An Indexing Framework for Queries on Probabilistic Graphs

Silviu Maniu, Reynold Cheng, Pierre Senellart

▶ **To cite this version:**

Silviu Maniu, Reynold Cheng, Pierre Senellart. An Indexing Framework for Queries on Probabilistic Graphs. ACM Transactions on Database Systems, 2017, 10.1145/3044713 . hal-01437580

HAL Id: hal-01437580
https://inria.hal.science/hal-01437580

Submitted on 17 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

An Indexing Framework for Queries on Probabilistic Graphs

SILVIU MANIU, Université Paris-Sud, Université Paris-Saclay
REYNOLD CHENG, The University of Hong Kong
PIERRE SENELLART, École normale supérieure, PSL Research University and Inria Paris

Information in many applications, such as mobile wireless systems, social networks, and road networks, is captured by graphs. In many cases, such information is uncertain. We study the problem of querying a probabilistic graph, in which vertices are connected to each other probabilistically. In particular, we examine "source-to-target" queries (or ST-queries), such as computing the shortest path between two vertices. The major difference with the deterministic setting is that query answers are enriched with probabilistic annotations. Evaluating ST-queries over probabilistic graphs is #P-hard, as it requires examining an exponential number of "possible worlds" – database instances generated from the probabilistic graph. Existing solutions to the ST-query problem, which sample possible worlds, have two downsides: (i) a possible world can be very large and (ii) many samples are needed for reasonable accuracy. To tackle these issues, we study the *ProbTree*, a data structure that stores a succinct, or *indexed*, version of the possible worlds of the graph. Existing ST-query solutions are executed on top of this structure, with the number of samples and sizes of the possible worlds reduced. We examine lossless and lossy methods for generating the ProbTree, which reflect the trade-off between the accuracy and efficiency of query evaluation. We analyze the correctness and complexity of these approaches. Our extensive experiments on real datasets show that the ProbTree is fast to generate and small in size. It also enhances the accuracy and efficiency of existing ST-query algorithms significantly.

CCS Concepts: •**Information systems** →**Uncertainty;** *Graph-based database models;* •**Theory of computation** →*Database query processing and optimization (theory);*

Additional Key Words and Phrases: reachability, shortest path, SPQR, treewidth, triconnected component, tree decomposition, uncertain graph

ACM Reference format:
Silviu Maniu, Reynold Cheng, and Pierre Senellart. 2016. An Indexing Framework for Queries on Probabilistic Graphs. *ACM Trans. Datab. Syst.* 1, 1, Article 1 (January 2016), 33 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Graph data are prevalent in many important and emerging applications. In a road network, cities are composed of streets and are connected by highways [1]. Mobile devices form ad-hoc networks through Wi-Fi technologies [23]. In biological networks, proteins interact with each other in a complex manner [11]. In online social networks, such as LinkedIn and Facebook, friends are interconnected to form complex social networks [20]. To study social interactions among authors of Wikipedia entries, a graph can also be constructed by analyzing the authors' text inputs. In this graph, the nodes are the authors, and an edge between two nodes describes the authors' relationship (e.g., whether they have the same political opinion) [34]. Substantial research has been devoted to the effective processing of graph queries, including reachability [16], shortest paths [19], frequent subgraphs [32], and graph patterns [13].

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Database Systems*, http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn.

Data uncertainty is inherent in the applications above. In a wireless network, the connection between two mobile devices may or may not be established, as factors such as signal interference and antenna power may affect the connection of devices [23]. Due to hardware limitations, measurement errors also occur in biological databases (e.g., protein-to-protein interaction [11]) and road monitoring systems (e.g., traffic congestion data [28]). In a graph that depicts the relationship of authors of Wikipedia entries, the existence of edges may not be definite. As another example, viral marketing techniques [33] study the purchase behavior of users in a social network. A directed edge from Mary to John, for example, indicates that John's purchases are influenced by those of Mary. This is unlikely to be a deterministic relationship, for John may not always follow Mary's purchase behavior [4]. Querying these graphs without considering this uncertainty information can lead to incorrect answers.

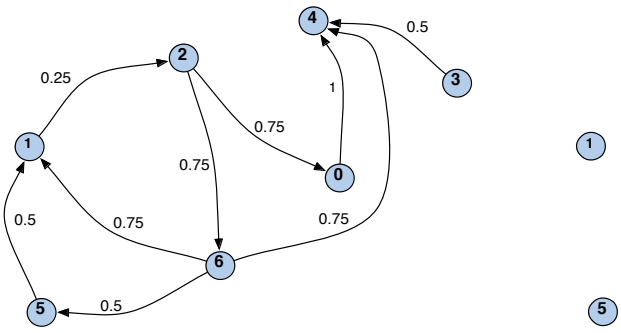
A natural way to capture graph uncertainty is to represent them as *probabilistic graphs* [12, 22, 28, 29, 46]. There exist two main representations of edge uncertainty in probabilistic graphs. In the *edge-existential model*, each edge is augmented with a probability value, which indicates the chance that the edge exists (Fig. 1a). This model captures reliability and failure in computer network connections [22, 29], and it can also represent uncertainty in social and biological networks [11]. In the *weight-distribution model*, each edge is associated with a probability distribution of weight values [28]. For example, the traveling time between two vertices in a road network can be represented by a normal distribution.

ST-queries. The problem of evaluating queries on large probabilistic graphs has been studied for a variety of tasks: reliability estimation [22, 29], searching nearest neighbors [39], and mining frequent subgraphs [49]. In this paper, we study the evaluation of an important query class, known as the *source-to-target* queries, or *ST-queries*, which are defined over a source vertex s and a target vertex t in a probabilistic graph. Example ST-queries include reachability queries (RQ) and shortest distance queries (SDQ). These queries provide answers with probabilistic guarantees. For example, the answer of an RQ tells us the chance that s can reach t; the SDQ returns the probability distribution of the distance between s and t.

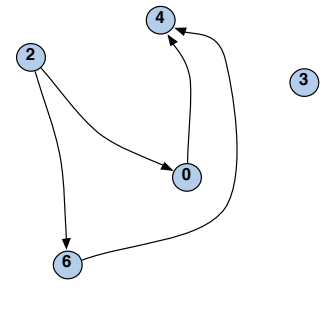
Evaluating an ST-query can be expensive. This is because these queries, executed on a probabilistic graph $\mathcal{G}$, adhere to the *possible world semantics* [17]. Conceptually, $\mathcal{G}$ encodes a set of possible worlds, each of which is a definite (non-probabilistic) graph itself. Fig. 1b shows a possible world of the probabilistic graph in Fig. 1a. Each possible world is given a probability of existence derived from edge probabilities. For example, the graph in Fig. 1b exists only if edges $0 \rightarrow 4$, $2 \rightarrow 0$, $2 \rightarrow 6$, and $6 \rightarrow 4$ exist, with a probability of 0.1, which is the product of the probabilities that edges in Fig. 1b exist, and the probabilities that other edges do not, i.e., $1 \times 0.75 \times 0.75 \times 0.75 \times (1 - 0.5) \times (1 - 0.25) \times (1 - 0.75) \times (1 - 0.5) \times (1 - 0.5)$. Evaluating a query q (e.g., an SDQ) on $\mathcal{G}$ amounts to running the deterministic version of q (e.g., computing the shortest distance between two vertices) on every possible world. This approach is intractable, due to the exponential number of possible worlds; and indeed the problem has been proved to be #P-hard [12, 17, 46].

To improve ST-query efficiency, *sampling* is usually employed [22, 29, 39, 49], where possible worlds with high existential probabilities are extracted. These algorithms, which examine fewer possible worlds than the possible world semantics, proved to be more efficient. However, they suffer from two major downsides, which can hamper query efficiency significantly:

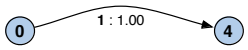
Issue 1 A possible world can be very large. Some of the probabilistic graphs used in our experiments, for example, have millions of vertices and edges. If we want to run an SDQ on a probabilistic graph, a shortest path algorithm needs to be executed once for each sampled possible world. Since a possible world can be a very big graph, query efficiency can be affected.



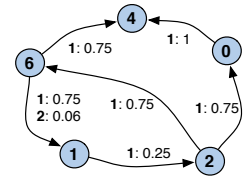
(a) A probabilistic graph $\mathcal{G}$ with unit-weighted edges



(b) A possible world of $\mathcal{G}$



(c) $\mathcal{G}(q)$ for $q(0, 4)$



(d) $\mathcal{G}(q)$ for $q(1, 4)$

Fig. 1. Illustrating (a) a probabilistic graph; (b) a possible world; and (c), (d) query-efficient representations

Issue 2 To achieve high accuracy, a large number of possible world samples may need to be generated. In our experiments, around 1,000 samples are required to converge to acceptable approximate values.

Our contributions. We improve the efficiency of ST-query evaluation by tackling the two issues above. The main idea is to evaluate the query on $\mathcal{G}(q)$, a weight-distribution probabilistic graph derived from $\mathcal{G}$. Let $q(s, t)$ be an ST-query with source vertex s and target vertex t. The result of running $q(s, t)$ on $\mathcal{G}(q)$ should be highly similar (or ideally, identical) to that of $q(s, t)$ executed on $\mathcal{G}$. If $\mathcal{G}(q)$ can be generated quickly, and $\mathcal{G}(q)$ is smaller than $\mathcal{G}$, then executing q (on $\mathcal{G}(q)$) can be faster.

Example 1.1. Let us consider an RQ, $q(0, 4)$, run on the graph in Fig. 1a. There is only one path of probability 1 between vertices 0 and 4. Correspondingly, $\mathcal{G}(q)$ is a directed edge $0 \rightarrow 4$, with $\{1 : 1.0\}$ denoting a unit-length path between vertices 0 and 4 of probability 1, as shown on Fig. 1c. Answering $q(0, 4)$ on $\mathcal{G}(q)$ is the same as evaluating $q(0, 4)$ on $\mathcal{G}$; in both cases, vertex 0 reaches vertex 4 with probability 1.

Fig. 1d illustrates $\mathcal{G}(q)$ for $q(1, 4)$. Here, edge $3 \rightarrow 4$ is not included, since it does not affect the result of $q(1, 4)$. Also, the subgraph containing vertices 1, 5, and 6 is abstracted by a directed edge $6 \rightarrow 1$. This edge represents the existence of two possible shortest paths: one with length 1 and probability 0.75 (the original edge $6 \rightarrow 1$) and the other with length 2 (the path going through edges $6 \rightarrow 5$ and $5 \rightarrow 1$) and probability $(1 - 0.75) \times (0.5 \times 0.5) \approx 0.06$.

In these examples, $\mathcal{G}(q)$ is smaller than $\mathcal{G}$, and hence efficiency of query evaluation is less affected by Issue 1. Moreover, $\mathcal{G}(q)$ contains fewer possible worlds than $\mathcal{G}$ does. Consequently, the sampling error is decreased, alleviating the impact due to Issue 2. Hence, an ST-query algorithm executed

Table 1. Summary of the ProbTree Structures.

ProbTree	Space & constr. time	Retrieval time	Query quality	Section
SPQR	linear	linear	lossy (with bound)	5
FWD	linear	linear	lossless (for $w \leqslant 2$)	6
LIN	quadratic	linear	lossless	7

on $\mathcal{G}(q)$ is faster than if it is processed on $\mathcal{G}$. As we will explain, in some of our experiments, the result of q is more accurate on $\mathcal{G}(q)$ than on $\mathcal{G}$.

How can a small $\mathcal{G}(q)$ be obtained quickly then? We answer this question by proposing the *ProbTree*, an indexing framework that facilitates ST-query execution. Given $q(s,t)$, one can efficiently generate from the ProbTree the corresponding $\mathcal{G}(q)$, which has small size. In this way, the speed of q, evaluated on $\mathcal{G}(q)$, can be improved. We formalize three design requirements for the ProbTree: (1) it should be generated efficiently; (2) it enables $\mathcal{G}(q)$ to be obtained quickly; and (3) it has a size comparable to $\mathcal{G}$.

We next investigate ProbTree construction. Our first contribution is to first prove that tree structures are the only structures fit to be ProbTree indexing structures. Based on this result, we study the *SPQR tree* [18, 24] and the *fixed-width tree decomposition* (FWD) [40], which are query-efficient structures for traditional, non-probabilistic graphs. We study their feasibility for probabilistic graphs, and build ProbTrees based on them – by appropriately incorporating edge uncertainty information. FWD ProbTrees allow an ST-query to be answered correctly but they are not as efficient as SPQR ProbTrees. On the other hand, SPQR may introduce some bounded error in the query result, making them *lossy* in general. For both structures, the construction and retrieval times, as well as the space overhead, are linear in the size of $\mathcal{G}$. We further show that the efficiency of FWD can be further enhanced without generating lossy query results, by keeping the full *lineage* of pre-computed edges, at the expense of occupying quadratic amount of space. We call this variant of FWD the *lineage tree* (or LIN). Our three solutions can be evaluated on the two major probabilistic graph classes – i.e., *edge-existential* [22, 29] and *weight-distribution* [28]. Moreover, any existing ST-query algorithm (e.g., [29]) can be executed on a graph retrieved from a ProbTree without any modification. Table 1 summarizes the main properties of the three approaches.

We have evaluated our approaches on five real-world large datasets. All our solutions significantly improve the performance of existing ST-query algorithms. The LIN structure achieves the highest efficiency (by a factor of up to 10), with a space overhead quadratic in the worst case, but reasonable in practice (in our experiments, the largest blow-up was a factor of 10, see Section 8, Fig. 8). Moreover, using SPQR trees and FWDs in conjunction with a state-of-the-art distance-constraint reachability algorithm [29] achieves speedups of 5 to 7 times.

Outline. The rest of our paper is organized as follows. We discuss related work in Section 2. Section 3 defines probabilistic graphs, ST-queries, and the ProbTree framework. We present the details of ProbTree in Section 4. In Sections 5, 6, and 7, we examine, respectively, SPQR trees, FWD, and LIN ProbTrees. We present our experimental results in Section 8.

2 RELATED WORK

We now discuss some relevant work, in the areas of probabilistic graph querying and indexing of graphs using trees.

Probabilistic graph queries. Recently, several efficient query algorithms for probabilistic graphs have been proposed. These techniques, which sample possible worlds, have been studied for ST-queries (e.g., reachability [22, 29]), k-nearest neighbors [39], and frequent subgraph discovery [49]. Our ProbTree index generates a small probabilistic graph for querying purposes. This graph has a smaller number of possible worlds, and can be used for any ST-query. It would be interesting to extend ProbTree to support k-nearest neighbor queries and frequent subgraph discovery.

We remark that the issue of indexing probabilistic graphs has been studied only recently. In [38, 48], an indexing solution was proposed for subgraph retrieval. A pruning and indexing framework for reliability search has been proposed in [31]. This technique does not extend in any direct manner to support a general ST-query, which is the topic of the present article.

Tree indexes of graphs. We create tree-like structures from graphs, by using *tree decompositions* [40], or *SPQR trees* [18, 24]. While such forms of query-efficient indexes have been used to generate indexes for efficient shortest-path-query execution [5, 47], they are designed for "certain graphs" with no probabilities. Extending their usage to probabilistic graphs is not trivial. The triangle inequality of distances in certain graphs allows the preprocessing of a large portion of the graphs. Unfortunately, this property cannot be exploited in the same way for distances in probabilistic graphs. In this article, we show how to use tree decompositions and SPQR trees to support probabilistic graph queries. In [30], the authors study how to use junction tree decompositions to evaluate queries in correlated databases [42]. They evaluate joint probabilities on the correlation DAG. Their solution does not address our problem, since we deal with general graphs rather than DAGs. Moreover, we are interested in evaluating paths over a graph, not the joint distributions in its nodes.

In a general sense, probabilistic query evaluation is shown to be tractable (in data complexity) on any tree-like structure of bounded treewidth [6, 7].

Given a probabilistic graph $\mathcal{G}$ and a ST-query q, our ProbTree can generate $\mathcal{G}(q)$, which is typically smaller than $\mathcal{G}$ and has fewer possible worlds. Hence, our approach can be considered in some sense as a *graph compression* algorithm – but compression of the possible worlds. For certain graph databases, graph compression is often used to reduce the size of a graph for higher query efficiency (e.g., neighborhood, reachability, and graph pattern queries [15, 21, 25, 43]). However, these solutions are not designed for probabilistic graphs. To the best of our knowledge, no other work has studied how to compress the possible worlds of a probabilistic graph.

In our preliminary work [35], we briefly explained how SPQR tree decompositions can be used for probabilistic graphs. In this paper, we introduce a formal indexing framework for these graphs, and we further present the detailed theoretical and experimental results for three instantiations of ProbTree (i.e., SPQR tree, FWD, and LIN). These issues were not discussed in [35]. We note that – due to errors in the software used for experiments – the results in [35] are superseded by those in the present paper.

3 INDEXING PROBABILISTIC GRAPHS

We now give definitions for probabilistic graphs and ST-queries. We also introduce our probabilistic graph indexing framework, which we call the *probabilistic indexing system.*

Definition 3.1. A *probabilistic graph* is a triple $\mathcal{G} = (V, E, p)$ where:

(i) V is a set of *vertices*;
(ii) $E \subseteq V \times V$ is a set of *edges*;

(iii) $p : E \to 2^{\mathbb{Q}^+ \times (0,1]}$ is a function that assigns to each edge a *finite probability distribution of edge weights*, i.e., each edge e is associated with a partial mapping $p(e) : \mathbb{Q}^+ \to (0,1]$ with finite support $\mathrm{supp}(p(e))$ such that $\sum_{w \in \mathrm{supp}(p(e))} p(e)(w) \leqslant 1$.

We denote $V(\mathcal{G})$, $E(\mathcal{G})$, $p_{\mathcal{G}}$ the vertex set, the edge set, and the probability assignment function of $\mathcal{G}$ respectively.

Note that the probability that an edge e does not exist in $\mathcal{G}$ is $1 - \sum_{w \in \mathrm{supp}(p(e))} p(e)(w)$. Definition 3.1 is essentially the *weight-distribution model* [28], where each edge is associated with a finite probability distribution of weights. This definition also captures the *edge-existential model* [22, 29], where an edge with existential probability p can be represented as a weight distribution $\{(1, p)\}$. Like these previous works, we assume that the probability distributions on different edges are independent.

Definition 3.2. Let $\mathcal{G} = (V, E, p)$ be a probabilistic graph. The (weighted) graph $G = (V, E_G, \omega)$ with $E_G \subseteq V \times V$ and $\omega : E_G \to \mathbb{Q}^+$ is a *possible world* of $\mathcal{G}$ if $E_G \subseteq E$ and for every edge $e \in E_G$, $\omega(e) \in \mathrm{supp}(p(e))$. We write $G \sqsubseteq \mathcal{G}$. The *probability* of the possible world G is:

$$\Pr(G) \coloneqq \prod_{e \in E_G} p(e)(\omega(e)) \times \prod_{e \in E \setminus E_G} \left(1 - \sum_{w' \in \mathrm{supp}(p(e))} p(e)(w')\right).$$

A probabilistic graph has an exponentially large number of possible worlds:

Proposition 3.3. *Let $\mathcal{G}$ be a probabilistic graph. Let $\mathrm{PW}(\mathcal{G})$ denote the set of non-zero probability possible worlds of $\mathcal{G} = (V, E, p)$; formally, $\mathrm{PW}(\mathcal{G}) = \{ G \mid G \sqsubseteq \mathcal{G}, \Pr(G) > 0 \}$.*
Then $\prod_{e \in E} |\mathrm{supp}(p(e))| \leqslant |\mathrm{PW}(\mathcal{G})| \leqslant \prod_{e \in E} (|\mathrm{supp}(p(e))| + 1)$, and $\sum_{G \in \mathrm{PW}(\mathcal{G})} \Pr[G] = 1$.

Proof. To choose a possible world G, one needs to choose for every edge e in E whether to include it in G and, if it is included, which edge weights among those in $\mathrm{supp}(p(e))$ to select e in E. There are therefore $|\mathrm{supp}(p(e))| + 1$ possibilities for each edge. If $\sum_{w' \in \mathrm{supp}(p(e))} p(e)(w') = 1$ for a given edge e, then not choosing this edge results in a possible world of zero probability, and this is the only case when this can happen. There are thus at least $|\mathrm{supp}(p(e))|$ possibilities per edge that result in a non-zero possible world.

The second part of the proposition ($\sum_{G \in \mathrm{PW}(\mathcal{G})} \Pr[G] = 1$) can be shown by a simple induction on the number of edges in the probabilistic graph. □

ST-queries. In this paper, we study *source-target distance* queries (or *ST-queries*), which is a common query class for probabilistic graphs. This kind of query requires two inputs: source vertex s and target vertex t, where $s, t \in V$. Typical example ST-queries include:

Reachability (RQ) [16] Probability that t is reachable from s.

Distance-constraint reachability (d-RQ) [29] Probability that t is reachable from s within distance d.

Expected shortest distance (SDQ) [12] The expected value of the *distance distribution* $p(s \to t)$ between s and t. Formally, $p(s \to t)$ is a set of tuples (d_i, p_i), where p_i is the probability that the shortest distance between s and t is d_i.

To evaluate these queries, we can conceptually obtain $p(s \to t)$, from which the result of any of these queries can be derived. Unfortunately, these queries are hard to evaluate, as stated below:

Theorem 3.4 ([12, 46]). *Evaluating RQ, d-RQ (for $d \geqslant 2$), and SDQ is $\mathsf{FP}^{\#\mathsf{P}}$-complete*[1].

[1] A *counting* problem is in #P if it is the number of accepting paths of some non-deterministic polynomial-time Turing machine. A *computation* problem is in $\mathsf{FP}^{\#\mathsf{P}}$ if it is solvable by a deterministic polynomial-time Turing machine with access

Without loss of generality, in the rest of the paper we assume that the answer of an ST-query is $p(s \to t)$, where any ST-query is answered from it. In fact, our solution can deal with *any* ST-query that depends only on $p(s \to t)$.

An indexing framework. We now propose an indexing framework for probabilistic graphs. First, we define the notion of *transformation system*.

Definition 3.5. A *probabilistic graph transformation system* is a pair (index, retrieve) where:

- index is a function that takes as input a probabilistic graph $\mathcal{G}$ and outputs an object $\mathcal{I} = \mathsf{index}(\mathcal{G})$ called an *index*;
- retrieve is an operator that, given an ST-query $q(s,t)$ in $\mathcal{G}$, and the index $\mathcal{I}$, produces a probabilistic graph $\mathcal{G}(q) = \mathsf{retrieve}_q(\mathcal{I})$, such that $\{s,t\} \subseteq V(\mathcal{G}(q))$.

Essentially, a transformation encodes a probabilistic graph $\mathcal{G}$ into an index structure, which can generate another probabilistic graph $\mathcal{G}(q)$ for a given pair of vertices (s,t), representing the query q. Since s and t can be found in $\mathcal{G}(q)$, $q(s,t)$ can be evaluated on $\mathcal{G}(q)$.

We consider two important properties for queries evaluated on the transformed graph $\mathcal{G}(q)$:

(i) the *loss* – the difference between the result of q evaluated on $\mathcal{G}(q)$ and $\mathcal{G}$; and
(ii) the *efficiency* – the cost of evaluating q on $\mathcal{G}(q)$.

We formalize these properties below.

Definition 3.6. Let (index, retrieve) be a probabilistic graph transformation system. Given a probabilistic graph $\mathcal{G} = (V, E, p)$, and a query load of ST-queries $\mathcal{Q}$ (e.g., RQs), the transformation loss of (index, retrieve) on $\mathcal{G}$ for $\mathcal{Q}$ is:

$$\mathsf{MSE}_{\mathcal{Q}}(\mathcal{G}, (\mathsf{index}, \mathsf{retrieve})) := \frac{1}{|Q|} \sum_{q \in \mathcal{Q}} \left(q^{\mathcal{G}} - q^{\mathcal{G}(q)}\right)^2,$$

where $q^{\mathcal{G}}$ is the result of q evaluated on $\mathcal{G}$. A transformation system (index, retrieve) is lossless for $\mathcal{Q}$ if, for every probabilistic graph $\mathcal{G}$, $\mathsf{MSE}_{\mathcal{Q}}(\mathcal{G}, (\mathsf{index}, \mathsf{retrieve})) = 0$; otherwise, it is lossy.

The definition above quantifies the loss of a transformation based on the classical definition of mean squared error, and we study both lossless and lossy transformation systems here.

A transformation system is called an *indexing system*, if it is efficient for answering a given kind of query.

Definition 3.7. A transformation system (index, retrieve) is said to be an *indexing system for query class* $\mathcal{Q}$ if the following hold:

(i) index is a polynomial-time function;
(ii) for every probabilistic graph $\mathcal{G}$, $|\mathsf{index}(\mathcal{G})| = \mathrm{O}(|\mathcal{G}|)$ (i.e., the space occupied by the index is bounded by a linear function of the space occupied by the original graph);
(iii) for every query $q \in \mathcal{Q}$, $\mathsf{retrieve}_q$ is linear-time computable.

Let us give an example transformation system that is not an indexing system. Given query class $\mathcal{Q}$, consider a system that pre-computes all pairwise results, i.e., the index operator. This system satisfies Property (iii), since $\mathsf{retrieve}_q$ builds a trivial two-vertex graph. Evaluating q over the resulting graph is very efficient, since it just involves looking up the distance probability distribution on the edges of this graph, in $\mathrm{O}(1)$ time. However, neither Property (i) nor (ii) holds,

to a #P oracle. Hardness for #P is defined in terms of Karp reductions, while for $\mathsf{FP}^{\#\mathsf{P}}$ it is in terms of Turing reductions. As a consequence, $\mathsf{FP}^{\#\mathsf{P}}$-hardness is equivalent to #P-hardness, though membership differ for both classes. See [37] for general definitions and [2] for details on reductions.

since indexing is intractable unless #P is tractable (which would imply P = NP), and since the index is at least quadratic in size.

We aim for indexing systems that allow efficient query evaluation (for a query class $\mathcal{Q}$) on the transformed graph: for every probabilistic graph $\mathcal{G}$ and query $q \in \mathcal{Q}$, the running time of retrieve_q on $\text{index}(\mathcal{G})$, together with the running time of q on $\mathcal{G}(q)$, should be less than evaluating q on $\mathcal{G}$. By summing the time for retrieve_q and the evaluation time on $\mathcal{G}(q)$, we aim at a retrieve_q operator with low overhead. One such indexing system is the ProbTree, as presented next.

4 INDEPENDENCE AND PROBTREE

We now address an important question: can we obtain an efficient index for probabilistic graphs, with zero or limited loss? We show that the answer to this question is positive, by proposing the *ProbTree*. The ProbTree is a *tree decomposition* of the probabilistic graph $\mathcal{G}$, where *independent subgraphs* of $\mathcal{G}$ are identified and reduced. We now introduce the concept of independent subgraphs.

Independent subgraphs. Recall that each edge in a probabilistic graph, along with its associated probability distribution, is independent of probability distributions of the other edges. Thus, one way to derive a lossless indexing system is to collapse larger subgraphs to edges, such that *independence* is maintained:

Definition 4.1. We define an *independent subgraph* of a probabilistic graph $\mathcal{G}$ as a (weakly) connected induced subgraph $S \subseteq \mathcal{G}$ with arbitrarily many *internal vertices* and at most two *endpoint vertices* v_1, v_2 such that each internal vertex has edges only to/from other internal vertices of S, or to/from the endpoint vertices.

We can use these independent subgraphs to reduce the graph to an equivalent subgraph by replacing S with edges $v_1 \to v_2$ and $v_2 \to v_1$, with corresponding probability distributions $p(v_1 \to v_2)$ and $p(v_2 \to v_1)$ computed from S. To understand why this is possible, let us introduce the notion of joint distance probability distributions:

Definition 4.2. Given a probabilistic graph $\mathcal{G} = (V, E, p)$ and a subset $V' = \{ v_1 \dots v_n \}$ of V, the *joint distance distribution* for V' in $\mathcal{G}$ is the probability distribution over tuples of n^2 rational numbers that gives for every tuple $(d_{ij})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant n}}$, where d_{ij} is a rational-valued distance, the probability $\Pr\left[\bigwedge_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant n}} (\text{shortest distance from } v_i \text{ to } v_j \text{ is } d_{ij}) \right]$.

The above characterizes the semantics of the probabilistic graph in terms of ST-queries: a query on any pair of vertices on the subset V' will yield the same result on any two graphs that have the same joint distribution but different structure. We now show a fundamental (and non-trivial) result: in the case of *undirected* graphs, independent subgraphs are exactly those that can be removed from the graph while preserving joint distance probability distributions for non-removed vertices.

Theorem 4.3. *Let $\mathcal{G} = (V, E, p)$ be a probabilistic graph with $(u, v) \in E \Leftrightarrow (v, u) \in E$ and V' a non-empty subset of vertices of V that are connected in $\mathcal{G}$. We assume for each $e \in E$, $\sum_{w \in \text{supp}(p(e))} p(e)(w) < 1$.*

There exists a probabilistic graph $\mathcal{G}' = (V \backslash V', E', p')$ such that the joint distance distributions for $V \backslash V'$ is the same in $\mathcal{G}'$ as in $\mathcal{G}$ if and only if *V' is the set of internal vertices of an independent subgraph of $\mathcal{G}$.*

Proof. Let us first show that such a $\mathcal{G}'$ exists if V' is the set of internal vertices of an independent subgraph S. If there are zero or one endpoint vertex in S, we do not modify the graph: indeed, no shortest distance between vertices of $V \backslash V'$ can be realized through vertices of V'. Assume

there are two endpoints v_1 and v_2. We add (or replace if they already existed) edges $v_1 \to v_2$ and $v_2 \to v_1$ whose distance distribution is given by the distance distribution from v_1 to v_2 and v_2 to v_1 either directly or through vertices of V'. Then shortest paths in G between vertices in $\mathcal{G}'$ that went through V' now go through $v_1 \to v_2$ or $v_2 \to v_1$ and result in the same shortest path distribution. Note that for this direction we do not use the fact that all edges are probabilistic.

For the other direction, assume by way of contradiction that V' is not the set of internal vertices of an independent subgraph, and that there is such a $\mathcal{G}'$. This means that vertices of V' are linked in $\mathcal{G}'$ to at least three vertices outside of V', v_1, v_2, and v_3. Since the vertices of V' are connected in $\mathcal{G}'$, there is a simple path p_{21} from v_2 to v_1 going through vertices of V' and a simple path p_{13} from v_1 to v_3 going through vertices of V' such that p_{21} and p_{13} share an edge e. Since all edges may be missing, there is a world where the only path between v_2 and v_1 (resp., between v_1 and v_3) is achieved by paths formed of edges of p_{21} and p_{13}. We denote by $i \to^{\mathcal{G}} j$ the probabilistic event "there is a path from i to j in $\mathcal{G}$", by $i \not\to^{\mathcal{G}} j$ its complement, and by $X^{\mathcal{G}}$ the event: "for all pairs of vertices $(i,j) \in (V \backslash V')^2$ with $i \neq j$ and either i or j not in $\{v_1, v_2, v_3\}$, $i \not\to^{\mathcal{G}} j$. This event is realizable jointly with $v_2 \to^{\mathcal{G}} v_1$ and $v_1 \to^{\mathcal{G}} v_3$ by considering the world where all edges not connecting to V' are removed. Since the joint distance distributions of $\mathcal{G}$ and $\mathcal{G}'$ are the same, we have:

$$\begin{aligned}&\Pr\left[v_2 \to^{\mathcal{G}} v_1 \wedge v_1 \to^{\mathcal{G}} v_3 \middle| v_2 \to^{\mathcal{G}} v_3 \wedge X^{\mathcal{G}}\right] \\ =&\Pr\left[v_2 \to^{\mathcal{G}'} v_1 \wedge v_1 \to^{\mathcal{G}'} v_3 \middle| v_2 \to^{\mathcal{G}'} v_3 \wedge X^{\mathcal{G}'}\right].\end{aligned}$$

Now, observe that

$$\begin{aligned}&\Pr\left[v_2 \to^{\mathcal{G}'} v_1 \wedge v_1 \to^{\mathcal{G}'} v_3 \middle| v_2 \to^{\mathcal{G}'} v_3 \wedge X^{\mathcal{G}'}\right] \\ =&\Pr\left[v_2 \to^{\mathcal{G}'} v_1 \middle| v_2 \to^{\mathcal{G}'} v_3 \wedge X^{\mathcal{G}'}\right] \Pr\left[v_1 \to^{\mathcal{G}'} v_3 \middle| v_2 \to^{\mathcal{G}'} v_3 \wedge X^{\mathcal{G}'}\right] \\ =&\Pr\left[v_2 \to^{\mathcal{G}} v_1 \middle| v_2 \to^{\mathcal{G}} v_3 \wedge X^{\mathcal{G}}\right] \Pr\left[v_1 \to^{\mathcal{G}} v_3 \middle| v_2 \to^{\mathcal{G}} v_3 \wedge X^{\mathcal{G}}\right]\end{aligned}$$

since in $\mathcal{G}'$, $v_2 \to^{\mathcal{G}'} v_1$ and $v_1 \to^{\mathcal{G}'} v_3$ are conditionally independent given $X^{\mathcal{G}'}$ and $v_2 \to^{\mathcal{G}'} v_3$ (in $\mathcal{G}'$ the only possible worlds where X is realized are those where only v_1, v_2, v_3 may be connected to each other). But $v_2 \to^{\mathcal{G}} v_1$ and $v_1 \to^{\mathcal{G}} v_3$ are *not* conditionally independent given $X^{\mathcal{G}}$ and $v_2 \to^{G} v_3$ since they are correlated by the presence of the edge e. Contradiction. □

In other words, Theorem 4.3 states that the independent subgraph approach is the *unique* manner in which a lossless indexing system can be obtained for a probabilistic graph, at least for undirected graphs. The case of directed graphs is more complex; the tools that we use (in particular, tree decompositions) are more robust and better understood in the setting of undirected graphs [40] than in that of directed ones [41]. For that reason, we leave open the possibility of using techniques that exploit directedness of edges in order to obtain better indexing systems, in the case of directed graphs, than techniques based on the independent subgraph approach.

ProbTree. Our definition of independent subgraphs relies on vertices in the graphs which separate the graph into two independent components. We can *decompose* the graphs into the corresponding independent subgraphs in a recursive way, by repeatedly identifying endpoints and sub-dividing the subgraphs until it is not longer possible to do so. Put aside for now the question of choosing the independent subgraphs to decompose – we will propose different solutions to this problem in Sections 5 and 6. Whatever the choice, it is straightforward to verify that such a recursive decomposition – our desired index $\mathcal{I} = \mathsf{index}(\mathcal{G})$ – results in a tree where nodes are independent subgraphs and edges appear between subgraphs having common endpoints. We call such a tree decomposition a *ProbTree*.

Definition 4.4. Let $\mathcal{G}$ be a probabilistic graph. A *ProbTree* for V is a pair $(\mathcal{T}, \mathcal{B})$ where $\mathcal{T}$ is a tree (i.e., a connected, acyclic, undirected graph) and $\mathcal{B}$ is a function mapping each node of T to a *probabilistic graph* (called the *internal graph* or *bag* of n) with vertex set a subset of V. We further require that for every subtree $\mathcal{T}'$ of $\mathcal{T}$, the set of vertices in bags of nodes of $\mathcal{T}'$ induces an independent subgraph of $\mathcal{G}$.

Example 4.5. To understand what a ProbTree looks like, consider the tree depicted in Fig. 2 which is a ProbTree for the graph of Fig. 1 (it is more precisely an SPQR tree, as we will introduce in the next section). The tree $\mathcal{T}$ is defined by the black lines between bags; a Greek letter identifier is given on the right of each bag. Ignore for now the edges inside bags and the R, S, P labels and focus on vertices within each bag. The vertices in bags of any subtree of $\mathcal{T}$ induce an independent subgraph in the graph of Fig. 1: for example, the subtree rooted at node (δ) contains vertices 1, 2, 5, and 6, which indeed form an independent subgraph with endpoints 2 and 6. The nodes in white represent the endpoints of the independent subgraphs induced by the bag's respective subtree: here, for node (δ), these are 2 and 6. We will come back to this example in the next section and explain how to interpret the rest of the ProbTree structure, as well as how such a ProbTree may be obtained.

How can we efficiently find independent subgraphs, build a ProbTree, and use this ProbTree as an index for query answering? In the next two sections, we present two solutions: SPQR trees (Section 5) that provide an optimal and unique decomposition, but which result in a lossy indexing system; tree decompositions (Section 6) that yield weaker decompositions, but a lossless indexing system.

5 SPQR TREES

We introduce in this section a first method for indexing probabilistic graphs into a ProbTree: SPQR trees [18]. First, we need some graph theory basics on k-connectedness.

For a graph G, a vertex set $S \subseteq V(G)$ is called a *separator* for G if the graph induced by $V(G) \backslash S$ is disconnected. Given an integer k, a graph G is called *k-connected* if $V(G) \backslash S$ is connected for all $S \subseteq V(G), |S| < k$, i.e., there exists no separator for G of size less than k. 0-connected graphs are connected graphs in the usual sense, 1-connected graphs contain *cut vertices* which disconnect the graph into *biconnected* components, and 2-connected graphs have *separation pairs* of vertices which separate the graph into *triconnected* components. These definitions link directly to our desired properties for independent subgraphs. Connected, biconnected, and triconnected components are exactly independent subgraphs of 0, 1 and 2 endpoints, and we aim to decompose the graph into a tree containing them.

Tutte [45] studied the structure of the triconnected components of a graph, and Hopcroft and Tarjan [26, 27] gave optimal algorithms for decomposition. They showed that the triconnected components of a graph are unique:

Theorem 5.1 ([26, 27, 45]). *The biconnected and triconnected components of a graph G are unique and the inclusion relationships among them forms a tree.*

Using Theorem 4.3 and Theorem 5.1, we can derive the corollary that decomposing the graph into its biconnected and triconnected components is the unique manner in which we can obtain a lossless indexing of a probabilistic (undirected) graph – since biconnected and triconnected components are independent subgraphs and they are unique.

Hopcroft and Tarjan's algorithms were refined, by using SPQR trees [18] and Gutwenger's linear implementation of them [24]. Our approach uses these algorithms as a first step to obtain the decomposition. We go beyond simple decompositions in our approach, and show how distance

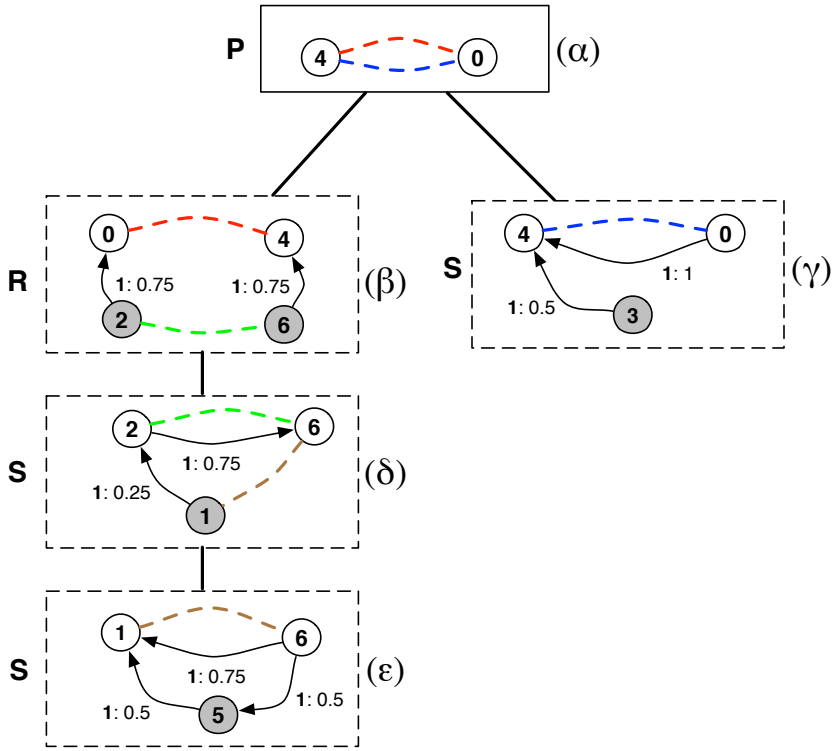


Fig. 2. SPQR tree of the graph in Fig. 1.

distributions on the interface edges can be computed, and how they can be propagated inside a ProbTree. We also show how to retrieve a query-equivalent graph from the decomposition.

In the following, we will consider the underlying deterministic graph G for a probabilistic graph $\mathcal{G}$, having the same edges and vertices as $\mathcal{G}$.

Indexing. Our ProbTree $\mathcal{T}$ consists of nodes corresponding to the triconnected components of the graph. Two types of edges are present in the bags of the ProbTree: *real* edges already existing in G, and *skeleton* edges, which correspond to the reduced triconnected components in the tree children. The decomposition of the graph $\mathcal{G}$ in the resulting index $\mathcal{I} = \mathsf{index}(\mathcal{G})$ corresponds exactly to the construction of the SPQR tree, together with the computation of the probability distributions for each skeleton edge in the graph.

There are three types of internal (triconnected) graphs in an SPQR tree, and by extension in $\mathcal{T}$ [18]:

(1) a path of at least two edges between the two endpoints; the corresponding tree bags are called *serial* or *S-bags*;
(2) two vertices having parallel edges; the corresponding bags are called *parallel* or *P-bags*; and
(3) a triconnected graph with neither of the previous two structures; the corresponding bags are called *rigid* or *R-bags*.

Example 5.2. We present in Fig. 2 the SPQR ProbTree resulting from the graph in Fig. 1. Note that each edge of the original graph (shown solid, while skeleton edges are dashed) is present only

in one bag, but vertices can be repeated across bags. The SPQR ProbTree is composed of three S-bags, one P-bag, and one R-bag. Each bag contains the union of the induced subgraph of $\mathcal{G}$ and the skeleton edges. Moreover, each bag contains a triconnected component.

Take bag (δ) as an example. It consists of three vertices and two edges of $\mathcal{G}$ ($1, 2, 6$ and $1 \to 2$, $2 \to 6$), and a skeleton edge propagated from node (ε), summarizing paths from 6 to 1 in node (ε) (there is no path from 1 to 6 in node (ε)). Vertices 2 and 6 are a separation pair for the subgraph induced by the vertices in bags (δ) and (ε), i.e., vertices $1, 2, 5, 6$. Bag (β) is an R-bag, and the bag (α) is a P-node, containing two parallel undirected skeleton edges, corresponding to the two branches of the SPQR tree.

The construction of an SPQR tree is a fairly elaborate process [18, 24] that is out of scope of this article. However, for the sake of illustration, we present an intuitive description of how the tree of Fig. 2 can be obtained from the graph of Fig. 1a. The goal is to decompose the (underlying undirected) graph in its triconnected components. We are thus looking for endpoints of triconnected components.

Nodes 1 and 6, for example, are the endpoints of a triconnected component that includes 1, 5, and 6 and another one that includes all vertices but 5. We thus build a node (ε) for the first triconnected component, which cannot be decomposed further and has an S form (an undirected cycle between vertices 1, 5, and 6). We then look for triconnected components in the rest of the graph. It is clear that 2 and 6 are endpoints of a component that contains vertex 1, and of another that contains vertices 0, 3, 4. Building a node for the former yields (δ). Now, we are left with vertices 2, 6, 0, 4, 3. On the induced subgraph, 0 and 4 separate the graphs into two triconnected components, one including 3 and one including all other nodes, yielding (γ) and (β). The latter cannot be further decomposed; the former can actually be further decomposed by isolating the two endpoints 0 and 4 (in node (α)) from a component containing 3 itself. This last refinement may seem to be a detail, but is performed since it is necessary to model paths from 0 to 4 either through the vertices of the subtree rooted at (β) or through the original edge in node (γ).

Note that the original formulation of SPQR trees contained also a fourth type of bag, the *Q-bag* or *trivial bag*, which were simply each edge of the graph in a single tree node, for ease of abstraction. In the most recent linear implementation [24] and in this paper, these bags are ignored and the edges are simply copied to the nearest ancestor in $\mathcal{T}$.

Algorithm 1 details the index operator using SPQR trees. It outputs a ProbTree $(\mathcal{T}, \mathcal{B})$.

The first step is the application of the SPQR tree algorithms from [18, 24], which creates a tree $\mathcal{T}$ and a mapping $\mathcal{B}$ from bags of $\mathcal{T}$ to sets of vertices of $\mathcal{G}$. We omit here the details of the SPQR algorithm, as it is not our focus, and we direct the reader to [24] for an up-to-date description of the working of the decomposition algorithm. Bags $\mathcal{B}(n)$ are then populated with the original edges from $\mathcal{G}$ which are between vertices in $\mathcal{B}(n)$.

The second step – and most important for correct query evaluation – is the pre-computation and upwards propagation of distance probabilities of the separation pairs in $\mathcal{T}$, i.e., function `precompute-propagate`$^{\text{SPQR}}$. We use here the observation that the distance distributions between endpoints can be computed in two directions. For example, take bag (β). Edge $0 \to 4$ can either be computed as coming from the independent subgraph defined by bags (α) and (γ), or by the independent subgraph defined by bags (β), (δ), and (ε). This bi-directional computation is very useful for the retrieve operator, as we shall see. We can perform this computation in an optimal manner, by successively rooting $\mathcal{T}$ at each of its leaves l, and then propagate the computation upwards.

For every node n of $\mathcal{T}$, we first need to collect the computed distributions of the separation pairs corresponding to bags of children of n. Then the probability distribution corresponding to

ALGORITHM 1: index$^{\text{SPQR}}(\mathcal{G})$

input : a probabilistic graph $\mathcal{G}$, width parameter w
output : index index$^{\text{SPQR}}(\mathcal{G}) = (\mathcal{T}, \mathcal{B})$

```
/* decompose the graph using SPQR trees                                    */
G ← undirected, unweighted graph of 𝒢;
(ℬ, 𝒯) ← compute-spqr(G);
for n node of 𝒯 do
    copy the edges of 𝒢 to ℬ(n);
/* compute edges between uncovered vertices and propagate up               */
for l, leaf of 𝒯 do
    root 𝒯 at l;
    for h ← height(𝒯) to 0 do
        for node n of 𝒯 s.t. level(n) = h do
            precompute-propagate^SPQR(ℬ(n), 𝒯);
root the tree at the node with largest bag;
return (𝒯, ℬ);
```

the endpoints $\{v_1, v_2\}$, i.e., $p(v_1 \rightarrow v_2)$ and $p(v_2 \rightarrow v_1)$, is computed, if it has not been computed previously when rooting the tree at other leaf bags. If it has been computed previously, then we do not need to perform the computation again, which means that each of the two directions for each pair of endpoints in each bag will only be computed *once*.

Depending on the type of bag, we have two ways of computing the endpoint distance distributions. For S-bags and P-bags, these can be computed *exactly* using min- and sum-convolutions of distance distributions – denoted as $\odot$ and $\oplus$ here. Fig. 3 illustrates the $\odot$ and $\oplus$ operators on distance distributions. For more details on the computation of convolutions of probability distributions, we refer the reader to [10].

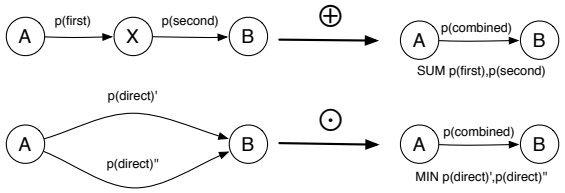


Fig. 3. Probability compositions of simple paths.

The convolutions depends on the configuration of the path we wish to pre-compute. In the case of a P-bag – equivalent to several parallel edges between the same endpoints – the final distance distribution between endpoints can be computed using a MIN convolution – denoted in the following as $\odot$ – of all the parallel edges in the bag. In other words, we compute the distribution of the minimal distance between the two endpoints. The MIN convolution is linear in the maximum distance of the input distributions. In the case of an S-bag – or a serial path between the endpoints with a direct edge between the endpoints – the distribution can be computed by applying a SUM convolution of the serial path between v_1 and v_2 – denoted as $\oplus$ – followed by a MIN convolution with the direct edge distribution. The SUM convolution is the distribution of the sum of the

distances in the serial path. Its computation is quadratic in the maximum distance of the input distributions.

For R-bags, it is expensive to compute exactly the endpoint distribution in the general case, as the graph present in the bag can have an arbitrary configuration. In this case, we can compute the endpoint distribution using sampling, choosing the number of samples by applying the Chernoff and Hoeffding inequalities, to obtain an (ε, δ) multiplicative guarantee [39]. We can then use the per-bag guarantees to compute the overall guarantees on the distributions in the root bag, in the spirit of [44].

Finally, to maximize the chances that an ST-query does not need any retrieve operation, we root $\mathcal{T}$ at the bag which contains the largest number of vertices.

Example 5.3. Returning to the SPQR tree in Fig. 2, we illustrate how the exact computation would work for bag (ε) in the tree, an S-bag. For the $6 \to 1$ direction, we first have to compute the SUM convolution of $p(6 \to 5)$ and $p(5 \to 1)$:

$$p'(6 \to 1) = p(6 \to 5) \oplus p(5 \to 1) = \{(d = 2, p = 0.5 \times 0.5 = 0.25)\}.$$

Then, the final $p^c(6 \to 1)$ is computed as the MIN convolution of the existing $p(6 \to 1)$ and the computed $p'(6 \to 1)$:

$$\begin{aligned} p(6 \to 1) &= p(6 \to 1) \odot p'(6 \to 1) \\ &= \{(d = 1, p = 0.75), (d = 2, p = (1 - 0.75) \times 0.25 = 0.0625)\}. \end{aligned}$$

There is no configuration in which $1 \to 6$ has a finite distance, hence $p^c(1 \to 6) = \emptyset$. The two distributions will be propagated up in the tree, to bag (δ), where they will serve for the computation of the distribution between endpoints 2 and 6.

Algorithm 2 details the pre-computation step. Note that for P-bags, we do not need to do anything in the second step, as the collection of children nodes will already take care of the MIN convolution of the parallel edges.

ALGORITHM 2: precompute-propagate$^{\text{SPQR}}(B, \mathcal{T})$

input: bag B, tree $\mathcal{T}$

```
/* propagating computations from children                                  */
for distribution p^c(u → v) in children of B do
    p(u → v) ← p(u → v) ⊙ p^c(u → v) ;
/* computing pairwise distributions                                        */
for edge v1 → v2 between endpoints v1, v2 do
    if p^c(v1 → v2) ∉ computed(B) then
        if type(B) = R then
            p^c(v1 → v2) ← sample(v1, v2, B) ;
        else if type(B) = S then
            p'(v1 → v2) ← p(v1 → u1) ⊕ ··· ⊕ p(uj → v2) ;
            p^c(v1 → v2) ← p(v1 → v2) ⊙ p'(v1 → v2) ;
        add p^c(v1 → v2) to computed(B);
```

Retrieval. When answering (s, t) ST-queries on the ProbTree we have two main cases. First, when both s and t are present in the root node, we only need to query the root bag with no need to look in the decomposition. The second case is the most interesting one: when at least one of s, t

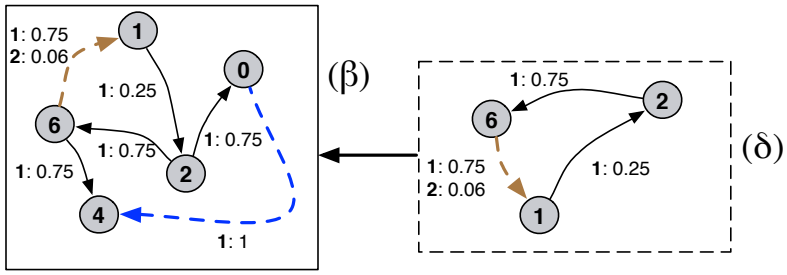


Fig. 4. Retrieval for the pair (1, 4).

are not in the root, but are vertices in the decomposition bags. In this case, the query vertices need to be propagated to the root node.

The bi-directional property of computed new edges means that we can simply assume that the root of the tree is located at one of the bags containing s or t, and then propagate only the edges corresponding to the other query vertex. It is not important which node is chosen – it is easy to verify that the number of edges propagated will be the same – so we will assume we root the tree at the node whose bag contains t in the following.

The original edges in ancestors of the bags containing the query vertices are propagated up, all the way to the new root, in a bottom-up manner. The previous pre-computations of edges in areas of the graphs not containing the query vertices and in the subtree of the bags containing the query vertices are not affected by this change. Recomputing the edges on these parts of the tree is not necessary, and this ensures that only a fraction of the bags in the tree is affected by the retrieval. Algorithm 3 details this operation.

ALGORITHM 3: retrieve$^{\text{SPQR}}(\mathcal{T}, \mathcal{B}, s, t)$

input : ProbTree $(\mathcal{T}, \mathcal{B})$, source s, target t
output: probabilistic graph $\mathcal{G}$

root the tree at one of the bags containing t;
/* propagate edges up the new tree */
for $h \leftarrow$ height$(\mathcal{T})$ **to** 0 **do**
  for *node* n *of* $\mathcal{T}$ *s.t.* level$(n) = h$ **do**
    $B \leftarrow \mathcal{B}(n)$;
    if $V(B) \cap \{s\} \neq \emptyset$ **then**
      delete p^c in parent(B) resulting from B;
      $E(\text{parent}(B)) \leftarrow E(\text{parent}(B)) \cup E(B)$;
      $V(\text{parent}(B)) \leftarrow V(\text{parent}(B)) \cup V(B)$;
return $\mathcal{B}(\text{root}(\mathcal{T}))$

Example 5.4. Let us return to the decomposition in Fig. 2, and exemplify how a retrieval for the query pair (1, 4) proceeds. Fig. 4 illustrates the execution of Algorithm 3 for this pair.

First, since 1 and 4 are on the same branch of $\mathcal{T}$, we can root the tree at bag (β). Moreover, one can notice that there is no need to recompute endpoint distributions on bags (α), (γ), and (ε). Hence, the computed edge $6 \to 1$ will be used from bag (ε) and $0 \to 4$ from (α). However, the computed edges $6 \to 2$ and $2 \to 6$ will not be propagated from bag (δ) to bag (β), as their

computation involves a query vertex, in this case vertex 1. Hence, all vertices and edges from bag (δ) will be propagated to bag (β), and joined by the original edge in (α), $0 \to 4$. The resulting graph in the new root – bag (β) – is a graph which will output equivalent results for the query on $(1, 4)$ as the original graph in Fig. 1a.

Properties. It is easy to check that the index and retrieve operators define an indexing system where queries run faster on the retrieved graph than on the original graph. Theorem 4.3 ensures the validity of the approach. The implementation of SPQR trees of [24] is linear in the size of $\mathcal{G}$. The `precompute-propagate` function only pre-computes endpoint distributions once per bag. The computation itself is polynomial, either the MIN and SUM convolutions, or the sampling of the R-bags using a set number of sampling rounds. The above two results verify Property (i) of Definition 3.7. Moreover, it is a known result that the number of skeleton edges added in the triconnected components tree is $O(E)$ (more precisely, it is upper-bounded by $3|E| - 6$, as shown in [45]), thus verifying Property (ii).

Each retrieve will output a graph that is at most as big as the original graph, and hence the standard shortest-path algorithms [19] would execute in less time for each sample[2]. Moreover, the retrieval is linear in the number of tree bags, which is itself linear in the size of $\mathcal{G}$, verifying Property (iii). Hence ($\mathsf{index}^{\mathrm{SPQR}}$, $\mathsf{retrieve}^{\mathrm{SPQR}}$) is an indexing system.

SPQR ProbTrees display a lot of the advantages we desire for our indexing systems, i.e., their optimality and their linear space and time costs. They, however, also have a big disadvantage. The presence of R-bags, along with the fact that we cannot trivially remove them from the structure, makes them *lossy* in the general case, even if this loss can be controlled by approximation guarantees.

In the next section, we introduce a different indexing technique by applying another classical graph decomposition technique, namely *fixed-width tree decompositions* (FWDs). We show how this technique can lead to *lossless* decompositions (either by bounding the width of the decomposition, or by introducing extra bookkeeping structures).

6 FIXED-WIDTH DECOMPOSITIONS

Tree decomposition of graphs [40] is a classic technique to solve NP-hard problems in linear time [9], where the input is constrained to be a graph with bounded treewidth. In this section, we leverage tree decompositions as a second method for indexing probabilistic graphs into a ProbTree.

6.1 Preliminaries on Tree Decompositions

Following the original definitions in [40], we start by defining a tree decomposition:

Definition 6.1 (Tree Decomposition). Given an *undirected graph* $G = (V, E)$, its *tree decomposition* is a pair (T, B) where $T = (I, F)$ is a tree and $B : I \to 2^V$ is a labeling of the nodes of T by subsets of V, with the following properties:

(i) $\bigcup_{i \in I} B(i) = V$;
(ii) $\forall (u, v) \in E$, $\exists i \in I$ s.t. $u, v \in B(i)$; and
(iii) $\forall v \in V$, $\{i \in I \mid v \in B(i)\}$ induces a subtree of T.

Intuitively, a tree decomposition groups the vertices of a graph into *bags* so that they form a tree-like structure, where a link between bags is established when there exists common vertices in both bags. Based on the number of vertices in a bag, we can define the concept of *treewidth*:

[2]We assume that the sampling from a distance probability distribution on an edge incurs constant-time cost.

Definition 6.2 (Treewidth). For a graph $G = (V, E)$ the *width* of a tree decomposition (T, B) is equal to $\max_{i \in I}(|B(i)| - 1)$. The *treewidth* of G, $w(G)$ is equal to the minimal width of all tree decompositions of G.

Given a width, a tree decomposition can be constructed in linear time [14]. However, determining the treewidth of a given graph is NP-complete [8]. This means that determining if a graph has a bounded treewidth, and thus being able to create its tree decomposition, cannot be reasonably performed on large-scale graphs.

Note that algorithms which solve NP-hard problems in linear time when restricted to graphs of bounded treewidth – including k-terminal reliability – have been proposed in [9]. They have two main disadvantages:

(i) they use bottom-up dynamic programming for the computation of optimal values, but they retain an exponential dependence on the treewidth w; and
(ii) the practical appeal is limited, as the computation of the query answers is made at the same time as the construction of the decompositions.

Our solution, in contrast, is linear in the size of the graph, and is computed only once to be used for any query.

In real-world graphs or complex networks, it has been observed that graphs have a dense core together with a tree-like fringe structure [36]. It is consequently possible to decompose the fringe, and finally to place the rest of the graph in a "root" node. Based on this, indexes for faster exact shortest path query answering have been proposed using *fixed-width decompositions* [5, 47] (FWDs) in the context of exact graphs: the idea is to fix a given treewidth and decompose the graph as much as possible to obtain a *relaxed* tree decomposition where only the root node may have a large number of nodes. Note that these indexes can be used for shortest paths by exploiting the triangle inequality property in definite graphs and thus any width can be used as a parameter. This is not possible in probabilistic graphs, and thus the algorithms cannot be readily used. In the following – supported by our main result in Theorem 4.3 – we study the suitability of FWDs for indexing probabilistic graphs.

6.2 Algorithm

Indexing. We now present in Algorithm 4 the index operator. It consists of three stages: the main decomposition, the building of the FWD ProbTree and the pre-computation of paths.

As for the SPQR decomposition, the first stage of Algorithm 4 (lines 1–14) is the adaptation of the algorithms in [5, 47], which build the decomposition tree. At each step, a vertex having a degree at most w is chosen, marked as *covered*, and its neighbors are added into the bag, along with the probabilistic edges from $\mathcal{G}$. Then, the covered vertex is removed form the undirected graph G and a clique between the neighbors is created. This process repeats until there are no such vertices left. Finally, the rest of the uncovered vertices and the remaining edges are copied in the root graph $\mathcal{R}$.

The second stage is the creation of the tree $\mathcal{T}$. We visit in creation order each bag and define as their parent the bag whose vertex set contains all uncovered vertices of the visited bag. If no such bag exists, the parent of the bag will be the root graph.

The final stage is similar to the SPQR tree approach. In each bag B, and for each pair (v_1, v_2), we need to compute $p(v_1 \rightarrow v_2)$ by using the information about the link configuration between v_1, v_2 and the covered vertex v, using MIN and SUM convolutions. More precisely: $p(v_1 \rightarrow v_2) = p(v_1 \rightarrow v_2) \odot (p(v_1 \rightarrow v) \oplus p(v \rightarrow v_2))$. This is followed by the bottom-up propagation of computed probabilities, in a manner similar to SPQR. At each step pairwise probabilities are computed among the vertices which are not the covered vertex v of the respective bag. In order to compute these probabilities, for each bag B, the first step is to "collect" the computed edges

ALGORITHM 4: index$^{\text{FWD}}(\mathcal{G})$

input : a probabilistic graph $\mathcal{G}$, width parameter w
output : index index$^{\text{FWD}}(\mathcal{G}) = (\mathcal{T}, \mathcal{B})$

/* decompose the graph into bags of size $\leqslant w$ */
$G \leftarrow$ undirected, unweighted graph of $\mathcal{G}$;
$\mathcal{S} = \emptyset, \mathcal{T} = \emptyset$;
for $d \leftarrow 1$ **to** w **do**
  while *there exists a vertex v with degree d in G* **do**
    create new bag B;
    $V(B) \leftarrow v$ and all its neighbors;
    for *all unmarked edge e in $\mathcal{G}$ between vertices of $V(B)$* **do**
      $E(B) \leftarrow E(B) \cup \{e\}$; mark e;
    covered $(B) \leftarrow \{v\}$;
    remove v from G and add to G a $(d-1)$-clique between v's neighbors;
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{B\}$;
/* create the root graph and the bag tree */
$V(\mathcal{R}) \leftarrow$ all vertices in $\mathcal{G}$ not in covered$(\mathcal{B})$;
$E(\mathcal{R}) \leftarrow$ all unmarked edges in $\mathcal{G}$;
for *bag B in $\mathcal{S}$* **do**
  mark B;
  if $\exists$ *an unmarked bag B' s.t.* $V(B) \backslash \text{covered}(B) \subseteq B'$ **then**
    update $(\mathcal{T}, \mathcal{B})$ so that B' is parent of B;
  else update $(\mathcal{T}, \mathcal{B})$ so that $\mathcal{R}$ is parent of B ;
/* compute edges between uncovered vertices and propagate up */
for $h \leftarrow \text{height}(\mathcal{T})$ **to** 0 **do**
  for *bag B s.t.* $\text{level}(B) = h$ **do**
    precompute-propagate$^{\text{FWD}}(B)$;
root $\mathcal{T}$ at $\mathcal{R}$;
return $(\mathcal{T}, \mathcal{B})$;

from B's children and combine them using the $\odot$ operator. Then, for each pair (v_1, v_2) we compute distances using the $\oplus$ operator between the edges $v_1 \to v$ and $v \to v_2$. Finally, the direct edge $v_1 \to v_2$ is combined to get the final probability distribution. At the final level – the root bag $\mathcal{R}$ – the computed pairwise distance distributions are simply copied to the edge set of $\mathcal{R}$. Note that we do not compute the distance distributions by using other possible paths between endpoints, and we restrict the computations only between the direct endpoint edge and the unique path going through the covered node. We do this to allow tractability of the convolution computations and allow the same semantics of the edges in $\mathcal{R}$, i.e., each resulting edge between endpoints can be independently sampled.

Unlike the SPQR tree approach, we cannot compute the bi-directional edges, at least for $w > 2$. Hence, only a single bottom-up propagation is made, from the leaves to $\mathcal{R}$. The resulting precompute-propagate$^{\text{FWD}}$ is similar to the SPQR version, and we omit it here.

Example 6.3. We give in Fig. 5 the result of applying Algorithm 4 on the graph in Fig. 1, for $w = 2$. The resulting decomposition consists of five bags in $\mathcal{B}$ and a root graph of two vertices, 6 and 0. Originally, the root graph does not contain any original edges, but it will have computed

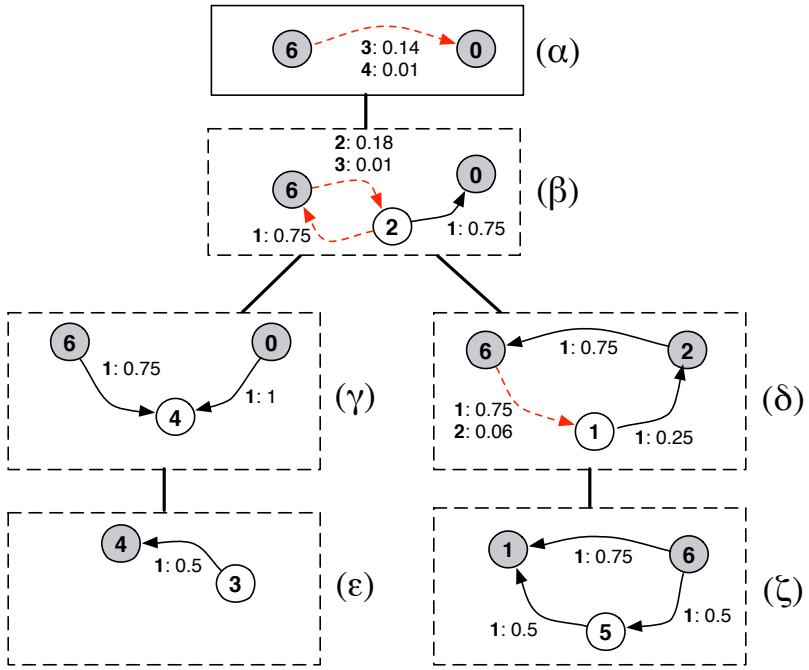


Fig. 5. The $w = 2$ decomposition of the example graph. Vertices in white are the vertices covered by each bag, and dashed red edges are edges which are computed from children. Each edge has a distribution of distance probabilities associated to it.

edges resulting from the bottom-up propagation. In the figure, the dashed red edges represent the edges which have been computed from the children.

On the left-hand side of the tree, bags (γ) and (ε) do not propagate any edges up the tree, as they either do not have 2 endpoints, as is the case of bag (ε), or there exist no paths between the endpoints, as for (γ). On the right-hand side, bag (ζ) will provide a $6 \rightarrow 1$ edge to bag (δ). Bag (δ) also propagates edges $6 \rightarrow 2$ and $2 \rightarrow 6$ to bag (β). Finally, bag (β) propagates edge $6 \rightarrow 0$ to the root bag (α).

In terms of time complexity, we know that computing the FWD itself is linear in the number of vertices in the graph [5, 47]. The computation of pairwise probability distributions, for each bag, is quadratic in w.

Proposition 6.4. *The complexity of precompute-propagate*$^{\text{FWD}}$ *is* $O(w^2 d)$*, where* d *is the maximum distance having non-zero probability in the graph.*

Proof. The number of endpoint pairs in a bag is $O(w^2)$. The computation of the SUM convolutions is quadratic in the maximum distance of each distribution, but cannot exceed d, which is bounded in connected graphs. The computation of the MIN convolution is linear in the maximum distance in the two distributions, and is upper bounded by d. The proposition follows. □

While it is conceivable that a possible world exists in which a shortest distance path between two vertices visits all edges in a graph thus having $d = \Omega(|E|)$, this does not occur in practice. Moreover, for $w \leqslant 2$ (a case which we will explore in more detail), there are only two pairs to

generate, and each bag is visited only once by Algorithm 4. Hence, in this setting, the complexity of propagating computations is linear in the number of vertices in the graph.

Retrieval. The retrieve operator is similar to the one applied for SPQR trees, with a single major difference. Since the bi-directional distance probabilities are not computed in the decomposition phase, we will not root $\mathcal{T}$ at a bag containing t. Instead, the edges from the bags containing s and t will always be propagated to $\mathcal{R}$, if they do not already belong to $\mathcal{R}$. Looking at Fig. 5 and for query $(1, 4)$, we would have to propagate the edges of bags (γ), (δ) and (β) to (α), resulting in the same equivalent graph as in Fig. 1.

The complexity of the retrieval for tree decompositions is the same as in the case of SPQR ProbTrees, i.e., linear in the size of the graph.

6.3 Analysis for $w \leqslant 2$

We first analyze in more detail FWDs for $w \leqslant 2$. We show that, in this special case, the computations performed by precompute-propagate$^{\text{FWD}}$ are correct:

Proposition 6.5. *precompute-propagate$^{\text{FWD}}$ computes correct probability distributions, i.e., does not induce any error, for decompositions of $w \leqslant 2$.*

Proof. A bag of size at most 2 has at most three vertices, the endpoints v_1, v_2 and the covered vertex v. $p(v_1 \to v_2)$ is uniquely defined by two paths: $v_1 \to v_2$ and $v1 \to v \to v_2$, resulting in $p(v_1 \to v_2)^{new} = p(v_1 \to v_2) \odot (p(v_1 \to v) \oplus (p(v \to v_2))$. Similarly, $p(v_2 \to v_1)^{new} = p(v_2 \to v_1) \odot (p(v_2 \to v) \oplus (v \to v_2))$. This can be computed exactly and efficiently, hence no error due to applying sampling or equivalent methods is induced.

Since we assume that all previous edges are independent, and none of the terms appear in both equations, it follows that $v_1 \to v_2$ and $v_2 \to v_1$ are independent. Their propagation to the parent maintains their independence and that of already present edges in the parent bag, their computed edges will also be independent. Hence no error is induced by not maintaining the independence property of edges.

Finally, the root bag $\mathcal{R}$ will only have already existing edges (which are independent by definition) or computed edges, which are independent, as shown above. It follows that all computed probabilities in the decomposition are exact. □

In addition, for $w \leqslant 2$ the decomposition defines a tree of independent subgraphs, i.e., a ProbTree. It follows that every computed edge in the root graph $\mathcal{R}$ corresponds to an independent subgraph.

Proposition 6.6. *Let $(\mathcal{T}, \mathcal{B})$ be a* FWD *ProbTree of $w \leqslant 2$. Then every bag B in $\mathcal{B}(\mathcal{T})$ defines an independent subgraph, having as endpoints its uncovered vertices and as internal vertices all covered vertices in the subtree of $\mathcal{T}$ rooted at B.*

Proof. A decomposition of $w \leqslant 2$ can only have at most 2 uncovered vertices in each bag. By definition, a covered vertex of a bag can only have links with the uncovered vertices of a bag, and hence the leaf bags in $\mathcal{T}$ define independent subgraphs of size 1.

For the bag above leaf vertices, we know again that the covered vertices can only have links with the uncovered vertices. These links can be from the original graph $\mathcal{G}$ or computed from children. The computed edges from children correspond to independent subgraphs themselves. Hence the covered vertex can only have links with other covered vertices or endpoints and thus is an internal vertex of an independent subgraph. □

Combining the previous results with the complexity bounds on the index$^{\text{FWD}}$ and retrieve$^{\text{FWD}}$ operators established in the previous section, we obtain that, for $w \leqslant 2$, (index$^{\text{FWD}}$, retrieve$^{\text{FWD}}$) is

a lossless indexing system. On the other hand, since few datasets have treewidth $\leqslant 2$, the FWD is generally not an optimal decomposition into independent subgraphs.

This lossless indexing system provide gains in efficiency close to those of the lossy SPQR indexes. In some cases – such as denser networks – their efficiency is still not fully satisfactory (see Section 8, Fig. 9). It is thus natural to consider FWDs for larger values of w.

6.4 Analysis for $w > 2$

Unfortunately, decompositions for $w > 2$ are not lossless, due to the correlations induced by pre-computing the distributions in bags, as witnessed by the following counter-example. Imagine a bag resulting from a $w > 2$ decomposition with covered vertex v and neighbor vertices $v_1, v_2, v_3, \ldots, v_w$ and the following edges: $v_1 \to v$ and $v \to v_2, \ldots, v \to v_w$. In this case, the computable edges would be $v_1 \to v_2, \ldots, v_1 \to v_w$. For every $1 < i \leqslant w$, $p(v_1 \to v_i) = p(v_1 \to v) \oplus p(v \to v_i)$. $p(v_1 \to v)$ appears in all equations, meaning that the computed edges would not be maintaining their independence, hence leading to lossy indexing. No guarantees can be obtained for them either, unlike in the case of SPQR.

As we show in the next section, we can use FWDs with $w > 2$ as a starting point to design an index structure that is lossless and improves on query time efficiency w.r.t. FWDs with $w \leqslant 2$, at the cost of an increase in space requirements, by representing explicitly the correlations introduced with higher treewidths.

7 LINEAGE TREES

As we have seen previously, for FWDs with $w > 2$, it is not generally possible to pre-compute the edges between endpoints in the decompositions bags, due to the correlations possibly introduced. This means that directly sampling the pre-computed edges is error-prone. Hence, sampling the pre-computed distance distributions directly from $\mathcal{R}$ or the graph returned by retrieve is not advisable.

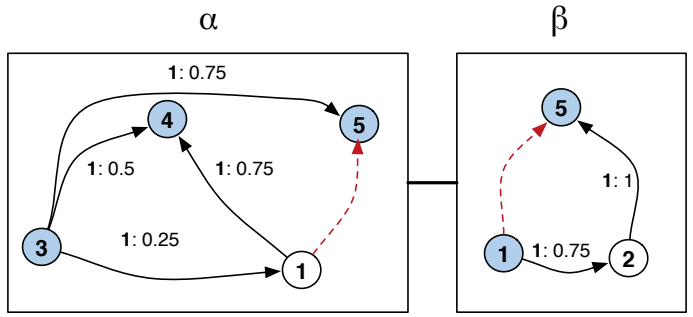
Instead, we can compute the full *lineage* of the distance distributions between endpoints at pre-processing, and leave the handling of the correlations at query time. For this, we compute the lineage at tree decomposition time and build a *lineage tree*, i.e., a parse tree of the path between endpoints.

Definition 7.1. A *lineage tree* of a probabilistic graph $\mathcal{G}$ is a binary tree whose leaves are labeled with pairs of nodes of $\mathcal{G}$ and whose internal nodes are labeled with either $\odot$ or $\oplus$.

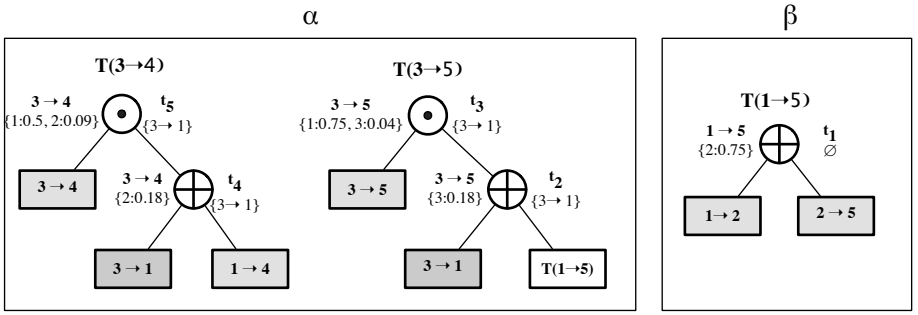
The *distance distribution* represented by a lineage tree is defined inductively given a distribution of distances on leaf nodes: the distributions of $\odot$- and $\oplus$-labeled internal nodes are given by MIN- or SUM-convolutions of probability distributions of children, following Fig. 3.

Such lineage trees, that will represent the actual distance distribution between two given nodes in a tree, can be efficiently computed at decomposition time by adding tree nodes "on top" of existing tree pointers, coming from previous bags. To enable efficient evaluation of edges which introduce correlations – hereby named *dependency edges* – we annotate each tree node T with a little information:

(i) the set of dependency edges $\mathsf{dependent}(T)$, i.e., the edges which introduce correlations in the entire subtree T;
(ii) the pre-computed distance distribution, $T.dst$, i.e., the distance distribution computed as if $\mathsf{dependent}(T) = \emptyset$; and
(iii) the edge being pre-computed, $T.edge$.



(a) Original decomposition



(b) Lineage trees

Fig. 6. Example of dependent path lineages and annotated lineage trees

Both $T.edge$ and $T.dst$ can be computed directly at decomposition time, just as in the previous decompositions, SPQR and FWD.

The dependent(T) computation goes on as follows. We first compute T with the dependency annotations. For each subtree t that originates from a previous bag in the tree decompositions, we union its dependent(t) to the current dependent(T). Then, for each bag processed in `precompute-propagate`$^{\text{FWD}}$ and for each distance distribution between endpoints, we keep the set of its lineage edges *only from the current bag*, linedges(T). Finally, for each pair of computed endpoint trees T_1 and T_2, we compute linedges(T_1) ∩ linedges(T_2) and add it to both dependent(T_1) and dependent(T_2), by set union. This ensures that each subtree will contain the correct set of dependency edges.

Example 7.2. Let us take the FWD decomposition, with $w = 3$, in Fig. 6a, composed of two bags α and β.

We wish to precompute the edges $3 \to 4$ and $3 \to 5$ in α. For that, we start at β and precompute $1 \to 5$ and its associated lineage tree, $T(1 \to 5)$, in Fig. 6b. It contains only one convolution node, t_1, and does not have any dependent edges because it belongs to a leaf bag of width $\leqslant 2$.

This lineage tree will be propagated to α and will help in the computation of $3 \to 4$ and $3 \to 5$ and their associated lineage trees, $T(3 \to 4)$ and $T(3 \to 5)$. $T(1 \to 5)$ is involved in the computation of $T(3 \to 5)$, as a pointer node that is a child of the convolution node t_2.

The edge $3 \to 1$ introduces a dependency between $3 \to 4$ and $3 \to 5$ because it appears in the pre-computation of several edges in the same bag, and hence the dependent edge annotation of $T(3 \to 4)$ and $T(3 \to 5)$ is $\{3 \to 1\}$.

ALGORITHM 5: propagate-dependent(T)

input : tree pointer T
output : distance distribution d

```
1  if dependent(T) = ∅ then
       /* subtree does not contain dependencies                    */
2      d ← T.dst;
3  else if T is a leaf then
       /* reached a leaf, sample the edge                          */
4      if T.edge is dependent then
5          if ∄sampled(T.edge) then
6              sampled(T.edge) ← sample(T.edge);
7          d ← sampled(T.edge);
8      else d ← dist(T.edge); ;
9  else
       /* evaluate branches                                        */
10     dl ← propagate-dependent(T.left);
11     dr ← propagate-dependent(T.right);
       /* compute convolutions                                     */
12     if T.oper = ⊕ then d ← dl ⊕ dr ;
13     else d ← dl ⊙ dr ;
14 return d;
```

The lineage trees described above can be added to computed edges of FWDs. Note that there is no need for lineage trees for FWDs with $w \leqslant 2$, as bags are always independent of other parts in the graph. Specifically, the only change that occurs is in Algorithm 2, `precompute-propagate`, and only for bags that have $w > 2$: instead of applying the convolution operators in lines 2, 8, and 9, we construct the lineage tree by adding the $\odot$ and $\oplus$ gates as needed. Hence, each edge for bags of $w > 2$ will be associated to a lineage tree. Then, at sampling time, each time a such an edge is encountered we evaluate the corresponding lineage tree, as detailed below.

Given a lineage tree on an edge, evaluating the distance distribution from a tree pointer T is done as in Algorithm 5. This algorithm is called at sample time for a tree pointer T. If the tree pointed by T does not contain any dependency edges, then we simply return the distance distribution $T.dst$. If, on the other hand, the pointer points to a leaf of the tree – which points to a graph edge – and this edge is a dependency edge, we need to sample it in this possible world. To ensure that we keep the correlation in all other possible trees which have this edge as a dependency edge, we need to ensure that the sampled distance is the same in all trees. For this we keep a map sampled which contains the sampled edges in the current possible world, ensuring no sampling of a dependency edge is repeated. If the edge is not a dependency edge, we can return its distance distribution. Finally, for intermediary tree nodes, we recursively evaluate the left and right branches and then compute the convolution indicated by the node, either $\oplus$ or $\odot$. The returned distance distribution d can be sampled by our sampler of choice.

Example 7.3. Let us return to the tree $T(3 \to 5)$ in Fig. 6b. At sampling time, edge $3 \to 1$ needs to be sampled because it is a dependency edge, i.e., it introduces a correlation with tree $T(3 \to 4)$.

When $T(3 \rightarrow 4)$ needs to be evaluated, we need to use this, previously sampled, distance for $3 \rightarrow 1$. Edge $3 \rightarrow 5$ and $T(1 \rightarrow 5)$ can use their distributions without sampling, as they do not have correlations anywhere else in the decomposition.

The problem with this lineage-based method, that we call LIN in what follows, is that it is not generally space-efficient. On each bag of width w, we only potentially remove $2w$ edges – two for each endpoint covered node pair –, while we can introduce $w(w-1)$ edges to the graph – one edge for all possible pairs of the w endpoints. For $w > 2$, this can add edges to the graph. We thus obtain a quadratic theoretical upper bound on the size of the resulting structure, though, as we will show, the blow-up is not nearly as bad in practice. Consequently, LIN does not satisfy our definition of an indexing system (Definition 3.7).

As we shall show in experiments, the number of computed edges added to $\mathcal{R}$ has a direct influence on query evaluation. Yet, as we will see in the next section (Fig. 9), LIN can achieve considerable increases in efficiency, even on dense graphs such as Wiki, where SPQR and FWD fare relatively poorly, meaning it still has good practical applicability.

8 EXPERIMENTAL EVALUATION

We now report on our experimental evaluation showing the efficiency of SPQR decompositions, FWDs, and LINs for ST-query evaluation over probabilistic graphs. Our experiments were performed on the graph datasets described below:

(1) The Wiki *social network* dataset, representing Wikipedia text interactions between contributors. Each edge has distance 1 and has associated as probability a value that is proportional to the number of positive interactions over the number of total interactions. Positive interactions represent text interactions which do not involve the deletion or replacement of another contributor's text, and edges in the graphs represent the probability that two authors agree on a topic. This can roughly be interpreted as a measure of editing similarity between users. The graph has 252,335 vertices and 2,544,312 edges.
(2) The Comm *communication* dataset, obtained from the SNAP website[3] representing the P2P connections between Gnutella hosts. Each edge is uniformly assigned a probability from $\{0.25, 0.5, 0.75, 1\}$, representing the probability that two hosts will establish a P2P connection. The graph has 62,561 vertices and 147,878 edges.
(3) The United States *road network* graphs[4], in which the edges represent roads between geographic locations, and have weights representing the average driving time. We have attached to each edge a (discretized) normal distribution whose mean is the driving time, and a standard deviation of 5% of the mean. We have experimented on two graphs, corresponding to roads of two US states: the Nh road network of 115,055 vertices and 260,394 edges, and the Ca road network of 1,595,577 vertices and 3,919,162 edges.
(4) The entire transport network of England, processed from XML dumps of OpenStreetMap[5], where edges represent roads, walkways, waterways or train lines between geographic locations, and weights represent the traversal time. The probabilities are directly proportional to the inverse of the traversal time. The resulting dataset, En, has 9,061,293 nodes and 15,305,006 edges.

Our ProbTree framework was implemented in C++, and all experiments were run on a Linux machine with a quad-core 3.6GHz CPU and 48 GB of RAM. The fixed-width decomposition algorithm

[3]http://snap.stanford.edu/data/p2p-Gnutella31.html
[4]http://www.dis.uniroma1.it/challenge9/data/tiger
[5]https://www.openstreetmap.org/

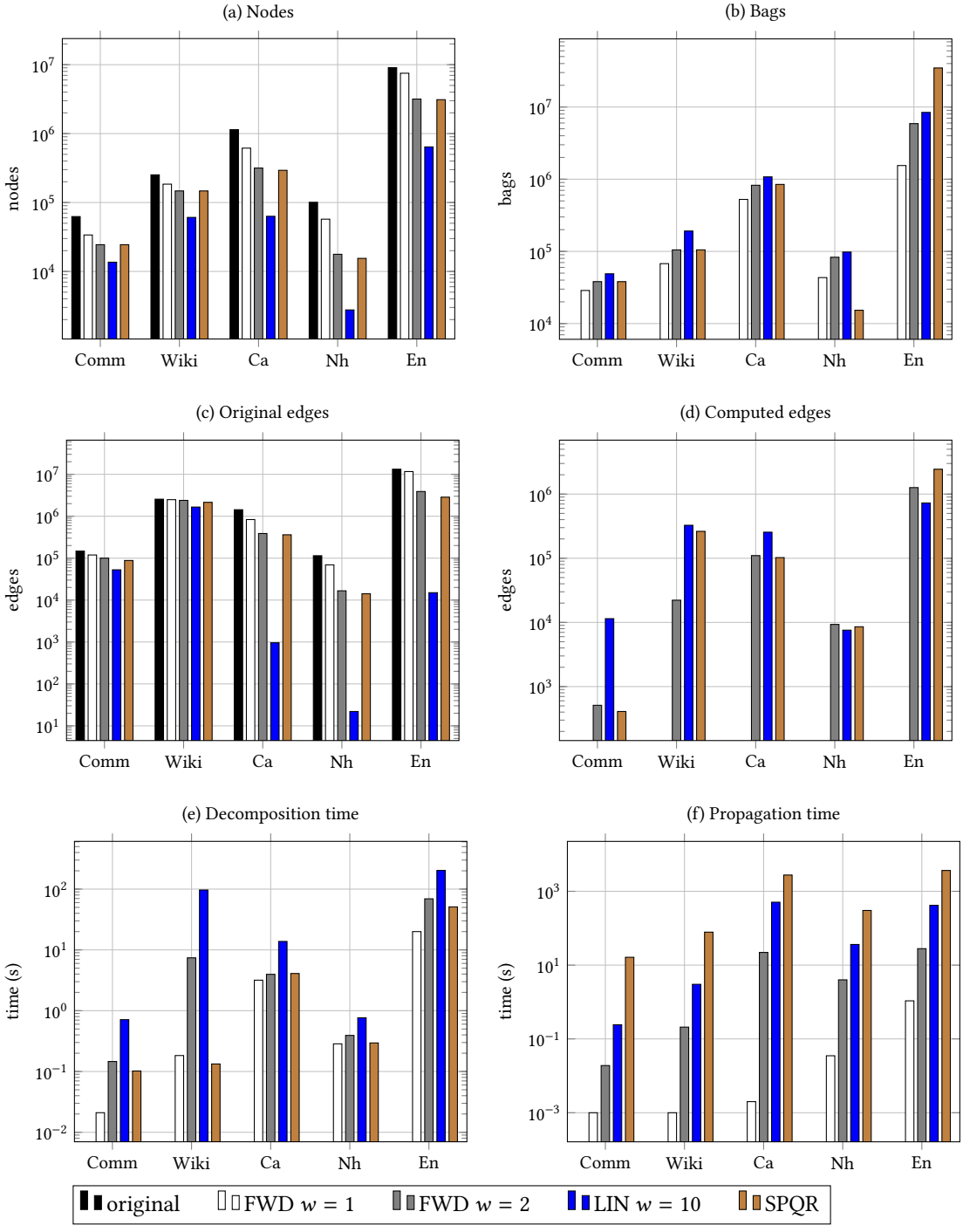


Fig. 7. ProbTree properties (log y-axis scale)

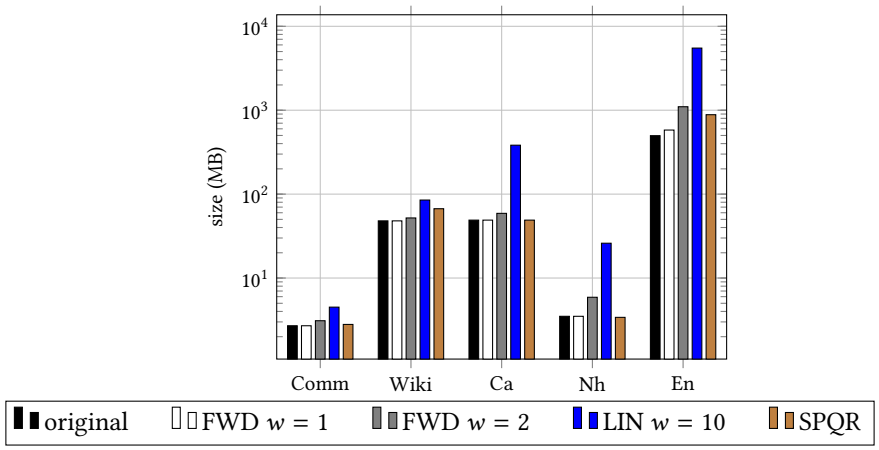


Fig. 8. ProbTree index size (log y-axis scale)

was implemented by us, while the deterministic part of the SPQR decomposition was done using the implementation in the Open Graph Drawing Framework library[6]. This implementation is an efficient implementation of the linear decomposition algorithm presented in [24].

ProbTree properties. For each dataset, we have generated both the lossless FWD, i.e., $w \in \{1, 2\}$, the SPQR ProbTrees, and the LIN decompositions for $w \in \{5, 10, 20\}$. For the R-bags of the resulting SPQR tree, we have computed the probabilities of the separation pairs by using 1,000 rounds of sampling. We have also generated SPQR ProbTrees using a different number of sampling round in the R-bags, but have noticed that the loss incurred by the samples remains relatively constant, for values over 1,000 samples. All time plots for the SPQR decomposition include the time taken for the sampling of R-bags. Moreover, for legibility, we only plot the LIN $w = 10$ for the decomposition properties and query times.

Fig. 7a-d illustrate the properties of the resulting indexes, from applying ProbTree on the four graphs. We show there the number of nodes in the root bag, the number of resulting bags, the number of original edges retained in the root, and the number of computed edges added to the root.

The number of bags seen in Fig. 7b represents the size of the decomposition tree without the root. This is equal to the number of independent graphs in the decomposed graph; note that this number depends on the decomposition one uses. Note that the number of nodes in the root (Fig. 7a) plus the number of bags (Fig. 7b) are always equal to the number of nodes in the original graph. As can be seen, the number of bags – which gives an indication on the number of independent graphs – represents a large percentage of the total number of nodes in the graph, regardless of the dataset. It is, however, generally higher in transport networks, for reasons we shall discuss later.

Fig. 7c (resp., Fig. 7d) shows the number of original (resp., computed) edges in the root. Ideally, we wish that the sum of the original edges and the computed edges is less than the original graph size. Ideally, for ProbTree to be efficient, we wish that the root bag is much smaller that the original graph – as we shall see, the sampling procedures directly depend on the number of edges. The number of vertices in the root decreases significantly with w, which can be explained by the fact that the graph degrees show long-tail distributions. A less pronounced effect is seen for the number of edges removed, especially in the case of the Wiki graph. This is also expected since even if one

[6]http://ogdf.net/

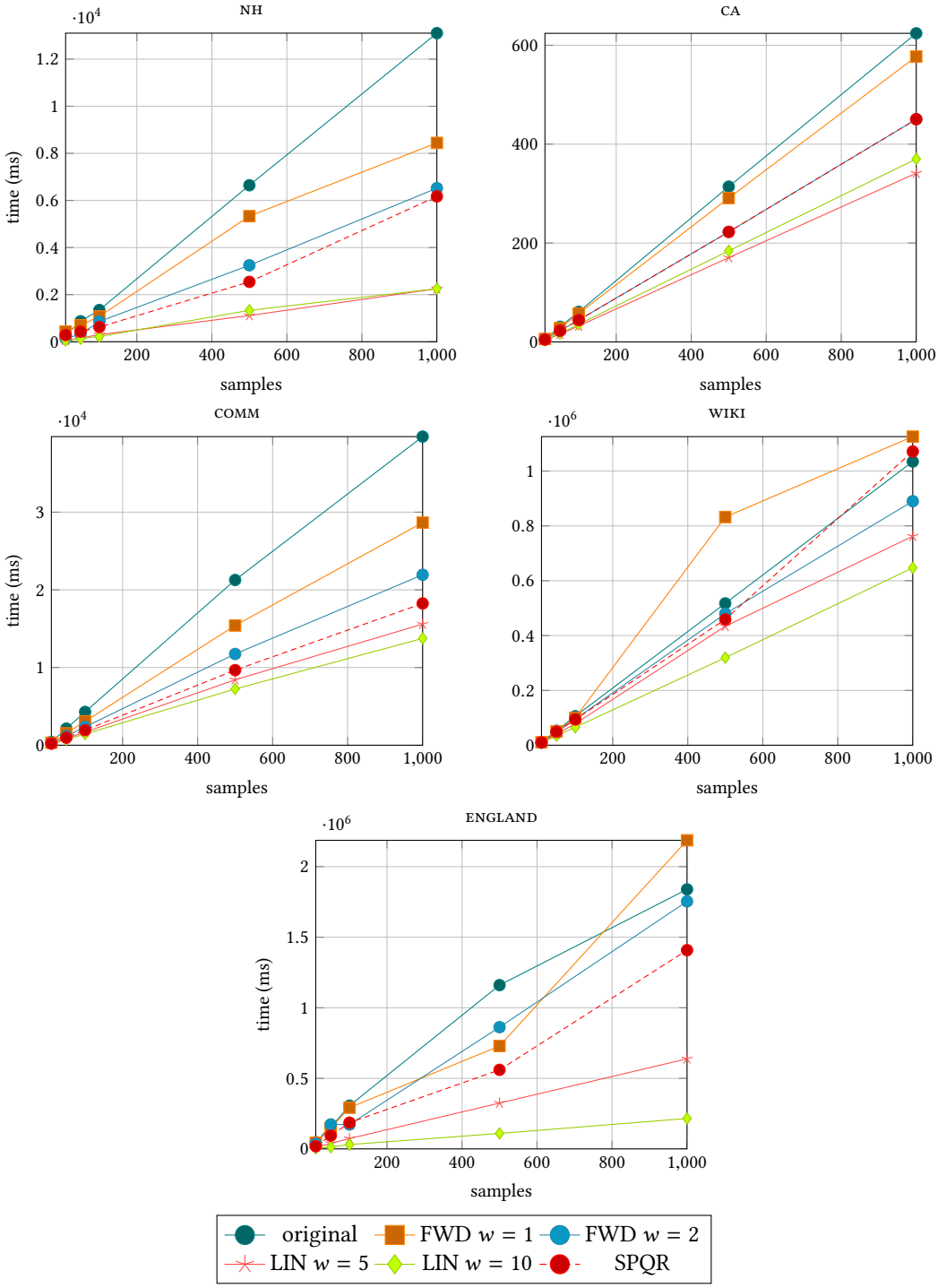


Fig. 9. Running time versus number of samples

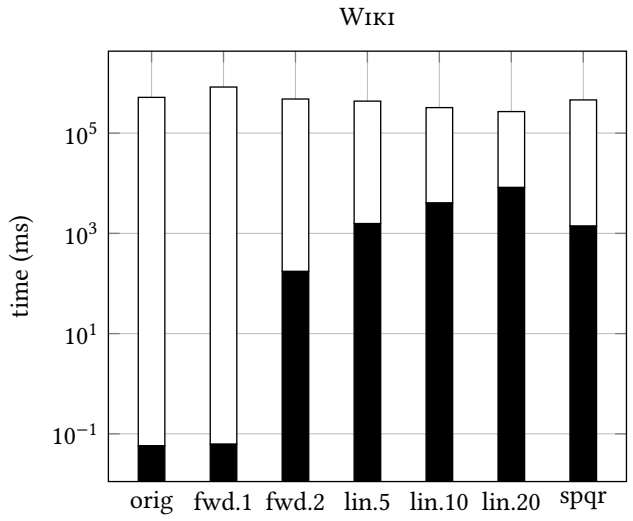


Fig. 10. Proportion of retrieve (black) out of the total query time (white) (log y-axis scale)

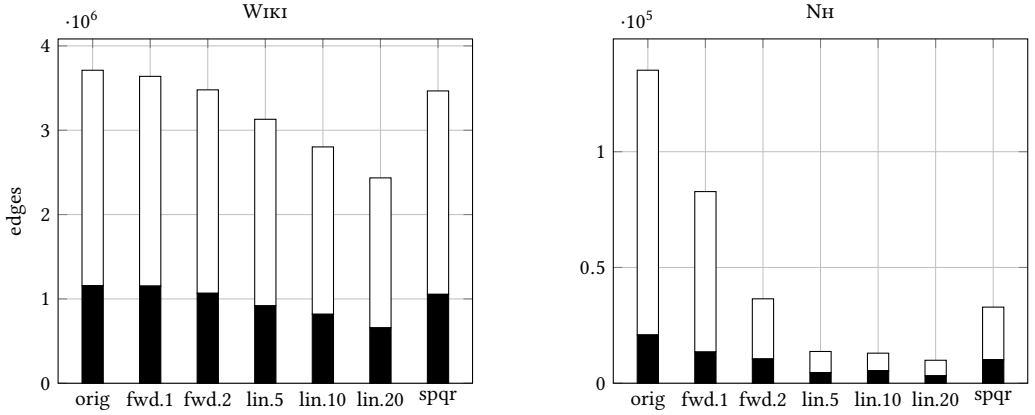


Fig. 11. Independent graph size (white) versus sampled edges (black).

removes the long tail of the degree distribution, the high-degree nodes – where most of the edges of the graph are concentrated – are still retained in the root. They, however, are always significantly lower than the original graph size. For the SPQR decomposition, it can be seen that the root[7] is smaller than the root of the largest lossless FWD decomposition, $w = 2$.

Interestingly, for the road graphs (En, Nh and Ca) and LIN, the corresponding roots contain very few original edges and are almost entirely composed of computed edges. We conjecture that this is due to the relative sparsity of the road network datasets as compared to the other datasets. Other possible explanations may come from the near-planar character of the road networks – although this has no bearing on whether the treewidth is bounded – and their low *highway dimension* [3].

[7]In the SPQR case we define the root as the largest bag in the tree.

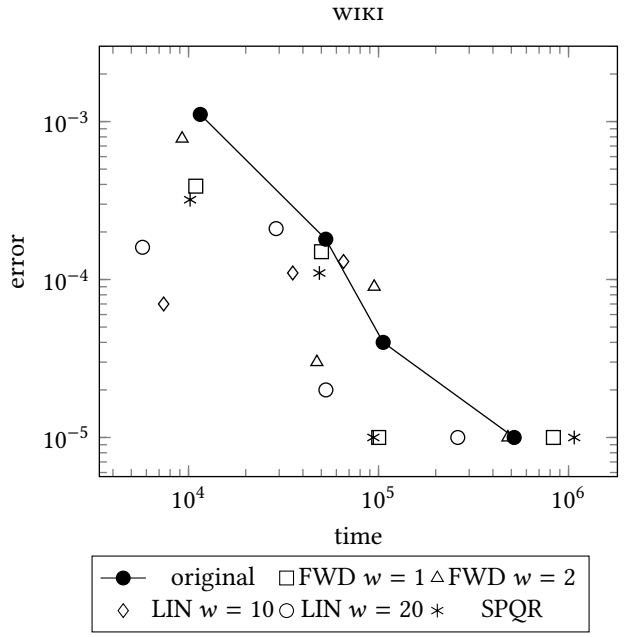


Fig. 12. Relative error vs. time (log-log axis)

Fig. 7e-f shows the preprocessing execution time of ProbTree. As can be noticed, the index operation is very efficient, running in the order of seconds even on large graphs, except the LIN computation of the dense Wiki graph, which takes around 2 minutes. The same observations hold for the pre-computation and propagation of distance distributions for the fixed-width decompositions. However, due to the overhead of sampling the R-bags, the SPQR pre-computation takes significant more time than FWD, but still under an hour. The exception to this behaviour are the LIN decompositions of the road networks, where the distance propagation can take a few hours, due to the computation of the lineage trees.

The space overhead of $\mathcal{I}$ (Fig. 8) is also reasonable. Generally, the ProbTree for FWD $w \leqslant 2$ and SPQR only incurs between 10% (Wiki) and double (Nh) space overhead compared to the space cost of the original graph. As expected, the LIN decompositions for the road networks increase significantly in size compared to their original graph size, even reaching a few gigabytes in the case of Ca and En, and up to 10 times in the size of the original graph in the case of En. Since the higher widths of the tree decompositions no longer retain the linear size increase property, i.e., can be quadratic in the original size, this is not surprising. Nevertheless, the query time savings of the lossless decompositions of En, Nh and Ca are very important, as we shall see next.

Running time. For evaluating the execution time, we used the following experimental setup. For each dataset, a randomly generated query workload of 1,000 vertex pairs from the original graphs were generated. For a workload of 1,000 queries, the standard error for the running time and a 95% confidence interval is ±3%; this significance level is reached for every comparison between the running time of the baseline algorithm and the running time on the different decompositions we used.

For each query workload, we generated the ground truth probabilities via 10,000 rounds of sampling. Please note that for each query pair we generated the actual distance distribution

between the vertices, by applying Dijkstra's shortest path algorithm from the source vertex, on each sampling round. For testing, we executed the workloads for a number of samples between 10 and 1,000.

As Fig. 9 shows, the efficiency gains are important when queries are executed on ProbTree indexes. The gains on the lossless decompositions are up to 2–3 times in the case of NH. In most cases, SPQR is more efficient that the lossless FWDs, but only marginally so. Denser networks, such as WIKI, do not have such an important increase in efficiency for lossless decompositions. For that kind of networks, the most efficient is to use LIN decompositions, which can achieve a two-fold increase in efficiency. LIN even achieves 10-fold increases in efficiency for the road datasets, especially EN and NH. For the reason that WIKI is the worst-performing of the datasets, we proceed to evaluate it next more closely under the lens of query times, of the impact of RETRIEVE, and of accuracy. As we shall see, WIKI can benefit from LIN, with no sacrifice in accuracy.

We next explain how the retrieve operator time influences the execution of the queries, in Fig. 10 (note the log y-axis scale). The bars represent the average query time, using 500 samples on the WIKI graph. Out of this time, a portion is spent on the retrieve operation, represented by the black bars in the figure. This operation, corresponding to the retrieval of equivalent graphs, does not take a significant time out of the total execution time. In the worst case, it is roughly 1% of the execution time. Hence, the sampling time greatly dominates the query time and applying retrieve is highly efficient. Note that since we count the retrieve time as part of the total execution time, we do not assume that retrieved graphs are kept in any way. Each time a query is executed, an equivalent graph is retrieved and is then discarded after query evaluation.

To understand more about the difference in running time savings of the decomposition, we plot, in Fig. 11, the size of the equivalent graph $\mathcal{G}(q)$ resulting from applying retrieve for each type of decomposition (white bars), along with the proportion of actual sampled edges in these graphs (black bars). The sizes represent average sizes over all 1,000 queries in the query load. In the case of WIKI, we observe that the $\mathcal{G}(q)$ graphs are relatively close in size to the original graphs. Generally, the proportion of actually sampled edges in the graph remains constant (around 40% in the case of WIKI, and lower in the case of NH); this means that, indeed, reducing the number of edges in the equivalent graphs can reduce query times. The large decrease in the query times for the transport networks – such as the illustrated NH – can be directly attributed to their efficient decompositions, resulting in relatively more independent graphs, which in turn result in much smaller equivalent graphs.

Error vs. time. The question we wish to answer now is the following: Are such approaches better overall than sampling algorithms? That is, is the error vs. time trade-off – especially for SPQR and LIN – enough to justify using our algorithms, and not simply use more sampling rounds? To check this, we have plotted the running time of applying sampling on ProbTree versus its error – expressed in terms of the mean squared error as compared to the ground truth results. For brevity, we only track the results for the reachability – or 2-terminal reliability – queries. As query answers are derived directly from the distance distribution, results for other types of queries have similar relative error results.

Fig. 12 presents the results for the WIKI graph (note the log-log axes). The black dots represent the results on sampling the original graph, for a number of sample rounds between 10 (top left) and 1,000 (bottom right). Intuitively, we want the points corresponding to ProbTree variants (drawn for the same amount of samples) to lie "below" the line induced by the black points, meaning that they yield a better time-accuracy trade-off. As seen before, the gains in execution time when using the decompositions are important. The results also show that the relative error can be even slightly improved when using ProbTree. For instance, note that the FWD and LIN errors in the WIKI graph

Table 2. Reachability relative error ($\times 10^{-5}$) for selected number of samples

	samples		
Decomp.	50	100	1,000
orig.	18	4	1
SPQR	11	1	1
FWD, $w = 2$	15	9	1
LIN, $w = 20$	21	2	1

Table 3. Distance-constraint reachability running time / sample (ms) for d-RQ estimators

	$d = 3$		$d = 4$		$d = 5$	
Decomp.	RHH	RHT	RHH	RHT	RHH	RHT
orig.	5	4	6	6	7	7
SPQR	3	2	4	2	5	3
FWD, $w = 2$	3	2	4	3	5	4
LIN, $w = 10$	2	1	2	1	2	1

are slightly lower than the corresponding black dots, i.e., the original graph, suggesting an increase in accuracy.

To make this clearer, we present in Table 2 the errors for selected decompositions. It can be seen that indeed the decomposition may be both more efficient and more effective for distance queries. This may occur because computed edges get sampled only once – as opposed to their original component edges –, which minimizes the propagation of sampling errors. Another interesting result is that 1,000 samples seem to be enough to make the error near-zero.

Comparison with other algorithms. One of our arguments in using ProbTree as a pre-computed index is that it can be directly applied to existing solutions. To check this, we apply the distance-constraint reachability (d-RQ) estimators studied in [29] to the FWD, SPQR and LIN versions of the Wiki graph. We use the advanced samplers from [29] – RHH, RHT – and apply directly the authors' implementation. We use the same experimental setup as [29], and we transform the $\mathcal{G}(q)$ versions of the input graphs into their edge-existential versions to serve as direct input to d-RQ algorithms, by varying $d \in \{3, 4, 5\}$.

Table 3 summarizes the results. First of all, it can be easily noted that, indeed, applying ProbTree decompositions directly affects the running time of any of the three estimators, up to a 7-times increase in efficiency, for the RHT estimators. On both estimators, the FWD and SPQR estimator are comparable in running time to DCR on the original graph. On the other hand, LIN gets an increased efficiency on the advanced estimators, even more than on the simple estimator used in the previous experiments – which corresponds to Dagger sampling. This shows that the significantly smaller size of the LIN graphs – even in Wiki –, combined with the increased theoretical efficiency of RHH and RHT – as proven in [29] – can lead to very efficient query processing.

9 CONCLUSIONS

In this paper, we studied efficient ST-query evaluation in probabilistic graphs. We formally define an indexing framework on such graphs, and propose the ProbTree, with two variants: the SPQR tree

and the FWD. SPQR trees have the advantage of an optimal decomposition, enabling higher query efficiency, at the cost of being lossy; FWDs, with $w = 2$, are lossless, and achieve good performance on real-world datasets, especially when graphs are sparse. To achieve further efficiency, we show how FWDs can be enriched with lineage information to return sound query results; the downside is a theoretical quadratic blow-up, which in practice rarely happens. The graphs produced can also be easily used by existing query algorithms, and we show how pre-processing based on ProbTree can increase efficiency and accuracy in state-of-the art probabilistic query processing algorithms. In the future, we will develop query-efficient representations for other kinds of queries (e.g., k-nearest neighbors [39] and frequent subgraph discovery [49]).

ACKNOWLEDGMENTS

Reynold Cheng and Silviu Maniu were supported by the Research Grants Council of HK (Project HKU 17205115 and 17229116) and HKU (Projects 102009508 and 104004129). We would like to thank the reviewers for their insightful comments.

REFERENCES

[1] Juancarlo Añez, Tomás De La Barra, and Beatnz Pérez. 1996. Dual graph representation of transport networks. *Transportation Research Part B: Methodological* 30, 3 (1996).
[2] Serge Abiteboul, T.-H. Hubert Chan, Evgeny Kharlamov, Werner Nutt, and Pierre Senellart. 2011. Capturing continuous data and answering aggregate queries in probabilistic XML. *ACM Trans. Database Syst.* 36, 4, Article 25 (2011), 45 pages.
[3] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. 2010. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *SODA*.
[4] Eytan Adar and Christopher Re. 2007. Managing Uncertainty in Social Networks. *IEEE Data Eng. Bull.* 30, 2 (2007).
[5] Takuya Akiba, Christian Sommer, and Ken-ichi Kawarabayashi. 2012. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *EDBT*.
[6] Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. 2015. Provenance circuits for trees and treelike instances. In *ICALP*.
[7] Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. 2016. Tractable lineages on treelike instances: Limits and extensions. In *PODS*.
[8] Stefan. Arnborg, Derek G. Corneil, and Andrzej Proskuworski. 1987. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods* 8, 2 (1987).
[9] Stefan Arnborg and Andrzej Proskurowski. 1989. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics* 23, 1 (1989).
[10] Robert B. Ash and Catherine A. Doléans. 1999. *Probability & Measure Theory* (2nd ed.). Academic Press.
[11] Saurabh Asthana, Oliver D. King, Francis D. Gibbons, and Frederick P. Roth. 2004. Predicting Protein Complex Membership Using Probabilistic Network Reliability. *Genome Research* 14, 6 (2004).
[12] Michael O. Ball. 1986. Computational Complexity of Network Reliability Analysis: An Overview. *IEEE Trans. Reliability* 35, 3 (1986).
[13] Pablo Barceló, Leonid Libkin, and Juan L. Reutter. 2014. Querying Regular Graph Patterns. *J. ACM* 61, 1, Article 8 (2014), 54 pages.
[14] Hans L. Bodlaender. 1996. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM J. Comput.* 25, 6 (1996).
[15] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In *KDD*.
[16] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003).
[17] Nilesh N. Dalvi and Dan Suciu. 2007. Efficient query evaluation on probabilistic databases. *VLDB J.* 16, 4 (2007).
[18] Giuseppe Di Battista and Roberto Tamassia. 1990. On-Line Graph Algorithms with SPQR-Trees. In *ICALP*.
[19] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959).
[20] Pedro Domingos and Matthew Richardson. 2001. Mining the network value of customers. In *KDD*.
[21] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query preserving graph compression. In *SIGMOD Conference*.
[22] George S. Fishman. 1986. A Comparison of Four Monte Carlo Methods for Estimating the Probability of s-t Connectedness. *IEEE Trans. Reliability* 35, 2 (1986).

[23] Joy Ghosh, Hung Q. Ngo, Seokhoon Yoon, and Chunming Qiao. 2007. On a Routing Problem Within Probabilistic Graphs and its Application to Intermittently Connected Networks. In *INFOCOM*.
[24] Carsten Gutwenger and Petra Mutzel. 2000. A Linear Time Implementation of SPQR-Trees. In *Graph Drawing*.
[25] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. 2005. Compact reachability labeling for graph-structured data. In *CIKM*.
[26] John E. Hopcroft and Robert Endre Tarjan. 1973. Dividing a Graph into Triconnected Components. *SIAM J. Comput.* 2, 3 (1973).
[27] John E. Hopcroft and Robert Endre Tarjan. 1973. Efficient Algorithms for Graph Manipulation [H] (Algorithm 447). *Commun. ACM* 16, 6 (1973).
[28] Ming Hua and Jian Pei. 2010. Probabilistic path queries in road networks: traffic uncertainty aware path selection. In *EDBT*.
[29] Ruoming Jin, Lin Liu, Bolin Ding, and Haixun Wang. 2011. Distance-Constraint Reachability Computation in Uncertain Graphs. *PVLDB* 4, 9 (2011).
[30] Bhargav Kanagal and Amol Deshpande. 2010. Lineage processing over correlated probabilistic databases. In *SIGMOD Conference*.
[31] Arijit Khan, Francesco Bonchi, Aristides Gionis, and Francesco Gullo. 2014. Fast Reliability Search in Uncertain Graphs. In *EDBT*.
[32] Michihiro Kuramochi and George Karypis. 2001. Frequent Subgraph Discovery. In *ICDM*.
[33] David Liben-Nowell and Jon M. Kleinberg. 2007. The link-prediction problem for social networks. *JASIST* 58, 7 (2007).
[34] Silviu Maniu, Bogdan Cautis, and Talel Abdessalem. 2011. Building a Signed Network from Interactions in Wikipedia. In *Databases and Social Networks (DBSocial, SIGMOD '11)*. 19–24.
[35] Silviu Maniu, Reynold Cheng, and Pierre Senellart. 2014. ProbTree: A Query-Efficient Representation of Probabilistic Graphs. In *Proc. BUDA*. Workshop without formal proceedings.
[36] Mark E. J. Newman, Steven H. Strogatz, and Duncan J. Watts. 2001. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev. E* 64, Article 026118 (Jul 2001), 17 pages. Issue 2.
[37] Christos H. Papadimitriou. 1994. *Computational Complexity*. Addison Wesley Pub. Co., Reading, USA.
[38] Odysseas Papapetrou, Ekaterini Ioannou, and Dimitrios Skoutas. 2011. Efficient discovery of frequent subgraph patterns in uncertain graph databases. In *EDBT*.
[39] Michalis Potamias, Francesco Bonchi, Aristides Gionis, and George Kollios. 2010. k-Nearest Neighbors in Uncertain Graphs. *PVLDB* 3, 1 (2010).
[40] Neil Robertson and Paul D. Seymour. 1984. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B* 36, 1 (1984).
[41] Mohammad Ali Safari. 2005. D-Width: A More Natural Measure for Directed Tree Width. In *MFCS*.
[42] Prithviraj Sen and Amol Deshpande. 2007. Representing and Querying Correlated Tuples in Probabilistic Databases. In *ICDE*.
[43] Stephan Seufert, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum. 2013. FERRARI: Flexible and Efficient Reachability Range Assignment for Graph Indexing. In *ICDE*.
[44] Asma Souihli and Pierre Senellart. 2013. Optimizing approximations of DNF query lineage in probabilistic XML. In *ICDE*.
[45] William T. Tutte. 1966. *Connectivity in graphs*. Mathematical Expositions, Vol. 15. University of Toronto Press.
[46] Leslie G. Valiant. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.* 8, 3 (1979).
[47] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *SIGMOD Conference*.
[48] Ye Yuan, Guoren Wang, Haixun Wang, and Lei Chen. 2011. Efficient Subgraph Search over Large Uncertain Graphs. *PVLDB* 4, 11 (2011).
[49] Zhaonian Zou, Hong Gao, and Jianzhong Li. 2010. Discovering frequent subgraphs over uncertain graph databases under probabilistic semantics. In *KDD*.