Truss Decomposition in Large Probabilistic Graphs

by

Mahsa Daneshmandmehrabani
B.Sc. (Mathematics), Sharif University of Technology, Iran, 2015
M.Sc. (Mathematics), Brock University, Canada, 2017

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Mahsa Daneshmand, 2019
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.
Truss Decomposition in Large Probabilistic Graphs

by

Mahsa Daneshmandmehrabani
B.Sc. (Mathematics), Sharif University of Technology, Iran, 2015
M.Sc. (Mathematics), Brock University, Canada, 2017

Supervisory Committee

Dr. Alex Thomo, Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Committee Member
(Department of Computer Science)
ABSTRACT

Truss decomposition is an essential problem in graph mining, which focuses on discovering dense subgraphs of a graph. Detecting trusses in deterministic graphs is extensively studied in the literature. As most of the real-world graphs, such as social, biological, and communication networks, are associated with uncertainty, it is of great importance to study truss decomposition in a probabilistic context. However, the problem has received much less attention in a probabilistic framework. Furthermore, due to computational challenges of truss decomposition in probabilistic graphs, state-of-the-art approaches are not scalable to large graphs.

Formally, given a user-defined threshold $k$ (for truss denseness), we are interested in finding all the maximal subgraphs, which are a $k$-truss with high probability. In this thesis, we introduce a novel approach based on an asynchronous $h$-index updating process, which offers significant improvement over the state-of-the-art. Our extensive experimental results confirm the scalability and efficiency of our approach.
# Contents

[Supervisory Committee] ii

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

Acknowledgements ix

Dedication x

1 Introduction 1

1.1 Motivation 2

1.2 Contribution 3

1.3 Organization 4

2 Graphs and Trusses 6

2.1 Graphs 6

2.1.1 Probabilistic Graphs 7
# Cohesive Subgraphs

## The Importance of $k$-trusses

## $K$-Truss in Deterministic Graphs

### Preliminaries

### Properties of $k$-trusses

### Discovering Maximal Trusses

### Related Works on $K$-truss Decomposition in Deterministic Graphs

## $k$-Truss in Probabilistic graphs

### Local and Global $(k, \gamma)$-truss

### Probabilistic Truss Decomposition’s Challenges

### Related Works on $k$-truss Decomposition in Probabilistic Graphs

#### Edge Support Computation Using Dynamic Programming

#### Updating Edge Supports

#### Probabilistic Truss Decomposition Using Central Limit Theorem

# The New Probabilistic Truss Decomposition Algorithm

## New Approach

### Optimized Dynamic Programming

## Correctness of the Proposed Algorithm

# Experiments, Results and Discussions

## The Machine

## The Datasets and Webgraph
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Initial $\gamma$-support, values obtained by Phase I and Phase II, respectively. $\gamma = 0.2$ for the example.</td>
<td>46</td>
</tr>
<tr>
<td>4.1</td>
<td>Dataset Statistics</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Max $\eta$-support, max probabilistic trussness, $\gamma$.</td>
<td>51</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1 A probabilistic graph $G$ .................................................. 9
Figure 2.2 A possible world of $G$ .................................................. 9
Figure 2.3 Graph $G$ and its $k$-classes for $k \leq k \leq 5$ [41] .......... 13
Figure 2.4 A 4-truss which does not contain a 4-clique ................. 14
Figure 2.5 A 4-truss with vertex connectivity of 1 .......................... 16
Figure 2.6 A probabilistic graph in which the $\gamma - \text{sup}(e = (3,5)) = 1$ for $\gamma = 0.2$. .................................................. 25
Figure 2.7 A probabilistic graph $G$ .................................................. 26
Figure 2.8 Local $(4,0.216)$-truss $H_1$ ............................................. 26
Figure 3.1 Example ........................................................................... 46
Figure 4.1 Running time of our $h$-index based updating algorithm versus edge peeling for probabilistic truss decomposition .......... 52
Figure 4.2 Average error (average of difference from true trussness) versus iterations for $\gamma = 0.1$ .................................................. 53
Figure 4.3 Average error (average of difference from true trussness) versus iterations for $\gamma = 0.3$ .................................................. 54
Figure 4.4 Average error (average of difference from true trussness) versus iterations for $\gamma = 0.5$. .................................................. 54
ACKNOWLEDGEMENTS

I would like to thank:

Dr. Alex Thomo, for being supportive and providing constant encouragement. I highly appreciate his mentoring and advices throughout my studies at University of Victoria.

my family, for their endless love, support and always believing in me.

Fatemeh Esfahani, for collaboration on this project and sharing her experiences and inspiration.

University of Victoria, for funding me with a Scholarship.
DEDICATION

This work is dedicated to the soul of my wonderful teacher; Dr. Babak Farzad, who had a great influence on my life.
Chapter 1

Introduction

Analysis of network data is crucial in various scientific fields such as social, biological, and information networks. Real-world networks typically are not deterministic and are associated with uncertainty. This uncertainty in many domains expresses intensity or the confidence of the relations among components of a dataset \cite{7, 14}. The uncertainty may have various reasons, such as the data collection process, using methods based on machine learning to process the data, privacy-related issues, or fuzzy join of information (cf. \cite{5, 18, 21, 22, 37, 38, 39}). Mostly, these uncertain data and networks are modeled as \textit{probabilistic graphs}, which sometimes are called \textit{uncertain graphs} as well in the literature. Edges in probabilistic graphs are labeled with probabilities, each of which expresses the existence likelihood of the corresponding edge \cite{5, 11, 24}. As an example, in PPI networks, a probabilistic model for the protein interaction network is introduced in which a probability-weighted edge represents the interaction and its strength between each protein pair \cite{2, 13}. In social networks, interactions between users can also be modeled using probabilities to express peer
influence and link prediction [18, 25, 30, 42].

A fundamental task for network data analysis is to identify densely connected (also called cohesive) subgraphs [20, 44, 45]. An interesting notion of a dense subgraph of a graph is $k$-truss.

### 1.1 Motivation

Among different notions of cohesive subgraphs in the literature, the truss notion tells more about hierarchical structure of the graph [44]. Knowing the hierarchical structure of a graph helps to focus on small but most important and informative components of a network as a graph’s trusses are nested in each other. The $k$-truss of a given graph is defined as the maximal subgraph in which each edge is contained in at least $k - 2$ triangles. A triangle $(a, b, c)$ is a set of three edges $(a, b)$, $(b, c)$, and $(c, a)$, where $a, b, c$ are vertices in the graph. If an edge is contained in a triangle, we say that the triangle supports the edge. For example, triangle $(a, b, c)$ supports edges $(a, b)$, $(b, c)$, and $(c, a)$. Intuitively, the notion of support captures the idea of “endorsement” in social networks, e.g. in the above triangle user $c$ endorses the social connection between $a$ and $b$. The number of all triangles that support an edge is called the support of the edge.

Applications of $k$-truss include visualization of complex networks [46], and providing the foundation of several community models [27]. It can also be useful in connection analysis between users of social network, or finding the most influential users in viral marketing and ask them to promote a product. Therefore, it is important to discover $k$-trusses.
Truss decomposition in deterministic graphs is a well-known graph-analytics task and has been widely studied in the literature [44, 17]. However, in probabilistic graphs, truss computation is significantly more challenging and has received much less attention. In this thesis, we use the notion of local probabilistic \((k, \gamma)\)-truss introduced in [18].

1.2 Contribution

The standard approach to compute \(k\)-truss decomposition is an edge peeling process. To find \(k\)-truss, all edges with less than \(k\) triangles as a support are removed repeatedly until no such edges remain. Edges should be kept sorted during the algorithm which requires maintaining global information about the graph at each step of the algorithm. This prevents the scalability of the algorithm considerably. Furthermore, since each step depends on the result from the previous one, approximations on final truss values for each edge are not possible. These problems become more challenging in probabilistic graphs, since triangle count in such graphs has a combinatorial nature [18]. In [18], the support probability of an edge \(e = (u, v)\) is computed using dynamic programming (DP), and then the edge peeling algorithm is used to compute truss decomposition. However, the algorithm is not scalable to large graphs. Thus, it is of great importance to propose an efficient algorithm for truss decomposition in probabilistic graphs.

In deterministic graphs, the \(h\)-index of each edge is initially set to be its triangle count. At each iteration, the \(h\)-index of an edge is updated based on the \(h\)-indices of its neighbors; those edges which form a triangle with the given edge. This process is
repeated until convergence. Upon termination, the final \( h \)-index value of each edge equals its trussness (or truss score). Note that, the process only uses local information about an edge.

Unfortunately, this idea does not work for probabilistic graphs by itself, since it does not consider uncertainty in such graphs, thus resulting in wrong trussness values. In this thesis, we extend the \( h \)-index updating algorithm to work for probabilistic graphs. In particular, we design a procedure which considers properties of probabilistic truss subgraphs and maintains proper upper-bounds on trussness of edges until convergence to true trussness values. In summary, our contributions are as follows:

- We propose an algorithm based on \( h \)-index updating which works for probabilistic graphs. Our proposed algorithm is exact with respect to final result, but also progressive allowing the user to see near-results along the way, and it works by processing only one edge and its immediate neighbors at a time, thus being a small memory-footprint approach.

- We evaluate the performance of our approach on a wide range of data sets. Our experimental results confirm the scalability and efficiency of our algorithm, significantly outperforming state-of-the art [18] for large datasets.

1.3 Organization

This thesis is organized as follows:

Chapter [2]

this chapter presents the definitions and notations on graphs and trusses, and also
states properties of trusses. Moreover, important truss decomposition algorithms in the literature are brought in this chapter.

Chapter 3
this chapter introduces our new truss decomposition algorithm on probabilistic graphs and gives a theoretical analysis of the correctness of the algorithm.

Chapter 4
this chapter presents our experiments and results.

Chapter 5
concludes the thesis.
Chapter 2

Graphs and Trusses

This chapter presents the necessary background and definitions on graphs and trusses.

2.1 Graphs

Typically graphs are the best representation one can choose to indicate relationships between two entities. Graph Theory is the study of graphs and their properties. Because of various applications and interesting features of graphs, representing the data in form of a graph can reveal new information about our data.

A graph is a pair of sets $G = (V, E)$, where $V$ is a set of nodes and $E$ is a set of edges. Edges are formed by pairs of vertices. An edge between vertices $u$ and $v$ are denoted by $(u, v)$. The number of vertices and edges are denoted by $n$ and $m$, respectively. That is, $|V| = n$ and $|E| = m$. For a vertex $u \in V$, the set of its neighbors is denoted by $N(u)$ such that $N(u) = \{v : (u, v) \in E\}$. The degree of $u$ is the number of $u$'s neighbors and is denoted by $\text{deg}(u)$. If edges of a graph have no direction, the graph is called an undirected graph. On the other hand, when edges
have direction, the graph is a *directed* graph. In this thesis, we consider undirected graphs.

In the following paragraph we briefly list some important graph-theoretic notions which might or might not have a direct relation to our truss computation problem.

A *path* in a graph $G$ is a finite (or infinite) sequence of edges that connects a sequence of distinct vertices. The *distance* between two vertices in a graph is measured by the number of edges in the shortest path that connects those two vertices. The *diameter* of a given graph is defined as the maximum distance between any two vertices of a graph $G$. Two vertices $u$ and $v$ in $G$ are said to be connected if there is a path connecting $u$ and $v$ in $G$. The graph $G$ is called *connected* if there is a path between any two vertices of $G$; otherwise $G$ is called *disconnected*. Each partition of a disconnected graph is called a *connected component* of $G$. In other words, a connected graph has exactly one connected component while a disconnected graph has more than one connected component. The *edge connectivity* of a graph $G$ is the minimum number of edges that divide the graph into separate connected components if they are removed from $G$. A *cut set* is a set of edges whose removal will make the graph disconnected. The *vertex connectivity* of graph $G$ is the minimum number of vertices which make the graph disconnected.

### 2.1.1 Probabilistic Graphs

Most of the real-world networks and data are associated with uncertainty due to noise, incompleteness and inaccuracy \[47\]. The best way to represent this type of data is modeling them using *probabilistic graphs*.

A probabilistic graph $\mathcal{G} = (V, E, p)$, is defined over a set of vertices $V$, a set of
edges \( E \) and a probability function \( p : E \to (0, 1] \) which maps every edge \( e \in E \) to an existence probability. Typically, the existence probability of each edge is considered to be independent of other edges. Probabilistic graphs also may have direction on edges similar to deterministic graphs. In this thesis, we focus on undirected probabilistic graphs.

To investigate and analyze probabilistic graphs, the concept of *possible worlds* is defined. Possible worlds are deterministic graph instances of \( G \). A possible world \( G \) of \( G \) is obtained by performing a coin flip for each edge in \( G \) biased by the probability of the edge. If the coin flip results in head we keep the edge, otherwise, if the coin flip results in tail, we remove the edge. For a possible world \( G \) obtained this way from \( G \), we write \( G \sqsubseteq G \). The probability of a possible world \( G \) is obtained as follows:

\[
P(G) = \prod_{e \in E_G} p(e) \prod_{e \notin E_G} (1 - p(e))
\]

Equation 2.1 states that the probability of any possible world of a probabilistic graph can be obtained by multiplying the existing edges’ probabilities by those ones that do not appear in the possible world. Figure 2.1 shows a probabilistic graph, in which the number on each edge represents the edge’s existence probability. Figure 2.2 is a possible world of \( G \) in Figure 2.1. The probability of this possible world is:

\[
0.7 \times 0.33 \times (1 - 0.21) \times (1 - 0.1) \times 046 \times 0.65 \times 0.5 \times (1 - 0.83) = 0.004174
\]
2.2 Cohesive Subgraphs

Much research has been done in mining and querying probabilistic graphs \cite{12, 32, 15, 16, 48, 19, 47, 23}. Several problems have been defined and investigated on uncertain graphs.

An important category of problems is detecting dense or cohesive components within graphs. In this thesis, as elsewhere in the literature, the notions of dense and cohesive subgraphs are used interchangeably.

To analyze huge graphs, it is helpful to only target the smaller but more important sections of the graph. Studying cohesive subgraphs can reveal important information about connectivity and centrality features of the network. Therefore, identifying cohesive subgraphs attracted many researchers’ attention. Different notations are suggested to detect cohesive subgraphs in the literature. \textit{Cliques} are the first and basic representations of cohesive subgraphs \cite{28}. A clique is a subgraph in which every two vertices are connected by an edge. Cliques are not a very appropriate measure due to their rarity problem \cite{9}. That is, the cliques of a few vertices are very...
common while the cliques of a larger number of vertices are rare. On the other hand, detecting cliques does not scale well and makes the clique enumeration impractical.

\( n \)-clique is a relaxed version that allows the distance between any two vertices within the subgraph be \( n \) rather than 1 in cliques. In other words, \( n \)-clique is a set of vertices such that the distance between any two vertices is at most \( n \). The distance is measured in the full graph, not just the subgraph induced by the vertices in the \( n \)-clique. As such, \( n \)-clique has some drawbacks as a measure for a cohesive subgraph: it is probably too scattered over the entire graph which would not be useful. Moreover, the \( n \)-clique enumeration problem is not better than the clique enumeration problem and is still impractical. Furthermore, vertices of a \( n \)-clique may be connected via some intermediaries which are not members of the \( n \)-clique [1] and hence can be disconnected. As a result, \( n \)-cliques cannot depict the tightness and connectedness properties well [29].

To address the problems of \( n \)-cliques, two similar other notions are defined [1, 29]: \( n \)-clans and \( n \)-clubs. The first notion, \( n \)-clan, starts with the \( n \)-cliques that are identified in a network and excludes those that have a diameter greater than \( n \). The other notion, \( n \)-club, defines a new entity, a maximal \( n \)-diameter subgraph. Unfortunately, both \( n \)-clan and \( n \)-club are still computationally impractical.

\( k \)-plex [36] is another notation as a dense subgraph in which the degree of each vertex is \( c - k \) in a subgraph of \( c \) vertices. However, the enumeration and scalability problems still exist in this case.

On the other hand, Seidman [36] introduced the \( k \)-core. A \( k \)-core is is a maximal subgraph such that every vertex of it, is adjacent to at least \( k \) other vertices. A clique
is a $k$-core which has $k + 1$ vertices. The advantage of the $k$-core over generalizations above is that $k$-core can be detected in polynomial time.


2.2.1 The Importance of $k$-trusses

Wang and Cheng [44] claimed that the truss notation tells more about hierarchical structure of the graph. $k$-trusses are cohesive subgraphs similar to $k$-cores but more dense. $k$-cores are subgraphs in which every vertex has degree of at least $k$. $k$-trusses are subgraphs that every edge within the subgraph participates in at least $k - 2$ triangles. While, $k$-cores are based on vertices and their degrees, $k$-trusses depend on edges and the number of triangles that contain edges. Triangles indicate a strong connection between three vertices, i.e. their friends within a social network. Hence $k$-trusses by imposing each edge to participate in at least $k - 2$ triangles, strengthen edges connection by at least $k - 2$ ties [44]. Wang and Chang [44] also showed that $k$-truss is a more suitable cohesive subgraph and it defines tightly-knit clusters of a graph.

2.3 $K$-Truss in Deterministic Graphs

Cohen [9] gives an example to explain the truss concept: If two actors on a social network are strongly connected, they probably share connections to other people on the network. Therefore, Cohen [9] defines the $k$-truss structure such that a connection between two actors $A$ and $B$ is considered legitimate if $k - 2$ other actors support
that connection and are connected to A and B as well. In the following, we define the $k$-truss formally and describe some of its properties in details.

### 2.3.1 Preliminaries

Let $G = (V, E)$ be an undirected graph, such that $V$ is the set of vertices and $E$ is the set of edges. A *triangle* in $G$ is defined as a cycle of vertices $\{u, v, w \in V\}$ such that all the edges $uv$, $vw$ and $uw$ exist. This triangle is denoted by $\Delta_{uvw}$. Then, the support of an edge $e \in E$ denoted by $sup(e)$ can be defined as the number of triangles in graph $G$ containing $e$. The $k$-truss is defined as the non-trivial and one component subgraph in which every edge is augmented by at least $k - 2$ pairs of edges constructing a triangle with the edge. In other words, $k$-truss is the largest subgraph $T_k$ of $G$ in which every edge $e \in E_{T_k}$ has $sup(e) \geq k - 2$ for $k \geq 2$. The *trussness* or *truss number* of an arbitrary edge $e \in E$ denoted by $\phi(e)$ is the maximum value of $k$ such that $T_k$ contains $e$ but $T_{k+1}$ does not. The maximum truss number of any edge $e$ in $G$ is denoted by $k_{max}$. Based on mentioned definitions, we can define the $k$-class of $G$ as: the set of edges that have the same truss number of $k$. In other words, $k$-class of $G$ is: $\Phi(k) = \{e : e \in E, \phi(e) = k\}$.

### 2.3.2 Properties of $k$-trusses

Now we discuss important properties of trusses which are derived from Cohen’s paper [9].

**Observation 2.3.1.** A *clique of order $k$ is a $k$-truss.*
Figure 2.3: Graph $G$ and its $k$-classes for $k \leq 5$.

Every clique over $k$ vertices can be seen as a $k$-truss since each edge connects two vertices that have at least $k - 2$ common neighbors. Therefore, each edge is supported by $k - 2$ pairs of edges.

**Observation 2.3.2.** Every non-trivial component of a graph is a maximal 2-truss.

Cohen [9] presented a naive but sufficient algorithm to find $k$-trusses of a given graph in polynomial time.

**Algorithm 1** Discovering $k$-trusses of $G$

1. function REDUCE_GRAPH_TO_K_TRUSS(Graph $G$)
2. while No change do
3. for each edge $e = (a, b) \in G$ do
4. if $|N(a) \cap N(b)| < k - 2$ then remove $e$ from $G$;
5. Remove isolated vertices;

The testing step in algorithm 1 takes $d(a) + d(b)$ time (order of $n$). The inside for loop checks for every edge. Therefore, it is executed at most $|E|$ times. The outside loop also is executed at most $|E|$ times (total number of edges that can be removed). Therefore, the time complexity of the algorithm is at most $n|E|^2 + n$. 


Observation 2.3.3. Every clique having \( k \) vertices in the graph \( G \), is contained in a \( k \)-truss of \( G \).

Remember that trusses are relaxed versions of cliques. This can be very useful to enumerate the cliques. One can first discover trusses, and then look for the cliques in trusses.

Observation 2.3.4. A \( k \)-truss does not necessarily contain a clique of order \( k \).

For example figure 2.4 shows a 4-truss in which there is no 4-clique.

![Figure 2.4: A 4-truss which does not contain a 4-clique](image)

Proposition 2.3.1. In the \( k \)-truss \( M \), every vertex \( v \) is of degree at least \( k - 1 \);

\[
\forall v \in V(M) : d(v) \geq k - 1
\]

Proof. Consider an arbitrary vertex \( v \) in a \( k \)-truss \( M \). \( v \) has an edge \((v,u)\) (because there is no isolated vertex in a \( k \)-truss, then \( v \) has at least 1 edge). Based
on the definition of a $k$-truss, $u$ and $v$ have at least $k - 2$ common neighbors which are distinct from $u$ and $v$. That is, vertex $v$ has at least $1 + (k - 2) = k - 1$ neighbors.

**Observation 2.3.5.** Each $k$-truss of $G$ is a subgraph of a $(k - 1)$-truss of $G$, for $K > 2$.

Observation 2.3.5 is useful for truss computation process. To find all $k$-trusses for different $k$ values ($k = 3, 4, \ldots, t$), first we can detect the 3-trusses, and then 4-trusses from 3-trusses, and so on.

**Proposition 2.3.2.** Every $k$-truss of $G$ is a subgraph of a $(k - 1)$-core of $G$.

**Proof.** Consider a $k$-truss $M$ of graph $G$. Based on proposition 2.3.1, every vertex in $M$ has at least $k - 1$ neighbors. Hence, $M$ is qualified to be a $(k - 1)$-core of $G$. The maximality of cores needs that all of $M$ be in the same core.

Based on proposition 2.3.2 and observation 2.3.1, each truss is somewhere between the clique which is too strict, and the core, which is too loose.

**Proposition 2.3.3.** Every $k$-truss of $G$ has an edge connectivity of at least $k - 1$.

**Proof.** Suppose $C$ is minimum cut set in $k$-truss $M$. Consider an arbitrary edge $e = (a, b) \in C$. We consider the cases which make vertices $a$ and $b$ be in separate components. By cut set definition, if all edges in $C$ are deleted, then vertices $a$ and $b$ will be placed in separate components. Note that if removal of $e$ was not necessary to make this happen, then the minimality of set $C$ is violated. On the other hand, $e$ is supported by other $k - 2$ pairs of edges which are $k - 2$ independent paths of length
2 in $M$. Therefore, a minimum of $k - 2$ edges are required to be removed to make $a$ and $b$ separated. Hence, $|C| \geq 1 + (k - 2) = k - 1$.

**Observation 2.3.6.** Note that a graph with edge connectivity $k - 1$ is not necessarily a $k$-truss. For example, a cycle of order 6 has edge connectivity 2 but it’s not a 3-truss.

**Observation 2.3.7.** A $k$-truss can have vertex connectivity of 1 no matter how large $k$ is. See figure 2.5.

**Proposition 2.3.4.** A cutpoint of a $k$-truss connects $k$-trusses. (A cutpoint is a vertex that its removal will make the graph disconnected).

**Proof.** Suppose $v$ is the cutpoint of $k$-truss $M$. Let $S_1, \ldots, S_l$ are subgraphs which are connected by $v$. Since $M$ is a $k$-truss, each edge in subgraph $S_1$ is supported by other $k - 2$ edges. In other words, endpoints of each edge in $S_1$ have $k - 2$ common
neighbors such that they are also in $S_1$, otherwise, the selection of $v$ as a cutpoint is violated. Therefore, $S_1$ is also a $k$-truss. Similarly each $S_i$ for $2 \leq i \leq l$ are $k$-trusses.

**Corollary 2.3.1.** If a $k$-truss has a cutpoint $v$, then $d(v) \geq 2(k-1)$.

**Proof.** According proposition 2.3.4, $v$ connects at least 2 $k$-trusses. Each of these $k$-trusses, contribute $k-1$ to the degree of vertex $v$.

### 2.3.3 Discovering Maximal Trusses

Finding the maximal $k$-truss of the given graph $G$ is an interesting problem. Cohen [9] believes that the maximal $k$-trusses can be detected by deleting the components which are not part of the maximal $k$-truss. He presents an approach to discover maximal trusses of a given graph. We discuss his approach in this section in details.

**Algorithm 2** Discovering Maximal $k$-trusses of $G$

```plaintext
1: function REDUCE_GRAPH_TO_K_TRUSS(Graph G, k)
2:     function REDUCE_TO_K_CORE(G, k - 1);
3:     function DELETE_UNSUPPORTED_EDGES(G, k - 2);
4:     function DELETE_ISOLATED_VERTICES(G);
```

Algorithm 2 is an outline of the approach to detect maximal $k$-trusses introduced by Cohen [9]. First, algorithm takes the graph and cuts it down to its maximal $(k - 1)$-core. It helps to ignore many vertices and their adjacent edges which are not sufficiently supported. Then the edges that are not appropriate and isolated vertices would be removed. Functions REDUCE_TO_K_CORE and DELETE_UNSUPPORTED_EDGES are described in algorithms 3 and 4 respectively.
In Algorithm 3, the function takes the input graph $G$ and value $j$ and outputs the maximal $j$-core of $G$. Then the degree $d(v)$ of each vertex $v \in G$ is computed. A queue $q$ is defined to store the vertices having degree less than $j$. Then, the algorithm iteratively removes each vertex with vertex degree of less than $j$ and decrements the degree of its neighbors by 1. Then the updated neighbors need to be tested as well. The initialization steps are executed in order of $(n + |E|)$. The removal process in the inner loop can be done in no more than $\sum_{v \in V(G)} d(v) = 2|E|$ time. Therefore, the whole function’s time complexity is of order $(n + |E|)$. 
Algorithm 3 Reducing the graph $G$ to $j$-core

1: function REDUCE_GRAPH_TO_K_CORE(Graph $G$, $k$)
2: 
3: function REDUCE_TO_K_CORE($G$, $j$);
4: 
5: for $\forall v \in G$ do
6: $d(v) = 0$;
7: 
8: for $\forall e = (a, b) \in E(G)$ do
9: $d(a) \leftarrow d(a) + 1$ and $d(b) \leftarrow d(b) + 1$;
10: 
11: $q \leftarrow \emptyset$;
12: 
13: for $\forall v \in G$ do
14: if $d(v) < j$ then
15: $q \leftarrow q \cup \{v\}$;
16: 
17: while $q \neq \emptyset$ do
18: Pull $v$ from $q$;
19: 
20: for $\forall u \in N(v)$ do
21: $d(u) \leftarrow d(u) - 1$;
22: 
23: if $d(u) < j$ then
24: $q \leftarrow q \cup \{u\}$;
25: 
26: Delete $v$ from $G$;
Algorithm 4 Removing edges with insufficient support
1: function delete Unsupported Edges(Graph $G$, $j$)
2: \hspace{1em} $q \leftarrow \emptyset$; \hfill $\triangleright$ queue that keeps the edges for removal
3: \hspace{1em} for \hspace{1em} $\forall e = (a, b) \in E(G)$ \hspace{1em} do
4: \hspace{2em} place each member of $N(a)$ in a hash table $T$
5: \hspace{1em} $c \leftarrow 0$;
6: \hspace{1em} for \hspace{1em} $\forall v \in N(b)$ \hspace{1em} do
7: \hspace{2em} if \hspace{1em} $v \in T$ \hspace{1em} then
8: \hspace{3em} $c \leftarrow c + 1$;
9: \hspace{2em} if \hspace{1em} $c < j$ \hspace{1em} then
10: \hspace{3em} $q \leftarrow q \cup \{e\}$; remove $e'$ from $G$
11: \hspace{2em} else
12: \hspace{3em} $C(e) \leftarrow c$; \hfill $\triangleright$ stores the number of triangles supporting $e$
13: \hspace{2em} while \hspace{1em} $q \neq \emptyset$ \hspace{1em} do
14: \hspace{3em} Pull $e = (a, b)$ from $q$;
15: \hspace{3em} place members of $N(b)$ in a hash table $T$;
16: \hspace{3em} $I \leftarrow \emptyset$;
17: \hspace{3em} for \hspace{1em} $\forall v \in N(b)$ \hspace{1em} do
18: \hspace{4em} if \hspace{1em} $v \in T$ \hspace{1em} then
19: \hspace{5em} $I \leftarrow I \cup \{v\}$;
20: \hspace{3em} for \hspace{1em} $\forall e'$ connecting $a$ or $b$ to a member of $I$ \hspace{1em} do
21: \hspace{4em} $C(e') \leftarrow C(e') - 1$;
22: \hspace{4em} if \hspace{1em} $C(e') < j$ \hspace{1em} then
23: \hspace{5em} $q \leftarrow q \cup \{e'\}$; remove $e'$ from $G$
The main work is done in algorithm 4. It eliminates edges that are not supported by at least $j$ triangles (edge pairs). The first part of the algorithm calculates the number of triangles supporting each edge. The second part, deletes the edges which are not sufficiently supported. Whenever each edge is placed in the queue of removal, its neighbors need to be tested as well since their number of supporting triangles will change too. An arbitrary edge $e = (a, b)$ will not be disregarded if $a$ and $b$ have $j$ common neighbors. This test takes $d(a) + d(b)$ times. In the case that test fails, removing the edge and placing the neighbors in the queue of removal also takes $d(a) + d(b)$ times. Each edge can be pushed into the queue at most one time. Therefore, the total work takes:

$$\sum_{e=(a,b)} (d(a) + d(b)) = \sum_{v \in G} d^2(v)$$

Hence, the algorithm 4 is of order $O(\sum_{v \in G} d^2(v))$.

Note that in algorithm 2, function DELETE_UNSUPORTED_EDGES takes the $(k - 1)$-core of the original graph and works on it rather than working on the original graph \cite{9}.

**Truss Decomposition Problem.** The truss decomposition problem for a given graph $G$ is about discovering the $k$-trusses of $G$ for any $2 \leq k \leq k_{max}$. 
2.3.4 Related Works on $K$-truss Decomposition in Deterministic Graphs

One of the earliest algorithms that has been introduced for $k$-truss decomposition problem was Cohen’s in-memory algorithm [9]. His algorithm starts with computing the initial support of all edges in $G$. This initial support for edge $e = (u, v)$ is the number of common neighbors of $u$ and $v$ that is equal to the number of vertices that form triangles with vertices $u$ and $v$. In other words, initial support: $\text{sup}(e) = |N(u) \cap N(V)|$. Then, for every value of $3 \leq k \leq k_{\text{max}}$, the algorithm eliminates edges having support less than $k - 2$. By removing an edge $e = (u, v)$ whose support is less than $k - 2$, all the triangles containing $e$ are not valid anymore (they are not triangles). Therefore, the support of other two edges of those triangles also should be decreased by 1. This procedure is done iteratively until no edge with support less than $k - 2$ remains. Then the algorithm outputs $G$ as the $k$-truss for each $k$. In this algorithm, the input graph need to be located in main memory. On the other hand, it requires $O(m + n)$ space to store the entire graph and also edges’ supports. Whereas, real world graphs are usually huge and cannot fit in memory. Hence, the memory usage and as well as high time complexity of this algorithm make it inefficient for large scale graphs with vertices of high degrees.

In other work, Wang and Cheng [44] improved Cohen’s algorithm. Their improved truss decomposition algorithm accesses edges which are sorted in an ascending order of their supports while Cohen’s algorithm accesses the edges randomly. The algorithm starts with computing the initial support of each edge in $G$ by using a triangle enumeration algorithm [26 35]. Then, for each $k$ value, it sorts all edges in ascending order based on their support values. As the next step, for all edges having support
less that \( k - 2 \), the algorithm removes such an edge \( e \) and increments the support values of all other edges which constitute a triangle with edge \( e \). Then the edges need to be reordered based on their new support values. This process terminates whenever no edge with support of \( k - 2 \) is left. The main difference between Wang and Cheng [44] improved algorithm and Cohan’s approach [9] is regarding the search for triangles. Therefore, Wang and Cheng [44] algorithm enhanced the time complexity of Cohan’s algorithm to \( O(m^{1.5}) \). However, this algorithm is still inefficient in terms of memory usage as it has \( O(m + n) \) memory complexity which is a very big number for large graphs. Wang and Cheng [44] also suggested two I/O efficient algorithms for the case that the input graph does not fit in main memory. Their bottom-up algorithm starts from lowest \( k \) value, i.e., \( k = 2 \). Then, a lower bound on edges’ truss numbers are obtained. Afterwards, it searches for a candidate subgraph containing edges participating in \( k \)-class, \( \Phi_k \) and then \( \Phi_k \) is obtained in main memory. Then, algorithm moves on to calculating the \( \Phi_{k+1} \). These steps are repeated until all edges are eliminated from \( G \). The top-down algorithm is similar to bottom-up approach and the difference is that the algorithm starts from largest value for \( k \) and instead of calculating a lower bound, an upper bound on edges’ truss numbers are computed. After calculating \( \Phi_k \) in main memory, the algorithm calculates \( \Phi_{k-1} \). Their experiments on both small and large datasets showed a better performance of their I/O efficient algorithms compared to Cohen’s approach.
2.4 *k*-Truss in Probabilistic graphs

Huang et al. [18] extends the *k*-truss notation to probabilistic graphs. Their definition for truss concept in probabilistic graphs is similar to what Bonchi et al. [5] accomplished. Bonchi et al. [5] extended the core decomposition problem from deterministic to probabilistic graphs and defined \((k, \eta)-core\) as a subgraph in which the probability of any edge have a degree of at least \(k\) is not less than \(\eta\). In our work, we follow Huang et al. [18] definitions to study the truss decomposition problem in probabilistic graphs.

2.4.1 Local and Global \((k, \gamma)-truss\)

As mentioned above, Huang et al. [18] expanded the truss concept into probabilistic framework. They defined two different notions for truss in probabilistic graphs: *local* and *global* \((k, \gamma)-truss\). In the following we discuss each of these definitions in details.

Suppose \(G\) be a probabilistic graph. Consider an arbitrary vertex \(v \in G\). The *structural neighbors* of \(v\) in \(G\) are defined as the neighbors of \(v\) ignoring the edge probabilities. Let \(k_e = |N(u) \cap N(v)|\) for any edge \(e = (u, v) \in E_G\). Let \(sup_G(e)\) be the support of edge \(e = (u, v)\) in graph \(G\). Note that \(sup_G(e)\) is a random variable that can have any value from 0 to \(k_e\). We define the \(\gamma\)-support denoted by \(\gamma - sup_G(e)\), as the maximum \(k\) such that \(Pr_{G \subseteq G}[sup_G(e) \geq k] \geq \gamma\). Based on the possible worlds concept, this probability is taken over all the possible worlds \(G \subseteq G\).

For instance, consider Figure 2.6, edge \(e = (3, 5)\), and \(\gamma = 0.2\). With the assumption that \(e\) exists (with probability \(p(e) = 0.5\)), edge \(e\) has support at least 2 with probability \(0.5 \cdot (0.53 \cdot 0.67) \cdot (0.3 \cdot 0.8) = 0.042612\) (product of the probabilities
that triangles \( \Delta_{035} \) and \( \Delta_{135} \) exist in a possible world), and it has support at least 1 with probability \( 0.5 \cdot (1 - (1 - 0.53 \cdot 0.67) \cdot (1 - 0.3 \cdot 0.8)) = 0.2549 \) (complementary probability that none of the two triangles are in a possible world). Since 0.2549 is greater than the threshold \( \gamma \), the \( \gamma \)-support of edge \( e \) is 1.

According to possible worlds definitions, the probability that \( \text{sup}_G(e) \geq s \) for any \( s \in [0, k_e] \) is the sum of the probability of all possible worlds \( G \sqsubseteq \mathcal{G} \) such that the support of \( e \) in \( G \) is at least \( s \). That is,

\[
Pr(\text{sup}_G(e) \geq s) = \sum_{G \sqsubseteq \mathcal{G}} Pr[G|\mathcal{G}] \cdot I(\text{sup}_G(e) \geq s),
\]

where, \( I(\text{sup}_G(e) \geq s) \) is an indicator and is equal to 1 if \( \text{sup}_G(e) \geq s \), and 0 otherwise. In case \( e \notin E_G \), \( I(\text{sup}_G(e) \geq s) = 0 \).

\[\text{Figure 2.6: A probabilistic graph in which the } \gamma - \text{sup}(e = (3, 5)) = 1 \text{ for } \gamma = 0.2.\]

A subgraph \( \mathcal{H} \) of probabilistic graph \( \mathcal{G} \) is considered as a cohesive subgraph if any edge in \( \mathcal{H} \) has support of at least a threshold with high probability. This intuition can lead to the following definition of \( \text{Local} (k, \gamma) - \text{truss} \):
Definition 2.4.1. (Local \((k, \gamma)\)-truss). Let \(\mathcal{G} = (V, E, p)\) be a probabilistic graph and \(\gamma \in [0, 1]\) be a given threshold. Suppose \(\mathcal{H} \subseteq \mathcal{G}\) be any connected subgraph of \(\mathcal{G}\). For any integer \(k \geq 2\), \(\mathcal{H}\) is called to be a local \((k, \gamma)\)-truss if and only if for any edge \(e \in E_{\mathcal{H}}\),

\[
\Pr[\sup_{\mathcal{H}}(e) \geq k - 2] \geq \gamma
\]  

(2.4)

For example, figure 2.8 shows a local \((4, 0.216)\)-truss \(\mathcal{H}_1\) of probabilistic graph depicted in figure 2.7. Each edge in \(\mathcal{H}_1\) has support at least 2 with probability no less than 0.216.

The “local” term in definition indicates that edges’ supports are computed individually and no global constraint need to be satisfied. Now, the Local Probabilistic Truss Decomposition problem can be defined as follows:

**Local Probabilistic Truss Decomposition Problem.** Let \(\mathcal{G} = (V, E, p)\) be a probabilistic graph and \(\gamma \in [0, 1]\) be a given threshold. The problem of finding all maximal local \((k, \gamma)\)-trusses of \(\mathcal{G}\), for any \(2 \leq k \leq k_{\text{max}}\), is called local probabilistic...
truss decomposition problem.

Similar to the deterministic scenario, local \((k, \gamma)\)-trusses are unique and nested into each other [11].

Huang et al. [18] also presented a stronger definition of a \((k, \gamma)\)-truss. Their second definition satisfies a graph-wise global constraint. Their global \((k, \gamma)\)-truss definition makes sure that the \((k - 2)\) triangles supporting edges in subgraph \(\mathcal{H}\) exist at the same time in at least \(\gamma\) percent of the possible worlds. Therefore, it reinforces the cohesiveness of the truss.

Definition 2.4.2. (Global \((k, \gamma)\)-truss). Suppose \(G = (V, E, p)\) is a probabilistic graph. For integer \(k \geq 2\) and threshold \(\gamma \in [0, 1]\), connected subgraph \(\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H})\) is a global \((k, \gamma)\)-truss if and only if for any edge \(e \in E_\mathcal{H},\)

\[
\alpha_k(\mathcal{H}, e) = \sum_{H \subseteq \mathcal{H}} Pr[H|\mathcal{H}] \cdot I(H, k, e) \geq \gamma, \quad (2.5)
\]

Where \(I\) is an indicator and is equal to 1 if the possible world \(H\) is connected, deterministic \(k\)-truss containing \(e\), and 0 otherwise.

Huang et al. [18] proved that every global \((k, \gamma)\)-truss is also a local \((k, \gamma)\)-truss. This means that global \((k, \gamma)\)-truss is a more powerful and stricter definition. Similarly, truss decomposition problem can be defined based on global \((k, \gamma)\)-truss notation as follows:

Global Probabilistic Truss Decomposition Problem. Let \(\mathcal{G} = (V, E, p)\) be a probabilistic graph and \(\gamma \in [0, 1]\) be a given threshold. The problem of finding all
maximal global \((k, \gamma)\)-trusses of \(G\), for any \(2 \leq k \leq k_{\text{max}}\), is called global probabilistic truss decomposition problem.

In this thesis, we focus on local \((k, \gamma)\)–truss notation as \([18]\) shows that computing the global \((k, \gamma)\)–trusses is intractable and is a \#P-hard problem.

### 2.4.2 Probabilistic Truss Decomposition’s Challenges

Truss decomposition is a challenging task in probabilistic framework because of uncertainty nature of probabilistic graphs. Hence, the basic idea of edge removal to obtain trusses in deterministic graphs does not work in the probabilistic case. For example, we need to struggle with triangles that contain an edge \(e\) while they also have combinatorial nature. Therefore, triangle counting is not as straightforward as in deterministic graphs.

### 2.4.3 Related Works on \(k\)-truss Decomposition in Probabilistic Graphs

Truss decomposition in probabilistic graphs has not been studied much. Huang et al. \([18]\) were the first researchers that investigated this problem on probabilistic graphs. They proposed an algorithm based on dynamic programming to overcome the challenges that uncertainty cause. The basic idea of their algorithm is based on edge removal approach in deterministic case. However, they suggested a dynamic programming algorithm to compute and update edge supports. We discuss their local probabilistic truss decomposition algorithm in detail as we used an optimized version of dynamic programming on our proposed approach.
Similar to deterministic framework, first the support scores of edges are computed. Then, their algorithm iteratively detects all local \((k, \gamma)\)-trusses for \(k\) starting from 2. For each \(k\), if there is an edge \(e\) that belongs to a local \((k, \gamma)\)-truss while does not belong to any \((k + 1, \gamma)\)-truss, then \(e\)'s truss score is set to \(k\) and will be eliminated from \(G\). Whenever an edge is removed, the truss score of all other edges that construct a triangle with removed edge should be updated. This process stops when all edges are removed. The difference between this algorithm and the deterministic case is the way that truss scores are computed and updated at each iteration which we discuss below.

For an arbitrary edge \(e \in E(G)\), \(\sigma(e, t)\) is defined as the probability that the support of edge \(e\) is at least \(t\) in probabilistic graph \(G\). That is,

\[
\sigma_G(e, t) = Pr[\text{sup}_G(e) \geq t]
\]  

Then, the edge support probability vector is denoted by \(\sigma_G(e) = [\sigma_G(e, 1), \ldots, \sigma_G(e, k_e)]\) that represents the vector of support probabilities of \(e\). For purpose of computing \(\sigma_G(e)\), it is supposed that edge \(e\) exists and then the true edge support probability can be obtained by multiplying edge existence probability \(p(e)\) by \(\sigma_G(e)\).
Algorithm 5 Local \((k, \gamma)\)-truss decomposition

1: function LOCAL\(_{(k, \gamma)}\)-TRUSS\_DECOMPOSITION\((\mathcal{G} = (V, E, p), \gamma \in [0, 1])\)
2:     for \(\forall e \in E(\mathcal{G})\) do
3:         function COMPUTE\(_{\sigma}(e)\);
4:     for \(k \leftarrow 2\) to \(n\) do
5:         while \(\exists e = (u, v)\) such that \(\sigma(e, k-1)p(e) < \gamma\) and \(\sigma(e, k-2)p(e) \geq \gamma\) do
6:             \(\tau(e) \leftarrow k;\)
7:             delete \(e\) from graph \(\mathcal{G}\);
8:         for \(w \in N(u) \cap N(v)\) do
9:             function UPDATE\(_{\sigma}(e' = (w, u))\) and
10:             function UPDATE\(_{\sigma}(e'' = (w, v))\)
11: if \(\mathcal{G}\) is empty then
12:     break;

Algorithm 5 describes the approach used in [18] to detect local trusses in probabilistic graphs. It starts with computing edge support probability vector \(\sigma(e)\) for each edge (calls function COMPUTE\(_{\sigma}(e)\)). Then, iteratively local \((k, \gamma)\)-trusses are detected for \(k\) value from 2 to \(n\). For each \(k\), trussness score of \(k\) is assigned to an edge if that edge belongs to a local \((k, \gamma)\) truss, but belongs to no local \((k+1, \gamma)\) truss. Whenever, an edge \(e\) is removed from the graph, trussness scores of all other edges that construct triangles with \(e\) need to be updated (calls function UPDATE\(_{\sigma}(e)\)). The iterations are stopped when all edges are removed from the graph. Note that the algorithm computes the trussness score of each edge. Then, for any value of \(k\),
maximal local \((k, \gamma)\)-trusses can be obtained by piecing together edges with \(\tau(e) \geq k\).

In the following, we explain how edge support probability vectors are computed and updated in algorithm 5.

2.4.3.1 Edge Support Computation Using Dynamic Programming

Recall definition of \(\sigma(e, t) = Pr[\sup_G(e) \geq t]\). By complement rule we have the following equation:

\[
\sigma(e, t) = 1 - \sum_{i=0}^{t-1} Pr[\sup_G(e) = i]
\]  

(2.7)

To compute \(Pr[\sup_G(e) = i]\) in equation 2.7, Huang et al. [18] considered two cases for edge \(e = (u, v)\) as follows:

I Zero triangles. One possibility is that no triangle contain edge \(e\). We want to calculate \(Pr[\sup_G(e) = 0]\). This can happen because of two reasons:

(a) \(e\) does not exist which happens with probability of:

\[
P(\text{edge } e \text{ does not exist}) = 1 - P(e)
\]  

(2.8)

(b) \(e\) exists but no triangle contains \(e\) and this happens with probability of:

\[
P(\text{no triangle contain } e) = P(e) \cdot \prod_{w \in N(u) \cap N(v)} (1 - [p(w, u)p(w, v)])
\]  

(2.9)
Therefore,

$$\Pr[\sup_G(e) = 0] = \Pr(\text{edge } e \text{ does not exist}) + \Pr(\text{no triangle contain } e)$$

$$= (1 - \Pr(e)) + \Pr(e) \prod_{w \in N(u) \cap N(v)} (1 - [p(w, u)p(w, v)]) \quad (2.10)$$

II Multiple triangles. Second possibility is that \(e\) is contained in multiple triangles. To calculate the probability of the case that \(e\) participates in \(i\) triangles; \(\Pr[\sup(e) = i]\), Huang et al. [18] developed a dynamic programming algorithm.

Consider an arbitrary edge \(e = (u, v)\) and a fixed neighbor \(w \in N(u) \cap N(v)\). If \(e\) is a part of \(i\) triangles, there are two cases:

(a) Vertices \(u, v\) and \(w\) do not constitute a triangle (\(\Delta_{u,v,w}\) does not exist) and \(e\) is a part of \(i\) other triangles.

(b) \(\Delta_{u,v,w}\) exist and \(e\) participates in other \(i - 1\) other triangles

Therefore, a recursive formula can be defined to compute \(\Pr[\sup(e) = i]\). Let \(W\) be the set of all common neighbors of \(u\) and \(v\); \(W = N(u) \cap N(v) = \{w_1, \ldots, w_l\}\). Consider \(W_i\) which is a subset of common neighbors of \(u\) and \(v\); \(W_i \subseteq W\). The probability that \(e\) participates in \(i\) triangles with common neighbors from \(W_i\) is denoted by \(\Pr[\sup(e) = i|W_i]\). Then, the following recursive equation can be used to compute the desired probability.

$$\Pr[\sup(e) = i|W_i] = p(w_i, u)p(w_i, v)\Pr[\sup(e) = i - 1|W_{i-1}]$$

$$+ (1 - [p(w_i, u)p(w_i, v)])\Pr[\sup(e) = i|W_{i-1}] \quad (2.11)$$

Where \(-1 \leq i \leq |N(u) \cap N(v)|\) and \(0 \leq l \leq |N(u) \cap N(v)|\). The base cases of
this relation is defined as follows:

- \( \Pr[\sup(e) = 0|W_0] = 1 \)
- \( \Pr[\sup(e) = -1|W_l] = 0; \) for any \( 0 \leq l \leq |N(u) \cap N(v)| \)
- \( \Pr[\sup(e) = i|W_l] = 0; \) for \( i > l \)

For every \( i, \) \( \Pr[\sup(e) = i|W_{N(u)-N(v)}] \) is the edge support probability \( \Pr[\sup(e) = i] \).

Algorithm 6 represents the dynamic programming used to compute the edge support probability vectors (function \text{COMPUTE}_\sigma(e) \) in algorithm 5). The algorithm first computes the initializations (lines 1-6). Then, it calculates \( \Pr[\sup(e) = i|W_l] \) probabilities for different \( i \) and \( l \) values (lines 7-9). Note that in the algorithm, \( P(i,j) \) is used as a shorthand for \( \Pr[\sup(e) = i|W_j] \). Finally, edge support probabilities are computed (lines 10-12) by using following equations:

\[
\sigma(e, t - 1) = \Pr[\sup(e) \geq t - 1] \\
= \Pr[\sup(e) = t - 1] + \Pr[\sup(e) \geq t] \\
= \Pr[\sup(e) = t - 1] + \sigma(e, t) \\
= P(t - 1, |N(u) \cap N(v)|) + \sigma(e, t) \\
\Rightarrow \sigma(e, t) = \sigma(e, t - 1) - P(t - 1, |N(u) \cap N(v)|) 
\]
Algorithm 6 Edge support probability computing using dynamic programming

1. function COMPUTE_\(\sigma(e)\)(\(G = (V, E, p)\), \(e = (u, v) \in E_G\))

2. \(P(0, 0) \leftarrow 1;\)

3. \(P(-1, l) \leftarrow 0, \text{ for } 0 \leq l \leq |N(u) \cap N(v)|;\)

4. for \(l = 0\) to \(|N(u) \cap N(v)|\) do

5. for \(i = l + 1\) to \(|N(u) \cap N(v)|\) do

6. \(P(i, l) \leftarrow 0;\)

7. for \(i = 0\) to \(|N(u) \cap N(v)|\) do

8. for \(l = 1\) to \(|N(u) \cap N(v)|\) do

9. \(P(i, l) = p(w_l, u)p(w_l, v)P(i - 1, l) + (1 - [p(w_l, u)p(w_l, v)])P(i, l - 1)\)

10. \(\sigma(e, 0) \leftarrow 1;\)

11. for \(j = 1\) to \(|N(u) \cap N(v)|\) do

12. \(\sigma(e, j) \leftarrow \sigma(e, j - 1) - P(j, |N(u) \cap N(v)|);\)

In algorithm 6, the dynamic programming calculations takes \(O(|N(u) \cap N(v)|^2)\) time. Computation of support edge probabilities (lines 10-12) also needs \(O(|N(u) \cap N(v)|)\) time. Hence, the time complexity of the dynamic programming algorithm is \(O((\min\{d(u), d(v)\})^2)\). Note that \(|N(u) \cap N(v)| \leq (\min\{d(u), d(v)\})\).

2.4.3.2 Updating Edge Supports

After edge elimination step (line 7 in algorithm 5), the support score of some of the edges needs to be updated. Huang et al. [18] update edge supports using a recursive formula instead of calculating the new supports from scratch. Suppose we want to update the support score of edge \(e = (u, v)\) in the case that \(\Delta_{u,v,w}\) does not exist
anymore due to deletion of either $e' = (u, w)$ or $e'' = (v, w)$. Assume that $e' = (u, w)$ is eliminated from the graph. Then,

$$
Pr[	ext{sup}(e) = i|W_i] = (1 - p(e')p(e''))Pr[	ext{sup}(e) = i|W_i \setminus \{w\}] + p(e')p(e'')Pr[	ext{sup}(e) = i - 1|W_i \setminus \{w\}]
$$

From above equation we can see that,

$$
Pr[	ext{sup}(e) = i|W_i \setminus \{w\}] = \frac{Pr[	ext{sup}(e) = i|W_i] - p(e')p(e'')Pr[	ext{sup}(e) = i - 1|W_i \setminus \{w\}]}{(1 - p(e')p(e''))}
$$

Then, in order to update the support score $\sigma(e)$ of edge $e$, we can use the following recursion formula:

$$
P_{\text{new}}(i, |N(u) \cap N(v)|) = \frac{P_{\text{old}}(i, |N(u) \cap N(v)|) - p(e')p(e'')P_{\text{new}}(i - 1, |N(u) \cap N(v)|)}{1 - p(e')p(e'')}\quad (2.14)
$$

Note that old and new superscripts mean before and after deletion of edge $e'$. $\sigma(e)$ can be updated in $O(|N(u) \cap N(v)|)$ time using $P_{\text{new}}$.

**Time complexity of Local truss decomposition algorithm.** Computing $\sigma(e)$ in algorithm 3 (lines 2-3) takes

$$
O(\sum_{(u,v) \in E} (\min\{d(u), d(v)\})^2) \subseteq O(d_{\max} \sum_{(u,v) \in E} \min\{d(u), d(v)\}) \subseteq O(d_{\max} \rho |E|)\quad (2.15)
$$
time where $d_{\text{max}}$ is the maximum degree and $\rho$ is arboricity of $G$. Then, in lines 4-12 the algorithm eventually delete all edges from the graph, and after each deletion, algorithm 5 updates support probabilities of neighboring edges. If edge $e = (u, v)$ is removed, at most $2|N(u) \cap N(v)| \in O(\min\{d(u), d(v)\})$ need to be updated (since each $w \in N(u) \cap N(v)$ is connected to both $u$ and $v$ and therefore for every common neighbor, 2 edges $uw$ and $vw$ should be updated). As mentioned above, the updating step can be completed in $O(|N(u) \cap N(v)|) \subseteq O(\min\{d(u), d(v)\})$. Therefore, the total time needed to do updates corresponding each edge removal is $O(\sum_{(u,v) \in E} (\min\{d(u), d(v)\}))^2 \subseteq O(d_{\text{max}}\rho|E|)$. Therefore, the time complexity of the algorithm is $O(d_{\text{max}}\rho|E|)$.

**Space complexity of Local truss decomposition algorithm.**

For every edge $e = (u, v)$, two arrays should be stored: $P(i-1, *)$ and $P(i, *)$ which needs $O(\min\{d(u), d(v)\})$. Edge support vectors $\sigma(e)$ also consumes $O(\min\{d(u), d(v)\})$ space. To store the whole graph, $O(|E|)$ space is required. Therefore, totally, the space complexity of algorithm 5 is $O(\min\{d(u), d(v)\}) + O(|E|) \subseteq O(\rho|E|)$.

### 2.4.3.3 Probabilistic Truss Decomposition Using Central Limit Theorem

Esfahani et al. \[1\] suggested a different approach to calculate the $\gamma$-supports rather than dynamic programming. They used a specific version of Central Limit Theorem (CLT) to estimate support probabilities. They also performed a theoretical analysis on their approach and obtained error bound of their approximations.

---

1 Minimum number of spanning forests needed to cover all edges of a graph is called the arboricity of the graph. A tree is an undirected, connected and acyclic graph. A forest is an undirected graph in which any two vertices are connected by at most one path, or equivalently a disjoint union of trees.
Esfahani et al. [11] proposed \emph{CLT\_based\_PA} algorithm which decomposes a probabilistic graph into trusses and is summarized below:

1. Initial $\gamma$-supports are computed using Lyapunov’s central limit theorem
2. Edges are sorted in an ascending order based on their $\gamma$-supports
3. For each $k$ value, the edges having $\gamma$-supports less than $k$ are removed.
4. The $\gamma$-supports of other edges that form a triangle with the removed edge and have supports greater than the its support, are updated.
5. Edges are re-sorted since some of the edge’s supports are updated.
6. The algorithm is stopped when all edges are removed and therefore all trusses are detected.

The results of their algorithm show the efficiency of their proposed \emph{CLT\_based\_PA} algorithm and also represent it performs much faster than the peeling algorithm based on DP approach.
Chapter 3

The New Probabilistic Truss Decomposition Algorithm

In this section we propose our algorithm based on $h$-index updating approach [45, 54].

As explained before in the thesis, the “edge-peeling” process is a broadly used approach for computing truss decomposition. To reiterate the main salient points described so far, edge-peeling is as follows: (1) recursively delete the edge $e$ of the smallest support (2) set $e$’s trussness to be its support at the time of removal, and (3) update the support of all the edges which form a triangle with $e$ while keeping the edges sorted. This approach is associated with two major drawbacks: each step depends on the result from the previous step. Moreover, since edges should be kept sorted at all times during the algorithm, additional data structures are required to store edges and their position in the data structure as well as keeping track of removed edges at each step of the algorithm.

An alternative way to compute $k$-trusses is the $h$-index updating algorithm. It
processes edges in a random order with less memory usage, since it does not require extra data structures as the edge-peeling does.

## 3.1 New Approach

The $h$-index updating algorithm for truss decomposition has been introduced in the context of deterministic graphs [34]. Given a set of real numbers, the $h$-index of the set is defined as the largest number $h$ such that there are at least $h$ elements in the set that are equal to $h$ or higher. For instance, the $h$-index of the set $\{1, 2, 2, 3\}$ is 2. In the algorithm, the $h$-index of each edge is initialized to the edge’s initial support.

At each step of the algorithm, the $h$-index of each edge is updated by computing the $h$-index of its neighbor set, until no updates would happen (cf. [45] for more details). This process was significantly extended for computing the probabilistic $k$-core in [10]. Here we extend the $h$-index based process to computing the probabilistic $k$-truss.

To start off, the extension of neighbor set for an edge is defined as a set of all edges which form a triangle with it. In our algorithm, we refer to this process as **phase I**.

Since the **phase I** does not take into account an edge having enough support probability to be part of a $k$-truss, it may not converge to true probabilistic trussness of the edges by itself, even if the $h$-index of each edge is initialized to the edge’s probabilistic support. We tackle the problem by **phase II** which considers uncertainty in probabilistic truss decomposition. Although the **phase I** does not work for computing exact truss values, it can provide a tight upper-bound on them which can be used in **phase II**. Therefore, we combine **phase II** with **phase I** to speed up convergence as **phase I** is faster than **phase II**.
Theorem 3.1.1. The phase I provides an upper-bound on true truss values in the input probabilistic graph.

Proof. If the $h$-index of an edge $e$ is fixed at $\bar{k}_e$, it means that there are at least $\bar{k}_e$ triangles (regardless of their existence probability) which contains $e$, and for each edge $e'$ in these triangles we have: $\bar{k}_{e'} \geq \bar{k}_e$. However, given the threshold $\gamma$, and considering subgraph $\mathcal{H}$ whose edges has support at least $\bar{k}_e$, $\Pr[\text{sup}_{\mathcal{H}}(e) \geq \bar{k}_e]$ might be less than $\gamma$. Due to monotonous property of tail probability of edge support, true value for trussness of $e$ should be in the interval $[0, \bar{k}_e]$. On the other hand, based on definition of $h$-index, $e$ cannot be contained in $k > \bar{k}_e$ triangles whose edges have $h$-index at least $k$. Thus, $\Pr[\text{sup}_{\mathcal{H}'}(e) \geq k] < \gamma$, where $\mathcal{H}'$ is a subgraph which contains edges with support at least $\bar{k}_e$. As a result, the theorem follows.

The major steps of our proposed algorithm are summarized in Algorithm 7. In line 3 the $h$-index of each edge is initialized to its initial $\gamma$-support. Checking whether an edge requires processing or not is done by array scheduled, which is initialized to true for each edge. The variable updated records whether there is some edge with its $h$-index changed or not. Line 4 invokes deterministic $h$-index updating (Phase I) for each edge. Then, the algorithm starts Phase II which processes the $h$-index (upper-bound on true trussness) of all the edges for the possibility of gap between the current value and its true trussness (line 7). If after Phase II terminates, there is some edge with its $h$-index updated, Phase I (line 10) is started again, and this process continues until each edge’s $h$-index achieves convergence (lines 6-10). The final trussness or truss score of each edge is obtained by adding 2 to the final $h$-index (line 11).

Probabilistic $h$-index updating (Phase II) is given in Algorithm 8. This part
differentiates our approach from \( h \)-index based algorithms for deterministic graphs.

In probabilistic graphs, the \( h \)-index of an edge \( e \) is assigned \( h[e] \), if it satisfies the condition of having support at least \( h[e] \) with enough probability: \( \Pr[\sup(e) \geq h[e] | \Gamma, e \text{ exists}] \geq \gamma \). The computation is done with respect to the set of triangles \( \Gamma \) which contains \( e \), and the \( h \)-index of each edge in these triangles is at least \( h[e] \). This is because only these triangles can contribute to trussness of edge \( e \) in final truss subgraph. This process is done using function \textit{Phase II} in Algorithm 8. For each scheduled edge \( e \), lines 4-10 find the set \( \Gamma \), and validity of the edge’s \( h \)-index is verified in line 11. If the \( h \)-index \( h[e] \) is not valid, all the values less than \( h[e] \) is checked until a valid \( h \)-index is detected (lines 12-14). Each time the set \( \Gamma \) is updated to correspond to the \( h \)-index being validated for edge \( e \). Once a new \( h \)-index for edge \( e \) is obtained, all the neighbour edge set of \( e \) whose \( h \)-index is between \( e \)'s new \( h \)-index and \( e \)'s current \( h \)-index are scheduled to be processed in the next iterations (lines 17-19). This is because the \( h \)-index of these neighbour edges may have a chance to change.

The time complexity of our algorithm for only one single iteration is \( O(d_{\text{max}} | E|) \). This is because, at each iteration the \( h \)-index of each edge and its neighbours are processed; \( O(d_{\text{max}}) \), and this is done for all edges of the graph. Therefore, the total time complexity is \( O(d_{\text{max}} | E|) \).

### 3.1.1 Optimized Dynamic Programming

In the while loop of \textit{Phase II} algorithm, the main challenge is efficient computation of \( \Pr[\sup(e) \geq H | \Gamma, e \text{ exists}] \) (line 11 Algorithm 8), for different values of \( H \), until a desired \( H \) is obtained. For this, we can use a modified dynamic programming (DP)
process to avoid computation of these probabilities from scratch. Once \( \Pr[\sup(e) \geq H | \Gamma, e \text{ exists}] \) is computed using DP, we cache the following probabilities:

\[
\Pr[\sup(e) = 0 | \Gamma, e \text{ exists}], \ldots, \Pr[\sup(e) = H | \Gamma, e \text{ exists}].
\]

These probability values can be used in computation of \( \Pr[\sup(e) \geq (H - 1) | \Gamma \cup \mathcal{T}^{(H-1)}, e \text{ exists}] \), where set \( \mathcal{T}^j \) is given in (line 13), \( j = 0, \ldots, H \). In fact, since we already computed support probability with respect to \( \Gamma \), we only need to look at set \( \mathcal{T}^{(H-1)} \) for computing new tail probability for value \( H - 1 \). This way computation can be very fast, because \( \mathcal{T}^{(H-1)} \) is usually small (e.g., about 50 in our evaluated graphs).

In the following, we describe our optimized DP process in more detail.

Given an edge \( e = (u, v) \), and a triangle \( \triangle = uvw \) which contains \( e \), set \( \rho_\triangle \) as the minimum \( h \)-index of edges \( (u, w) \) and \( (v, w) \). Let assume that we have computed \( \Pr[\sup(e) = k | \Gamma, e \text{ exists}] \), where \( k = 0, \ldots, H \), and \( \Gamma \) is a sequence of triangles \( \triangle \) with \( \rho_\triangle \geq H \). Also, let \( \mathcal{T}^{H-1} = \{\triangle_1, \ldots, \triangle_x\} \). For each triangle \( \triangle_i = (u, v, w_i) \), we have \( \rho_{\triangle_i} = H - 1, i = 1, \ldots, x \). Thus, we have:

\[
\Pr[\sup(e) = k | \Gamma \cup \mathcal{T}^{H-1}, e \text{ exists}] = \Pr[\sup(e) = k | \Gamma \cup \{\triangle_1, \ldots, \triangle_x\}, e \text{ exists}] = T(x, k),
\]

By \( T(x, k) \) we denote the probability that \( e \) participates in \( k \) triangles selected from \( \Gamma \cup \{\triangle_1, \ldots, \triangle_x\} \). For a fixed triangle \( \triangle_x = (u, v, w_x) \), the event that \( e \) participates in \( k \) triangles implies that either (1) \( \triangle_x \) exists and \( e \) participates in \( k - 1 \) triangles selected from \( \{\triangle_1, \ldots, \triangle_x\} \) excluding \( \triangle_x \), or (2) \( \triangle_x \) does not exist and \( e \) participates
in $k$ other triangles. Thus, it holds that

$$T(x, k) = p(u, w_x)p(v, w_x)T(x - 1, k - 1)$$

$$+ (1 - p(u, w_x)p(v, w_x))T(x - 1, k),$$

The base cases for the above formula is:

$$T(0, k) = \Pr[\sup(e) = k \mid \Gamma, e \text{ exists}], \quad 0 \leq k \leq H$$

$$T(x, -1) = 0,$$

As can be seen, in the base case of the recursive formula, we are using the previously computed support probabilities to compute new probability values. This results in saving time significantly.

### 3.2 Correctness of the Proposed Algorithm

We prove this by contradiction. Suppose that after the last iteration of Phase II, there is an edge $e_1$ such that $\bar{h}[e_1] = H' > H_1$, where $\bar{h}$ represents the $h$-index fixed by the algorithm, and $h[e_1] = H_1$ is the true $h$-index for $e_1$. The value $H_1$ is the maximum value such that $\Pr[\sup_F(e_1) \geq H_1] \geq \gamma$, and for all $i > H_1$, $\Pr[\sup_{F'}(e_1) \geq i] < \gamma$, where $F$ and $F'$ are maximal induced subgraphs in which each edge has $\gamma$-support at least $H_1$ and $i$, respectively. Based on the properties of truss in a graph, we know that $F' \subseteq F$. If all the neighbor edges of $e_1$ in $F$ have true $h$-index $H_1$, then $e_1$ will not have any neighbor with true $h$-index greater than $H_1$ in $F'$, so the algorithm eventually set the $h$-index of $e_1$ to $H_1$ which is a contradiction. Similar
reasoning holds if all the neighbour edge set of $e_1$ in $\mathcal{F}$ have true $h$-index greater than $H_1$. Since $h[e_1] = H_1$ and $\bar{h}[e_1] = H' > H_1$, there should exist two edge neighbours $e'_1$ and $e''_1$, with $\min\{h[e'_1], h[e''_1]\} = H_1$, and $\min\{\bar{h}[e'_1], \bar{h}[e''_1]\} > H_1$, such that they form a triangle $\triangle$ with $e$, and $\Pr[\sup_{\mathcal{F}_2 \cup \{e'_1 \cup e''_1\}}(e_1) \geq H'] \geq \gamma$. Otherwise, $\Pr[\sup_{\mathcal{F}_1}(e_1) \geq H'] < \gamma$. In other words, due to existence of these neighbor edges $\bar{h}[e] > H_1$. Let $E_i = \{e_1, e'_1, e''_1\}$. Applying the same reasoning for $e'_1$ and $e''_1$, we can build a sequence of such edges: $S = \{E_i, E_{i+1}, E_{i+2}, ..., E_j = E_i\}$. For each edge set $E_i$, let $N_{E_i}$ be the set of all edge neighbors of $E_i$’s edges in $\mathcal{F}'$. Defining $\mathcal{F}''$ to be a subgraph which contains $S \cup (\cup N_{E_i})$, it is proved that $\mathcal{F}''$ is a $H'$-truss, since for each edge $e \in \mathcal{F}''$, we have $\Pr[\sup_{\mathcal{F}''}(e) \geq H'] \geq \gamma$. This is a contradiction to the assumption of $h[e] = H$, and maximality of $\mathcal{F}'$.

\begin{algorithm}
\textbf{Algorithm 7} Probabilistic $h$-index truss (proHIT) decomposition
\begin{algorithmic}[1]
\Function{h-index Updating}{$\mathcal{G}, \text{support}$}
\ForAll{edge $e \in E$}
\State $h[e] \leftarrow \text{support}[e], \text{scheduled}[e] \leftarrow \text{true}$
\EndFor
\State \Comment Deterministic $h$-index updating
\State $\text{Phase I} (\mathcal{G}, h)$
\State $\text{updated} \leftarrow \text{false}$ \Comment True if any $h[e]$ is updated
\While{true}
\State \Comment $\text{Phase II} (\mathcal{G}, h, \text{scheduled})$
\If{$\text{updated}$ is false} break \EndIf
\Else
\State \Comment $\text{Phase I} (\mathcal{G}, h)$, $\text{updated} \leftarrow \text{false}$
\ForAll{edge $e \in E$} $\text{truss score}[e] \leftarrow h[e] + 2$
\EndFor
\State \Return $\text{truss score}$
\EndWhile
\EndFunction
\end{algorithmic}
\end{algorithm}
Algorithm 8 Probabilistic $h$-Index Updating

1: function Phase II($\mathcal{G}, h, scheduled$)

2: \hspace{1em} for all edge $e \in E$ do

3: \hspace{2em} if $scheduled[e]$ is false then continue

4: \hspace{2em} $N \leftarrow$ empty list, $\mathcal{T}^H \leftarrow$ empty list, $H \leftarrow h[e]

5: \hspace{2em} for all $\triangle$ contains $e$ do

6: \hspace{3em} $e_1, e_2 \leftarrow$ the two edges in $\triangle$ other than $e$

7: \hspace{3em} $N.add(e_1), N.add(e_2)$

8: \hspace{3em} $\rho \leftarrow \min \{h[e_1], h[e_2]\}$

9: \hspace{3em} if $\rho \geq h[e]$ then $\mathcal{T}^H.add(\triangle)$

10: \hspace{2em} $\Gamma \leftarrow \mathcal{T}^H$

11: \hspace{2em} while $\Pr[\sup(e) \geq H \mid \Gamma, e \text{ exists}] < \gamma$ or $H \geq 0$ do

12: \hspace{3em} $H \leftarrow H - 1$

13: \hspace{3em} $\mathcal{T}^H \leftarrow$ all $\triangle$ which contains $e$, with $\rho_\triangle = H$

14: \hspace{3em} $\Gamma \leftarrow \Gamma \cup \mathcal{T}^H$

15: \hspace{2em} if $H \neq h[e]$ then

16: \hspace{3em} $updated \leftarrow$ true

17: \hspace{3em} for all edge $e_N \in N$ do

18: \hspace{4em} if $H < h[e_N] \leq h[e]$ then

19: \hspace{5em} $scheduled[e_N] \leftarrow$ true

20: \hspace{4em} $h[e] \leftarrow H$, $scheduled[e] \leftarrow$ false

\textbf{Example:} Consider the graph in Figure 3.1 with $\gamma = 0.2$. Table 3.1 shows the
main steps of our algorithm. First column shows each edge $e = (i, j)$ connecting vertices $i$ and $j$ for different $i$ and $j$ values in the graph. Each entry in second column is written in form of a sequence separated by colons. Elements of the sequence correspond to $\gamma$-support values, $h$-index values obtained at the end of Phase I, Phase II and true truss scores (final scores), respectively. The $\gamma$-support of edges are obtained using Dynamic programming (DP).

![Figure 3.1: Example](image)

<table>
<thead>
<tr>
<th>edge</th>
<th>$\gamma$-support</th>
<th>phase I</th>
<th>phase II</th>
<th>true score</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(i, j), 1 \leq i &lt; j \leq 4$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$(0, j), j = 1, 2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$(i, 5), i = 1, 4$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$(i, 6), i = 2, 3$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$(i, 7), i = 3, 4$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*Table 3.1: Initial $\gamma$-support, values obtained by Phase I and Phase II, respectively. $\gamma = 0.2$ for the example.*

*Phase I* starts with initializing the $h$-index of each edge to its initial $\gamma$-support, and
updates the values based on the $h$-indices of each edge’s neighbors. For instance, consider edge $e = (1, 2)$. Let $T = \{ \triangle_{0,1,2}, \triangle_{1,2,3}, \triangle_{1,2,4} \}$ be the set of all triangles containing $e$. For each triangle in $T$, phase I finds the minimum $h$-index of two edges other than $e$, and adds it to a list $L$. Thus, $L = \{ \min(h[(0, 1)], h[(0, 2)]), \min(h[(1, 3)], h[(2, 3)]), \min(h[(1, 4)], h[(2, 4)]) \} = \{1, 2, 2\}$. Therefore, Phase I sets 2 as the $h$-index of edge $e$. The values for the remaining edges are color coded red in Table 3.1. As can be seen, there are some edges whose $h$-indices are different from their true values. Also, further execution of Phase I does not update any $h$-index.

We tighten the upper-bound using Phase II function. For each edge, we check the validity of the assigned $h$-index obtained by Phase I. In particular, the condition $\Pr[\sup(e) \geq H \mid \Gamma, e \text{ exists}] \geq \gamma$ is checked for satisfaction. For instance, given edge $e = (1, 2)$, the $h$-index obtained by phase I is equal to $H = h[e] = 2$ which needs to be verified. $\Gamma = \{ \triangle_{1,2,3}, \triangle_{1,2,4} \}$ is the set of the triangles which contain $e$, and the minimum $h$-index of the other two edges in these triangles is at least 2. With the assumption that $e$ exists, $\Pr[\sup(e) \geq 2 \mid \Gamma, e \text{ exists}] = 1 \times (1 \times 0.3) \times (1 \times 0.5) = 0.15 < 0.2$. Therefore, edge $e$ cannot have $h$-index of 2. Next, for $H \in \{0, 1\}$ (values less than 2) should be checked. Algorithm picks 1 and similarly checks $\Pr[\sup(e) \geq 1 \mid \Gamma, e \text{ exists}] = 1 - \Pr[\sup(e) = 0 \mid \Gamma, e \text{ exists}] = 1 \times (1 - (1 - 1) \times (1 - 0.3) \times (1 - 0.5)) = 1 \geq 0.2$, where the set $\Gamma$ is updated to $\Gamma = \{ \triangle_{0,1,2}, \triangle_{1,2,3}, \triangle_{1,2,4} \}$ in which each edge has $h$-index at least 1. Since the edge $e$ has enough support probability to have $h$-index of 1, thus its $h$-index is updated to 1. In a similar way, the $h$-index of other edges is tightened further. The obtained values at the end of Phase II are shown in Table 3.1. No updates happen in the next iteration, so convergence is obtained in a single iteration.
Chapter 4

Experiments, Results and Discussions

4.1 The Machine

Our implementations are in Java and the experiments are conducted on a commodity machine with Intel i7, 2.2Ghz CPU, and 12Gb RAM, running Ubuntu 14.03. The hard disk is Seagate Barracuda ST31000524AS 2TB 7200 RPM.

4.2 The Datasets and Webgraph

The statistics for the datasets we use are shown in Table 4.1.
Flicker, dblp, and biomine are real datasets with edge probabilities [5]. Dataset ljournal-2008 is from Laboratory of Web Algorithms: [http://law.di.unimi.it/datasets.php](http://law.di.unimi.it/datasets.php). For this dataset we generated probability values uniformly distributed in [0, 1].

To store and access the datasets, we use WebGraph [3] which is an excellent graph compression framework providing simple methods to handle very large graphs. WebGraph enables fitting large graphs in memory and therefore reduces the memory footprint. More specifically, WebGraph is a framework for graph compression exploiting modern compression techniques. According to the website [4], it consists of:

- A set of flat codes suitable for storing web graphs.

- Algorithms for compressing web graphs that exploit gap compression. The algorithms are controlled by several parameters, which provide different trade-offs between access speed and compression ratio.

- Algorithms for accessing a compressed graph without actually decompressing it, using lazy techniques that delay the decompression until it is actually necessary.

### Table 4.1: Dataset Statistics

| Graph   | |V| | |E| | |\(\Delta\) |
|---------|---|---|---|---|
| flickr  | 24,125 | 300,836 | 8,857,038 |
| dblp    | 684,911 | 2,284,991 | 4,582,169 |
| biomine | 1,008,201 | 6,722,503 | 93,716,868 |
| ljournal-2008 | 5,363,260 | 49,514,271 | 411,155,444 |
• Algorithms for analyzing very large graphs.

• A complete, documented implementation of the algorithms above in Java.

• Datasets for very large graphs (e.g., a billion of links).

Some of the works that have successfully used WebGraph for scaling various algorithms to big graphs are [8, 20, 40, 41, 43].

4.3 Running Time Comparison

Figure 4.1 represents the running times of our proposed approach (h-index_based), versus the state-of-the-art peeling approach [18]. We refer to it as DP_peeling. For biomine we present the results for different values of $\gamma$ ranging from 0.1 to 0.5. However, for the other datasets, we only show the results for $\gamma = 0.1$, and omit results for $\gamma = 0.2, \ldots, 0.5$, since they are similar to those for $\gamma = 0.1$ and their performance trend is similar to what we see for biomine.

As can be seen, h-index_based algorithm is significantly faster than DP_peeling. For instance, for biomine, which is a large dataset, the gain of our algorithm is 84 percent, making h-index_based six times faster than DP_peeling.
Table 4.2: Max $\eta$-support, max probabilistic trussness, $\gamma$.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$\gamma$-suppmax</th>
<th>kmax</th>
<th>$\gamma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>flickr</td>
<td>49</td>
<td>47</td>
<td>0.1</td>
</tr>
<tr>
<td>dblp</td>
<td>42</td>
<td>14</td>
<td>0.1</td>
</tr>
<tr>
<td>ljournall-2008</td>
<td>1030</td>
<td>51</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>151</td>
<td>33</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>143</td>
<td>30</td>
<td>0.2</td>
</tr>
<tr>
<td>biomine</td>
<td>135</td>
<td>28</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td>25</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>121</td>
<td>18</td>
<td>0.5</td>
</tr>
</tbody>
</table>

$h$-index-based produced the results in about 1.5 minutes and 1 minute for flickr and DBLP, respectively. Although flickr is smaller than dblp in terms of the number of vertices and edges, its probabilistic maximum truss is much greater at value 47 compared to 15 in dblp (see Table 4.2). On biomine which is a large dataset, our proposed algorithm completed in about 15 minutes on average; which is quite impressive. In contrast, DP_peeling produced the results in more than 2 hours. The running time for ljournal-2008 increases, which is quite reasonable, because this graph has 49 million edges with probabilistic support of 1030 when $\gamma = 0.1$.

4.3.1 The Effect of $\gamma$ Values on Running Times

The running time of both algorithms decrease as $\gamma$ becomes large. In terms of the effect of $\gamma$ on truss decomposition and support values, as can be seen in Table 4.2, the maximum truss and maximum initial probabilistic support decrease as $\gamma$ increases.
Figure 4.1: Running time of our $h$-index based updating algorithm versus edge peeling for probabilistic truss decomposition.
Figure 4.2: Average error (average of difference from true trussness) versus iterations for $\gamma = 0.1$

4.3.2 Convergence Speed

To further evaluate the execution of proHIT as it unfolds with time, we look at the average distance from the true truss values over the sequence of iterations (see Figure 4.4). The average error decreases fast for flicker, dblp, and biomine, and more gradually for ljournal-2008. These results show that proHIT can produce high-quality near-results in only a fraction of iterations needed for completion. For instance, for ljournal-2008 with $\gamma = 0.1$, the average error becomes less than at 0.01 at iteration 20, about one third of the total number of required iterations (about 60, see the end of the curve). This can be a desirable property in graph mining where the user would like to see near-results as the execution progresses.

We ignored comparing CLT-based peeling to our approach since CLT-based peeling algorithm provides approximations on true truss values while our approach outputs the true values.
Figure 4.3: Average error (average of difference from true trussness) versus iterations for $\gamma = 0.3$.

Figure 4.4: Average error (average of difference from true trussness) versus iterations for $\gamma = 0.5$. 
In this thesis, we studied the $k$-truss decomposition problem in probabilistic graphs. In chapter 2, essential definitions on graphs and also trusses were presented. Furthermore, the $k$-truss decomposition problem in both deterministic and probabilistic frameworks were discussed. We also reviewed properties and main existing truss decomposition algorithms on deterministic and probabilistic graphs. Truss decomposition has been broadly studied in deterministic graphs. However, not much research has been conducted on probabilistic graphs truss decomposition.

We used the local $(k, \gamma)$-truss notation used in [18] to define trusses in probabilistic graphs. Our novel algorithm computes truss decomposition in large scale probabilistic graphs. It is based on asynchronous $h$-index updating approach. Unlike the edge peeling strategy, our algorithm accesses the edges in an asynchronous order which makes it very memory efficient. The new algorithm has two main phases: *phase I* is responsible for edges’ $h$-index updating process and *phase II* takes care of probabilistic nature of truss decomposition. The results of our experiments show that the new
presented algorithm is significantly faster than edge peeling approach while requiring much less memory footprint.

Finally, we list three of several future directions we plan to explore.

Firstly, we would like to investigate new approaches and develop even more efficient algorithms for finding \((k, \gamma)\)-trusses which scale to bigger graphs.

Secondly, we would like to explore the applications of probabilistic truss decomposition to foundational problems on uncertain graphs of influence. One of the most important problems in such graphs is finding a set of seeds to propagate eventual influence to nodes of the graph. This is an intractable problem and even approximation solutions are computationally intensive (cf. [30, 31]). Selecting initial seeds based on endpoints of edges with a high truss value might prove to be a beneficial start in the search for high influential seeds. An example of the application of this problem would be in viral marketing. Detecting high influential seeds (users) can be beneficial to promote a product in social network.

Thirdly, we would like to extend truss decomposition to directed graphs. In those graphs we do not talk about triangles but “triads” (cf. [33]). There are 7 types of triads in directed graphs, and defining truss in terms of them might prove to be quite challenging.
Bibliography


