Efficient Probabilistic Truss Indexing on Uncertain Graphs

Zitan Sun
Hong Kong Baptist University
Hong Kong, China
sunzitan@comp.hkbu.edu.hk

Xin Huang
Hong Kong Baptist University
Hong Kong, China
xinhuang@comp.hkbu.edu.hk

Jianliang Xu
Hong Kong Baptist University
Hong Kong, China
xujl@comp.hkbu.edu.hk

Francesco Bonchi
ISI Foundation
Turin, Italy
francesco.bonchi@isi.it

ABSTRACT

Networks in many real-world applications come with an inherent uncertainty in their structure, due to e.g., noisy measurements, inference and prediction models, or for privacy purposes. Modeling and analyzing uncertain graphs has attracted a great deal of attention. Among the various graph analytic tasks studied, the extraction of dense substructures, such as cores or trusses, has a central role.

In this paper, we study the problem of (k, γ)-truss indexing and querying over an uncertain graph $\mathcal{G}$. A (k, γ)-truss is the largest subgraph of $\mathcal{G}$, such that the probability of each edge being contained in at least $k-2$ triangles is no less than γ. Our first proposal, CPT-index, keeps all the (k, γ)-trusses: retrieval for any given k and γ can be executed in an optimal linear time w.r.t. the graph size of the queried (k, γ)-truss. We develop a bottom-up CPT-index construction scheme and an improved algorithm for fast CPT-index construction using top-down graph partitions. For trading off between (k, γ)-truss offline indexing and online querying, we further develop an approximate indexing approach (ϵ, Δ_r)-APX equipped with two parameters, ϵ and Δ_r, that govern tolerated errors.

Extensive experiments using large-scale uncertain graphs with 261 million edges validate the efficiency of our proposed indexing and querying algorithms against state-of-the-art methods.

CCS CONCEPTS

• **Information systems** → **Clustering**; **Database query processing**; • **Mathematics of computing** → *Graph algorithms*; • **Theory of computation** → Probabilistic computation.

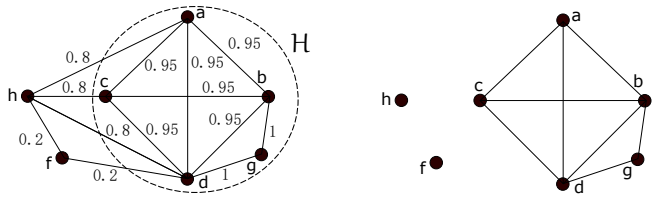
KEYWORDS

probabilistic k-truss, uncertain graph, indexing, query

ACM Reference Format:
Zitan Sun, Xin Huang, Jianliang Xu, and Francesco Bonchi. 2021. Efficient Probabilistic Truss Indexing on Uncertain Graphs. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3442381.3449976

WWW '21, April 19–23, 2021, Ljubljana, Slovenia

ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3449976



(a) Uncertain graph $\mathcal{G}$ (b) Possible world H

Figure 1: An example of uncertain graph $\mathcal{G}$. $\mathcal{H}$ is the $(3, 0.9)$-truss of $\mathcal{G}$. The deterministic graph H is a possible world of $\mathcal{G}$.

1 INTRODUCTION

Graph is a widely used model to represent entities and their relationships in many application domains, such as biological networks, social and communication networks, or traffic networks, just to mention a few. In many such applications, uncertainty on the network structure is inherently part of the analysis. In fact, the network is rarely directly observable: in most of the cases its structure must be inferred from noisy data and by means of correlation analyses, machine learning modeling, and other imprecise processes. Such uncertainty can be captured and modeled by means of *probabilistic graphs*, or *uncertain graphs*, where each edge is associated with tis own probability of existence [31]. Uncertainty on the network structure can also be explicitly adopted as a privacy measure [6].

One key graph analytic task is the identification of dense substructures. In the literature, there exist a plethora of different dense subgraph definitions, both in deterministic and uncertain graphs, including cliques, quasi-cliques [30], n-clans [26], n-club [26], k-core [33], k-truss [36], (k, η)-core [7], and (k, γ)-truss [42][18][14]. In a deterministic graph, k-truss is the largest subgraph, such that every edge is contained in at least $(k-2)$ triangles. For example, consider the graph G in Figure 1(a) ignoring the edge probabilities: the whole graph is a 3-truss, as every edge is contained in at least one triangle. Given an uncertain graph $\mathcal{G}$ and two parameters $k \geq 2$ and $\gamma \in (0, 1]$, a (k, γ)-*truss* is a subgraph $\mathcal{H} \subseteq \mathcal{G}$ in which the probability of each edge being contained in at least $k-2$ triangles is no less than γ [42][18]. In Figure 1(a), the probability of edge (f, d) being contained in one triangle is $0.2 \times 0.2 \times 0.8 = 0.032$, and those of other edges are no less than 0.032; hence, the whole graph $\mathcal{G}$ is a $(3, 0.032)$-truss.

In this paper, we tackle the problem of constructing an offline index to support efficient (k, γ)-truss retrieval for any k, γ. The

(k,γ)-truss indexing is particularly useful in applications in which many (k,γ)-truss retrieval queries must be issued in order to find a solution. Some examples include *probabilistic triangle densest subgraph discovery* [35], *team formation* and *community search* [15, 17], and *module detection* for critical medical assessments, such as in cancer diagnoses [18]. In Sections 8.3 and 8.4, we validate the usefulness of (k,γ)-truss indexing in practice for speeding-up task-driven team formation [7, 18] and approximating triangle densest subgraph discovery [35] over uncertain graphs.

Challenges and contributions. Efficient (k,γ)-truss indexing and retrieval presents significant challenges, mostly due to the enumeration of all possible k and γ combinations.

For a given γ, it exists a (k,γ)-truss decomposition method to find all the k-trusses of an uncertain graph $\mathcal{G}$ [18]. Unfortunately, a straightforward extension of such method is inefficient for (k,γ)-truss indexing and retrieval. On the one hand, as there exist infinite choices for value of $\gamma \in (0,1]$, building a decomposition for many values of γ would lead to combinatorial blow-up and inefficiency. On the other hand, an online search approach applying (k,γ)-truss decomposition [18] each time from scratch for a given k and γ, would not benefit of an index and would incur in high time complexity especially for large real-world graphs. The need for an index gets even worse when many (k,γ)-truss queries are repeatedly issued.

To tackle problems efficiently, we analyze and find several useful structural properties of (k,γ)-truss. The key observation is that, for a given k, the maximum probability γ of an edge that this edge is contained in a (k,γ)-truss, can be computed in polynomial time. Thus, the trussness of every edge for all possible k can be pre-computed. Moreover, we analyze the hierarchical properties of (k,γ)-truss and design a compact and elegant CPT-index to keep all the (k,γ)-trusses. To construct the CPT-index, we develop a bottom-up CPT-Basic construction to keep the (k,γ)-truss information from the smallest k to the largest k. To improve the efficiency, we further propose a fast algorithm CPT-Fast to construct CPT-index from the largest k to the smallest k using graph partitions.

Besides these two exact algorithms, which generate bottom-up and top-down CPT-index constructions, we also propose an approximate indexing scheme (ϵ,Δ_r)-APX. (ϵ,Δ_r)-APX builds up an approximate index, which keeps the approximate trussnesses within a given error Δ_r for partial edges that are contained in the (k,γ)-truss for $\gamma \geq \epsilon$. Built upon CPT-Fast, (ϵ,Δ_r)-APX achieves a good level of efficiency to balance out the trade-off between index construction and (k,γ)-truss retrieval.

The contributions of this paper can be summarized as follows:

- We formulate the problem of building an index to maintain all the (k,γ)-trusses of an uncertain graph $\mathcal{G}$ for any value of k and γ, for efficient (k,γ)-truss retrieval (Section 3).
- Exploiting structural properties of (k,γ)-truss we devise a compact CPT-index to keep the probabilistic trussnesses for all edges in $\mathcal{G}$. To construct the CPT-index, we propose a bottom-up indexing approach, CPT-Basic, through which to compute all the (k,γ)-trusses (Section 5).
- We present a top-down algorithm, CPT-Fast for building the CPT-index. To speed up efficiency, CPT-Fast delays the initialization and update of probabilistic support vectors. CPT-index based (k,γ)-truss retrieval can be carried out in an optimal linear time for any given k and γ (Section 6).
- To achieve efficient (k,γ)-truss indexing on large graphs, we develop an approximate indexing scheme named (ϵ,Δ_r)-APX (Section 7).
- Our extensive empirical analysis on nine real-world uncertain graphs (Section 8) demonstrates that (i) our index-based approaches are several orders of magnitude faster than the online search approach [18], (ii) have useful applications in task-driven team formation and approximating probabilistic triangle densest subgraph discovery, and (iii) that our (ϵ,Δ_r)-APX scheme scales to a large graph with 261M edges.

2 RELATED WORK

Our work is related to *uncertain graph analytics* and *k-truss mining*.

Uncertain Graph Analytics. In the literature, numerous kinds of graph analytic problems have been extended from deterministic graphs to uncertain graphs [9, 19, 22, 23, 28, 38, 39], such as dynamic skyline queries [3], k-nearest neighbor searches [31], top-k maximal cliques [41], graph sparsification [29], and (k,η)-core [7], just to mention a few.

The study most related to our work is the probabilistic k-core indexing proposed by Yang et al. [37]. The key distinctions with our work are as follows. (1) While [37] studies k-core we focus on a different dense subgraph model, k-truss, which has typically higher density and clustering coefficient than k-core in both deterministic [36] and uncertain graphs [18]. The (k,γ)-truss decomposition has also a higher computational complexity than (k,η)-core decomposition [37], which makes the (k,γ)-truss indexing task more challenging. (2) Different optimization techniques are developed. In particular, [37] develops techniques to accelerate the probabilistic vector initialization, while our top-down CPT-Fast method accelerates both phases of vector initialization and vector update. (3) Compared with (k,η)-core indexing [37], we not only have an exact indexing method for optimal linear time query processing, but also propose a novel approximate indexing algorithm by keeping partial indexes to be scalable on large real-world graphs.

K-Truss Mining. Many recent works have studied k-truss mining [15][21][4][8], which finds a dense subgraph of k-truss such that each edge has at least $k-2$ triangles within the k-truss. The discovery of k-trusses has been widely studied over various kinds of graphs, including deterministic graphs [36], uncertain graphs [14, 18, 42], dynamic graphs [15, 40], directed graphs [25, 34], attributed graphs [16], simplicial complexes [32] and public-private networks [13]. Moreover, truss decomposition that finds all k-trusses on a graph has in-memory algorithms [36], shared-memory algorithms [20], external-memory algorithms [42], and cloud computing [10]. Chen et al. [10] design graph-parallel algorithms for truss decomposition in the scenario of cloud computing. Huang et al. [18] study the probabilistic truss decomposition on uncertain graphs, which finds the (k,γ)-truss for a given parameter γ. Fatemeh et al. [14] use the central limit theorem to accelerate probability

calculations in regard to a (k,γ)-truss, instead of using dynamic programming techniques [18]. Different from the above studies, this paper investigates a different problem regarding (k,γ)-truss indexing, to keep all possible (k,γ)-trusses in a data structure and develop novel algorithms.

3 PRELIMINARIES

In this section, we present the definitions and problem formulations.

3.1 Uncertain Graphs

Let $\mathcal{G} = (V, E, p)$ be an *uncertain* (or *probabilistic*) graph, where V is the set of $n = |V|$ vertices, E is the set of $m = |E|$ edges, and $p\colon E \to [0,1]$ is a probabilistic function. Without loss of generality, we assume that the existence probability of an edge $e \in E$ is non-negative and denoted by $p(e) \in (0,1]$. Each edge is supposed to exist independently [31]. The larger its probability, the larger chance that it exists. Let $N(v)$ be the neighbor set of vertex v, i.e., $N(v) = \{u|(v,u) \in E\}$. The degree of v is $d(v) = |N(v)|$. A commonly used method of analyzing uncertain graphs is to apply the *possible worlds* model, as follows.

Definition 1 (Possible World). *Given a possible world $G = (V, E_G)$ of an uncertain graph $\mathcal{G}$, G is a deterministic graph where the set of edges $E_G \subseteq E$ exists for certain, denoted by $G \sqsubseteq \mathcal{G}$.*

Note that a possible world $G \sqsubseteq \mathcal{G}$ keeps all the same vertices of $\mathcal{G}$. The existence probability of a possible world $G \sqsubseteq \mathcal{G}$ can be calculated as follows.

$$\Pr[G|\mathcal{G}] =_{\text{def}} \prod_{e\in E_G} p(e) \prod_{e\in E\setminus E_G} (1-p(e)). \quad (1)$$

Example 1. *Consider a probabilistic graph $\mathcal{G}$ in Figure 1(a) and a possible world $H \sqsubseteq \mathcal{G}$ in Figure 1(b). By applying Eq.* (1)*, we have* $\Pr[H|\mathcal{G}] = 0.95^6 \times 1^2 \times (1-0.8)^3 \times (1-0.2)^2 = 0.00376367048$.

3.2 Probabilistic (k,γ)-truss

We first define the k-truss in deterministic graphs and then extend its definition to the probabilistic (k,γ)-truss in uncertain graphs. Given a deterministic graph G, a subgraph $H(V(H), E(H))$ of G has the vertex set $V(H) \subseteq V(G)$ and the edge set $E(H) = \{(v,u) \in E(G)|v,u \in V(H)\}$. A triangle is a cycle of three vertices, v,u,w, denoted as $\triangle_{uvw}$. For an edge $e = (v,u)$ in a graph H, we define the number of triangles containing e as the *support* of e in H, denoted by $\sup_H(e) = |\{\triangle_{vuw} \mid (v,u),(v,w),(u,w) \in E_H\}|$. Based on the definition of support, we formulate a dense subgraph of k-truss as follows.

Definition 2 (k-truss). *A subgraph H is the k-truss of G if and only if H is the largest subgraph of G, such that $\forall e \in E(H)$ is contained in at least $k-2$ triangles in H, i.e.,* $\sup_H(e) \geq k-2$.

The k-truss of G is represented in T_k. The largest value of k, such that a non-empty $T_k \subseteq G$, is denoted by k_{max}. Let $k_{\max} = \max\{k \in \mathcal{Z} : \exists \text{ a non-empty } T_k \subseteq G\}$. Thus, graph G can be decomposed into multiple k-trusses $\{T_k|2 \leq k \leq k_{max}\}$.

Example 2. *Consider the deterministic graph H in Figure 1(b). The triangle $\triangle_{bdg}$ is formed by three vertices, b, d, g. Thus, the support of edge (b,g) is* $\sup_H((b,g)) = 1$. *The whole graph H is the 3-truss as T_3. In addition, we have $k_{max} = 4$ for H as the induced subgraph of H by the vertex set $\{a,b,c,d\}$ is the 4-truss T_4 in H.*

Uncertain Support. We consider the uncertain support of edges. Given an edge e in an uncertain graph $\mathcal{H}$, the triangles containing e exist with an uncertain probability. For example, the triangle $\triangle_{hdf}$ in Figure 1(a) exists with a probability value of $0.2 \times 0.2 \times 0.8 = 0.032$. Based on the possible world model, the uncertain support of $e \in E(H)$ is defined as the number of triangles containing e in $\mathcal{H}$, denoted as $\sup_{\mathcal{H}}(e)$. Let $\Pr[\sup_{\mathcal{H}}(e) = t]$ be the probability that the support $\sup_{\mathcal{H}}(e)$ equals t in $\mathcal{H}$, which is the sum of the probability mass of all possible worlds $H \sqsubseteq \mathcal{H}$ that e is contained in exactly t triangles in H. Specifically, we have the following definition.

Definition 3 (Uncertain Support). *Given an edge e in graph $\mathcal{H}$, the probability of uncertain support* $\sup_{\mathcal{H}}(e) = t$ *is*

$$\Pr[\sup_{\mathcal{H}}(e) = t] = \sum_{H\sqsubseteq\mathcal{H}} \Pr[H|\mathcal{H}] \cdot \mathbf{I}(\sup_H(e) = t), \quad (2)$$

where $\mathbf{I}(\sup_H(e) = t)$ *is an indicator function which takes on* 1 *if* $\sup_H(e) = t$*, and* 0 *otherwise.*

Consequently, we define the probability that $\sup_{\mathcal{H}}(e) \geq t$ as

$$\Pr[\sup_{\mathcal{H}}(e) \geq t] = \sum_{H\sqsubseteq\mathcal{H}} \Pr[H|\mathcal{H}] \cdot \mathbf{I}(\sup_H(e) \geq t), \quad (3)$$

where $\mathbf{I}(\sup_H(e) \geq t)$ is an indicator function which takes on 1 if $\sup_H(e) \geq k$, and 0 otherwise. Obviously, $\Pr[\sup_{\mathcal{H}}(e) \geq t] = 1 - \sum_{i=0}^{t-1} \Pr[\sup_{\mathcal{H}}(e) = i]$ holds. To simplify this, we denote $\Pr[\sup_{\mathcal{H}}(e) \geq t]$ by $\sigma_{\mathcal{H}}(e,t)$, i.e., $\sigma_{\mathcal{H}}(e,t) =_{\text{def}} \Pr[\sup_{\mathcal{H}}(e) \geq t]$. Moreover, let $\sigma_{\mathcal{H}}(e) =_{\text{def}} [\sigma_{\mathcal{H}}(e,0), \sigma_{\mathcal{H}}(e,1), \ldots, \sigma_{\mathcal{H}}(e,t_e)]$ be the vector of support probabilities of $e = (u,v)$ from 0 to t_e, where $t_e = |N(u) \cap N(v)|$ is the number of common neighbors between u and v. When it is clear from the context, we drop the subscripts $\mathcal{H}$ from all the notations, e.g., $\sigma(e)$, and $\sigma(e,t)$.

Based on the definitions of uncertain support and k-truss above, we show that the deterministic k-truss can be extended to probabilistic graphs as (k,γ)-truss [18] as follows.

Definition 4 ((k,γ)-truss). *Given an uncertain graph $\mathcal{G}$, and two numbers, k and γ, $\mathcal{H}$ is the (k,γ)-truss of $\mathcal{G}$ if and only if $\mathcal{H}$ is the largest subgraph in $\mathcal{G}$, such that each edge $e \in E(\mathcal{H})$ has the probability of* $\sup_{\mathcal{H}}(e) \geq k-2$ *no less than γ, i.e., $\sigma_{\mathcal{H}}(e, k-2) \geq \gamma$.*

Example 3. *Consider an uncertain graph $\mathcal{G}$ in Figure 1(a). For an edge $e = (a,h)$, we have $e \in \triangle_{ach}$ and $e \in \triangle_{adh}$. Thus, the uncertain supports of e are calculated as* $\Pr[\sup_{\mathcal{G}}((a,h)) = 2] = 0.8^3 \times 0.95^2 = 0.46208$ *and* $\Pr[\sup_{\mathcal{G}}((a,h)) = 1] = 2 \times 0.8 \times 0.95 \times 0.8 \times (1 - 0.95 \times 0.8) = 0.29184$. *Therefore, the support probability vector of e is $\sigma((a,h)) = [1, 0.75392, 0.46208]$. In addition, the whole graph $\mathcal{G}$ is $(3, 0.032)$-truss because $\forall e \in E(\mathcal{G})$, $\sigma(e,1) \geq 0.032$.*

3.3 Problem Formulation

In this paper, we are interested in indexing and querying (k,γ)-trusses for all possible values of k and γ in a given graph $\mathcal{G}$. We formulate this problem of *probabilistic truss indexing and querying* as *(k,γ)-truss indexing.*

Problem 1. *Given a probabilistic graph $\mathcal{G} = (V, E, p)$, the problem of (k,γ)-truss indexing on $\mathcal{G}$ involves constructing a (k,γ)-truss index and answering the (k,γ)-truss retrieval for all possible $2 \leq k \leq k_{max}$ and $\gamma \in (0,1]$.*

Consider a query of (k,γ)-truss retrieval on $\mathcal{G}$ in Figure 1(a) for $k=3$ and $\gamma=0.9$. The answer of $(3,0.9)$-truss is $\mathcal{H}$ shown in Figure 1(a), which is an induced subgraph of $\mathcal{G}$ by vertices $\{a,b,c,d,g\}$. Note that all proposed techniques in this paper are naturally extended to discover the connected components of (k,γ)-truss as the *maximal connected* (k,γ)*-truss* [18].

4 EXISTING ALGORITHM

In this section, we introduce an algorithm of (k,γ)-truss decomposition for a given γ [18]. The general idea of (k,γ)-truss decomposition uses a peeling strategy, which iteratively removes a disqualified edge from a graph until the remaining graph is the (k,γ)-truss. To identify the qualification of an edge for the (k,γ)-truss, the γ-based maximum support is defined for an edge e, denoted by $\sup^{\gamma}(e)=\max_t\{t\in\mathcal{Z}_0^+|\sigma(e,t)\geq\gamma \text{ and } \sigma(e,t+1)<\gamma\}$. Thus, for the given k and γ, all edges e with $\sup^{\gamma}(e)<k-2$ can be removed from graph $\mathcal{G}$ for (k,γ)-truss decomposition.

(k,γ)-Truss Decomposition Algorithm. Algorithm 1 presents the pseudo-code for the (k,γ)-truss decomposition procedure [18]. The algorithm first computes the edge support probability vector $\sigma(e)$ for all edges e in the uncertain graph $\mathcal{G}$ (lines 1-2). Next, it uses the peeling strategy to iteratively find all (k,γ)-trusses, starting from the smallest $k\geq 2$ (lines 3-9). Specifically, it first identifies the value of k as the smallest γ-based maximum support among all existing edges (line 4). If there exists an edge $e=(u,v)$, such that $\sup^{\gamma}(e)\leq k-2$, it indicates that e belongs to the (k,γ)-truss but $e\notin(k+1,\gamma)$-truss. Thus, e is assigned a trussness value of k, denoted by $\tau^{\gamma}(e)=k$ and is removed from graph $\mathcal{G}$. After the edge removal of e, other edges in the neighborhood of e shall be updated accordingly (lines 8-9). This iterative process is repeated until all edges are removed from $\mathcal{G}$. The last value of k when the algorithm stops is $k_{\max}$. Note that all edges e with $p(e)<\gamma$ can be directly removed from $\mathcal{G}$ for preprocessing at the beginning of Algorithm 1.

Drawbacks of Algorithm 1 for indexing and querying (k,γ)-truss. Although Algorithm 1 can compute all the (k,γ)-trusses using dynamic programming techniques, several drawbacks still exist. First, we consider the limitations of Algorithm 1 for (k,γ)-truss indexing. The most significant drawback of Algorithm 1 is that it only works for a given γ. Thus, if we apply Algorithm 1 for (k,γ)-truss indexing, it needs to enumerate all possible values of γ and invoke the (k,γ)-truss decomposition for every γ on $\mathcal{G}$. However, the choices of the real number $\gamma\in[0,1]$ are infinite. This implies that a straightforward extension of Algorithm 1 leads to combinatorial blow-ups and that it is inefficient in regard to (k,γ)-truss indexing. Second, we consider the limitations of Algorithm 1 for (k,γ)-truss querying. We can directly apply Algorithm 1 without any index in order to find the (k,γ)-truss for any given k and γ. However, it still requires very expensive costs to compute and update all $\sigma(e)$ vectors in the time complexity of $O(d_{\max}\rho m)$, where $d_{\max}$ is the maximum degree and ρ is the graph arboricity of $\mathcal{G}$ with $\rho\leq\min\{d_{max},\sqrt{m}\}$ [11]. Thus, the online approach for (k,γ)-truss querying is inefficient in regard to large uncertain graphs. Motivated by the above limitations, we propose several novel approaches for efficient (k,γ)-truss indexing and querying in the following sections.

Algorithm 1 Local (k,γ)-Truss Decomposition

Input: $\mathcal{G}=(V,E,p)$; $\gamma\in[0,1]$;
Output: trussness score $\tau^{\gamma}(e)$ of each edge $e\in E$
1: **for all** $e\in E$ **do**
2: compute $\sigma(e)$ using dynamic programming [18];
3: **while** $E\neq\emptyset$ **do**
4: $k\leftarrow 2+\min_{e\in E}\{\sup^{\gamma}(e)\}$;
// The following loop is for (k,γ)-truss identification.
5: **while** $\exists e=(u,v)\in E$ s.t. $\sup^{\gamma}(e)\leq k-2$ **do**
6: Assign the trussness to e as $\tau^{\gamma}(e)=k$;
7: Remove edge e from graph $\mathcal{G}$;
8: **for** $w\in N(u)\cap N(v)$ **do**
9: Update $\sigma((u,w))$ and $\sigma((v,w))$ using a linear scan of dynamic programming algorithm [18];
10: **return** $\tau^{\gamma}(e)$ for all edges $e\in E$;

5 (k,γ)-TRUSS INDEXING

In this section, we propose a new index structure for keeping the **C**omplete information of all **P**robabilistic (k,γ)-**T**russ, referred to as the CPT-index. The CPT-index can support (k,γ)-truss queries with any parameters of k and γ. Before introducing the CPT-index, we first analyze several useful structural properties of (k,γ)-truss for efficient index construction. Then, we present a bottom-up CPT-index construction algorithm and a CPT-index-based method for (k,γ)-truss querying. Finally, we theoretically analyze the algorithm complexities.

5.1 Properties of (k,γ)-Truss

In the following, we first analyze the structural property of (k,γ)-truss and then define a key concept of *probabilistic trussness*.

Lemma 1. *(Hierarchical Property) Given two parameter pairs of (k_1,γ_1) and (k_2,γ_2), where $k_1\leq k_2$ and $\gamma_1\leq\gamma_2$, then two probabilistic trusses of $\mathcal{G}$ are (k_1,γ_1)-truss and (k_2,γ_2)-truss, denoted by $\mathcal{H}_1$ and $\mathcal{H}_2$ respectively, and $\mathcal{H}_2\subseteq\mathcal{H}_1$ holds.*

Proof. This property of $\mathcal{H}_2\subseteq\mathcal{H}_1$ holds naturally by the definition of (k,γ)-truss in Def. 4. For an edge $e\in E(\mathcal{H}_2)$, the uncertain support $\sup_{\mathcal{H}_2}(e)\geq(k_2-2)$ has a probability of no less than γ_2, i.e., $\sigma_{\mathcal{H}_2}(e,k_2-2)\geq\gamma_2$. As $k_1\leq k_2$ and $\gamma_1\leq\gamma_2$, $\sigma_{\mathcal{H}_2}(e,k_1-2)\geq\sigma_{\mathcal{H}_2}(e,k_2-2)\geq\gamma_2\geq\gamma_1$. As a result, each edge of H_2 satisfies the uncertain support constraint of (k_1,γ_1)-truss, indicating that $\mathcal{H}_2$ is a subgraph of (k_1,γ_1)-truss $\mathcal{H}_1$. Thus, $\mathcal{H}_2\subseteq\mathcal{H}_1$ holds. □

This lemma shows the relationship between different (k,γ)-trusses in $\mathcal{G}$. We define the *probabilistic trussness* as follows.

Definition 5 (Probabilistic Trussness). *Given an edge e in $\mathcal{G}$ and an integer $k\geq 2$, the probabilistic trussness is defined as the largest probability γ, such that there exists a non-empty (k,γ)-truss $\mathcal{H}\subseteq\mathcal{G}$ containing e, denoted by $\gamma_k^*(e)=\max_{\gamma}\{\gamma\in(0,1]|\ e\in E(\mathcal{H})$ and $\mathcal{H}$ is a non-empty (k,γ)-truss of $\mathcal{G}\}$.*

The probabilistic trussness is a very important indicator with which to judge the existence of an edge e in a given (k,γ)-truss. Given two parameters k and γ, we can easily use the probabilistic trussness to identify the existence of $e\in(k,\gamma)$-truss as follows.

Table 1: An example of the CPT-index on graph $\mathcal{G}$ in Figure 1(a)

$k\backslash\gamma^*$	0.32	0.46208	0.75392	0.7737809	0.9409691	0.95
3	(d,f) (h,f)		(a,h) (d,h) (c,h)		(a,b) (b,c) (c,d) (a,d) (a,c)	(b,d) (d,g) (b,g)
4		(a,h) (d,h) (c,h)		(a,b) (b,c) (a,c) (a,d) (b,d) (c,d)		

Lemma 2. *Given an edge e in graph $\mathcal{G}$ and an integer k, the edge e belongs to the (k,γ)-truss* if and only if $\gamma \le \gamma_k^*(e)$.

According to Lemma 2, we can infer that, for any number $\gamma > \gamma_k^*(e)$, e does not belong to the (k,γ)-truss in $\mathcal{G}$. For instance, in Figure 1(a), edge (a,h) has $\gamma_3^*(e)$ with 0.75392, so edge (a,h) is in $(3,0.75392)$-truss or in $(3,0.7)$-truss but not in $(3,0.8)$-truss. We analyze the non-increasing property of probabilistic trussness.

Lemma 3 (Non-increasing Probability). *For an edge e in graph $\mathcal{G}$, and two integers k_1, k_2 with $k_1 < k_2$, we have the probabilistic trussnesses of e with $\gamma_{k_1}^*(e) \ge \gamma_{k_2}^*(e)$.*

Proof. This property can be inferred by the hierarchical property of (k,γ)-truss, i.e., $(k_2,\gamma_{k_2}^*(e))$-truss $\subseteq$ $(k_1,\gamma_{k_2}^*(e))$-truss for $k_2 > k_1$. Let e belong to $(k_2,\gamma_{k_2}^*(e))$-truss; thus, $e \in (k_1,\gamma_{k_2}^*(e))$-truss and $e \in (k_1,\gamma_{k_1}^*(e))$-truss. Based on Lemma 2, $\gamma_{k_2}^*(e) \le \gamma_{k_1}^*(e)$ holds. □

In Figure 1(a), the edge (a,h) has $\gamma_3^*((a,h)) = 0.75392$ and $\gamma_4^*((a,h)) = 0.46208 \le \gamma_3^*((a,h))$.

5.2 CPT-Index and (k,γ)-Truss Retrieval

We introduce the CPT-index based on probabilistic trussnesses.

CPT-Index. The data structure of the CPT-index is a table. It consists of all edges $e \in E(\mathcal{G})$ with their probabilistic trussness $\gamma_k^*(e)$ for all possible values of k. Specifically, for each number $2 \le k \le k_{max}$, the CPT-index maintains a sorted list of probabilistic edges in the increasing order of $\gamma_k^*(e)$, which can imply all possible (k,γ)-truss for $\gamma \in (0,1]$. Table 1 shows an example of the CPT-index for a graph $\mathcal{G}$ in Figure 1(a). It has only three values $k \in \{2,3,4\}$, so $k_{max} = 4$ and any (k,γ)-truss with $k > 4$ is empty. Edges (d,f) and (h,f) belong to set $(k = 3, \gamma^* = 0.32)$, so these edges have $\gamma_3^* = 0.32$. Note that an edge may appear in multiple (k,γ)-trusses but in a unique $(k,\gamma_k^*(e))$-truss for a specific k.

Index-based Retrieval of (k,γ)-Truss. Next, we show how to use the CPT-index to efficiently find (k,γ)-truss. Assuming that $k = 3$ and $\gamma = 0.9$, we seek to find the (k,γ)-truss of $\mathcal{G}$ in Figure 1(a). Based on the CPT-index in Table 1, we can directly look to the row with $k = 3$, which takes $O(1)$ time and merge all the edges e with probabilistic trussness $\gamma_3^*(e) \ge 0.9$, which takes $O(|Ans|)$ time, where $|Ans|$ is the number of edges in the answer (k,γ)-truss, if we compare probabilistic trussnesses from large to small. Finally, we obtain the $(3,0.9)$-truss, which consists of the edges $(a,b),(b,c),(a,c),(a,d),(c,d),(b,d),(b,g)$ and (d,g). In summary, the CPT-index-based (k,γ)-truss retrieval takes $O(|Ans|)$ time, indicating an optimal retrieval of (k,γ)-truss based on the CPT-index.

Algorithm 2 CPT-Basic

Input: $\mathcal{G} = (V,E,p)$
Output: The CPT-index of $\mathcal{G}$

```
1: Extract a deterministic graph G = (V, E) from 𝒢;
2: Apply the truss decomposition on G and get trussness k_max;
3: for k ← 2 to k_max do
4:    Let H be the k-truss of G;
5:    Let ℋ = (V_H, E_H, p) be an induced subgraph of 𝒢 by H;
6:    Compute the support vector σ(e) for each edge e in ℋ;
7:    while ℋ ≠ ∅ do
8:       Find an edge ê with the smallest σ(ê, k − 2) from ℋ;
9:       γ*_k(ê) ← σ(ê, k − 2);
10:      An edge queue: Q ← {ê}; An edge set: S ← ∅;
11:      while Q ≠ ∅ do
12:         Pop out an edge (u, v) from Q;
13:         Remove (u, v) from ℋ;
14:         S ← S ∪ {(u, v)};
15:         for w ∈ N(u) ∩ N(v) do
16:            Update the support vectors σ_(w,u) and σ_(w,v);
17:            if σ((w, u), k − 2) ≤ γ*_k(ê) then
18:               Q ← Q ∪ {(w, u)};
19:            if σ((w, v), k − 2) ≤ γ*_k(ê) then
20:               Q ← Q ∪ {(w, v)};
21:      Add (k, e, γ*_k(ê)) for all edges e ∈ S into the CPT-index;
22: return {(k, e, γ*_k(e)) | 2 ≤ k ≤ k_max, e ∈ E(𝒢), γ*_k(e) > 0};
```

5.3 CPT-Index Construction

We start with an observation.

Lemma 4. *[Minimum Edge Support Probability] Given an uncertain graph $\mathcal{H}$ and number k, each edge $e \in E(\mathcal{H})$ has the probability $\sigma(e,k-2) > 0$. There exists an edge $\hat{e} \in E(\mathcal{H})$ has the minimum edge support probability $\sigma(\hat{e},k-2)$ in $\mathcal{H}$ and $\gamma_k^*(\hat{e}) = \sigma(\hat{e},k-2)$.*

Proof. First, $\sigma(\hat{e},k-2) > 0$ as $\hat{e} \in E(\mathcal{H})$. Second, $\sigma(\hat{e},k-2)$ is the minimum one; thus, $\forall e \in E(\mathcal{H})$, $\sigma(e,k-2) \ge \sigma(\hat{e},k-2)$. Therefore, the whole graph $\mathcal{H}$ is $(k,\sigma(\hat{e},k-2))$-truss. For any $\gamma > \sigma(\hat{e},k-2)$, (k,γ)-truss requires all edges e to meet $\sigma(e,k-2) \ge \gamma$, so $\hat{e} \notin (k,\gamma)$-truss. In conclusion, $\gamma_k^*(\hat{e}) = \sigma(\hat{e},k-2)$. □

Note that not all edges e satisfy $\gamma_k^*(e) = \sigma(e,k-2)$. For example, consider two edges e_1 and e_2 that $\sigma(e_1,k-2) < \sigma(e_2,k-2)$, we suppose $\gamma_k^*(e_1) = \sigma(e_1,k-2)$. When edge e_1 is removed, edge e_2 should also be removed if it does not meet the truss requirements, and $\gamma_k^*(e_2)$ should be set to $\sigma(e_1,k-2)$.

Based on the observation above, we can develop a peeling algorithm of CPT-index construction in a bottom-up manner. The general idea is that we start from $k = 2$ and extract the $(k,0)$-truss of $\mathcal{G}$, which is a deterministic k-truss of $G(V,E)$. Then, we iteratively delete an edge $\hat{e}$ with the minimum edge support probability in graph $\mathcal{G}$ and assign it the probabilistic trussness of $\gamma_k^*(\hat{e})$; we repeat the peeling process of edge removal until the remaining graph is empty. After that, we increase k by one and repeat the above process until k exceeds k_{max}.

BC-Indexing Algorithm. The detailed procedure of CPT-Basic for CPT-index construction is presented in Algorithm 2. First, the algorithm extracts the deterministic graph $G = (V, E)$ of $\mathcal{G} = (V, E, p)$, which takes all edges by ignoring their edge probabilities (line 1). It then applies the algorithm of truss decomposition [36] on G and obtains the largest value of edge trussness as k_{max} (line 2). Next, the algorithm starts from $k = 2$ and iteratively computes the CPT-index for different values of k until k achieves k_{max} (lines 3-21). The algorithm extracts an uncertain subgraph $\mathcal{H}$ of $\mathcal{G}$ induced by the k-truss H (lines 4-5). During the edge peeling process, it finds an edge $\hat{e}$ with the smallest $\sigma(\hat{e}, k-2)$ in $\mathcal{H}$, and assigns the probabilistic trussness as $\gamma_k^*(\hat{e}) = \sigma(\hat{e}, k-2)$ (lines 8-9). Next, it deletes $\hat{e}$ from graph $\mathcal{H}$ and updates the support vectors of affected edges in $\mathcal{H}$ (lines 11-20). The process of edge removal is conducted in a BFS manner, which finds all edges not belonging to $(k, \gamma_k^*(\hat{e}))$-truss (lines 17-20) and puts them into an edge set S (line 14). After the edge removal, we update the CPT-index by adding all triple elements of $(k, e, \gamma_k^*(\hat{e}))$ for each edge $e \in S$ (line 21). The algorithm repeats the above process to enumerate all (k, γ)-trusses in $\mathcal{G}$ until $k = k_{max}$ and CPT-index construction is finished (lines 3-21).

Complexity Analysis. We denote the edge size of deterministic k-truss T_k as $m_k = |T_k|$ where $|E| = m \geq m_k$ for $k \in [2, k_{max}]$, and $\mathcal{T} = \sum_{k=2}^{k_{max}} m_k$. The arboricity of graph G is the minimum number of spanning forests needed to cover all edges of G, which is denoted as ρ and $\rho \leq \min\{d_{max}, \sqrt{m}\}$. The arboricity of T_k is denoted as ρ_k where $\rho_k \leq \rho$ for $k \in [2, k_{max}]$. Note that $\rho = \rho_2$ and $m = m_2$.

Theorem 1. CPT-Basic *in Algorithm 2 takes* $O(\rho\mathcal{T}(d_{max} + \log m))$ *time and* $O(\rho m)$ *space.*

Proof. First, we analyze the time complexity. Algorithm 2 extracts a deterministic graph G and performs the truss decomposition on G, which takes $O(\rho m)$ time (lines 1-2). Then, for each $k \in [2, k_{max}]$, it extracts the deterministic k-truss $H = T_k$ in $O(|T_k|) = O(m)$ time, and then applies the probabilistic truss decomposition to calculate the edge trussnesses (lines 4-21). It takes $O((\min\{d(u), d(v)\})^2)$ time to compute an initial vector $\sigma((u, v))$ and update it for an edge $(u, v) \in T_k$ (lines 6 & 15-16). Thus, the support vector for all edges in T_k takes $O(\sum_{(u,v)\in T_k} (\min\{d(u), d(v)\})^2) = O(d_{max} \sum_{(u,v)\in T_k} \min\{d(u), d(v)\}) \leq O(d_{max}\rho_k m_k)$. In addition, Algorithm 2 needs to maintain a balanced binary search tree, in order to get the edge with the smallest $\sigma(\hat{e}, k-2)$ (lines 8 and 13). The construction and maintenance of a balanced binary search tree in $\mathcal{H}$ take in a total of $O(\sum_{(u,v)\in T_k} \min\{d(u), d(v)\} \log |T_k|) = O(\rho_k m_k \log m_k)$ time. Overall, the time complexity of CPT-Basic in Algorithm 2 is

$$
\begin{aligned}
&O(\rho m + \sum_{k=2}^{k_{max}} (m_k + d_{max}\rho_k m_k + \rho_k m_k \log m_k)) \\
&= O(\rho \cdot (d_{max} + \log m) \cdot \sum_{k=2}^{k_{max}} m_k) \\
&= O(\rho\mathcal{T}(d_{max} + \log m)).
\end{aligned}
$$

Next, we analyze the space complexity. The vector $\sigma(e)$ for all edges in G takes $O(\sum_{(u,v)\in\mathcal{G}} \min\{d(u), d(v)\}) = O(\rho m)$ space. □

Theorem 2. *The* CPT-index *takes* $O(\mathcal{T})$ *space. For any* k *and* γ, CPT-index*-based* (k, γ)*-truss retrieval can be done in* $O(|Ans|)$ *time, where* $|Ans|$ *is the graph size of* (k, γ)*-truss answer.*

Proof. For any $k \in [2, k_{max}]$, each edge $e \in T_k$ has a unique $\gamma_k^*(e)$. For those edges e in G but not belong to T_k, we do not keep the trussness $\gamma_k^*(e)$. Thus, the space complexity of CPT-index is $O(\sum_{k=2}^{k_{max}} |T_k|) = O(\mathcal{T})$. The time complexity analysis of CPT-index-based (k, γ)-truss retrieval can be found in Section 5.2. □

6 FAST CPT-INDEX CONSTRUCTION

In this section, we present a fast CPT-index construction algorithm CPT-Fast, which constructs the optimal CPT-index from $k = k_{max}$ to 2 in a top-down manner. The general idea is to use Lemma 1 to delay the initialization and update of $\sigma(e)$ for all possible edges $e \in E$, which can speed up the index construction process.

Overview. Recall that CPT-Basic adopts the peeling strategy to remove an edge e with the smallest probabilistic support $\sigma(e, k-2)$in each iteration. Due to the removal of e, the probabilistic support of other edges needs to be updated (in line 16 of Algorithm 2). However, this part of the updating computation process can be saved.

We start with an observation. For a given $k \geq 2$, let be γ_1 and γ_2 with $\gamma_1 \leq \gamma_2$. According to Lemma 1, we have (k, γ_2)-truss $\subseteq$ (k, γ_1)-truss. Thus, for any edge e in (k, γ_2)-truss, it is unnecessary to compute $\sigma(e, k-2)$ to identify (k, γ_1)-truss. This is because the edge e will not be removed during the (k, γ_1)-truss peeling process. In this way, we can compute the trussness of all edges in (k, γ_1)-truss efficiently and avoid frequent updates.

However, we cannot know in advance the edges of (k, γ_2)-truss before computing (k, γ_1)-truss, as CPT-Basic computes edges with the smallest $\gamma_k^*(e)$ first. This creates significant challenges for efficient computation. To address this issue, we need to change the order of k in computing (k, γ)-truss. Specifically, we develop a new top-down algorithm to compute (k, γ)-truss by varying k from k_{max} to 2, which is different from CPT-Basic computing scratch from 2 to k_{max} in a bottom-up manner. For (k^+, γ_2)-truss, where $k^+ > k$, it is derived that (k^+, γ_2)-truss $\subseteq (k, \gamma_2)$-truss $\subseteq (k, \gamma_1)$-truss by Lemma 1. Hence, we make use of the obtained (k^+, γ_2)-truss for $k^+ > k$ and avoid updating $\sigma(e)$ for $e \in (k^+, \gamma_2)$-truss during the process of (k, γ_1)-truss computation.

Edge partition. In the following, we formally introduce how to leverage the obtained $(k+1, \gamma)$-truss to compute (k, γ)-truss via edge partition. A general framework is shown in Figure 2. For a given k, we denote by $\Gamma_k = \{\gamma_k^*(e) \in (0, 1] | e \in (k, \gamma)\text{-truss}\}$, which keeps all distinct values γ in (k, γ)-truss. Let be the $k^+ = k + 1$ and the set cardinality $|\Gamma_{k+1}| = x$. Without the loss of generality, $\Gamma_{k+1} = \{\gamma_1, \ldots, \gamma_x\}$ and $\gamma_1 < \gamma_2 < \ldots < \gamma_x$. Thus, we can partition all the edges e of (k, γ)-truss based on $\gamma_{k+1}^*(e)$ into $x + 1$ groups, i.e., $[\gamma_0, \gamma_1), [\gamma_1, \gamma_2), \ldots, [\gamma_x, \gamma_{x+1})$, where $\gamma_0 = 0$ and $\gamma_{x+1} = 1 + 10^{-20}$. Note that γ_{x+1} could be any number exceeding 1. Specifically, we have the following definition.

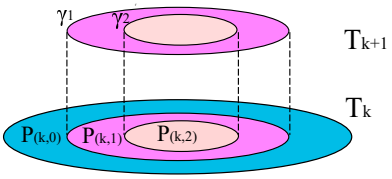


Figure 2: An example of $P(k)$ and how T_k is partitioned by Γ_{k+1}.

Definition 6 (Partition of Candidate Edges). *Given two integers $k \geq 2$ and $0 \leq i \leq |\Gamma_{k+1}| = x$, a partition of candidate edges is denoted as $P(k,i) = \langle E_k^i, \gamma_{lower}, \gamma_{upper} \rangle$, where $\gamma_{lower} = \gamma_i$, $\gamma_{upper} = \gamma_{i+1}$, and an edge set $E_k^i = \{e$ belongs to deterministic k-truss$/\ \gamma_{lower} \leq \gamma^*_{k+1}(e) < \gamma_{upper}\}$.*

$P(k,i)$ consists of three parts: an edge set E_k^i and two parameters of γ_{lower} and γ_{upper}. For each edge $e \in E_k^i$, it satisfies $\gamma_{lower} = \gamma_i \leq \gamma^*_{k+1}(e) < \gamma_{i+1} = \gamma_{upper}$. Note that, for $i = 0$, we have $E_k^0 = \{e \in E | e \in T_k \setminus T_{k+1}\}$, i.e., those edges belong to the deterministic k-truss, but do not belong to $(k+1)$-truss. Moreover, for $k = k_{max}$, we have $x = 0$ and $E_k^0 = T_{k_{max}}$. Thus, we have a total of $x+1$ partitions as $P(k) = \{P(k,0), P(k,1), \ldots, P(k,x)\}$. In general, we have x different $(k+1,\gamma)$-trusses for different valued γ in Γ_{k+1}, and then divide all candidate edges T_k into $x+1$ parts as $P(k)$.

Example 4. *Consider graph $\mathcal{G}$ in Figure 1(a). Assuming that Γ_4 has been computed, we use Γ_4 to partition the edges of T_3. For $k = 4$, we have $\Gamma_4 = \{0.46208, 0.7737809\}$ and $x = 2$. The three edges $\{(a,h),(c,h),(d,h)\}$ have the same probabilistic trussness $\gamma_4(e) = 0.46208$. The six edges $\{(a,b),(b,c),(b,d),(a,c),(a,d),(c,d)\}$ all have $\gamma_4(e) = 0.7737809$. As shown in Figure 2, the edges of T_4 have been divided into two parts with two $\gamma_4(*) = \{\gamma_1, \gamma_2\}$. Based on Γ_4, we partition the candidate edges of T_3 into three parts:* $[0, 0.46208)$, $[0.46208,\ 0.7737809)$, $[0.7737809, 1 + 10^{-20})$. *Therefore, $P(3,0) = \langle\{(d,f),(h,f),(b,g),(d,g)\}, 0, 0.46208\rangle$; $P(3,1) = \langle\{(a,h),(c,h),(d,h)\},\ 0.46208,\ 0.7737809\rangle$; $P(3,2) = \langle\{(a,b),(b,c),(b,d),(a,c),(a,d),(c,d)\},\ 0.7737809,\ 1 + 10^{-20}\rangle$.*

Hence, we have the following corollary.

Corollary 1. *Given $2 \leq k \leq k_{max} - 1$ and $0 \leq i \leq |\Gamma_{k+1}|$, each edge $e \in P(k,i)$ has the trussness $\gamma^*_k(e) \geq P(k,i).\gamma_{lower}$.*

Proof. This concludes from Lemma 1 and Def. 6. □

CPT-Fast Indexing Algorithm. Algorithm 3 outlines the CPT-Fast algorithm, which is a top-down decomposition for (k,γ)-truss indexing based on edge partitions. The algorithm first applies the truss decomposition on G and obtains k_{max} (line 2). Then, it adopts a top-down decomposition strategy of (k,γ)-truss indexing by traversing k from k_{max} to 2 (lines 3-26). For a specific $k \in [2, k_{max}]$, it partitions all edges of T_k into multiple partitions $P(k) = \{P(k,0), \ldots\}$ by Definition 6 (line 6). It starts from $i = 0$ and loads each partition $P(k,i)$ one by one into an edge set *calE* for (k,γ)-truss peeling processing (lines 8-26). For each edge $e \in P(k,i).E_k^i$, if the support vector $\sigma(e,k-1)$ can find, the vector $\sigma(e,k-2)$ can be computed by adding edges in *calE* for efficiency; otherwise, the vector needs to be computed in $\mathcal{H}$ from scratch (line 9). Next, the algorithm iteratively finds an edge $\hat{e}$ with the smallest support probability $\sigma(e,k-2)$ (line 12). If $\sigma(e,k-2) > P(k,i).\gamma_{upper}$, it needs to load a new partition $P(k,i+1)$ into *calE* for the calculation (lines 13-14); otherwise, it can skip the considerations of all edges in $P(k,j)$, where $j \geq i+1$, which saves the cost of the $\sigma(e)$ update (line 22). After obtaining $\hat{e}$, it removes all edges $\hat{e}$ disqualified for $(k, \sigma(\hat{e},k-2))$-truss from graph $\mathcal{H}$ and updates the support vector $\sigma(e)$ for candidate edges in *neiE*, where $neiE \leftarrow \{e \in calE | e \in \triangle_{uvw}, w \in N(u) \cap N(v), e \neq e^*\}$ (lines 17-24). This step is similar to CPT-Basic, but uses a smaller set *neiE* for the updating process. It keeps all edges and their probabilistic trussness in the CPT-index until $k = 2$ (line 25).

Algorithm 3 CPT-Fast

Input: $\mathcal{G} = (V, E, p)$
Output: The CPT-index of $\mathcal{G}$
1: Extract a deterministic graph $G = (V, E)$ from $\mathcal{G}$;
2: Apply truss decomposition [18] on G and get trussness k_{max};
3: **for** $k \leftarrow k_{max}$ **to** 2 **do**
4: Let $H = (V_H, E_H)$ be the k-truss of G where $E_H = T_k$;
5: Let $\mathcal{H} = (V_H, E_H, p)$ be an induced subgraph of $\mathcal{G}$ by H;
6: Partition all edges of E_H into $P(k)$ by Definition 6;
7: $calE \leftarrow \emptyset$; $i \leftarrow 0$;
8: **while** $\mathcal{H} \neq \emptyset$ **do**
9: Compute the support vector $\sigma(e)$ for $e \in P(k,i).E_k^i$ by using edges in *calE*;
10: $calE \leftarrow calE \cup P(k,i).E_k^i$;
11: **while** $calE \neq \emptyset$ **do**
12: $\hat{e} \leftarrow \arg\min_{e \in calE}\{\sigma(e,k-2)\}$;
13: **if** $\sigma(\hat{e},k-2) > P(k,i).\gamma_{upper}$ **then**
14: **break**;
15: An edge queue: $Q \leftarrow \{\hat{e}\}$; An edge set: $S \leftarrow \emptyset$;
16: $calE \leftarrow calE - \{\hat{e}\}$;
17: **while** $Q \neq \emptyset$ **do**
18: Pop out an edge $e^* = (u,v)$ from Q; $S \leftarrow S \cup \{e^*\}$;
19: A set of candidate edges to update $\sigma(e)$: $neiE \leftarrow \{e \in calE | e \in \triangle_{uvw}, w \in N(u) \cap N(v), e \neq e^*\}$;
20: Remove the edge e^* from graph $\mathcal{H}$;
21: **for** $e \in neiE$ **do**
22: update $\sigma(e)$;
23: **if** $\sigma(e,k-2) \leq \sigma(\hat{e},k-2)$ **then**
24: $Q \leftarrow Q \cup \{e\}$; $calE \leftarrow calE - \{e\}$;
25: Add $(k, e, \sigma(\hat{e},k-2))$ for $e \in S$ into CPT-index;
26: $i \leftarrow i+1$;
27: **return** $\{(k, e, \gamma^*_k(e)) \mid 2 \leq k \leq k_{max}, e \in E(\mathcal{G}), \gamma^*_k(e) > 0\}$;

Algorithm Analysis. Algorithm 3 computes the same CPT-index result as CPT-Basic in Algorithm 2. However, in contrast to Algorithm 2, Algorithm 3 has two efficiency advantages: initializing and updating $\sigma(e)$. First, Algorithm 3 initializes $\sigma(e)$ in batch for different partitions $P(k)$ in the iteratively reduced graph $\mathcal{H}$, which produces the initializations on smaller graphs much faster (lines 9-10 of Algorithm 3). Second, Algorithm 3 updates $\sigma(e)$ for a small number of edges in *neiE* (line 22 of Algorithm 3). In other words, those edges in partitions $P(k,j)$ where $j > i$ are skipped when initializing and updating $\sigma(e)$, speeding up the efficiency. As a result,

Algorithm 3 is more efficient than Algorithm 2. Furthermore, at each phase of edge removal, only the edges in a partition are considered in Algorithm 3. Thus, it takes a small cost to maintain the edges in the set *calE*, as the set has no more edges than the graph G. Let $|calE|$ be the number of edges in *calE*. A small-sized balance binary search tree is needed to maintain edge probabilistic support in $O(\mathcal{P}_{\max} \log \mathcal{P}_{\max}))$ time, where $\mathcal{P}_{\max}$ denotes the maximum size of *calE* in all iterations of Algorithm 3, i.e.,

$$\mathcal{P}_{\max} = \max_{k \in [2, k_{\max}], i \in [0, |\Gamma_k|]} |\bigcup_{j=i}^{|\Gamma_k|} P(k, j)| - |\bigcup_{j=i+1}^{|\Gamma_{k+1}|} P(k+1, j)|.$$

As a result, the time complexity of CPT-Fast in Algorithm 3 is $O(\mathcal{T}\rho(d_{\max} + \log \mathcal{P}_{\max}))$ and the space complexity is still $O(\rho m)$.

7 (ϵ, Δ_r)-APPROXIMATE INDEXING SCHEME

In this section we devise an approximate indexing scheme for (k, γ)-truss. Using the CPT-index, the retrieval of (k, γ)-truss can be completed with optimal time complexity w.r.t. the answer size. However, the CPT-index construction still takes a considerable amount of time when using large graphs. To trade off the efficiency of index construction and the online query processing of (k, γ)-truss retrieval, we propose an (ϵ, Δ_r)-approximate (k, γ)-truss indexing method to keep partial information regarding edge trussness.

7.1 (ϵ, Δ_r)-APX Index Construction

We first give two needed definitions of *floor* and *ceiling* functions.

Definition 7. *Given $\gamma, \Delta_r \in [0, 1]$, the floor function is $floor(\gamma, \Delta_r) = \lfloor \frac{\gamma}{\Delta_r} \rfloor \cdot \Delta_r$. Similarly, the ceiling function is defined as $ceil(\gamma, \Delta_r) = (\lfloor \frac{\gamma}{\Delta_r} \rfloor + 1) \cdot \Delta_r$.*

The (ϵ, Δ_r)-APX method is a slight modifications of CPT-Fast in Algorithm 3, which uses two new parameters ϵ and Δ_r to control the limits of pruning probability and approximation ratio, respectively.

Pruning probability ϵ. The approximate index keeps the *partial* edge trussnesses of (k, γ)-truss for all $k \geq 2$ and $\gamma \geq \epsilon$, where ϵ is a small value, e.g., $\epsilon = 10^{-20}$. For $k = 2$, we first remove all the edges e with $p(e) < \epsilon$ from the graph $\mathcal{G}$ before the actual index construction takes place. Thus, we keep no index information for edges with such low edge probabilities. For $k \geq 3$, we apply the truss decomposition on the probabilistic graph $\mathcal{G}$ to get (k, ϵ)-truss by [18], rather than on the deterministic graph G (line 2 of Algorithm 3). Therefore, we reduce the graph index size and improve the index construction efficiency.

Approximation ratio Δ_r. The approximate index uses a parameter $\Delta_r \in [0, 1]$ to maintain the trussness of each edge e for any $k \in [2, k_{max}]$ as an *approximate* value of $floor(\gamma_k^*(e), \Delta_r)$. In other words, we assgin an approximate trussness of $floor(\gamma_k^*(e), \Delta_r)$ for all edges e with an actual trussness $\gamma_k^*(e)$ satisfying $floor(\gamma_k^*(e), \Delta_r) \leq \gamma_k^*(e) < ceil(\gamma_k^*(e), \Delta_r)$. Specifically, we modify Algorithm 3 to remove a batch of edges e from graph $\mathcal{G}$, instead of an edge $\hat{e}$ with the smallest $\sigma(\hat{e}, k-2)$. Every removal edge e satisfies the condition of $\sigma(e, k-2) < ceil(\sigma(\hat{e}, k-2), \Delta_r)$ (lines 23-24 of Algorithm 3). Thus, we assign all removal edges with an approximate trussness of $floor(\gamma_k^*(\hat{e}), \Delta_r)$ and ensure $\gamma_k^*(e) - floor(\gamma_k^*(\hat{e}), \Delta_r) < \Delta_r$. Therefore, we avoid costly computations for exact update $\sigma(e)$ and achieve bulk deletion to reduce the removal iterations.

Table 2: $(0.1, 0.1)$-approximate index on graph $\mathcal{G}$ in Figure 1(a).

$k \backslash \gamma^*$	0.3	0.4	0.7	0.9
3	(d, f)		(a, h)	(a, b) (b, c) (b, d)
	(h, f)		(d, h)	(c, d) (a, d) (d, g)
			(c, h)	(a, c) (b, g)
4		(a, h)	(a, b) (b, c)	
		(d, h)	(a, c) (a, d)	
		(c, h)	(b, d) (c, d)	

For example, given $\Delta_r = 0.1$, the trussnesses $\gamma_k^*(e_1) = 0.11$, $\gamma_k^*(e_2) = 0.12$ and $\gamma_k^*(e_3) = 0.21$, our approximate indexing should keep the batch of edges $\{e_1, e_2\}$ together as the trussness of 0.1 due to $floor(\gamma_k^*(e_1), 0.1) = floor(\gamma_k^*(e_2), 0.1) = 0.1$, and keep the trussness of e_3 as $floor(\gamma_k^*(e_3), \Delta_r) = 0.2$. Table 2 shows the (ϵ, Δ_r)-approximate index on a uncertain graph $\mathcal{G}$ shown in Figure 1(a) where $\epsilon = 0.1$ and $\Delta_r = 0.1$. Note that the CPT-index is identical to the (ϵ, Δ_r)-approximate index for $\epsilon = 0$ and $\Delta_r = 0$. In addition, we keep the deterministic trussness for all edges in the deterministic graph G, which can be useful for query processing in case of the input query parameter $\gamma < \epsilon$ in (k, γ)-truss retrieval.

Complexity analysis. Let us denote $\mathcal{G}'$ the shrank graph after removing all edges e with $p(e) < \epsilon$ from graph $\mathcal{G}$, where the edge size of $\mathcal{G}'$ is $m' = |E(\mathcal{G}')| \leq m$. We also let ρ' and d'_{max} denote respectively the arboricity and the maximum degree in the deterministic graph of $\mathcal{G}'$. Moreover, $\mathcal{T}' = \sum_{k=2}^{k_{max}} |T'_k|$, where T'_k is the k-truss in deterministic graph of $\mathcal{G}'$.

Theorem 3. *Building (ϵ, Δ_r)-APX index takes $O(\mathcal{T}'\rho' d'_{\max})$ time and $O(\rho' m')$ space, where $\mathcal{T}' \leq \mathcal{T}$, $\rho' \leq \rho$, and $d'_{\max} \leq d_{\max}$.*

Proof. Recall that, in the CPT-Basic and CPT-Fast algorithms, we create a balanced binary search tree to maintain the edge with the smallest probabilistic edge support, as $\sigma(e, k-2)$ may be any float value. However, in the (ϵ, Δ_r)-APX algorithm, the value of $\sigma(e, k-2)$ is rounded as $floor(\sigma(e, k-2), \Delta_r)$. This implies that there exist finite values for probabilistic edge supports and, at most, $\lceil 1/\Delta_r \rceil$ distinct bins to handle a batch of edges. Thus, we can implement a sort bin algorithm to maintain the edges with the smallest amount of probabilistic edge support in $O(1)$ time. The total time complexity of (ϵ, Δ_r)-APX is $O(\mathcal{T}'\rho' d'_{max})$ using $O(\rho' m')$ space. □

7.2 (ϵ, Δ_r)-APX Index based (k, γ)-truss Retrieval

The algorithm involved in finding (k, γ)-truss based on the (ϵ, Δ_r)-approximate index is outlined in Algorithm 4. Using Lemma 1, Algorithm 4 exploits the pruning strategy to efficiently extract (k, γ)-truss for the given k and γ. Specifically, if $k_1 \geq k_2 \geq k_3$ and $\gamma_1 \geq \gamma_2 \geq \gamma_3$, (k_1, γ_1)-truss $\subseteq (k_2, \gamma_2)$-truss $\subseteq (k_3, \gamma_3)$-truss holds, according to Lemma 1. To obtain (k_2, γ_2)-truss, we only need to check a small part of the edges in $\{e | e \in (k_3, \gamma_3)$-truss, $e \notin (k_1, \gamma_1)$-truss$\}$ and compute trussness in order to query (k, γ)-truss. Thus, Algorithm 4 first extracts two subgraphs, $\mathcal{H}_u$ and $\mathcal{H}_l$, from the

Algorithm 4 (ϵ, Δ_r)-Approximate Index based (k, γ)-Truss Retrieval

Input: (ϵ, Δ_r)-approximate index; k; γ;
Output: (k, γ)-truss;

1: $\mathcal{H}_u \leftarrow$ Extract the (k, γ_u)-truss from (ϵ, Δ_r)-approximate index where $\gamma_u = ceil(\gamma, \Delta_r)$;
2: $\mathcal{H}_l \leftarrow$ Extract the (k, γ_l)-truss from (ϵ, Δ_r)-approximate index where $\gamma_l = floor(\gamma, \Delta_r)$;
3: $calE \leftarrow \{e | e \in \mathcal{H}_l, e \notin \mathcal{H}_u\}$;
4: Compute $\sigma_{\mathcal{H}_l}(e)$ for each edge $e \in calE$;
5: **while** $\exists e^* \in calE$ with $\sigma_{\mathcal{H}_l}(e^*, k-2) < \gamma$ **do**
6: Remove e^* from $\mathcal{H}_l$ and $calE$;
7: Update $\sigma_{\mathcal{H}_l}(e)$ for all necessary edges $e \in calE$;
8: **return** $\mathcal{H}_l$;

(ϵ, Δ_r)-approximate index (lines 1-2). $\mathcal{H}_u$ is the $(k, ceil(\gamma, \Delta_r))$-truss from the (ϵ, Δ_r)-approximate index; $\mathcal{H}_l$ is the $(k, floor(\gamma, \Delta_r))$-truss from (ϵ, Δ_r)-approximate index. It generates a set of (k, γ)-truss candidate edges as $calE \leftarrow \{e | e \in \mathcal{H}_l, e \notin \mathcal{H}_u\}$ (line 3). Then, it iteratively removes the disqualified edges from $\mathcal{H}_l$ using the definition of (k, γ)-truss (lines 5-7) and finally returns $\mathcal{H}_l$ as the answer (line 8). For example, consider the (0.1, 0.1)-approximate index in Table 2. For a (4, 0.5)-truss retrieval, the edges $\{(a, b), (b, c), (a, c), (a, d), (c, d), (b, d)\}$ that locate in (4, 0.7)-truss, will be directly obtained in the answer by avoding computations. The edges $\{(a, h), (c, h), (d, h)\}$ need to be checked for a qualified (4, 0.5)-truss.

Our (ϵ, Δ_r)-approximate index based (k, γ)-truss retrieval can achieve exact results for any given k and γ. This is because we follow the peeling strategy to remove edges with the smallest trussness and update the support vectors of the remaining edges, ensuring exact computation of all edges' trussnesses. The fast efficiency of (k, γ)-truss retrieval leverages (ϵ, Δ_r)-approximate index to obtain small-sized candidate graphs for edge removals.

Discussions. In summary, the approximate indexing scheme improves the efficiency of index construction. First of all, it ignores all the edges e with $\sigma(e, k-2) < \epsilon$, leading to a smaller graph $\mathcal{G}$ and a smaller corresponding k_{max}. Note that, if an input parameter $\gamma < \epsilon$, Algorithm 4 uses T_k as $\mathcal{H}_l$. T_k is often much larger than $\mathcal{H}_u$; so, in this case, the algorithm cannot significantly improve the efficiency. Fortunately, in real applications, we usually are not interested in the discovery of any (k, γ)-truss with a very small probability. Thus, it is possible to choose a suitable value of ϵ in order to speed up index construction. Second, we use the parameter Δ_r to index edges in a batch. A larger value of γ_i leads to a shorter index construction time but a longer online (k, γ)-truss retrieval time. Therefore, the approximate indexing scheme achieves a good balance by trading off between index construction and online query processing. As a result, (ϵ, Δ_r)-approximate indexing is more practical for real-world large graph datasets than other methods (CPT-Basic and CPT-Fast). On the other hand, CPT-Fast builds up the exact CPT-index and optimally deals with (k, γ)-truss retrieval, which is expected to outperform the other two methods on moderate and small graph datasets.

Table 3: Network statistics

Network	$\|V_\mathcal{G}\|$	$\|E_\mathcal{G}\|$	d_{max}	k_{max}
Fruit-Fly	3,751	3,692	27	5
ca-GrQc	5,242	14,496	81	44
wiki_vote	7,118	103,689	1,065	23
Flickr	24,125	300,836	546	218
DBLP	684,911	2,284,991	611	115
biomine	1,008,200	6,742,939	139,624	439
LiveJournal	4,847,571	42,851,237	20,333	352
Orkut	3,072,441	117,185,083	33,313	73
wise	58,655,849	261,321,033	278,489	80

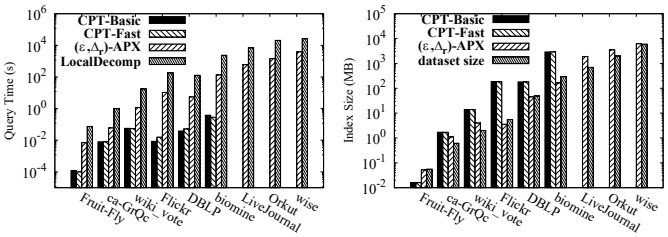


Figure 3: Performance of all algorithms for all datasets, in terms of query time (left), measured as average over 100 queries, and index size (right).

8 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the performance of our proposed solutions.

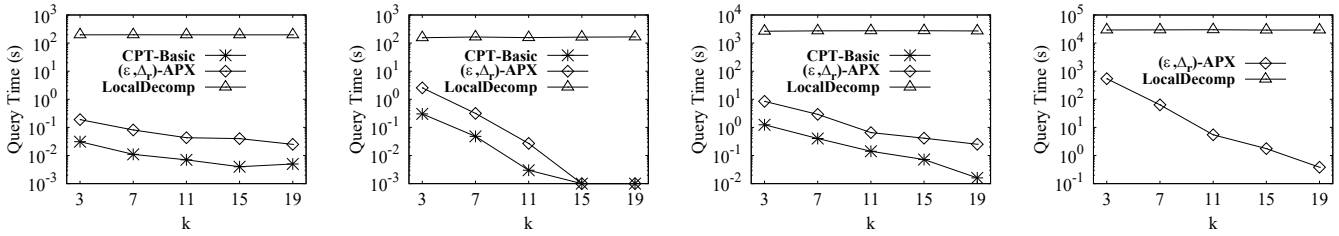
Datasets. We use nine real-world probabilistic graphs in the experiments. Table 3 reports the network statistics. Fruit-Fly is a protein-protein interaction (PPI) network [27] from the BioGRID database [1] and STRING database [31]. Flickr is an online photo sharing community network, in which vertices represent users and an edge probability represents the Jaccard coefficient of two users' interest groups [7, 31]. DBLP is a probabilistic collaboration network [2]. Biomine is a biological interaction network [12], in which the probability of an edge represents the confidence that the interaction actually exists [7, 31]. In the other five datasets, wiki_vote, ca-GrQc, LiveJournal, Orkut, and wise [24], the edge probabilities are uniformly assigned a random value from the interval [0, 1].

Comparison Methods. We compare different algorithms for index construction and (k, γ)-truss retrieval query processing, as follows. Our code is available at https://github.com/jinrdfh/P-truss.

- LocalDecomp: An online (k, γ)-truss search in Algorithm 1 [18], which finds (k, γ)-truss without any index.
- CPT-Basic: The basic (k, γ)-truss indexing scheme for exact CPT-index construction using Algorithm 2. The CPT-index-based (k, γ)-truss retrieval provides optimal answers.
- CPT-Fast: A fast (k, γ)-truss indexing scheme for exact CPT-index construction using Algorithm 3. CPT-Fast uses the same (k, γ)-truss retrieval method as CPT-Basic.
- (ϵ, Δ_r)-APX: An (ϵ, Δ_r)-approximate indexing scheme in Section 7, trades off the efficiency of index construction and query processing and achieves exact query answers.

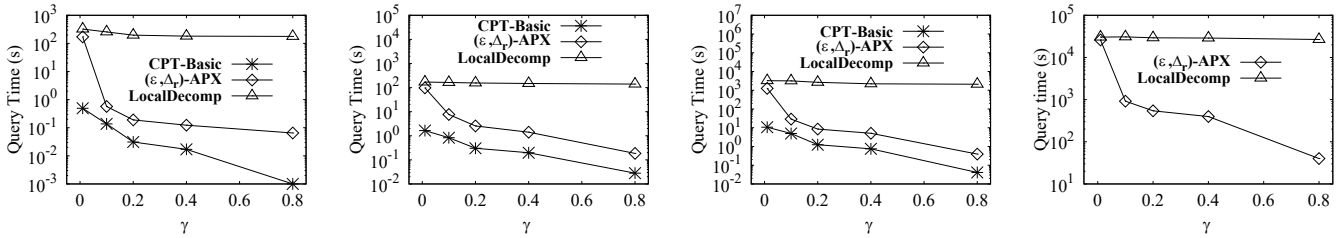
Table 4: Index construction time (in seconds) of three indexing algorithms. '-' represents that it exceeds the time limit.

	Fruit-Fly	ca-GrQc	wiki_vote	Flickr	DBLP	biomine	LiveJournal	Orkut	wise
CPT-Basic	4.7×10^{-2}	15	2.2×10^{2}	2.5×10^{4}	1.8×10^{3}	6.9×10^{5}	-	-	-
CPT-Fast	$\mathbf{2.7 \times 10^{-2}}$	**6.9**	87	1.7×10^{3}	4.9×10^{2}	5.6×10^{4}	-	-	-
(ϵ, Δ_r)-APX	9.1×10^{-2}	7.8	**53**	**72**	$\mathbf{2.3 \times 10^{2}}$	$\mathbf{1.2 \times 10^{4}}$	$\mathbf{5.0 \times 10^{4}}$	$\mathbf{7.7 \times 10^{4}}$	$\mathbf{9.9 \times 10^{4}}$



(a) Flickr (b) DBLP (c) biomine (d) wise

Figure 4: Parameter sensitivity evaluation of all query processing algorithms by varying k.



(a) Flickr (b) DBLP (c) biomine (d) wise

Figure 5: Parameter sensitivity evaluation of all query processing algorithms by varying γ.

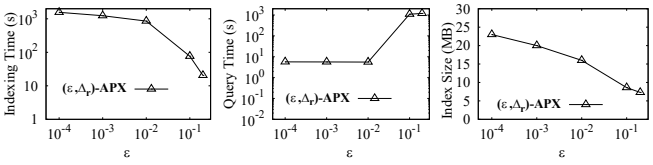
Evaluation Metrics and Parameters. We compare two tasks regarding index construction and query processing. For index construction, we use two metrics of the index construction time (in seconds) and the index size (in Megabytes). For (k, γ)-truss query processing, we input two parameters k, γ and evaluate the efficiency using query time (in seconds). For each query, we set the parameters $k \in [3, 40]$ and $\gamma \in (0, 1]$. We randomly test 100 sets of queries and report the average running time. We treat the running time of an index construction and a query as infinite if it cannot finish within 200 hours, which is marked by '-'. For (ϵ, Δ_r)-APX, we set the parameters $\epsilon = 0.1$ and $\Delta_r = 0.001$ by default.

8.1 Efficiency Evaluation

Exp-I: Index Construction Time. We compare the efficiency of three index construction algorithms: CPT-Basic, CPT-Fast and (ϵ, Δ_r)-APX. Table 4 reports the index construction time of the different algorithms for all datasets. (ϵ, Δ_r)-APX outperforms CPT-Fast and CPT-Basic on most datasets, except for the small datasets Fruit-Fly and ca-GrQc. Neither CPT-Fast nor CPT-Basic can finish the index construction within 200 hours on the large graphs LiveJournal, Orkut, and wise. CPT-Fast is faster than CPT-Basic on all other graphs, which validates the effectiveness of edge partitioning and delays updates for $\sigma(e)$ in the top-down framework of CPT-Fast.

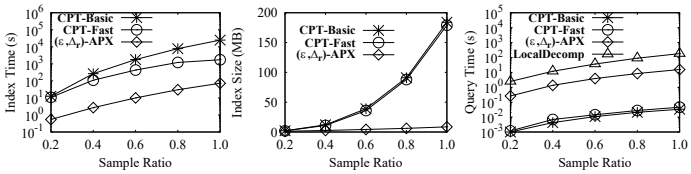
Exp-II: Query Time. We conduct the query efficiency evaluation of (k, γ)-truss retrieval for all datasets. We compare the query time of four algorithms: one online search algorithm (LocalDecomp) and three index-based (k, γ)-truss retrieval algorithms (CPT-Basic, CPT-Fast and (ϵ, Δ_r)-APX). Figure 3(left) reports the query time of all the methods. We can observe that LocalDecomp performs the worst for all datasets, due to the way in which the online computation does not have any index. Both CPT-Basic and CPT-Fast are orders of magnitude faster than LocalDecomp for all datasets with a successfully constructed index, except for large graphs. CPT-Basic and CPT-Fast have the same query time, as they are based on the same exact CPT-index. Moreover, (ϵ, Δ_r)-APX is faster than LocalDecomp and slower than CPT-Basic and CPT-Fast, as (ϵ, Δ_r)-APX uses an (ϵ, Δ_r)-approximate index. On the other hand, while CPT-Basic and CPT-Fast cannot complete index construction within 200 hours, (ϵ, Δ_r)-APX reduces the query time of LocalDecomp from 2.0-7.3 hours to 10-65 minutes on the three large graphs.

Exp-III: Index Size. Figure 3(right) reports the index sizes of different index construction algorithms for all datasets. The disk sizes of the original graphs are also reported for comparison. The CPT-index constructed by CPT-Basic and CPT-Fast take more space than the graph size and (ϵ, Δ_r)-APX, except for the small graph Fruit-Fly. In Fruit-Fly, many edges have the same probability, so the CPT-index can compress a great deal of space; moreover, (ϵ, Δ_r)-APX keeps the trussness index of the deterministic graph G and takes more index space than CPT-Basic and CPT-Fast. In regard to the other datasets, the (ϵ, Δ_r)-approximate index occupies the smallest space, which is comparable with the graph sizes. This is because (ϵ, Δ_r)-APX prunes those low probability edges $p(e) < \epsilon$ and merges many edges with the same probabilities.



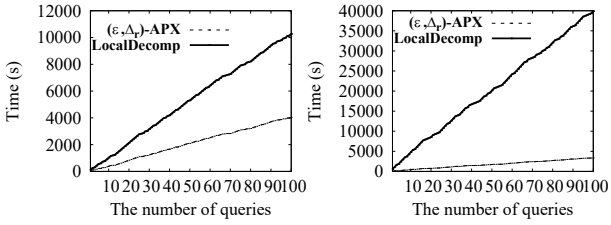
(a) Index Time (b) Query Time (c) Index Size

Figure 6: Parameter sensitivity evaluation of (ϵ, Δ_r)-APX by varying ϵ on Flickr dataset.



(a) Index Construction Time (b) Index Size (c) Query Time

Figure 7: Scalability of different algorithms on Flickr dataset in terms of index construction time, index size, and query time.



(a) Maximizing k (b) Maximizing γ

Figure 8: The cumulative query time of task-driven team formation. Here, $\gamma = 10^{-11}$ and $k = 3$ by default.

8.2 Parameter Sensitivity Evaluation

For query processing, we compare three algorithms: LocalDecomp, CPT-Basic, and (ϵ, Δ_r)-APX, as CPT-Basic and CPT-Fast achieve the same performance based on the same index.

Exp-IV: Varying Query Parameter k. We evaluate the performance of different query processing algorithms by varying the parameter k. We vary k from 3 to 19 and set $\gamma = 0.2$. Figure 4 reports the query time results of LocalDecomp, CPT-Basic, and (ϵ, Δ_r)-APX for four large datasets: Flickr, DBLP, biomine, and wise. The query time of the CPT-Basic and (ϵ, Δ_r)-APX algorithms decreases with the increased k. This is because the (k, γ)-truss has a denser and smaller graph size with a larger k, which can be retrieved faster. Note that CPT-Basic cannot answer the (k, γ)-truss query as it cannot finish index construction within a given time for wise, as shown in Figure 4(d). In addition, LocalDecomp has a stable query processing performance but runs much slower than CPT-Basic and (ϵ, Δ_r)-APX.

Exp-V: Varying Query Parameter γ. We further evaluate the performance of LocalDecomp, CPT-Basic, and (ϵ, Δ_r)-APX by varying the parameter γ. We vary the parameter γ from 0 to 1 and set $k = 3$. Figure 5 shows the query time results of all methods for four large datasets: Flickr, DBLP, biomine, and wise. The query time of CPT-Basic and (ϵ, Δ_r)-APX significantly decreases with the increased γ. (ϵ, Δ_r)-APX is faster than LocalDecomp and slower than CPT-Basic, which makes use of partial indexing information in its online search of (k, γ)-truss in an integrated way. Specifically, for $\gamma < 0.1$, (ϵ, Δ_r)-APX takes a much longer query time to compute (k, γ)-truss from T_k when $\gamma < 0.1$ with the indexing setting of $\epsilon = 0.1$. On the other hand, when $\gamma \geq 0.1$, (ϵ, Δ_r)-APX makes use of the (ϵ, Δ_r)-approximate index and obtains a small edge set $calE$ for quick (k, γ)-truss identification, using less time. Once again, LocalDecomp has the worst performance, and CPT-Basic is the best but cannot finish query processing on the largest graph wise, as shown in Figure 5(d).

Exp-VI: Varying Pruning Probability ϵ. We evaluate the parameter sensitivity of (ϵ, Δ_r)-APX by varying $\epsilon \in [10^{-4}, 0.2]$. Figure 6 reports the index construction time, query time, and index size of (ϵ, Δ_r)-APX in regard to Flickr. With a larger pruning probability ϵ, (ϵ, Δ_r)-APX has less costs regarding index construction, as shown in Figure 6(a), and generates a smaller CPT-index in Figure 6(c); meanwhile, the query time of (ϵ, Δ_r)-APX is greater, as shown in Figure 6(b). This is because we generate the queries by randomly picking the parameter $\gamma \in [0, 1]$, and (ϵ, Δ_r)-APX takes a longer running time to obtain (k, γ)-truss for $\gamma < \epsilon$.

Exp-VII: Scalability Test. We conduct a scalability test on Flickr. Similar trend results can be found as for the other datasets. To test the scalability of the proposed algorithms, we vary the graph size by randomly sampling edges in the Flickr graph with 20%, 40%, 60%, 80%, and 100%. Figures 7(a)-(c) report the results regarding index construction time, index size, and query time for all methods used. In Figure 7(a), the index construction time of CPT-Basic increases faster than the construction time of CPT-Fast, indicating good scalability in regard to CPT-Fast on index construction. (ϵ, Δ_r)-APX performs best across all datasets. The index size of (ϵ, Δ_r)-APX also increases slowly, as shown in Figure 7(b). Last, Figure 7(c) shows the query time of all the methods used. In Figure 7(c), LocalDecomp again performs the worst among all methods. (ϵ, Δ_r)-APX is slower than CPT-Basic and CPT-Fast and increases slightly as the sampling ratio increases, indicating the stable scalability of the (ϵ, Δ_r)-APX scheme in regard to (k, γ)-truss retrieval.

8.3 Application: Task-Driven Team Formation

In this experiment, we apply our (k, γ)-truss indexing to solve the task-driven team formation problem [7, 18]. Given a probabilistic collaboration graph $\mathcal{G} = (V, E, p)$ derived specifically for a topic task T, a set of query vertices $Q \subseteq V$, the problem is finding a connected subgraph of (k, γ)-truss H such that (i) H contains all nodes Q and (ii) H has the largest trussness [18]. We use the same dataset of DBLP collaboration network and the same topics of 'data' and 'algorithm' [18], where the edge probability represents the collaboration strength of papers between two authors related to the topics by LAD model [5]. We evaluate the strength of probabilistic trussness in two dimensions of k and γ. Specifically, one is maximizing the k w.r.t. a given γ, and the other one is maximizing the γ w.r.t. a given k. We randomly generate 100 sets of query nodes. Figure 8(b) shows the cumulative query time of LocalDecomp [18] and our (ϵ, Δ_r)-APX approach to find (k, γ)-truss team by maximizing k, w.r.t. a given $\gamma = 10^{-11}$. Both approaches find the same (k, γ)-truss

Table 5: The efficiency and quality results of probabilistic triangle densest subgraph discovery.

	Expected Triangle Density			Running Time (in seconds)		
	PTDS	Ours	loss (%)	PTDS	Ours	speedup
Fruit-Fly	1.87	1.84	**1.4**	0.05	0.001	**50x**
ca-GrQc	91.2	87.0	**4.7**	63	0.23	**277x**
Flickr	376	345	**8.1**	15,234	0.29	**52,080x**

results. (ϵ, Δ_r)-APX is faster than LocalDecomp, which achieves significantly incremental gains with more queries issued. In addition, Figure 8(a) shows the cumulative query time of LocalDecomp [18] and (ϵ, Δ_r)-APX by maximizing γ, w.r.t. a given $k = 3$. LocalDecomp runs much slower than (ϵ, Δ_r)-APX. This because that it needs to call LocalDecomp multiple times to find a feasible $(k, \bar{\gamma})$-truss, which uses a binary search over $\bar{\gamma} \in [0, 1]$ such that $|\bar{\gamma} - \gamma^*| \leq 0.1$ where γ^* is the largest trussness. On the other hand, our (ϵ, Δ_r)-APX approach runs once to achieve results for each team formation query.

8.4 Application: Probabilistic Triangle Densest Subgraph Discovery

We apply our (ϵ, Δ_r)-APX algorithm for a novel application of probabilistic triangle densest subgraph discovery. The problem of triangle densest subgraph discovery over a deterministic graph is formulated and studied in [35], while its problem over uncertain graphs has not been studied yet, to the best of our knowledge. Given a subgraph $\mathcal{H} \subseteq \mathcal{G}$, we define the expected triangles of an edge e in $\mathcal{H}$ is the expected number of triangles containing e based on its support vector, i.e., $\bar{\alpha}(e) = \sum_{i=0}^{t_e} i \cdot \sigma_{\mathcal{H}}(e, i)$. Thus, the expected triangle density of $\mathcal{H}$ is denoted as $\text{TDensity}(\mathcal{H}) = \frac{\sum_{e \in E(\mathcal{H})} \bar{\alpha}(e)}{3|V(\mathcal{H})|}$. The problem of probabilistic triangle densest subgraph is to find an induced subgraph $\mathcal{H}_S$ of $\mathcal{G}$ by a subset of vertices S with the largest expected triangle density, i.e., $S^* = \arg\max_{S \subseteq V} \text{TDensity}(\mathcal{H}_S)$. We implement a PTDS method by extending the $\frac{1}{3}$-approximation algorithm [35] from deterministic graphs to uncertain graphs. We compare two methods: one baseline is to apply PTDS on the whole graph $\mathcal{G}$ directly; the other one is to first extract (k, γ)-truss from $\mathcal{G}$ by (ϵ, Δ_r)-APX and then apply PTDS on a small (k, γ)-truss graph, denoted as (ϵ, Δ_r)-APX. Table 5 reports the running times and expected triangle density of PTDS and (ϵ, Δ_r)-APX on three datasets. Our (ϵ, Δ_r)-APX method obtains competitive triangle densities compared with PTDS, which falls in a small density loss of 8.1% but achieves a significant speedup of of 52,080**x** on Flickr.

9 CONCLUSIONS

This paper studies the problem of (k, γ)-truss indexing and querying on uncertain graphs. We propose a compact data structure of the CPT-index to keep the complete information of (k, γ)-truss for all k and γ. CPT-index-based querying algorithms can perform (k, γ)-truss retrieval with an optimal level time complexity for any given k and γ. To efficiently construct the CPT-index, we propose two index construction algorithms, CPT-Basic and CPT-Fast, in the bottom-up search and top-down partition manners. Based on CPT-Fast, we further develop (ϵ, Δ_r)-APX scheme to trade-off the index construction and online retrieval processes. Experiments on large datasets verify the superiority of our methods over state-of-the-art approaches.

ACKNOWLEDGMENTS

The work is supported by HK RGC Grants Nos. 22200320, 12200819, CRF C6030-18G, and Guangdong Basic and Applied Basic Research Foundation (No. 2019B1515130001). Xin Huang is the corresponding author.

REFERENCES

[1] http://thebiogrid.org.
[2] http://dblp.uni-trier.de.
[3] Suman Banerjee and Bithika Pal. 2020. DySky: Dynamic Skyline Queries on Uncertain Graphs. *CoRR* abs/2004.02564 (2020).
[4] M. Blanco, T. M. Low, and K. Kim. 2019. Exploration of fine-grained parallelism for load balancing eager k-truss on gpu and cpu. In *HPEC*. 1–7.
[5] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of Machine Learning Research* 3 (2003), 993–1022.
[6] Paolo Boldi, Francesco Bonchi, Aristides Gionis, and Tamir Tassa. 2012. Injecting Uncertainty in Graphs for Identity Obfuscation. *PVLDB* 5, 11 (2012), 1376–1387.
[7] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *KDD*. 1316–1325.
[8] Y. Che, Z. Lai, S. Sun, Y. Wang, and Q. Luo. 2020. Accelerating Truss Decomposition on Heterogeneous Processors. *PVLDB* 13, 10 (2020), 1751–1764.
[9] Lei Chen and Xiang Lian. 2012. Query processing over uncertain databases. *Synthesis Lectures on Data Management* 4, 6 (2012), 1–101.
[10] Pei-Ling Chen, C.K. Chou, and Ming-Syan Chen. 2014. Distributed algorithms for k-truss decomposition. In *International Conference on Big Data*. 471–480.
[11] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.
[12] Lauri E. and Hannu T. 2012. Biomine: predicting links between biological entities using network models of heterogeneous databases. *BMC Bioinformatics* 13 (2012).
[13] Soroush Ebadian and Xin Huang. 2019. Fast algorithm for K-truss discovery on public-private graphs. In *IJCAI*. 2258–2264.
[14] Fatemeh Esfahani, Jian Wu, V. Srinivasan, A. Thomo, and K. Wu. 2019. Fast Truss Decomposition in Large-scale Probabilistic Graphs. In *EDBT*. 722–725.
[15] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.
[16] Xin Huang and Laks VS Lakshmanan. 2017. Attribute-driven community search. *PVLDB* 10, 9 (2017), 949–960.
[17] Xin Huang, Laks VS Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *PVLDB* 9, 4 (2015).
[18] Xin Huang, Wei Lu, and Laks V. S. Lakshmanan. 2016. Truss Decomposition of Probabilistic Graphs: Semantics and Algorithms. In *SIGMOD*. 77–90.
[19] Ruoming Jin, Lin Liu, Bolin Ding, and Haixun Wang. 2011. Distance-constraint reachability computation in uncertain graphs. *PVLDB* 4, 9 (2011), 551–562.
[20] H. Kabir and K. Madduri. 2017. Shared-Memory Graph Truss Decomposition. In *HiPC*. 13–22.
[21] V. Kassiano, A. Gounaris, A. Papadopoulos, and K. Tsichlas. 2016. Mining uncertain graphs: An overview. In *Algorithmic Aspects of Cloud Computing*. 87–116.
[22] Arijit Khan, Francesco Bonchi, Aristides Gionis, and Francesco Gullo. 2014. Fast Reliability Search in Uncertain Graphs. In *EDBT*. 535–546.
[23] Arijit Khan, Francesco Bonchi, Francesco Gullo, and Andreas Nufer. 2018. Conditional reliability in uncertain graphs. *TKDE* 30, 11 (2018), 2078–2092.
[24] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[25] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based Community Search over Large Directed Graphs. In *SIGMOD*. 2183–2197.
[26] Robert J Mokken. 1979. Cliques, clubs and clans. *Quality & Quantity* 13, 2 (1979), 161–173.
[27] Arko Provo Mukherjee, Pan Xu, and Srikanta Tirthapura. 2013. Mining Maximal Cliques from an Uncertain Graph. *arXiv preprint arXiv:1310.6780* (2013).
[28] P. Parchas, F. Gullo, D. Papadias, and F. Bonchi. 2014. The pursuit of a good possible world: extracting representative instances of uncertain graphs. In *SIGMOD*. 967–978.
[29] Panos Parchas, Nikolaos Papailiou, Dimitris Papadias, and Francesco Bonchi. 2018. Uncertain graph sparsification. *TKDE* 30, 12 (2018), 2435–2449.
[30] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. Mining Cross-Graph Quasi-Cliques in Gene Expression and Protein Interaction Data. In *ICDE*. 353–354.
[31] Michalis Potamias, Francesco Bonchi, Aristides Gionis, and George Kollios. 2010. K-nearest neighbors in uncertain graphs. *PVLDB* 3, 1-2 (2010), 997–1008.
[32] G. Preti, G. De Francisci Morales, and F. Bonchi. 2021. STruD: Truss Decomposition of Simplicial Complexes. In *WWW*.
[33] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.L. Wu, and Ü. V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *PVLDB* 6, 6 (2013), 433–444.
[34] Taro Takaguchi and Yuichi Yoshida. 2016. Cycle and flow trusses in directed networks. *Royal Society Open Science* 3 (2016).

[35] Charalampos Tsourakakis. 2015. The k-clique densest subgraph problem. In *WWW*. 1122–1132.
[36] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *PVLDB* 5, 9 (2012), 812–823.
[37] B. Yang, D. Wen, L. Qin, Y. Zhang, L. Chang, and R. Li. 2019. Index-Based Optimal Algorithm for Computing K-Cores in Large Uncertain Graphs. In *ICDE*. 64–75.
[38] Ye Yuan, Guoren Wang, Lei Chen, and Haixun Wang. 2013. Efficient keyword search on uncertain graph data. *TKDE* 25, 12 (2013), 2767–2779.
[39] Ye Yuan, Guoren Wang, Haixun Wang, and Lei Chen. 2011. Efficient subgraph search over large uncertain graphs. *PVLDB* 4, 11 (2011), 876–886.
[40] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and Efficiency of Truss Maintenance in Evolving Graphs. In *SIGMOD*. 1024–1041.
[41] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. 2010. Finding top-k maximal cliques in an uncertain graph. In *ICDE*. 649–652.
[42] Zhaonian Zou and Rong Zhu. 2017. Truss decomposition of uncertain graphs. *Knowledge and Information Systems* 50, 1 (2017), 197–230.