Scalable Evaluation of $k$-NN Queries on Large Uncertain Graphs

Xiaodong Li¹, Reynold Cheng¹, Yixiang Fang¹, Jiafeng Hu¹, Silviu Maniu²
¹The University of Hong Kong, Hong Kong, ²Université Paris-Sud, France
{xdl,ckcheng,yxfang,jhu}@cs.hku.hk, silviu.maniu@lri.fr

ABSTRACT

Large graphs are prevalent in social networks, traffic networks, and biology. These graphs are often uncertain. For example, in a friendship network, an edge between two nodes $u$ and $v$ indicates that users $u$ and $v$ have a close relationship. This edge may only exist with a probability. To model such information, the uncertain graph model has been proposed, in which each edge $e$ is augmented with a probability that indicate the chance that $e$ exists. Given a node $q$ in an uncertain graph $G$, we study the $k$-NN query of $q$, which looks for $k$ nodes in $G$ whose distances from $q$ are the shortest. The $k$-NN query can be used in friend-search, data mining, and pattern-recognition. Despite the importance of this query, it has not been well studied. In this paper, we develop a tree-based structure called the $U$-tree. Given a $k$-NN query, the $U$-tree produces a compact representation of $G$, based on which the query can be executed efficiently. Our results on real and synthetic datasets show that our algorithm can scale to large graphs, and is 75% faster than the state-of-the-art solutions.

KEYWORDS

uncertain graph, tree indexing, $k$-NN query

1 INTRODUCTION

Graphs are prevalent in social networks [10, 12], traffic networks [11], biological networks [31], and mobile ad-hoc networks [13]. Due to noisy measurements [1], hardware limitation [2], inference models [9], and privacy-preserving perturbation [4, 21], these graphs are inherently uncertain. To model this error, the uncertain graph has been studied [1, 6, 15, 18]. In this graph, each edge is associated with a probability distribution. Figure 1(a) shows the protein-protein interaction (PPI) network as an uncertain graph, where each node denotes a protein, and each edge is an interaction between a pair of proteins. The value on each edge denotes the probability that the interaction exists (called existence probability). For instance, the existential probability that the interaction between nodes $A$ and $B$ exists is 0.7.

$k$-NN query. In this paper, we study the efficient evaluation of $k$-nearest neighbor ($k$-NN) queries [1] on uncertain graphs. Given a node $q$, the $k$-NN query returns $k$ nodes with the shortest "distances" from $q$, based on a distance function proposed in the literature (Table 1). The $k$-NN query can help biologists to perform tasks such as protein complex detection [29], link discovery [31], and protein function prediction [22]. Security experts can also use $k$-NN queries to design privacy-preserving algorithms to protect nodes in a graph from being identified [4].

Although $k$-NN queries are useful, the issues of evaluating them efficiently have only been briefly touched (e.g., [18]). Our experiments found that they are also not very efficient on large uncertain graphs. The major reason is that for correct query execution, queries running on the uncertain graph should follow the Possible World Semantics (or PWS in short). Figure 1(b) shows a possible world instance sampled from $G$ in Figure 1(a). A simple way to evaluate the query is to get the answer from each possible world and then collect all these answers to form the final answer. For example, given a $k$-NN query of computing the $k$ nearest neighbors from node $A$ in Figure 1(a), we obtain $2^3$ possible worlds, then for each of them, we compute the $k$ nearest neighbors, and finally obtain the $k$ nodes which are the closest to $A$ considering all the possible worlds (according to a function in Table 1). Because an uncertain graph has an exponential number of possible worlds, a naive solution can be extremely inefficient. Although existing solutions try to avoid enumerating all the possible worlds, they are still expensive.

The $U$-tree framework. In this paper, we develop a new indexing framework for $k$-NN requests, which (i) allows efficient and scalable $k$-NN query evaluation under the PWS, and (ii) can easily adapt to different distance functions (e.g., Table 1). As shown in Figure 2, our framework consists of two stages:

- **Offline index construction** (Figure 2(a)). Given an uncertain graph $G$ and a distance function $d$, we first employ decomposition techniques (Step A), which converts $G$ into a succinct index structure, whose edges are encoded with the probability information of $G$ [5, 14, 16, 20, 30, 32]. These techniques were not designed for $k$-NN queries. Hence, we perform customization of the index (Step B), which adjusts the information and structure of the index, in order to enhance the performance of $k$-NN queries.

![Figure 1](image1.png)

Figure 1: (a) an uncertain graph $G$ and (b) a possible world from $G$ with existence probability $0.3 \times 0.9 \times 0.6 \times 0.8 = 0.1296$.

<table>
<thead>
<tr>
<th>Function</th>
<th>Detailed formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most-probable [31]</td>
<td>$d_{mp}(s, t) = \arg \max \prod_{u \in \text{PATH}(s, t)} p(u)$</td>
</tr>
<tr>
<td>Reliability [34]</td>
<td>$d_{r}(s, t) = \sum_{d \in \text{child}(t)} P_{s, t}(d)$</td>
</tr>
<tr>
<td>Median [28]</td>
<td>$d_{me}(s, t) = \arg \max \left{ \sum_{d \in \text{child}(t)} P_{s, t}(d) \leq \frac{t}{k} \right}$</td>
</tr>
<tr>
<td>Majority [28]</td>
<td>$d_{ma}(s, t) = \arg \max_{</td>
</tr>
<tr>
<td>Expected [31]</td>
<td>$d_{e}(s, t) = \sum_{d \in \text{child}(t)} \frac{P_{s, t}(d)}{</td>
</tr>
<tr>
<td>Expected-reliable [34]</td>
<td>$d_{er}(s, t) = \sum_{d \in \text{child}(t)} \frac{P_{s, t}(d)}{P_{s, t}(d)}$</td>
</tr>
</tbody>
</table>
We first review the existing queries on uncertain graphs and then world semantics, which assumes that an uncertain graph has an

tures, which can significantly speed up the query process without

loss. (3) The last method is to build some elegant index struc-
tures, or only sample part of

the uncertain graph by removing some weak nodes and edges, or only sample part of

the uncertain graph in Step A. As we will point out later, there is a trade-

off between the U-tree’s size and the query cost. Later, we will
discuss a cost model that allows us to balance between these two

factors, in order to improve query efficiency without significantly

increasing the U-tree size.

Experiments. To evaluate the performance of the proposed
methods, we conduct experiments on both real and synthetic
datasets. We have also tested different decomposition methods
and distance functions in the U-tree framework. The results show
that U-tree is superior to the state-of-the-art algorithms, and the
query time is significantly reduced by 75%; the overhead of U-
tree is only 23% more than the index generated in Step A. Our
solution is also scalable to large uncertain graphs with over 1
million of nodes.

Organization. The rest of paper is organized as follows. We
review the related works in Section 2. In Section 3, we present
the formal definition of the problems, and discuss some basic
techniques. We present the U-tree indexing framework in Section
4. In Section 5, we present our experiment results. Section 6
concludes.

2 RELATED WORK

We first review the existing queries on uncertain graphs and then
discuss the probabilistic distance functions.

2.1 Queries on Uncertain Graphs

In recent years, the k-NN query on uncertain graphs has received
plenty of attention [18, 27, 28]. As aforementioned, to answer
queries on uncertain graphs, people often follow the possible
world semantics, which assumes that an uncertain graph has an
exponential number of instance graphs. To deal with the curse of
exponentiation, there are three methods in the literature: (1) The
first one finds some representative deterministic graphs from the
uncertain graph and answer the queries based on them [26]. (2)
The second one is to reduce the size of the uncertain graph by
removing some weak nodes and edges, or only sample part of
the uncertain graph [28]. This, however, will lead to information
loss. (3) The last method is to build some elegant index structures, which can significantly speed up the query process without

information loss, and thus has received plenty of attention re-
cently [18, 20]. Thus, in this paper we adopt the third method for
the k-NN query.

In [28], Potamias et al. studied the k-NN query on uncertain
graphs by using incremental Dijkstra [27] with Monte Carlo (MC)
sampling. In [18], Khan et al. proposed a novel index for the
probabilistic reliability search problem (reliable set query), which
aims to find all the nodes reachable from a given source node
with probability over a user-defined threshold. Moreover, it can
be adapted for answering the top-k reliability query. However,
it only focuses on reliability search, and it is not clear how to
support other distance functions. Recently, Maniu et al. [20]
proposed a novel tree index, called ProbTree, and studied how to
perform source-to-target query (STQ) using ProbTree. However,
as it is mainly designed for STQ, it works poorly for k-NN queries.

In summary, none of these work can answer the k-NN queries
with any given probabilistic distance functions. Also, they are
not generic for any kind of probability distributions on each

edge. Therefore, it is desirable to develop a generic framework
so that any specific probabilistic distance functions and graphs
with different probability distributions can be considered when
answering the k-NN queries.

2.2 Probabilistic Distance Functions

All the distance functions that can be used for k-NN queries are
summarized in Table 1, where \( P_{s,t}(a) = \sum_{G | d_s(x,t) = a} Pr(G) \) is the
probability that the shortest path distance (SPD) between two

nodes \( s \) and \( t \) equals to \( a \) [28]. Given a query node \( s \), a k-NN
query can return the top-k nodes with the smallest values of \( d_{me} \),
\( d_{ma} \), \( d_{ex} \), and \( d_{er} \) from \( s \), or to the top-k nodes with the biggest
values of \( d_{mp} \) or \( d_{re} \) from \( s \).

The function \( d_{mp} \) measures the length of the most-probable
path (i.e., a path with the highest probability) between nodes \( s \) and \( t \). The function \( d_{re} \) measures the probability that there exists
a path between nodes \( s \) and \( t \), which is meaningful in situations
such as delivering packages in a sensor network, but it cannot
deal with the cases that prefer distance rather than reliability.
The function \( d_{ma} \) measures the median SPD among all the possible
worlds, and it can be used when the user wants to get any \( k \)-th
order statistic, but its value may be infinite. The function \( d_{me} \)
computes the SPD that is the most likely to be observed when
sampling a graph from \( G \). It is useful in uncertain graphs with
irregular distance distributions on edges, where the value has a
limited discrete domain, but it cannot deal with infinity, e.g., most
values of \( d_{me} \) will become infinite when searching in the uncertain
graphs with small probability on each edge. The functions
\( d_{ex} \) and \( d_{er} \) are often used to compute the expected distances,
and \( d_{er} \) is better when there are infinite distances in uncertain
graphs. It is worth mentioning that, there is no consensus on
which distance function is the best, because different functions
can be used in different applications.

Consider an example graph in Figure 3(a), where the number
on an edge represents the probability that it exists and \( e \) is an
infinitely small number. Suppose that our goal is to compute the
SPD from \( S \) to \( T \). Then, \( d_{ma} \) will return the length of the path
on the top, while \( d_{mp} \) will return the length of the path on the
bottom as the SPD although it may contain infinite edges in the
path.

Because of space limitations, we mainly focus on Reliability
and Expected-reliable distances in the paper, as they are the

• online query evaluation (Figure 2(b)). This stage is used to evaluate

a k-NN request for a node \( q \) online. Particularly, we design an
efficient query algorithm (Step C) that uses the U-tree developed in

the offline stage, which then yields the \( k \) nodes that are the
closest to \( q \) according to distance function \( d \).

Cost model. Upon receiving a k-NN request, the U-tree will
generate another uncertain graph \( g \) (Step C above), which is
proper subgraph of \( G \), but contains the probability information

essential to answering the k-NN query. Because \( g \) is often much

smaller than \( G \), the query execution time can be significantly

reduced. To achieve this goal, more information needs to be in-
corporated to the index during customization, which also renders

a larger U-tree (Step B). As we will point out later, there is a trade-

off between the U-tree’s size and the query cost. Later, we will
discuss a cost model that allows us to balance between these two

factors, in order to improve query efficiency without significantly

increasing the U-tree size.

Figure 2: Answering k-NN query with the U-tree.

28
most well studied ones in recent years [18, 20, 28], but other distance functions can also be easily incorporated into our indexing framework.

3 PRELIMINARIES

In this section, we first introduce the problem we studied in Section 3.1, then discuss the MC sampling in Section 3.2, and finally present the uncertain graph decomposition (UGD) and UGD-based index in Sections 3.3 and 3.4.

3.1 Problem Definition

In line with previous studies [18, 28], we consider an uncertain graph as follows. Note that all the notations frequently used in the paper are summarized in Table 2.

**Definition 1 (Uncertain Graph).** An uncertain graph $G$ is a triple $(V, E, p)$, where $V$ is the set of nodes, $E$ is the set of edges, and $p$ is the function of assigning probabilities, i.e., for any edge $e \in E$, the probability that it has a value $w$ is $p(w|e)$, and $\sum_{w \in W} p(w|e) = 1$, where $W$ contains all the possible values on the edges.

Let us take the graph in Figure 1(a) as an example, where each edge follows a zero-one distribution (e.g., let $p(0|e) = 0.1$ and $p(1|e) = 0.9$). For each edge, if its value $w = 0$, it means that the edge does not exist; otherwise, it exists. Notice that the function $p$ allows any kind of probability distributions (e.g., binomial distribution and normal distribution) to be easily assigned for an edge.

According to the possible world semantics [18, 20, 28], an uncertain graph $G$ can be considered as an exponential number of possible worlds with different probabilities. The probability of a possible world is defined as follows.

**Definition 2.** Given an uncertain graph $G=(V, E, p)$, the probability of observing a possible world $G'=(V, E_G)$ is

$$Pr(G) = \prod_{e \in E_G} p(0|e) \prod_{e \in E \setminus E_G} p(1|e).$$

For simplicity, if the edges follow the zero-one distributions (see Figure 1(a)), the probability of observing a possible world is

$$Pr(G) = \prod_{e \in E_G} p(e) \prod_{e \in E \setminus E_G} (1 - p(e)).$$

Next, we formally define STQ and $k$-NN query as follows. We illustrate them via Example 1.

**Definition 3.** Given an uncertain graph $G=(V, E, p)$, a distance function $d$, and two nodes $s$ and $t$ $(s, t \in V)$, the source-to-target query (STQ) $q_d(s, t)$ aims to compute the distance between $s$ and $t$ based on the distance function $d$.

---

**Example 1.** Consider the uncertain graph in Figure 1(a). (1) Let $s = A$, $t = B$ and $d = d_{er}$. There are 3 kinds of possible SPD values between $A$ and $B$ on $G$, that is, $Pr(SPD = 1) = 0.7$, $Pr(SPD = 3) = 0.3 \times 0.9 \times 0.6 \times 0.8 = 0.1296$ (see Figure 1(b)) and $Pr(SPD = 0) = 0.3 \times (1 - 0.9 \times 0.6 \times 0.8) = 0.1704$. Therefore, $q(A, B) = (0.7 + 0.1296) / (1 - 0.1704) = 3.13$. (2) Let $s = A$, $k = 1$ and $d = d_{er}$. We first calculate the three SPD values $d_{er}(A, B) = 1.31$, $d_{er}(A, C) = 1.07$ and $d_{er}(A, D) = 2$. Therefore, $q_1(A) = \{C\}$.

Since the number of possible worlds is exponentially large, it is impractical to enumerate all the possible worlds. To alleviate this issue, people often sample a small number of possible worlds, then perform queries on these sampled graphs, and obtain the final answer by accumulating the results in a particular manner. Next, we will introduce Monte Carlo Sampling, which is the popular way for approximate query processing in an efficient way.

3.2 Monte Carlo Sampling

To address the curse of exponentiation, the Monte Carlo (MC) sampling is used to sample a small number of graph instances [20, 28], and estimate the query results by performing queries on these sampled graphs. To sample an instance graph $G$, we can sequentially consider each edge of $G$ and sample it as a deterministic edge in $G$ following the probability distribution on the edge. Intuitively, if a possible world is repeatedly sampled multiple times, it should have a high probability to be existed.

To achieve a theoretical estimation accuracy, the Chernoff bound [28] can be applied to determine the number of possible worlds needed for a $k$-NN query. Given an uncertain graph $G$, a distance function $d$, and a pair of nodes $s$ and $t$, the accuracy of estimating the value of $d(s, t)$ by MC sampling can be well guaranteed by Lemma 1.

**Lemma 1.** [28] To achieve an error rate of $\epsilon > 0$ with a failure probability of $\delta > 0$, i.e., $Pr(|d(s, t) - d'(s, t)| \geq \epsilon d(s, t)) \leq \delta$, the
Table 3: UGD methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Lossless</th>
<th>Linear</th>
<th>Time</th>
<th>Pros &amp; Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSD [32]</td>
<td>Yes</td>
<td>Cubic</td>
<td>Smaller decomposition result, accurate but slow.</td>
<td></td>
</tr>
<tr>
<td>SPQR [14]</td>
<td>No</td>
<td>Linear</td>
<td>Smaller decomposition result but not lossless.</td>
<td></td>
</tr>
<tr>
<td>FWD [30]</td>
<td>Yes</td>
<td>Linear</td>
<td>Information lossless when $W = 2$ with more redundancy.</td>
<td></td>
</tr>
<tr>
<td>LIN [20]</td>
<td>Yes</td>
<td>Linear</td>
<td>Compressing between SPQR and FWD.</td>
<td></td>
</tr>
</tbody>
</table>

number of samples needed is
\[ g(G, s, t, \rho, \delta) = \frac{3}{e^2d^*(s, t)} \cdot \phi(G)^2 \cdot \ln \left( \frac{2}{\delta} \right) \]
\[ d^*(s, t) \text{ is the estimated value of } d(s, t), \text{ and the function} \]
\[ \phi(G) = \max_{(s, t) \in V \times V} d(s, t) \text{ is the diameter of } G. \]

In practice, people usually focus on finding the pairs with a given threshold $\rho$. Note that in general $\rho$ is not too small [28], and thus we have $\phi(G)^2 \geq \frac{3}{e^2 \rho^2}$. Therefore, the number of samples is
\[ g(G, \epsilon, \delta) = \frac{\phi(G)^2}{2e^2} \ln \left( \frac{2}{\delta} \right) \]

### 3.3 Uncertain Graph Decomposition

To enable efficient $k$-NN queries, people often build an offline index [33, 35–37], which are often based on graph decomposition methods. In Table 2, we list all the uncertain graph decomposition (UGD) methods, including junction scan decomposition (JSD), SPQR decomposition [14], lineage tree decomposition (LIN) [20], probabilistic core decomposition (PCD) [5], and probabilistic truss decomposition (PTD) [16].

In [32], Na et al. proposed JSD² for dividing a deterministic graph into several partitions by finding the junction nodes. This method can also be adapted for UGD by simply regarding the probabilities of edges as their weights. SPQR [14] is named from the optimal tree obtained by decomposing the graph into a tree of tri-connected components. FWD [30] also decomposes the tree, but limits itself at bags which have at most a limited number of nodes, which is called tree width $W$. LIN [20] behaves the same as FWD, but keeps more information in the bags for better information retrieval. SPQR can decompose the uncertain graph uniquely, but during the process there will lose some information. FWD computes probabilities exactly in the bags, but can lead to decompositions that do not reduce much the graphs. LIN can both reduce the graph drastically and allow exact computation of probabilities, but comes at a high cost in space.

PCD [5] and PTD [16] are the layered uncertain graph decomposition methods recently proposed. They can extract a dense part of the uncertain graph called $(k, \eta)$-core and $(k, \eta)$-truss respectively, which has higher probability to exist, that is, more reliable. The extraction can be iteratively executed. Note that even though the exact algorithm for finding $(k, \eta)$-core or $(k, \eta)$-truss is #P-hard, approximation algorithms are provided so that we can complete them in linear time [5, 16]. For example, there are three $(k, \eta)$-cores in Figure 3(b), and the smaller ones are more dense and reliable in the uncertain graph.

### 3.4 UGD-based Tree Index

A naïve index is to pre-compute all the pairwise distances and store them in a matrix, and then answer a query by simply looking up the matrix. This index, however, has a space complexity of $O(n^2)$, and it is not affordable if $G$ is large. Thus, it is desirable to develop more effective indexes. Recently, people often build an index based on UGD. The rationality behind is that, by decomposing $G$ into several sub-graphs, the $k$-NN query can be performed on a few sub-graphs, rather than the entire graph, resulting in high query efficiency.

To build an index for $G$, a specific UGD method from Table 3 is used to decompose it into several bags, each of which contains a set of nodes of $G$. The bags are then organized into a tree structure. Below, we present a formal definition of the tree index.

**Definition 5.** Given an uncertain graph $G(V, E, \rho)$, the index of $G$ is a tuple $(B, T)$, where $B = \{B_i | i = 1, 2, \cdots, l\}$ is a set of $l$ bags resulted from the decomposition of $G$ and $T$ is a tree, such that:
1. $\cup_{B_i \in B} B_i = V$;
2. For each $(u, v) \in E$, there is $B_i \in B, s.t. u, v \in B_i$;
3. There is a link between $B_i$ and $B_j$ in $T$ if $B_i \cap B_j \neq \emptyset$.

We illustrate the index by taking FWD decomposition as an example. We first adopt FWD to compute all the bags limited by the tree width $W = 2$. After that, for each pair of bags, if they share some nodes, then we link them with an edge. Since only one bag serves as the root, this index is a tree structure. For example, the tree index in Figure 5 is an example from the FWD decomposition of Figure 4(a). Especially, for PCD and PTD, we only link the two bags when a bag is directly a subset of another bag, to keep the tree structure of the index. For example, if $(k_1, \eta_1)$-core $\subset (k_2, \eta_2)$-core and $(k_2, \eta_2)$-core $\subset (k_3, \eta_3)$-core, we only link $(k_1, \eta_1)$-core with $(k_2, \eta_2)$-core and $(k_2, \eta_2)$-core with $(k_3, \eta_3)$-core.

Recall that we have discussed six UGD methods in Section 3.3. Generally, all these methods can be adopted in the index. Due to the space limitation, in this paper we mainly focus on FWD and PCD, since most of existing $k$-NN queries on uncertain graphs are based on expected distance search [28] or density search [34]. For the expected distance based $k$-NN queries (e.g., the $k$-NN queries based on Expected-reliable distance [28]), FWD can be used; for the density based $k$-NN queries (e.g., top-$k$ reliability query [34]), PCD can be used, since it is useful for extracting the dense components (e.g., $(k, \eta)$-core in Figure 3(b)) with high existence probabilities.

### 4 THE U-TREE INDEXING FRAMEWORK

Even with the sampling methods or indexes proposed in section 3.3, it is challenging to answer $k$-NN queries in a large uncertain graphs. However, with an advanced index, we can speed up the searching process by generating an uncertain subgraph in much smaller size, with enough information to answer the $k$-NN query. Since the uncertain subgraph is smaller, it will be significantly efficient.

For example, the subgraph with essential information (Figure 4(b)) to answer the query $d_{G}(B, E)$ is quite smaller compared to the whole graph in Figure 4(a). Since the reduction on the nodes will lead to an exponential decreasing on the sampling times based on Definition 3.2, the tree index will definitely speed up the probabilistic searching in a dramatic way. In another aspect, tree index has a reasonable cost compared to other indexes. It
can decrease the number of distances to be calculated compared to the matrix index. For example, we only need to calculate $3 \, d_{er}$ values in Figure 4(a) rather than calculate the $d_{er}$ value for each pair of nodes in $G$, that is, 21 $d_{er}$ values.

Therefore, we will answer two questions in this section: how to build index to efficiently answer the $k$-NN queries on uncertain graphs (Step A, B in Figure 2), and how to query the index when faced with a $k$-NN query (Step C in Figure 2). To answer the second question, we propose a novel structure and a cost evaluation model.

### 4.1 U-tree Index

In this section, we propose U-tree for $k$-NN search on an uncertain graph. After decomposing the uncertain graph into several bags by Definition 5, we want to dig more on the index structure.

There are two steps to build U-tree: the basic index construction step, which is directly after the decomposition (Step A), and the index customization step (Step B), which will refine the index for efficiently $k$-NN search.

**Algorithm 1: Basic index construction**

**Input:** Uncertain graph $G$  
**Output:** ($\mathcal{B}, \mathcal{T}$) 
1. $\mathcal{B} \leftarrow \emptyset, \mathcal{T} \leftarrow \emptyset$;  
2. $G \leftarrow$ undirected, unweighted graph of $G$;  
3. for $d = 1$ to 2 do  
4. while $\text{degree}(v) = d \land v \in G$ do  
5. create bag $B$;  
6. $V(B) \leftarrow v$ and all its neighbors;  
7. for all unmarked $e \in V(B) \times V(B) \cap E(G)$ do  
8. $E(B) \leftarrow E(B) \cup \{e\}$;  
9. mark $e$;  
10. end  
11. Delete $v$ and link $v$’s neighbors in $G$;  
12. $\mathcal{B} \leftarrow \mathcal{B} \cup \{B\}$;  
13. end  
14. Create the root bag $R$ with all unmarked edges;  
15. $\mathcal{B} \leftarrow \mathcal{B} \cup \{R\}$;  
16. Organize the bags in $\mathcal{B}$ into $\mathcal{T}$ by their generating orders;  
17. Add an edge between two bags in $\mathcal{T}$ if they share nodes;  
18. Calculate distance distributions between nodes $v \in G$;  
19. return $(\mathcal{B}, \mathcal{T})$;

**Step A.** First we adapt the STQ index from [20] into a $k$-NN index as the basic index construction method (see Algorithm 1).

After the initialization (line 1 to 2), we iteratively add the node in degree 0 or 1 into a bag with its neighbors$^3$, as well as the edges among them to form a bag (line 3 to 14). Every time we add the node into one bag, it will be deleted from the original graph $G$. Then we create the root bag with the nodes left (line 15 to 16). Next, we link the bags and form a tree (line 17 to 18). This basic index is evaluated in the indexing framework named PTI in Section 5.

The index is a bottom-up structure, which means the root of the tree index contains more information, so when querying the index, it will also be launched in a bottom-up way until the root. This is why we always need several bags to answer a single sub-query. Then we consider if it is possible to use fewer bags to answer it after patching extra information into the index.

Note that a bag in a higher level of U-tree will contain more information. This is because we aggregate all the information into a root bag in a bottom-up way in Algorithm 1, making only the root bag have enough information to obtain global perspective of the whole uncertain graph. Also, only the root bag contains the complete distance distribution from the uncertain graph $G$ with respect to the nodes and edges in this bag. Then we can use the root bag itself to reduce an uncertain graph, but the other bags in $\mathcal{B}$ don’t have this character.

For example, node $B$ in bag $\beta$ of Figure 5 can only see the information from its child bag $\gamma$. Assume now we need to query $q_{er}(B, C)$, even though we have known the distance distribution on the edge $(B, C)$ in bag $\beta$, before we search the root bag $\alpha$, we do not know if there exits another path in another part of the uncertain graph $G$, that is, the subgraph of nodes $(C, D, E, F, G)$ in $G$. So if the starting bag is far from the root, then we must search plenty of bags before reaching the root bag, thus making the reduced sub-graph large in size and finally need more time to finish the querying process.

$^3$Here FWD (‘W = 2) is used to decompose the uncertain graph into several bags. Other decomposition methods can be used with minor changes.
Step B. To reduce the bags to be scanned, we propose a novel method to customize the basic index from Step A. We don’t stop after generating the root bag, but continue to make each edge contain both-side information. Here both-side information means the bag can see the information of its child bags as well as the parent bags. Then each bag will obtain a global view on \( \mathcal{G} \). Therefore, if the sub-query \( q(s, t) \) only concerns the nodes in one bag, we can use this single bag to answer \( q(s, t) \), and thus the size of the uncertain subgraph will be significantly reduced.

We summarize U-tree in Algorithm 2. First we obtain the set of bags \( B \) and the tree structure \( T \) from the basic index (line 1). Then we initialize a queue and add the root bag \( R \) of \( T \) into \( Q \) (line 2 to 3). Next, we iteratively pop out the head \( P \) of \( Q \) until it is not leaf node of \( T \) (line 4 to 7). For every edge that is shared by \( P \) and its child bags, we calculate the distance distribution in the corresponding child bags, with the help of the information provided by \( P \) (line 8 to 14).

Different from the basic index \( (B, T) \) from Algorithm 1, the two points are developed in the index from Algorithm 2. First, each bag is embedded with more information. Second, we actually change the structure of \( T \). Since every bag now can see the whole uncertain graph and thus can serve as the root, we can terminate the index searching earlier. Because fewer bags are searched and a smaller sub-graph is generated, query time is saved when running the query on this smaller sub-graph.

**Algorithm 2: U-tree construction**

**Input**: Uncertain graph \( \mathcal{G} 
**Output**: \( (B, T) \)

```
1 \( Q \leftarrow \emptyset \); \( P \leftarrow \emptyset \); null;
2 \( Q \text{.add}(R); \)
3 while \( Q \) is not empty do
4     while \( P \) is null or \( P \) is the leaf node of \( T \) do
5         \( P \leftarrow Q \text{.pop}(); \)
6     end
7     for each child \( S \) of \( P \) do
8         if edge \( e \) is shared by \( P \) and \( S \) then
9             Compute the distance distribution of \( e \) in \( S \);
10            \( Q \text{.add}(S); \)
11        end
12    end
13 return \( (B, T) \);  
```

For example, when answering \( q_c(S) \), we need to answer the sub-query \( q_{cr}(A, D) \). So we start searching the U-tree in Figure 6 from bag \( y \) and stop searching when reaching bag \( b \). However, if we answer it by the index in Figure 5, we can only terminate the searching when reaching the root bag \( a \). Thus, instead of generating the uncertain subgraph from \( \{a, b, y\} \), a smaller uncertain subgraph is generated from \( \{b, y\} \).

Compared to the basic index, U-tree may help us utilize graph locality [25]. This is because the nodes, which may be queried the users, are usually localized in some area of \( \mathcal{G} \) rather than uniformly distributed. For example, if \( s \) and \( t \) is within the same bag, a single bag is enough to answer the sub-query \( q(s, t) \). From graph locality, only few bags are searched when answering each sub-query and thus the query time is significantly reduced.

![U-tree for the uncertain graph in Figure 4(a).](image)

Figure 6: U-tree for the uncertain graph in Figure 4(a).

Also, the extra cost is reasonable. The space complexity keeps linear, and the index updating cost is only increased linearly. With this efficient index, we can design the query algorithm for the challenging k-NN queries now.

**Algorithm 3: U-tree query algorithm**

**Input**: U-tree \( (B, T) \), source node \( s \in V \), integer \( k \)
**Output**: the \( k \)-NN set \( C \)

```
1 \( C \leftarrow \emptyset; \)
2 From \( s \) to get \( r \) on \( \mathcal{G} \);  
3 Generate E-table and update the bags information;
4 Re-order E-table;
5 Initiate a queue \( Q \) and load the tuples into \( Q \);
6 Perform \( r \)-pruning;
7 if \( Q = \emptyset \) then
8     return \( NULL \)
9 end
10 \( p \leftarrow Q \text{.pop}(); \)
11 while \( Q \neq \emptyset \) do
12     \( q \leftarrow Q \text{.pop}(); \)
13     if \( \text{Cost}(p \cup q) > \text{Cost}(p) + \text{Cost}(q) \) then
14         \( p \leftarrow \text{merge}(p, q); \)
15     else
16         Generate \( \mathcal{G}_p \) and answer tuple \( p \) by sampling \( \mathcal{G}_p \);
17         \( p \leftarrow q; \)
18     end
19 Perform \( r \)-pruning in \( Q \);
20 end
21 Add the nodes with top-\( k \) smallest \( d_{er} \) values into \( C \);
22 return \( C \);  
```

4.2 Query processing by E-table

For better analyzing the search on U-tree for answering k-NN query, and then develop an efficient query algorithm, we propose
a novel data structure called E-table to keep track of the indexing querying process. It is named from the fact that it can help analyze the execution process of the query.

**Definition 6.** Given a k-NN query \( q_k(s) \), an Execution-table (E-table) is a collection of tuples \( u = (id, t, r, B) \) that:
1. \( u.id \) is the identifier for \( u \).
2. \( u.t \) is the target node in the sub-query \( q(s, t) \) mapped to \( u \).
3. \( u.r \) is the lower bound for \( d(s, t) \).
4. \( u.B \) is the set of minimum bags to answer \( q(s, t) \).

Note that each tuple \( u \) in the E-table corresponds to a sub-query \( q(s, t) \) for the given k-NN query \( q_k(s) \). A k-NN query \( q_k(s) \) can be divided into \((n - 1)\) sub-queries \( q(s, t) \) where \( t \in V \) and \( t \neq s \), which are actually STQs. For example, the \( d_{er} \) based NN query\(^4\) can be divided into \((n - 1)\) STQs and then return the node with the minimum \( d_{er} \) value. We demonstrate the use of E-table in Example 2.

**Example 2.** To find the nearest neighbors for node \( B \) in Figure 4(a), we can build an E-table like Table 4, in which each row is a sub-query and each column is an attribute of the sub-query, including the id, the target node, the lower bound for \( d_{er} \), and the bags needed to answer this sub-query. Note that for \( d_{er} \), \( r \) can be calculated from \( SPD(s, t) \) since \( SPD(s, t) \) can serve as the lower bound of \( d_{er} \).

### Table 4: E-table for NN(\( B \)).

<table>
<thead>
<tr>
<th>id</th>
<th>( t )</th>
<th>( r )</th>
<th>( B )</th>
<th>Sub-query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( A )</td>
<td>( \alpha, \beta, \gamma )</td>
<td>( d(B, A) )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( C )</td>
<td>( \alpha, \beta )</td>
<td>( d(B, C) )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( D )</td>
<td>( \alpha, \beta )</td>
<td>( d(B, D) )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( E )</td>
<td>( \alpha, \beta )</td>
<td>( d(B, E) )</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( F )</td>
<td>( \alpha, \beta, \delta, \epsilon )</td>
<td>( d(B, F) )</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>( G )</td>
<td>( \alpha, \beta, \delta )</td>
<td>( d(B, G) )</td>
<td></td>
</tr>
</tbody>
</table>

Obviously, it will be very time-consuming if we run every STQ in the E-table. Alternatively, we find that a tuple in the E-table can actually answer several sub-queries. For example, in Table 4, we can find that bag \( \alpha \) and \( \beta \) contain enough information to answer three sub-queries, that is, \( d_{er}(B, C), d_{er}(B, D) \) and \( d_{er}(B, E) \).

For a k-NN query to find the nodes \( t \) with top-k smallest \( d(s, t) \), we can filter the tuples \( u \) whose \( r > \) the current maximum \( d(s, t_k) \) named as \( d_{max} \). Here \( t_k \) is the node that has been searched and added in the k-NN set to be returned. We name the k-NN set to be returned as \( C \). It is actually the candidate set before it is returned. Note that if \( C \) is not full, the current maximum \( d(s, t_k) \) is set to \( +\infty \). The filtering theory can be formalized by Lemma 2.

**Lemma 2.** All the tuples \( t \) whose \( r > d(s, C) \) can be safely pruned if |\( C \)| > \( k \) is satisfied. Here \( d(s, C) = \max \max_{i \in C} d(s, i) \).

The Lemma can be easily proved by the fact that the k-NN solution can only be found in the subset whose \( r \) value is not bigger than the current \( d_{max} = d_{er}(s, C) \). We call this pruning technique as \( r \)-pruning.

Thus, our goal is to find the proper \( C \) and use it to prune the tuples in the E-table as more as possible, especially the tuple with a huge amount of bags. For example, it will be nice if tuple 5 and 6 in Table 4 are all pruned at the very first stage because these two tuples will generate uncertain graphs of bigger size.

So we have the following steps to generate the candidate set \( C \). First, we rank the E-table according to \( r \). Then, we rank the tuples according to the number of bags, that is, the tuple with fewer bags will be searched first even they have the same \( r \) value. It is from the instinct that the tuple \( u_i \) with fewer bags will generate a smaller uncertain sub-graph and thus may terminate in an earlier stage. We demonstrate the process in Example 3.

**Example 3.** For a NN query \( q_1 \), we load the first tuple \( u_1 \) and generate the corresponding reduced uncertain graph \( G_1 \) from \( B_1 \). Then, we sample the reduced uncertain graph \( G_2 \) and get the result of \( d_{er}(s, t_1) \). We add \( t_1 \) into \( C \) and update \( d_{max} = d_{er}(s, t_1) \). Next, we prune the tuples whose \( r \) value is even bigger than \( d_{er}(s, t_1) \). So on so forth, we load the tuple \( u_1 \) which is not filtered, and generate \( G_1 \) from \( B_1 \) to answer \( d_{er}(s, t_1) \) and then launch tuple filtering and if a smaller \( d_{er}(s, t_1) \) is found, update \( C = t_1 \) and \( d_{max} = d_{er}(s, t_1) \). After searching all the tuples that are not filtered, we can return \( C \).

### 4.3 Query scheduling by cost evaluation

In the steps above, we notice that the set of bags of one sub-query maybe the upper set of another sub-query, that is \( B_i \subset B_j \) happens time to time. At this time, we can safely use the upper set \( B_j \) to answer \( u_i \) and \( u_j \). For example, the uncertain graph generated from \( u_1 \) contains all the information needed to answer \( u_2, u_3 \) and \( u_4 \). However, some sub-queries’ bag sets have overlap but not belong to each other, e.g., \( B_1 \) and \( B_2 \). Faced with this condition, we have two choices. The first choice is to combine these two tuples and generate an uncertain graph of big size to answer these two sub-queries. The second choice is to generate two small uncertain graphs and answer the two sub-queries respectively. Thus, we can choose the one with smaller cost. The cost evaluation is summarized by the function below.

**Definition 7.** Cost Evaluation is a function \( f \) to compare the cost of combining two tuples with the cost of answering them respectively. Here each uncertain graph \( G_i \) is generated by the corresponding tuple \( t_i \), and \( G_{ij} \) is generated by \( B_i \cup B_j \). If \( f(B_i \cup B_j) < f(B_i) + f(B_j) \) then we will use \( B_i \) and \( B_j \) to generate a single uncertain sub-graph. Otherwise, we will generate two uncertain subgraphs respectively.

\[
\begin{align*}
  f(B_i \cup B_j) &= S(G_{ij}, \epsilon, \delta) \cdot |E(G_{ij})| \\
  f(B_i) + f(B_j) &= \sum_{p \in l} g(G_{p}, \epsilon, \delta) \cdot |E(G_{p})|
\end{align*}
\]

(5)

(6)

Actually, if we combine all the possible tuples to generate a single uncertain graph which contains all the information to answer the k-NN query, then this reduced uncertain graph is called Equivalent Uncertain Graph (EUG) of \( G \) considering this query. Figure 7 is the EUG for \( q_1(B) = N N(B) \) on the uncertain graph in Figure 4(a). When the candidate set is big in size or the

---

\(^{4}\) NN query is the k-NN query when \( k = 1 \).
$k$-NN query has a large $k$ value, the EUG cannot be reduced much. This shows another reason why we need a cost evaluation.

Example 4. If there comes three tuples in the E-table, and they need the bags $\{\alpha, \beta, \gamma\}, \{\alpha, \delta, \varepsilon\}$ and $\{\alpha, \beta, \delta\}$ respectively (see Figure 8). Thus we get three reduced uncertain graph $G_1, G_2$ and $G_3$. We can merge these reduced graphs to answer multiple tuples or answer them one by one. There are 5 ways to answer these tuples.

We notice that the basic index has a fixed root, and the reduced subgraph will not decrease much in size. U-node is far away from the root, the query time from the basic index retrieval may stop everywhere. So when the query answer them one by one. There are 5 ways to answer these tuples.

Given the initial uncertain graph $G_0 = (V_0, E_0, p)$ and two reduced uncertain graph $G_1 = (V_1, E_1, p), G_2 = (V_2, E_2, p)$, the function $\text{merge}(G_1, G_2)$ generates an uncertain graph $G_{1,2} = (V_1 \cup V_2, E, p)$ where $E = (V_1 \cup V_2)^2 \cap E_0$. The function $\text{pop}$ will pop up the queue’s head and delete it from the queue.

**Discussion.** We notice that the basic index has a fixed root, that is, every time when the index is queried, we need to reach the root. But every bag in the U-tree can serve the root, making the index retrieval may stop everywhere. So when the query node is far away from the root, the query time from the basic index will become unbearable, because many bags are required and the reduced subgraph will not decrease much in size. U-tree overcomes this drawback but the index construction time is increased. Therefore, we wonder if there is any trade-off and balance between the two indexes, and then we can enjoy both the quick indexing and efficient querying. We answer this question by graph locality. With the help of the history queries, we can obtain the popular nodes which may be frequently queried [17]. And with the help of graph locality, most $k$-NN queries will be around these nodes. Then we can build multiple basic indexes rooted on each popular node. Every time a query comes, it can be assigned to an index with the smallest cost. In this way, the cost of index updating is reduced and the query is accelerated.

## 5 Experimental Evaluation

We now present the experimental results. We first describe the datasets in Section 5.1, then introduce the competitors in Section 5.2, and finally report the experimental results on indexing cost, query time, time proportion, accuracy and generality in Sections 5.3, 5.4, 5.5, 5.6 and 5.7 respectively.

### 5.1 Datasets

#### 5.1.1 Real-world Graphs

We use six real-world datasets. Table 6 reports their statistics. All datasets are transformed into undirected uncertain graphs with meaningful probability distribution on each edge. Note that PPI, DBLP and FBN have zero-one distribution on each edge where the probability of one means the probability of the edge existing, but BJT has no limitation on the type of the distribution on each edge, and they do not follow any standard distributions [8].

For each uncertain graph, we calculate the diameter to better show the size and density of the dataset. To calculate the diameter, we ignore the probabilities and treat each uncertain graph as an unweighted graph, and then find the largest SPD. For the dataset with multiple connected components, we report the one with the biggest diameter. We evaluate scalability of the indexing framework by testing its performance when faced with datasets with different size (see Table 6) and different density (See Table 7).

**PPI** We have two protein-protein interaction (PPI) networks (PPIc, PPIk) [23]. We use these networks in which a node denotes a protein and an edge denotes a possible interaction encoded with a specific probability calculated from biology experiments, meaning the existence confidence of this edge. Collins’s network PPIc is the core dataset which is believed to have strong interactions among the nodes [7]. Krogan’s network PPIk, however, contains less reliable interactions and thus is bigger in size [19].

**DBLP** It is a subset of the co-authorship network. The nodes denote the authors and if two authors have co-authored a paper, there will be an edge between them. We follow the popular way of generating co-authorship probability between two authors [18]. If two authors co-authored $e$ times, the probability is $1 - e^{-\mu}$ on the corresponding where $\mu$ means smaller probabilities in general. Following the trend [18], we adopt $\mu = 5$ which makes the most edges with an existing probability around 0.05.

**FBN** The dataset called Facebook-like Social Network is from an online community for students at University of California, Irvine [24]. It includes the users that sent or received at least one message. Each node denotes a user and if there are messages between two users, there will be an edge. The probability on an edge means the probability of the corresponding users having an interaction. The method to generate the probability is same as the method in DBLP but we set $\mu = 2$ to make the most edges with a probability around 0.4.

---

2. http://www.informatik.uni-trier.de/~ley/db/

---
Table 6: Statistics of Real-world Datasets.

<table>
<thead>
<tr>
<th>Graph</th>
<th>#Nodes</th>
<th>#Edges</th>
<th>Diameter</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPIK</td>
<td>1,622</td>
<td>9,074</td>
<td>15</td>
<td>11.2</td>
</tr>
<tr>
<td>PPIK</td>
<td>3,672</td>
<td>14,317</td>
<td>10</td>
<td>7.8</td>
</tr>
<tr>
<td>DBLP</td>
<td>22,016</td>
<td>58,595</td>
<td>10</td>
<td>5.3</td>
</tr>
<tr>
<td>BJTA</td>
<td>317,080</td>
<td>1,033,668</td>
<td>23</td>
<td>6.6</td>
</tr>
<tr>
<td>BJTI</td>
<td>426,196</td>
<td>946,434</td>
<td>15</td>
<td>9.074</td>
</tr>
<tr>
<td>BJTA</td>
<td>1,285,215</td>
<td>2,690,296</td>
<td>10.529</td>
<td>4.2</td>
</tr>
</tbody>
</table>

Table 7: Statistics of Synthetic Datasets.

<table>
<thead>
<tr>
<th>Graph</th>
<th>#Nodes</th>
<th>#Edges</th>
<th>Diameter</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>1,000,000</td>
<td>1,115,373</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>S2</td>
<td>1,000,000</td>
<td>1,672,561</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>S3</td>
<td>1,000,000</td>
<td>2,433,185</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>S4</td>
<td>1,000,000</td>
<td>3,128,529</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>S5</td>
<td>1,000,000</td>
<td>3,728,130</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>S6</td>
<td>1,000,000</td>
<td>4,213,833</td>
<td>15</td>
<td>7</td>
</tr>
</tbody>
</table>

BJT\(^8\) The dataset is generated from mapping 15 million Beijing Taxi trajectories [38, 39] on a Beijing map with about 3 million road segments (BJT) and road segments respectively (BJTA, only the arterial roads) [32]. The probability distribution on an edge means the speed distribution of the vehicles on the corresponding road segments. It can be generated by analyzing the trajectories [8]. Note that the diameters of BJT and BJTA is obviously bigger than the other datasets, since long roads (tens of kilometers) will be separated into many road segments (tens of meters).

5.1.2 Synthetic Graphs. Given an unweighted graph, algorithms from [4] can generate a graph with disturbances on the edges and calculate a probability of existence for each edge. The algorithm from [21] can encrypt a given weighted graph into an uncertain graph encoded with a probability on each edge.

Here we encrypt a synthetic graph generated from the Barabási-Albert model [3], which is a widely used model to simulate real graphs. According to the general setting [36], we set the average degree as 3 for each synthetic graph. Next, we generate unweighted graphs with different size as the deterministic graphs and then encrypt them by (20, 10^{-3})-obfuscation [4]. We use these graphs to test the scalability of the indexes.

5.2 Competitors

In our experiments, we try to find the k-nearest neighbors of 100 randomly generated query nodes. We use three competitors, Incremental k-NN (IKNN) from [28], RQ-tree reliability search (RQRS) from [18], and the basic index (see Algorithm 1) adapted from ProbTree indexing framework (PTI) [20].

We adapt PTI into a single-source version to deal with k-NN queries. They have been discussed in the related works. IKNN is popular in finding k-NN in the uncertain graphs, and RQRS is the state-of-the-art probabilistic k-NN index based on reliability search. PTI is the current indexing framework which can adopt different decomposition methods and distance functions.

We implement the whole U-tree indexing framework in Java, and run experiments on a machine having a 4-core Intel i5-3570 3.40GHz processor and 16GB of memory, with Ubuntu installed.


Figure 9: Scalability Evaluation on index size and index construction Time on real-world datasets (top) and synthetic datasets (bottom).

5.3 Evaluation on Indexing Cost

Here we evaluate the space cost and the index construction time of U-tree, PTI and RQRS.

The space cost is composed by the tree structure and the distance distribution stored in the bags. From Figure 9, we see that the U-tree index needs more memory since it stores more distance distributions into each bag, and RQRS needs more memory because it stores the whole node set at each level of the tree index. Compared with the query efficiency improvement in Figure 13, we find that the extra memory cost in U-tree index is reasonable, because the space cost rises 23% but the query time is 75% shorter on average.

We compare the time cost to construct each index in Figure 9. Since RQRS needs to recursively perform balanced bi-partition clustering, it’s unbearable in the cases where uncertain graphs need to be updated.

To test the scalability of U-tree, we use the synthetic graph with the graph anonymization technique. We vary the size of the synthetic graph (see Table 7) and compare the index construction time and index storage in Figure 9. Semilog is used on the y-axis because of the big difference of the index size and construction time when the number of nodes sharply changes. From the figure, we find the index construction cost of U-tree is linear to the number of nodes in the uncertain graph. This shows good scalability over uncertain graphs of different size.

5.4 Evaluation on Query Time

Here we evaluate the query time on six real-world networks and six synthetic datasets. We use the popular probabilistic k-NN algorithm IKNN [28] to show the efficiency of the query time without index acceleration, and the ability of U-tree, PTI and RQRS to speed up the searching. This comparison makes sense because they all need to find the probability distribution among nodes, which is the most expensive part in the algorithms.

In Figure 13, we vary the k value to see the changes of each curve. For each k value, we randomly pick 100 nodes as the source nodes to initiate 100 probabilistic k-NN queries, and calculate
Also, when we run the queries on the index, the bags containing the uncertain subgraph. Because both RQRS and PTI construct a tree whose height \( h \in [10, 20] \) and they all have to search on the tree from the leaves until reaching the big root. These similarities make their retrieval time in the same level. Also, the figures show good scalability of U-tree despite of the dataset size and density.

5.5 Evaluation on Time Proportion

From Figure 10, U-tree performs good when searching the index and answering the \( k \)-NN queries. From Figure 10, U-tree performs good when searching the index and answering the \( k \)-NN queries. The bigger proportion of an IQT, the bigger probability that more bags are required to generate the uncertain subgraph, thus making the query time unbearable. From Figure 10, U-tree performs good when searching the index and answering the \( k \)-NN queries.

5.6 Evaluation on Accuracy

In U-tree indexing framework, there are two parts that may cause precision loss. The first part is the choice of decomposition methods. If a decomposition method which may lose information is used, e.g., FWD with \( 'W' > 2 \), the uncertain subgraph to be generated from searching the index will become not accurate, thus making error in \( k \)-NN results. The second part is the sampling method. Since sampling is a technique for approximate query processing, it will lose information while accelerating the querying process. In general, less sampling times will lead to bigger error in the \( k \)-NN results.

Here we vary the FWD tree width \( 'W' \) and construct the corresponding U-tree respectively. Then, we run \( k \)-NN query on these U-trees with different pair of parameters \((\epsilon, \delta)\) in the sampling. We set \((\epsilon, \delta) = (x, x)\) and vary \( x \) from 0.1 to 0.9.

The \( k \)-NN results are collected and compared with the \( k \)-NN result from a baseline which simply runs big number of samplings times to find the \( k \)-nearest neighbors on uncertain graphs. We run the experiment on PPIC and PPIK and \( d_{er} \) is used. Here \( k \) is set to 100 and the Mean Squared Error is reported.

From Figure 12, we see that the Mean Squared Error is generally small even when the sampling parameters are setting in a bad way, e.g., \( \epsilon \) and \( \delta \) are very near to 1. Especially, with parameters below 0.25, we can achieve high accurate \( k \)-NN results even with decomposition methods that may lose information.

5.7 Evaluation on Framework Generality

To show the generality of the indexing framework, we will use another distance function and another decomposition method to construct U-tree. We run two different \( k \)-NN queries on each U-tree, i.e., the \( k \)-NN query which asks for the set of nodes with top-\( k \) biggest reliabilities \( d_{er} \) from the query node, and the \( k \)-NN query which asks for the \( k \)-nearest neighbors in Expected-reliable distance \( d_{er} \).

We use three different decomposition methods to construct U-tree, i.e., JSD which can find the nodes and edges to divide the graph into several partitions [32], PCD (see Figure 3(b)), and FWD where we set the tree width \( 'W' = 2 \) [20].

We run this evaluation on DBLP. The results are collected in Figure 11. Even though the U-tree with PCD and the U-tree
with FWD are similar in performance, we can see that PCD is more suitable for $k$-NN in $d_e$ and FWD is more suitable for $k$-NN in $d_{er}$. The above two U-trees have significant advantage over the U-tree with JSD, since JSD performs poor in capturing hierarchical structure of an uncertain graph.

6 CONCLUSION

In this paper, we examine the $k$-NN query on uncertain graphs. We first propose a generic index which generally can be built based on any particular uncertain graph decomposition method. Then, based on the index, we develop an efficient $k$-NN query algorithm, which supports various probabilistic distance functions. Finally, we evaluate the proposed indexing frameworks on both real and synthetic large uncertain graphs. The experimental results show that our index is effective and scalable to large graphs, and our query algorithm is very efficient.

In the future, we will study how to efficiently maintain the index for dynamic graphs, in which the nodes and edges are inserted and deleted frequently. Another interesting direction is to study how to extend the indexing framework for answering $k$-NN queries on a distributed platform. We will also perform more experimental evaluation on other real uncertain graphs.

A APPENDIX

we attach the pseudo-codes of the competitors here.

---

Figure 13: Query efficiency comparison.

Algorithm 4: Incremental $k$-NN (IKNN)

**Input**: uncertain graph $G = (V, E, P, W)$, query node $s \in V$, sampling times $r$, distance increment $\gamma$, $k$

**Output**: $k$-NN result $T_k$

1. $T_k \leftarrow \emptyset$, $D \leftarrow 0$;
2. Initiate $r$ executions of Dijkstra from $s$;
3. while $|T_k| < k$ do
   4. $D \leftarrow D + \gamma$;
   5. for $i \leftarrow 1$ to $r$ do
      6. Continue visiting nodes until reaching $D$ by Dijkstra;
      7. for each node $t \in V$ visited do
         8. Update the distance distribution and get $d_{er}$;
      end
   end
   9. for node $t \notin T_k$ do
      10. if $d_{er}(s, t) < D$ then
          11. $T_k \leftarrow T_k \cup \{t\}$
      end
   end
16. end
17. return $T_k$;
Algorithm 5: RQ-tree reliability search (RQRS)

Input: uncertain graph $G = (V, E, P)$, query node $s \in V$, sampling times $r, k$
Output: $k$-NN result $T_k$
1. $T_k \leftarrow \emptyset$;
2. Initiate $RQ$-tree based on binary clustering;
3. Compute the upper bound $U_{out}$ for each cluster $C$;
4. Generate the candidate set $C^*$;
5. Travel the clusters in $RQ$-tree in bottom-up until $C^*(s, \eta) = \arg \max_{|S| \leq C, U_{out}(s, C)} U_{out}(s, C)$;
6. Get the reduced graph $G'$ by $C^*$;
7. Initialize $r$ executions of DFS on $G'$ from $s$;
8. Calculate the reliability and select the biggest top-$k$ as $T_k$;
9. return $T_k$;

Algorithm 6: ProbTree indexing (PTI) ($G$)

Input: uncertain graph $G = (V, E, P)$, query node $s \in V$, sampling times $r, k$
Output: $k$-NN result $T_k$
1. $\#$ Index construction:
2. Build the FWD tree with $W = 2$;
3. $\#$ Information Retrieval:
4. $T_k \leftarrow \emptyset$;
5. for $v \in V$ do
6.     $C_{er}(s, v)$;
7. end
8. Put the top-$k$ smallest $d_{er}$ in $T_k$;
9. return $T_k$;

REFERENCES