-based filter (MapReduce B)
5 Update support. (MapReduce C)
6 If there is any record with $sup - 1$, return to step 4.
7 Merge records which have the same edge. (MapReduce D)
8 Find the components of remaining graph; each is a truss.
```

We summarize our algorithm as follows. Step 1 and 2 are the same with the original algorithm. Step 3-5 are the core, which avoids re-calculating triangles in the loop. Step 5 ensures that all edges are legitimate and step 6 eliminates redundant messages. Finally, step 7 traverses all components.

3.3 Additional Conditions

After doing some experiments, we discover that our algorithm is better than the original algorithm in some cases but worse than the original in other cases. Thus, we try to explain the result and propose a method to deal with it. The key point is the accuracy which the guessed triads are closed. In the original algorithm, it guesses some potential closed triads and then tests whether they are indeed closed. For the phase 2 and 3 in our algorithm, the data size is $3 \cdot |Tri(G)|$ and the complexity is proportional to it. Thus, if the guessed accuracy of the original algorithm is larger than $\frac{1}{3}$, our algorithm is worse than the original. Because of the restriction, we set a condition that if the guessed accuracy is larger than $\frac{1}{5}$, we choose the original algorithm. The guessed accuracy could be obtained at the first triangle round. Another additional condition is that if the user number of social networks is smaller than 1 million, we choose our algorithm.

Chapter 4

Adaptive Algorithm for Truss Finding in Dynamic Social Networks

4.1 The Model of Adaptive Algorithm

Just like most of conventional SNA algorithms [11][15], we focused on the problem of static social networks in chapter 3. Although we have improved the speed of truss finding algorithm, it spends many hours for large scale networks. Many companies need to analyze social networks on a regular basis. For example, large-scale social networking service company, online shopping providers and telecommunication company. It costs a lot of time for recomputing whenever the network updates. While the changes rate of social networks could be fast, the amplitude of change might be slight. Thus, we propose an adaptive algorithm that uses the previous results to speed up the analysis of the current social graph.

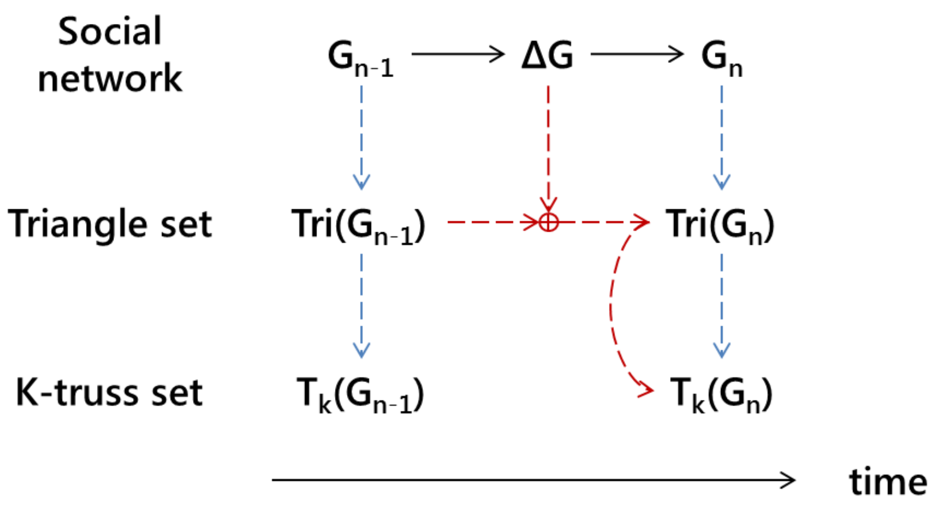

Figure 4.1: The model of adaptive algorithm

Our model is similar to [20] and we illustrate it in figure 4.1. Blue dash line represents the flowchat of original algorithm and red dash line represents adaptive algorithm. At time slot n, the adaptive algorithm utilizes information $Tri(G_{n-1})$ and ΔG instead of only using G_n to get $Tri(G_n)$ for decreasing computational time.

4.2 Problem Formulation

Suppose the dynamic social network is represented by a time series of graphs $\mathbf{G} = \{G_0, G_1, ..., G_n\}$. In particular, G_m is the social graph at the m-th time slot. Our goal is to devise an algorithm to quickly identify k-truss at any snapshot (exclude G_0) utilizing the information from the previous snapshot. For example, if someone wants to find $T_k(G_n)$, our algorithm can achieve the goal using the information including G_{n-1}, G_n and $Tri(G_{n-1})$.

4.3 The Concept of The Algorithm

We first elaborate on how truss structure changes according to social network change. When a member joins the network, he or she may communicate with other members in the network. There are four possibilities in the above situation: the first one is the relationship ties and the ties adjacent to it form a new truss; The second is new ties merges two or more trusses into a bigger truss; The third is it expends the scale of the existed truss; The last one is nothing happen.

If a member leaves the network, he or she may break relations between his or her friends in the network. The event also includes four possibilities: the first one is his or her original truss disappears from the network; The second is his or her original truss splits into two or more smaller trusses; The third is it reduces the scale of the existed truss; The last one is nothing happen.

Let us define four events using graph theory: *add a node*, *remove a node*, *add an edge*, and *remove an edge*. They sequentially represent a member joins; a member leaves; a relationship establish; a relationship break. In fact, *add a node* is composed of many *add an edge*, and *remove a node* is also composed of many *remove an edge*. Thus, we only discuss *add an edge* and *remove an edge* in the following paragraph.

We first discuss *add an edge*. When an edge is added into the graph, it may form a triangle with adjacent edges. If more than one edge adds into the graph simultaneously, they may form triangles with adjacent edges or form triangles by themselves. Let us describe the step in detail: when adding an edge set E_{add} into the original graph G_{n-1}, we first get the edge set E_{adj} which is adjacent to E_{add} in G_n. Then, the new triangle set in G is $Tri(E_{add} \bigcup E_{adj})$. Thus, if merging $Tri(E_{add} \bigcup E_{adj})$ and $Tri(G_{n-1})$, we can get $T_3(G_n)$ (recall the definition of 3-truss, each edge is supported by at least one triangle). In fact, the method may produce redundant triangles in a special case: E_{adj} may form triangles by themselves. Thus, we have to remove the redundant triangles later.

If we want to get $T_k(G_n)$ for $k > 3$ utilizing the same concept, we need knowing all indestructible structures (e.g. k-clique) for k-truss. In [15] Cohen proposed that a k-truss need not contain a clique of order k, and he give an example for $k = 4$. Thus, if we cannot find all indestructible structures excluding k-clique, we can't use the same way. Fortunately, if we use the improved algorithm proposed in Section 3.2, we can get $T_k(G_n)$ quickly after finishing the adaptive algorithm for 3-truss.

Now we consider *remove an edge*. When an edge is removed from the graph, it may destroy a triangle, a triad or even a k-truss. If more than one edge is removed from the graph simultaneously, the above phenomenon is more likely to occur. Let us describe the step in detail: suppose removing an edge set E_{remove} from the original graph G_{n-1}, we just delete the triangles including them from $Tri(G_{n-1})$. Because edges in each triangle can be repeated, we can get $T_3(G_n)$ using the above step. If we want to get $T_k(G_n)$ for $k > 3$ in removing phase, we can use the way like *add an edge*.

4.4 Algorithm Description

In this section we will introduce the adaptive algorithm for 3-truss in MapReduce. To easily understand the algorithm, we illustrate it using figure 4.2 in the following paragraph. In the figure, green graph G_0 represents original graph and the blue one G_1 represents the graph at next snapshot.

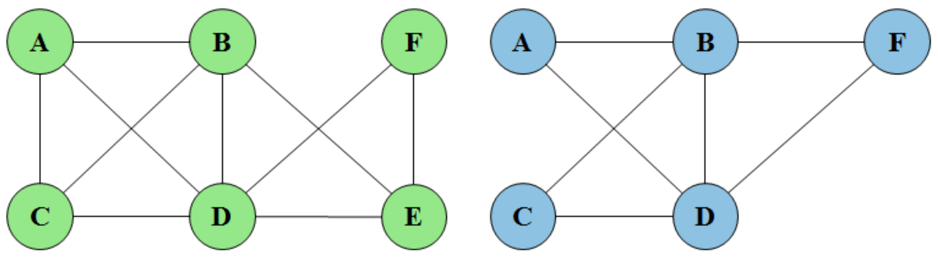

Figure 4.2: A dynamic graph with two snapshots

We first show the flowchat of our algorithm in figure 4.3. Note that the capital letter represent MapReduce phase.

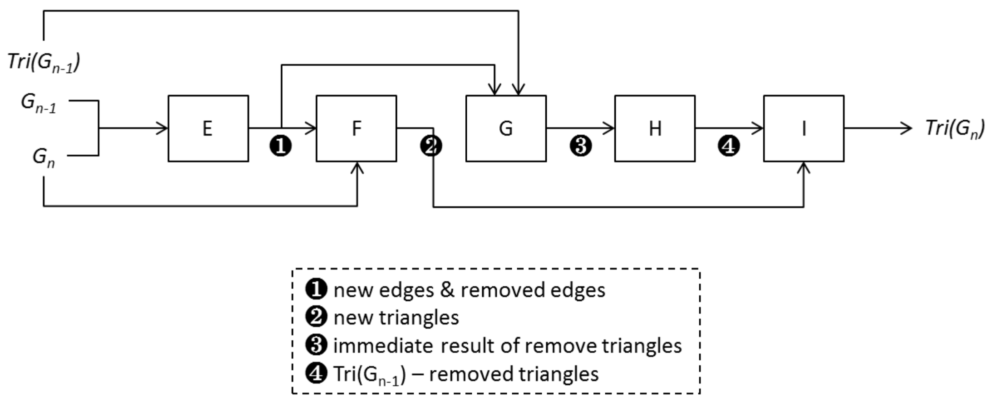
Tri(G_{n-1})
G_{n-1}
G_n
E
F
G
H
I
Tri(G_n)
❶ new edges & removed edges
❷ new triangles
❸ immediate result of remove triangles
❹ Tri(G_{n-1}) – removed triangles

Figure 4.3: The flowchat of adaptive algorithm

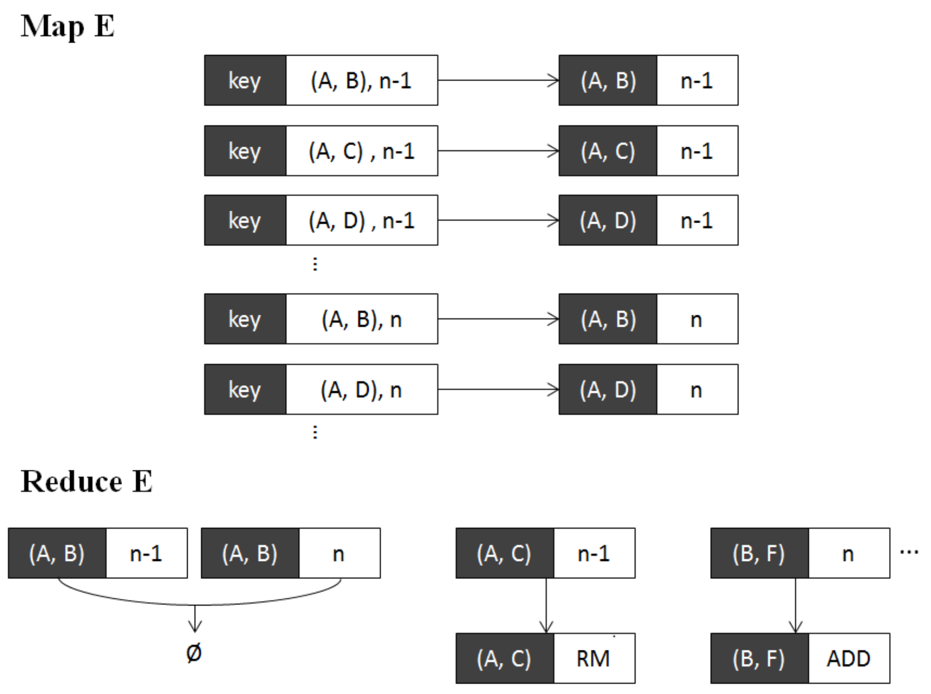

Figure 4.4: Find the changes of the network (phase 1)

The basic idea of phase 1 is to recognize which edges are added and which edges are removed. Let's see figure 4.4, the inputs of mapper are divided into two types which represent edges in G_{n-1} and G_n respectively. The mapper emits records by themselves. The mission of reducer is to recognize which edges are added and which edges are removed. The processing method according to three situations: if a record appears in both G_{n-1} and G_n, the reducer discards it; If a record appears in G_{n-1} but not in G_n, the reducer emits the edge is removed. If a record appears in G_n but not in G_{n-1}, the reducer emits the edge is added. After processing each edge, we find the changes of the network.

In this paragraph we discuss how to find new triangles utilizing edges newly added. We want to reserve E_{add} and E_{adj} in the phase. The inputs of the map are the output of the previous stage and G_n. There are two situations in Map F: if an edge contains *ADD* message, the mapper emits

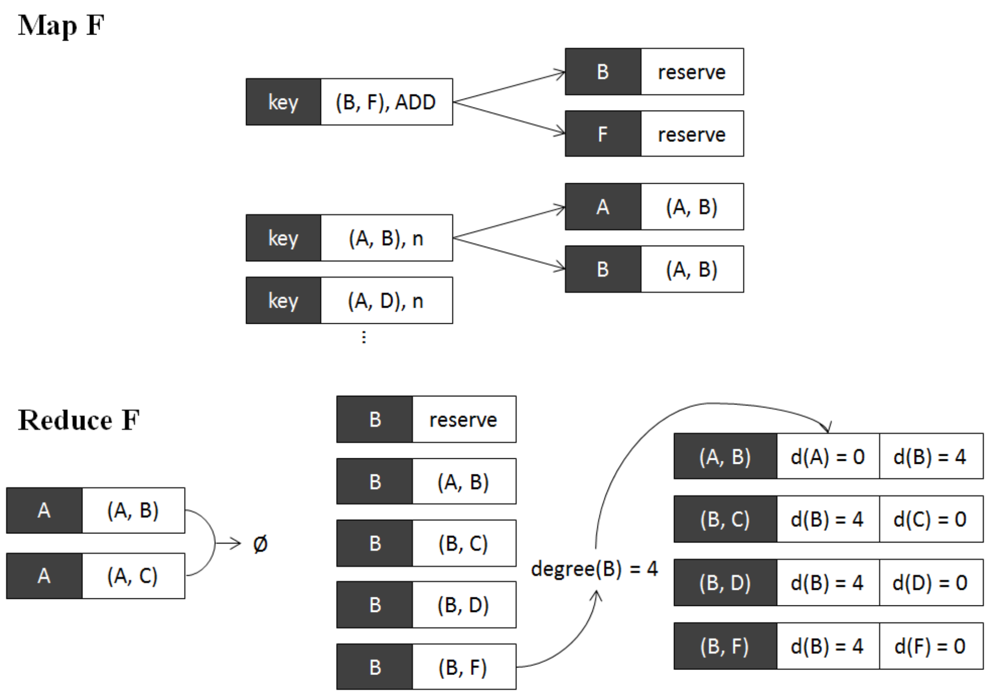

Figure 4.5: Reserve E_{add} and E_{adj}. Then, calculate their degrees. (phase 2)

two records whose keys are vertices of the edge and the values are *reserve* messages; If an edge contains no useful information, the mapper emits two records whose keys are vertices of the edge and the values are the edge. In Reduce F, there are two situations: if no record contains *reserve* message, the reducer emits nothing because there is no edge adjacent to newly added edges; Otherwise, the reducer counts the degree of the vertex and attaches it to the original record excluding *reserve* records. Note, the output format of phase 2 is equal to the stage 1 of Cohen's step 1 in algorithm 1. Thus, we can use his algorithm to finish finding $Tri(E_{add} \bigcup E_{adj})$. Here, we can merge $Tri(E_{add} \bigcup E_{adj})$ and $Tri(G_{n-1})$ to get $T_3(G_n)$ if there is no edge removed in G_n (we will illustrate how to eliminate redundant triangles later). There is still a detail when merging two half degree records. Some vertices' degree may be zero because the vertices is included by E_{adj} but not included by E_{add}. Fortunately, there is no interference for our algorithm.

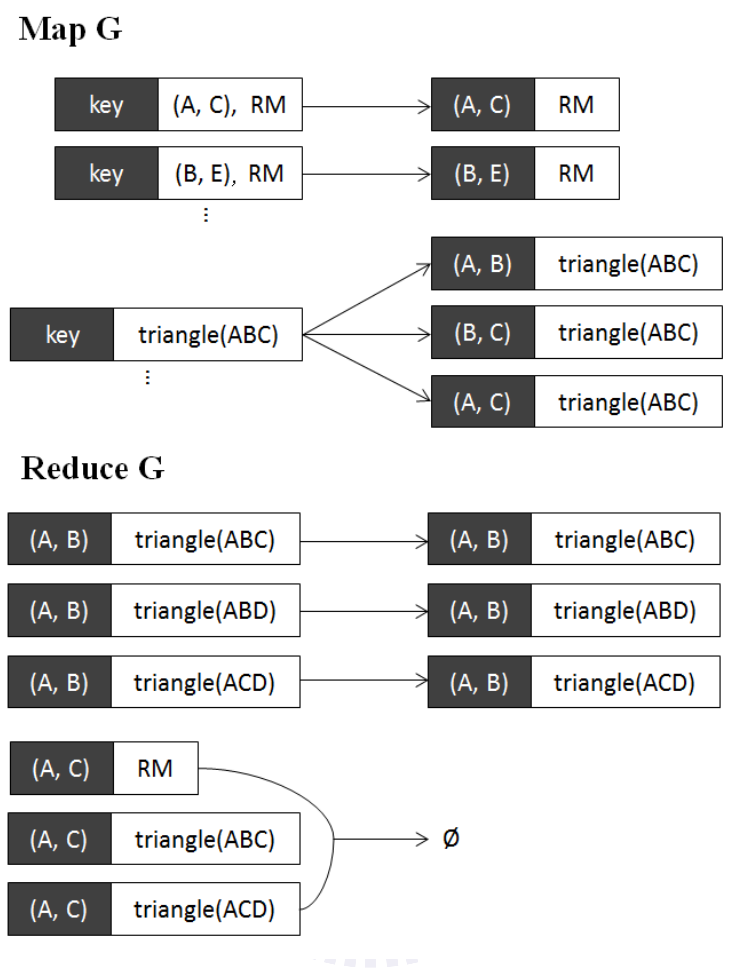

Figure 4.6: Discard the edges not appearing in current graph (phase 3)

Now let's consider removing edges from G_{n-1}. Recalling section 4.3, we propose if some edges are removed from the network, we can get $T_3(G_n)$ by removing triangles containing the edges from $Tri(G_{n-1})$. In order to discard the triangles, we first discard the edges not appearing in current graph. The example is shown in figure 4.6; If the record contains *RM* message in Map method, the mapper clones it; If the record represents a triangle, the mapper emits three records whose key is an edge of the triangle and value is the triangle. In Reduce method, there are two situations: if there is no *RM* message, the reducer emits the original records; Otherwise, it emits nothing.

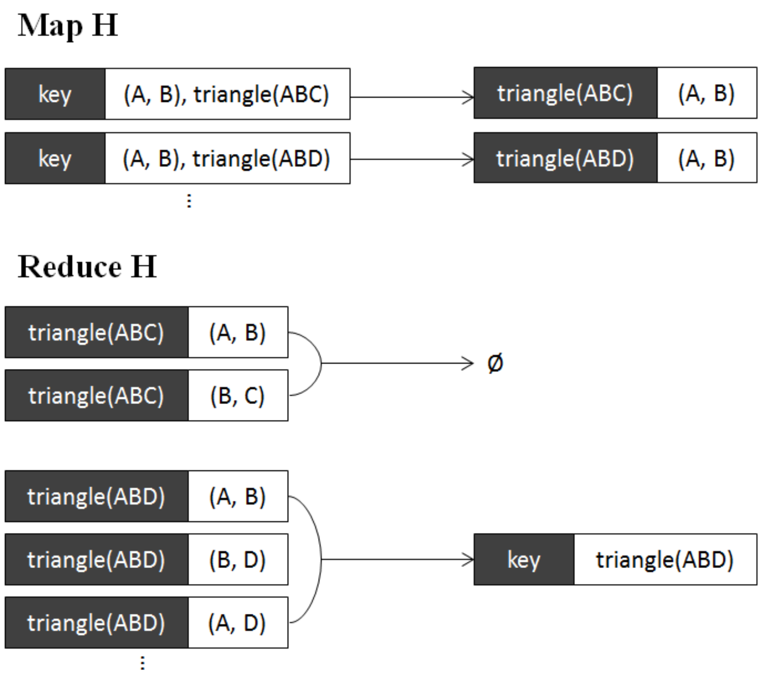

Figure 4.7: Discard the triangles not appearing in current graph (phase 4)

After discarding the edges not appearing in current graph, we can discard the triangle containing these edges. The example is shown in figure 4.7; In phase 4, the mapper changes the position of key and value to prepare for the reducer. In Reduce method, if a triangle possesses three edges, the reducer merges them to one triangle record; Otherwise, it emits no record. Here, we can get $T_3(G_n)$ if there is no edge added in G_n.

When having the algorithm for adding and removing edges, we can cascade them to process ΔG. Why? We can assume there exists an intermediate graph $G_{n-0.5}$ between G_{n-1} and G_n. It contains all the newly added edges in G_n but not removing any edge from G_{n-1}. In this perspective, we can explain why our algorithm is correct.

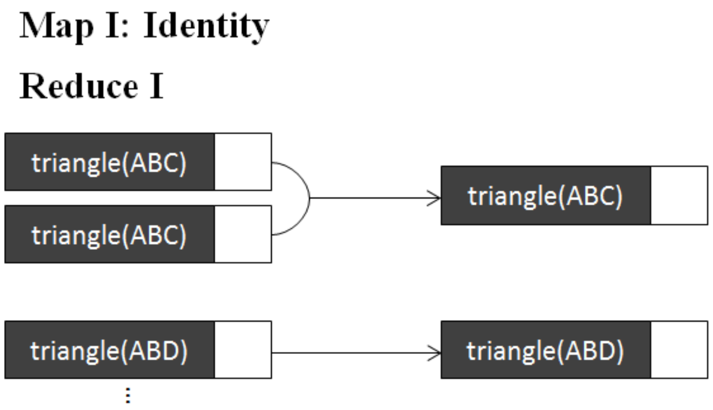

Figure 4.8: Delete redundant triangles (phase 5)

Recall that we have mentioned phase 2 (MapReduce F) may produce redundant triangles. Thus, we think up a method to solve the problem in this phase. An example is shown in figure 4.8 (it is not a correct example for figure 4.2 in logic). The Map method is identity and the reducer only emits one record for each triangle (key).

Algorithm 7: Adaptive algorithm for Truss finding in MapReduce

```
   Input: $G_n$, $G_{n-1}$ and $Tri(G_{n-1})$
   Output: $T_k(G_n)$
1  Find the changes of the network. (MapReduce E)
2  Reserve $E_{add}$ and $E_{adj}$. Then, calculate their degrees. (MapReduce F)
3  The stage 2 of augment the edges with vertex valences.
4  Enumerate triangles for $Tri(E_{add} \bigcup E_{adj})$.
5  Discard the edges not appearing in current graph. (MapReduce G)
6  Discard the triangles not appearing in current graph. (MapReduce H)
7  Delete redundant triangles. (MapReduce I)
8  Preprocessing. (MapReduce A)
9  Triangle-based filter (MapReduce B)
10 Update support. (MapReduce C)
11 If there is any record with $sup - 1$, return to step 9.
12 Merge records which have the same edge. (MapReduce D)
13 Find the components of remaining graph; each is a truss.
```

Let us conclude the algorithm and it is shown in algorithm 7. The input is G_n, G_{n-1} and $Tri(G_{n-1})$. The output is $T_k(G_n)$ (only complete adaptive for $k = 3$). Step 1 finds the variations of the network. Step 2-4 find newly added triangle in G_n. Step 5-6 delete non-existent triangles in G_n. Step 7 deletes redundant triangles. Step 8-12 find $T_k(G_n)$. Finally, Step 13 traverses all k-truss components.

Chapter 5

Experiment Results

5.1 Environment

The experiment cloud is composed of ten computers and the specifications are shown in table 5.1. The version of hadoop is 1.0.4 and the setting is default. In addition, the reducer number of our algorithm is set to eighteen.

Table 5.1: The computers' specification of the cloud

ID	CPU	RAM	HDD
1	Intel(R) Core(TM)2 Quad CPU Q9500 @ 2.83GHz	4.0GB	290GB
2	Intel(R) Core(TM)2 Quad CPU Q8200 @ 2.33GHz	2.0GB	460GB
3	Intel(R) Core(TM)2 i5 CPU 650 @ 3.20GHz	2.0GB	220GB
4	Intel(R) Core(TM)2 Quad CPU Q8300 @ 2.50GHz	2.0GB	460GB
5	Intel(R) Core(TM) i5-2300 CPU @ 2.80GHz	4.0GB	460GB
6	Intel(R) Core(TM)2 Quad CPU Q8300 @ 2.50GHz	2.0GB	460GB
7	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz	4.0GB	460GB
8	Intel(R) Core(TM) i5 CPU 760 @ 2.80GHz	2.0GB	460GB
9	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz	4.0GB	460GB
10	Intel(R) Core(TM)2 Quad CPU Q8400 @ 2.66GHz	2.0GB	290GB

5.2 Test Data

In this section, we will introduce test data in detail. Data is divided into two categories including artificial data and real world data. Synthetic data refers to data generated from random graph generator, and real world data refers to data obtained from real world (e.g. A graph crawled from Facebook).

5.2.1 Synthetic Data

From the viewpoint of complexity, we do not want the model of test data is too complex. Thus, we give priority to use Erdős–Rényi model [11]. An Erdős–Rényi random graph $ER(n, p)$ denotes an undirected graph in which each two distinct vertices are connected by an edge with probability p. We can use it to generate an initial random network. To test adaptive algorithm, we design an algorithm to change the initial network. It is shown in algorithm 8 and we will introduce it in detail in the following paragraph.

To get the next snapshot of the original ER graph, we need changing edges in the original graph (in this model we don't consider adding or removing vertices). First, we define the change rate for a graph as a fractional number whose numerator is the total number of newly added and removed edges, while denominator is the total number of the edges in the initial graph. Note that x is the rate of removed edges, y is the rate of new edges, $\mathbf{G_0}$ represents the initial random graph and E is the edge set for clique with parameter n.

Algorithm 8: Algorithm for changing Erdős–Rényi model

```
   Input: $G_0 : ER(n, p)$
   Output: $G_1$
1  Read the input file and construct data structure.
2  For each edge in $G_0$, it has the probability $\frac{c}{2}$ removed from the graph.
3  For each edge not in $G_0$ but in $E$, it has the probability $\frac{p}{2\cdot(1-p)} \cdot c$
   added into the graph.
4  Write the output file.
```

The values of x and y have to satisfy the following two conditions: ① The change rate is equal to c. ② The total number of newly added edges equals the total number of removed edges. Based on the above conditions, we have the following two equations:

$$|E(\mathbf{G_0})| \cdot x\% + (|E| - |E(\mathbf{G_0})|) \cdot y\% = |E(\mathbf{G_0})| \cdot c\% \tag{5.1}$$

$$|E(\mathbf{G_0})| \cdot x\% = (|E| - |E(\mathbf{G_0})|) \cdot y\% \tag{5.2}$$

Solving the above set of linear equations, we have

$$x = \frac{|E(\mathbf{G_0})|}{2 \cdot |E(\mathbf{G_0})|} \cdot c = \frac{c}{2} \tag{5.3}$$

$$y = \frac{|E(\mathbf{G_0})|}{2 \cdot (|E| - |E(\mathbf{G_0})|)} \cdot c \approx \frac{p \cdot |E|}{2 \cdot (1-p) \cdot |E|} \cdot c = \frac{p}{2 \cdot (1-p)} \cdot c \tag{5.4}$$

Finally, we set x and y to the probability $\frac{c}{2}$ and $\frac{p}{2\cdot(1-p)} \cdot c$ respectively. Thus, our model could satisfy the above two conditions.

5.2.2 Real World Data

The real data is obtained from the website [21] (Stanford Large Network Dataset Collection). The detail of the dataset is shown in Table 5.2. For Enron email, vertices represent email users, and edges represent the relation which two users send at least an email to each other. For Amazon, vertices represent products, and edges represent the relation which two products are frequently co-purchased. For Youtube, vertices represent registered users, and edges represent the relation which two users subscribe to each other. For LiveJournal and Orkut, vertices represent users, and edges represent the friend relation between two users. LiveJournal is a popular SNS in America and Russia, while Orkut is a popular SNS in Brazil and India.

Table 5.2: Real world dataset

Name	Nodes	Edges	Size	Category
Enron email	36,692	367,662	3.86 MB	Email communication network
Amazon	334,863	925,872	12.0 MB	Product network
Youtube	1,134,890	2,987,624	36.9 MB	Online social network
LiveJournal	3,997,962	34,681,189	478 MB	Online social network
Orkut	3,072,441	117,185,083	1.64 GB	Online social network

5.3 Correctness Verification

Because the results of truss finding algorithms are also big data, we design an algorithm in MapReduce to verify them. The inputs of the algorithm are the results $T_k(G)$ of original and improved algorithm in chapter 3 (or conventional and adaptive algorithm in chapter 4). The output of the algorithm is whether they are the same? Let's describe the process. For each edge, the mapper emits one record whose key is the edge that belongs to a truss and value is an empty string. For each key in Reduce, the reducer doesn't emit anything if there are just two records. Otherwise, it emits a message. After finishing the process, the output is set to they are the same if the reducer doesn't emit any record.

5.4 Evaluate Improved Algorithm

In this section, we focus on analyzing the runtime of datasets. For each dataset, we give a range of k to compare the runtime between original and improved truss finding algorithms. Furthermore, we show the runtime of each phase to understand the bottleneck of two algorithms.

Table 5.3 shows the runtime of original and improved algorithm. The table also shows the round number of the while loop used to find $T_k(G)$ (loop1) and which used to find components (loop2). It also shows the percentage of vertices and edges which k-truss versus the original graph. Furthermore, we visualize the runtime in figure 5.1 to make it easier to understand. In the figure, we can discover two points: ① The runtime of both algorithms is proportional to the round number of while loop. ② The round number and k are not related. Finally, we show the accumulate runtime of each phase for $k = 4, 5$ in table 5.4, 5.5 to give a detailed view. We use the framework to show our experiment results from section 5.4.1 to 5.4.3. However, because the runtime of LiveJournal (section 5.4.4) is too long, we only show the result for k = 4.

Let us conclude our experiment results. When the user number of social network is smaller than 1 million people, our algorithm is always better than the original. The reason is that the communication cost between computers cannot be ignored in this scale. Thus, our algorithm takes two phase to update triangles is better than which takes five phase to update triangles. When the user number of social network is larger than 1 million people, our algorithm is equal to or better than the original algorithm. The above is depend on the topology of social networks, in general, our algorithm has better performance in social networks with lower fraction of closed triangles.

5.4.1 Enron email

Enron email is an email communication network. It includes about 37 thousand vertices and 368 thousand edges, furthermore, the size of the file is 3.86 MB and the fraction of closed triangles is 0.0853.

Table 5.3: The runtime of Enron email

K	Original	Improved	loop1	loop2	Nodes	Edges
3	13 min 23 sec	13 min 19 sec	1	8	67%	46%
4	27 min 49 sec	20 min 52 sec	6	8	56%	44%
5	33 min 53 sec	23 min 17 sec	8	8	39%	38%
6	37 min 4 sec	24 min 0 sec	9	7	28%	33%
7	46 min 55 sec	28 min 8 sec	12	8	20%	27%
8	51 min 26 sec	29 min 7 sec	14	6	11%	21%
9	52 min 39 sec	32 min 22 sec	14	7	8%	17%
10	60 min 48 sec	32 min 20 sec	17	6	6%	15%
11	67 min 10 sec	34 min 39 sec	19	6	5%	13%
12	91 min 45 sec	44 min 38 sec	27	6	4%	10%

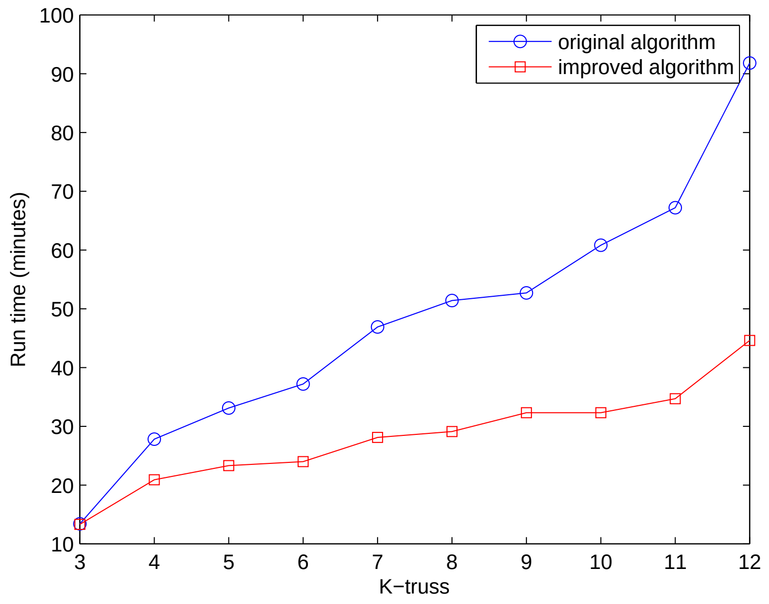

Figure 5.1: Visualize the runtime of Enron email

Table 5.4: The detailed runtime of Enron email (k = 4)

Phase	**Original**	**Improved**
Degree1	3 min 32 sec (13%)	0 min 36 sec (3%)
Degree2	3 min 29 sec (13%)	0 min 35 sec (3%)
Triangle1	3 min 28 sec (12%)	0 min 35 sec (3%)
Triangle2	4 min 0 sec (14%)	0 min 40 sec (3%)
DropEdge	3 min 29 sec (13%)	
Preprocessing		0 min 37 sec (3%)
TriangleBasedFilter		3 min 39 sec (17%)
UpdateSupport		3 min 42 sec (18%)
MergeEdge		0 min 35 sec (3%)
ComponentInit	0 min 35 sec (2%)	0 min 35 sec (3%)
Component1	4 min 40 sec (17%)	4 min 37 sec (22%)
Component2	4 min 36 sec (17%)	4 min 41 sec (22%)
Total	27 min 49 sec	20 min 52 sec

Table 5.5: The detailed runtime of Enron email (k = 5)

Phase	**Original**	**Improved**
Degree1	4 min 43 sec (14%)	0 min 35 sec (3%)
Degree2	4 min 37 sec (14%)	0 min 35 sec (3%)
Triangle1	4 min 39 sec (14%)	0 min 35 sec (3%)
Triangle2	5 min 20 sec (16%)	0 min 41 sec (3%)
DropEdge	4 min 40 sec (14%)	
Preprocessing		0 min 38 sec (3%)
TriangleBasedFilter		4 min 45 sec (21%)
UpdateSupport		4 min 45 sec (21%)
MergeEdge		0 min 35 sec (3%)
ComponentInit	0 min 37 sec (2%)	0 min 34 sec (2%)
Component1	4 min 38 sec (14%)	4 min 36 sec (20%)
Component2	4 min 39 sec (14%)	4 min 38 sec (20%)
Total	33 min 53 sec	23 min 17 sec

5.4.2 Amazon

Amazon is a product network. It includes about 335 thousand vertices, and 926 thousand edges. Furthermore, the size of the file is 12.0 MB and the fraction of closed triangles is 0.2052.

Table 5.6: The runtime of Amazon

K	Original	Improved	loop1	loop2	Nodes	Edges
3	61 min 47 sec	62 min 12 sec	1	47	79%	77%
4	67 min 59 sec	57 min 19 sec	8	36	49%	50%
5	41 min 1 sec	26 min 26 sec	10	9	22%	23%
6	31 min 10 sec	18 min 30 sec	7	5	5%	6%
7	20 min 42 sec	12 min 53 sec	4	3	0.1%	0.1%
8	15 min 1 sec	9 min 4 sec	3	1	0%	0%
9	14 min 58 sec	9 min 10 sec	3	1	0%	0%
10	11 min 49 sec	9 min 5 sec	3	1	0%	0%
11	11 min 47 sec	9 min 3 sec	3	1	0%	0%
12	11 min 45 sec	7 min 44 sec	2	1	0%	0%

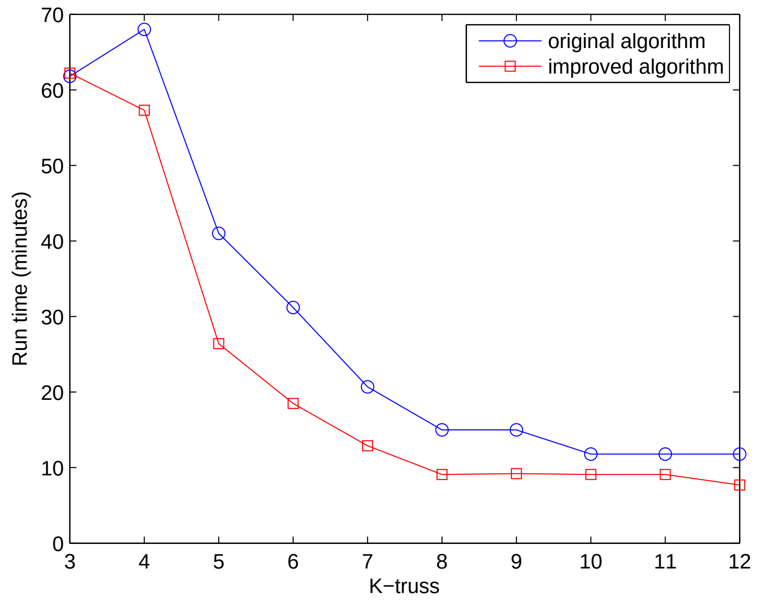

Figure 5.2: Visualize the runtime of Amazon

Table 5.7: The detailed runtime of Amazon (k = 4)

Phase	**Original**	**Improved**
Degree1	4 min 45 sec (7%)	0 min 41 sec (1%)
Degree2	4 min 37 sec (7%)	0 min 35 sec (1%)
Triangle1	4 min 39 sec (7%)	0 min 35 sec (1%)
Triangle2	5 min 24 sec (8%)	0 min 43 sec (1%)
DropEdge	4 min 37 sec (7%)	
Preprocessing		0 min 38 sec (1%)
TriangleBasedFilter		4 min 54 sec (9%)
UpdateSupport		4 min 57 sec (9%)
MergeEdge		0 min 35 sec (1%)
ComponentInit	0 min 34 sec (1%)	0 min 34 sec (1%)
Component1	22 min 12 sec (33%)	22 min 4 sec (38%)
Component2	21 min 11 sec (31%)	21 min 3 sec (37%)
Total	67 min 59 sec	57 min 19 sec

Table 5.8: The detailed runtime of Amazon (k = 5)

Phase	**Original**	**Improved**
Degree1	5 min 59 sec (15%)	0 min 41 sec (3%)
Degree2	5 min 48 sec (14%)	0 min 35 sec (2%)
Triangle1	5 min 48 sec (14%)	0 min 35 sec (2%)
Triangle2	6 min 36 sec (16%)	0 min 40 sec (3%)
DropEdge	5 min 50 sec (14%)	
Preprocessing		0 min 37 sec (2%)
TriangleBasedFilter		5 min 51 sec (22%)
UpdateSupport		5 min 52 sec (22%)
MergeEdge		0 min 35 sec (2%)
ComponentInit	0 min 35 sec (1%)	0 min 35 sec (2%)
Component1	5 min 12 sec (13%)	5 min 14 sec (20%)
Component2	5 min 13 sec (13%)	5 min 11 sec (19%)
Total	41 min 1 sec	26 min 26 sec

5.4.3 Youtube

Youtube is a video-sharing web site that includes a social network. It includes about 1.13 million vertices, and 2.99 million edges. Furthermore, the size of the file is 36.9 MB and the fraction of closed triangles is 0.0062.

Table 5.9: The runtime of Youtube

K	Original	Improved	loop1	loop2	Nodes	Edges
3	21 min 41 sec	21 min 38 sec	1	13	23%	47%
4	49 min 53 sec	36 min 52 sec	11	10	8%	28%
5	58 min 44 sec	39 min 5 sec	15	8	4%	18%
6	77 min 44 sec	45 min 56 sec	20	9	2%	13%
7	72 min 41 sec	42 min 23 sec	20	6	1%	9%
8	69 min 43 sec	39 min 58 sec	19	6	0.7%	7%
9	90 min 37 sec	49 min 9 sec	26	6	0.5%	5%
10	121 min 36 sec	60 min 58 sec	36	6	0.4%	4%
11	80 min 47 sec	43 min 35 sec	23	6	0.3%	3%
12	101 min 25 sec	50 min 59 sec	30	5	0.2%	2%

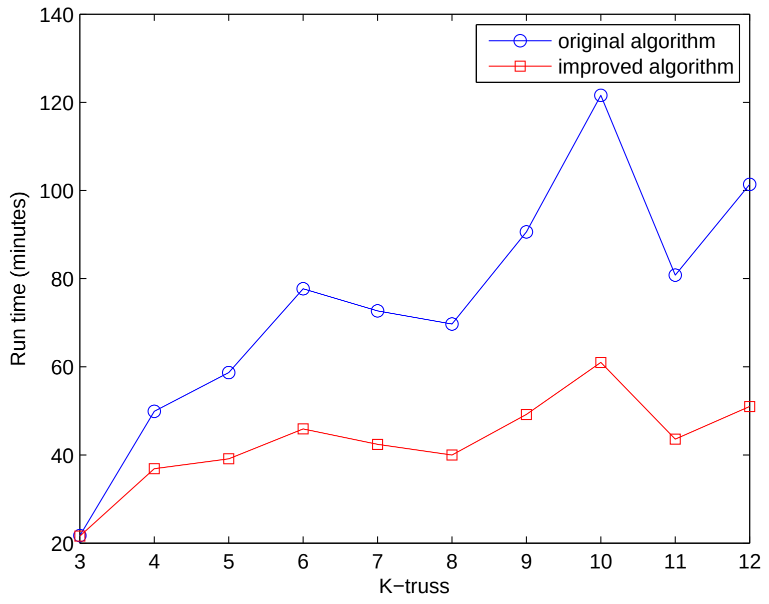

Figure 5.3: Visualize the runtime of Youtube

Table 5.10: The detailed runtime of Youtube (k = 4)

Phase	**Original**	**Improved**
Degree1	7 min 1 sec (14%)	1 min 19 sec (4%)
Degree2	6 min 31 sec (13%)	0 min 38 sec (2%)
Triangle1	7 min 37 sec (15%)	0 min 45 sec (2%)
Triangle2	8 min 39 sec (17%)	1 min 4 sec (3%)
DropEdge	7 min 6 sec (14%)	
Preprocessing		0 min 47 sec (2%)
TriangleBasedFilter		9 min 21 sec (25%)
UpdateSupport		9 min 9 sec (25%)
MergeEdge		0 min 41 sec (2%)
ComponentInit	0 min 35 sec (1%)	0 min 38 sec (2%)
Component1	6 min 17 sec (13%)	6 min 14 sec (17%)
Component2	6 min 7 sec (12%)	6 min 16 sec (17%)
Total	49 min 53 sec	36 min 52 sec

Table 5.11: The detailed runtime of Youtube (k = 5)

Phase	**Original**	**Improved**
Degree1	9 min 13 sec (16%)	1 min 15 sec (3%)
Degree2	8 min 43 sec (15%)	0 min 40 sec (2%)
Triangle1	9 min 44 sec (17%)	0 min 50 sec (2%)
Triangle2	11 min 25 sec (19%)	1 min 0 sec (2%)
DropEdge	9 min 33 sec (16%)	
Preprocessing		0 min 49 sec (2%)
TriangleBasedFilter		12 min 6 sec (31%)
UpdateSupport		11 min 36 sec (30%)
MergeEdge		0 min 39 sec (2%)
ComponentInit	0 min 34 sec (1%)	0 min 35 sec (2%)
Component1	4 min 48 sec (8%)	4 min 53 sec (12%)
Component2	4 min 44 sec (8%)	4 min 42 sec (12%)
Total	58 min 44 sec	39 min 5 sec

5.4.4 LiveJournal

LiveJournal is an online social network. It includes about 4.00 million vertices, and 34.7 million edges. Furthermore, the size of the file is 478 MB and the fraction of closed triangles is 0.1154. In this dataset, our algorithm detects the performance of our algorithm may worse than the original so it automatically switches to the original algorithm to avoid increasing computational cost.

Table 5.12: The detailed runtime of LiveJournal (k = 4)

Phase	**Original**	**Improved**
Degree1	40 min 19 sec (6%)	40 min 2 sec(6%)
Degree2	31 min 8 sec (4%)	31 min 28 sec (4%)
Triangle1	2 hour 12 min (19%)	2 hour 16 min (19%)
Triangle2	4 hour 54 min (42%)	4 hour 52 min (42%)
DropEdge	2 hour 29 min (21%)	2 hour 32 min (21%)
ComponentInit	1 min 57 sec (0.3%)	1 min 56 sec (0.3%)
Component1	28 min 49 sec (4%)	28 min 50 sec (4%))
Component2	25 min 58 sec (4%)	26 min 5 sec (4%)
Total	11 hour 43 min	11 hour 48 min

5.5 Evaluating the Adaptive Algorithm

In this section, we use the synthetic data described in section 5.2.1 to evaluate the performance of the adaptive algorithm. In addition, k is set to three and the change rate is from 1% to 8%. Note, we don't consider finding components because the adaptive algorithm doesn't focus on it. Adaptive algorithm is better than the original if the change rate is less than five. Actually, our algorithm has some limitations. Recall our algorithm is composed of adding triangle and removing triangle phase. In the adding triangle phase, the time complexity is proportional to $Tri(E_{add} \bigcup E_{adj})$. In the removing triangle phase, the time complexity is proportional to $3 \cdot |Tri(G)|$ as in chapter 3.

The first test graph is an ER random graph with 1 million vertices and the average degree is 20. In addition, the size of the file is 293MB and the fraction of closed triangles is smaller than millionth.

Table 5.13: The runtime of adaptive simulation 1

Change rate	**Original**	**Adaptive**	**Gain**
1%	10 min 9 sec	7 min 20 sec	28%
2%	9 min 27 sec	7 min 28 sec	21%
3%	10 min 5 sec	7 min 10 sec	29%
4%	9 min 40 sec	7 min 56 sec	18%
5%	10 min 0 sec	9 min 17 sec	7%
6%	9 min 25 sec	10 min 18 sec	-9%
7%	9 min 49 sec	10 min 32 sec	-7%
8%	9 min 36 sec	10 min 49 sec	-11%

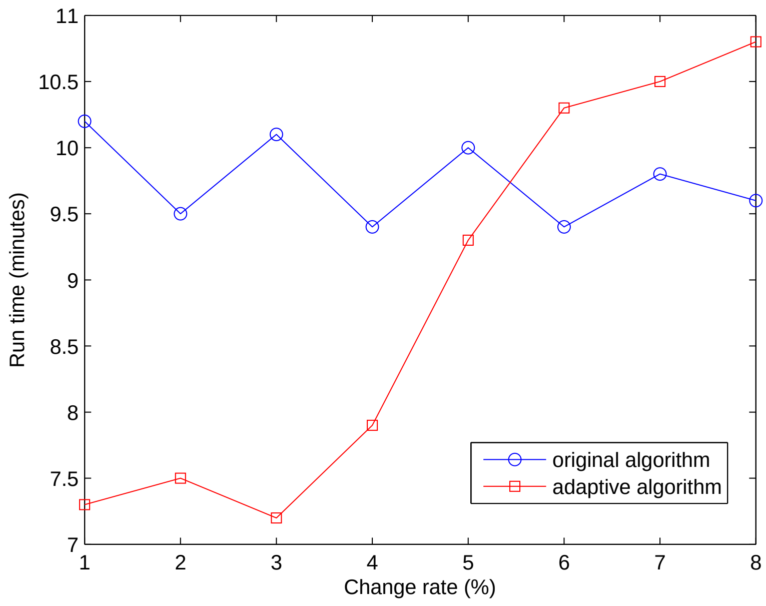

Figure 5.4: Visualize the runtime of adaptive simulation 1

The second test graph is an ER random graph with 4 million vertices and the average degree is 20. In addition, the size of the file is 599MB and the fraction of closed triangles is smaller than millionth.

Table 5.14: The runtime of adaptive simulation 2

Change rate	**Original**	**Adaptive**	**Gain**
1%	12 min 26 sec	8 min 10 sec	34%
2%	11 min 57 sec	8 min 47 sec	26%
3%	12 min 35 sec	9 min 43 sec	23%
4%	12 min 53 sec	11 min 16 sec	13%
5%	13 min 0 sec	11 min 38 sec	11%
6%	11 min 32 sec	12 min 18 sec	-7%
7%	12 min 1 sec	13 min 43 sec	-14%
8%	12 min 40 sec	13 min 52 sec	-10%

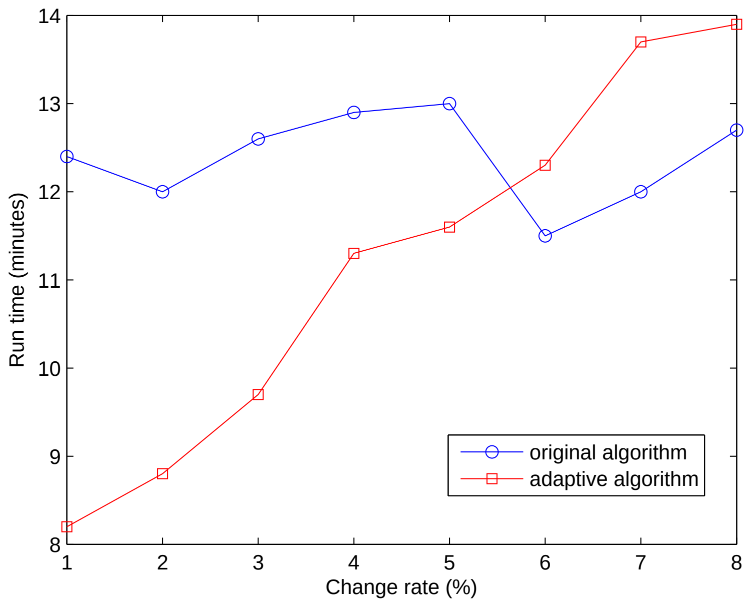

Figure 5.5: Visualize the runtime of adaptive simulation 2

Chapter 6

Conclusion

In this thesis, we focus on designing MapReduce clustering algorithm for analyzing massive online social networks. K-truss is a cohesive subgraph which can be computed in polynomial time. In addition, it could be used to represent a community in a social network. We propose two algorithms applied to two situations. The first one designed is for static social networks. In other words, it can analyze a social network at a snapshot. The second one is designed for dynamic social network. Given two close snapshots of a social network, we use the analytical results of the first one to speed up the analysis of the second one. The basic idea of our algorithms is to reuse the information and avoid unnecessary recomputing. The experimental results show that the first algorithm is is faster than the original algorithm if user population is less than 1 million or the guessed accuracy is larger than $\frac{1}{5}$. In addition, we propose an efficient method to predict if the proposed algorithm could improve the performance. For the second algorithm, the simulation results show that our algorithm is better than the original one if the change rate is less than five percent. Future works include applying our algorithms to real dynamic social networks. There exist some obstacles such as most of social networks has different properties from ER random graph and may out of our limitations.

Bibliography

[1] "Socialbakers." Retrieved May 15, 2013 from the web: `http://www.checkfacebook.com/`.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, pp. 107–113, January 2008.

[3] "Hadoop mapreduce tutorial." Retrieved May 15, 2013 from the web: `http://hadoop.apache.org/docs/r1.0.4/mapred_tutorial.html`.

[4] "Wiki mapreduce." Retrieved May 15, 2013 from the web: `http://en.wikipedia.org/wiki/MapReduce`.

[5] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce.* April 2010.

[6] J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham, "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, pp. 1312–1327, September 2011.

[7] I. Palit and C. K. Reddy, "Scalable and Parallel Boosting with MapReduce," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, pp. 1904–1916, October 2012.

[8] S. Sehrish, G. Mackey, P. Shang, J. Wang, and J. Bent, "Supporting HPC Analytics Applications with Access Patterns Using Data Restruc-

turing and Data-Centric Scheduling Techniques in MapReduce," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, pp. 158–169, January 2013.

[9] A. Nandi, C. Yu, P. Bohannon, and R. W Ramakrishnan, "Data Cube Materialization and Mining over MapReduce," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, pp. 1747–1759, October 2012.

[10] R. D. Luce and A. D. Perry, "A method of matrix analysis of group structure," *Psychometrika*, vol. 14, pp. 95–116, June 1949.

[11] M. van Steen, *Graph Theory and Complex Networks: An Introduction.* January 2010.

[12] R. J. Mokken, "Cliques, clubs and clans," *Quality and Quantity*, vol. 13, pp. 161–173, April 1979.

[13] S. B. Seidman and B. L. Foster, "A graph-theoretic generalization of the clique concept," *Journal of Mathematical Sociology*, vol. 6, pp. 139–154, 1978.

[14] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, p. 269–287, September 1983.

[15] J. Cohen, "Trusses: Cohesive Subgraphs for Social Network Analysis," *technical report*, pp. 1–29, 2008.

[16] J. Cohen, "Graph twiddling in a mapreduce world," *IEEE Computing in Science and Engineering*, vol. 11, pp. 29–41, July-August 2009.

[17] J. Wang and J. Cheng, "Truss decomposition in massive networks," *ACM Proc. VLDB Endowment (PVLDB)*, vol. 5, pp. 812–823, May 2012.

[18] D. Greene, D. Doyle, and P. Cunningham, “Tracking the Evolution of Communities in Dynamic Social Networks,” *IEEE/ACM ASONAM*, pp. 176–183, August 2010.

[19] Y.-R. Lin, Y. Chi, S. Zhu, H. Sundaram, and B. L. Tseng, “FacetNet: A Framework for Analyzing Communities and Their Evolutions in Dynamic Networks,” *WWW ’08 Proceedings of the 17th international conference on World Wide Web*, pp. 685–694, 2008.

[20] N. Nguyen, “Adaptive Algorithms for Detecting Community Structure in Dynamic Social Networks,” *IEEE INFOCOM*, pp. 2282–2290, April 2011.

[21] “Stanford large network dataset collection.” Retrieved May 30, 2013 from the web: `http://http://snap.stanford.edu/data/`.