An Indexing Framework for Queries on Probabilistic Graphs

Silviu Maniu
The University of Hong Kong
Hong Kong SAR, China
smaniu@cs.hku.hk

Reynold Cheng
The University of Hong Kong
Hong Kong SAR, China
ckcheng@cs.hku.hk

Pierre Senellart
Institut Mines-Télécom -
Télécom ParisTech
Paris, France
pierre@senellart.com

ABSTRACT

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) many samples are needed for reasonable accuracy, and (ii) a possible world can be very large. To tackle these issues, we study the *PTree*, 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 PTree, 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 PTree is fast to generate and small in size. It also enhances the accuracy and efficiency of existing ST-query algorithms significantly.

1. INTRODUCTION

Graph data are prevalent in many important and emerging applications. In online social networks, such as LinkedIn and Facebook, friends are interconnected to form complex social networks [17]. Mobile devices form ad-hoc networks through Wi-Fi technologies [20]. In a road network, cities are composed of streets and are connected by highways [1]. In biological networks, proteins interact with each other in a complex manner [7]. Substantial research has been devoted to the effective processing of graph queries, including reachability [12], shortest paths [16], frequent subgraphs [29], and graph patterns [9].

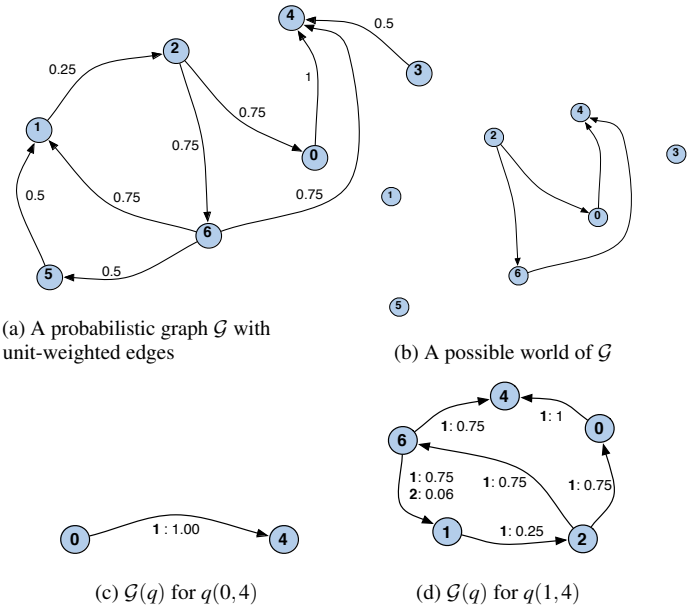
Data uncertainty is inherent in the applications above. For example, viral marketing techniques [30] study the purchase behavior of users in a social network. They study influence graphs, which depict the purchase influence among people, represented as graph vertices. A directed edge from Mary to John, for example, indicates that John's purchases are influenced by those of Mary. An edge on the influence graph is unlikely to be a definite relationship, for John may not always follow Mary's purchase behavior [2]. 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 [20]. Due to hardware limitation, measurement errors also occur in biological databases (e.g., protein-to-protein interaction [7]) and road monitoring systems (e.g., traffic congestion data [25]). 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* [38, 8, 19, 26, 25]. 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 (Figure 1a). This model captures reliability and failure in computer network connections [19, 26], and it can also represent uncertainty in social and biological networks [7]. In the *weight-distribution model*, each edge is associated with a probability distribution of weight values [25]. 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 a large probabilistic graph has not been addressed until recently. Some representative works include finding shortest paths and reliability estimation [19, 26], searching nearest neighbors [33], and mining frequent subgraphs [41]. 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 source vertex s and target vertex t in a probabilistic graph. In general, an ST-query is about getting information about paths that start from s and finish at t. 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 distance between two vertices, zsh:1: aucun fichier ou dossier de ce type: /This distances.

Evaluating an ST-query can be expensive. This is because these queries, operated on a a probabilistic graph $\mathcal{G}$, follow the *possible world semantics* (PWS) [13]. Conceptually, $\mathcal{G}$ encodes a set of possible worlds, each of which is a definite (non-probabilistic) graph itself. Figure 1b shows a possible world of the probabilistic graph in Figure 1a. Each possible world is given a probability of its existence derived from edge probabilities. For example, the graph in Figure 1b exists only if edges $0 \to 4$, $2 \to 0$, $2 \to 6$, and $6 \to 4$ exist, with a probability of 0.1.[1] Evaluating a query q (e.g., an SDQ) on $\mathcal{G}$

[1] This is the product of the probabilities that edges in Figure 1b exist, and the probabilities that other edges do not, i.e., $1 \times 0.75 \times 0.75 \times$



(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 $q(1,4)$

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

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 of possible worlds; and indeed the problem has been proved to be #P-hard [38, 8, 13].

To improve ST-query efficiency, researchers have proposed *sampling solutions* [19, 26, 33, 41], where possible worlds with high existential probabilities are extracted. These algorithms, which examine fewer possible worlds than the PWS, 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 over 250k vertices and 2.5 million 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.

Issue 2. To achieve high accuracy, a lot of possible world samples may need to be generated. In our experiments, around 1,000 samples are required to converge to near-zero errors.

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. Let us consider an RQ, $q(0,4)$, run on the graph in Figure 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. 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. Figure 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 paths: one with length 1 and probability 0.75, and the other with length 2 and probability 0.06.

In the examples above, $\mathcal{G}(q)$ is smaller than $\mathcal{G}$. The possible world graphs sampled from $\mathcal{G}(q)$ become smaller than those generated from $\mathcal{G}$. If $q(s,t)$ is executed on the possible worlds sampled from $\mathcal{G}(q)$, its efficiency 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 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 *PTree*, an indexing framework that facilitates ST-query execution. Given $q(s,t)$, PTree efficiently generates the corresponding $\mathcal{G}(q)$, which has a small size. In this way, the speed of q, evaluated on $\mathcal{G}(q)$, can be improved. We formalise three design requirements for PTree: (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 show theoretically that for these criteria to be satisfied, PTree needs to have a tree structure.

We next investigate the construction of PTree. Specifically, we study the *SPQR tree* [14, 21] and the *fixed-width tree decomposition* (FWD) [34], which are query-efficient structures for traditional, non-probabilistic graphs. By appropriately incorporating uncertainty information into these structures, two implementations of PTree can be obtained. We study the efficiency and accuracy of these two structures. It turns tout that FWDs allow an ST-query to be answered correctly. However, they are not as efficient as SPQR trees, which on the other hand may introduce bounded error in the query result, and we say that the SPQR tree is *lossy*. For both structures, the construction and retrieval times, as well as the space overhead, are linear to the size of $\mathcal{G}$. We further show that the efficiency of FWD can be enhanced without generating lossy query results, at the expense of occupying more 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* [19, 26] and *weight-distribution* [25]. Moreover, any existing ST-query algorithm (e.g., [26]) can be executed on a graph retrieved from a PTree without any modification. Table 1 summarizes the main properties of our three approaches.

Table 1: Summary of the PTree Structures.

PTree	Space	Time	Query Quality	Section
SPQR	linear	linear	lossy (with bound)	5
FWD($w=2$)	linear	linear	lossless	6.1
LIN	quadratic	linear	lossless	6.2

We have evaluated our approaches on four real-world large datasets. All our solutions significantly improve the performance of existing ST-query algorithms. For instance, using SPQR trees and FWDs in a state-of-the-art distance-constraint reachability algorithm [26] achieves speedup of 2 to 3 times, with query accuracy improved by up to 50%. The LIN structure achieves the highest efficiency, with reasonable space overhead in practice.

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 PTree framework. We present the details of PTree in Section 4. In Sections 5 and 6, we examine SPQR trees, FWD, and LIN. We present our experimental results in Section 7.

2. RELATED WORK

$0.75 \times (1-0.5) \times (1-0.25) \times (1-0.75) \times (1-0.5) \times (1-0.5)$.

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 [19, 26]), k-nearest neighbors [33], and frequent subgraph discovery [41]. Our PTree index generates a small probabilistic graph for querying purposes. This graph renders smaller possible world sizes, as well as fewer samples. It would be interesting to extend PTree to support k-nearest neighbor query and frequent subgraph discovery. We remark that the issues of indexing probabilistic graphs have only been recently studied. In [40, 32], an indexing solution was proposed for subgraph retrieval. A pruning and indexing framework for reliability search has been proposed in [28]. It is not clear how they support a general ST-query, which is studied intensively in our paper.

Tree decompositions. While tree decompositions have been used to generate indexes for efficient shortest-path-query execution [39, 3], 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 does not hold for distances in probabilistic graphs. In this paper, we show how to use tree decompositions to support probabilistic graph queries. In [27], the authors study how to use junction tree decompositions to evaluate queries in correlated databases [35]. 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.

Graph compression. Given a probabilistic graph $\mathcal{G}$ and a ST-query q, Our PTree can generate $\mathcal{G}(q)$, which is smaller than $\mathcal{G}$. Hence, our approach can be considered in some sense as a *graph compression* algorithm. 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 [11, 22, 18]). However, these solutions are not designed for probabilistic graphs. To our best knowledge, no other work has studied how to compress a probabilistic graph. In this paper, we examine how aprobabilistic graph can be compressed in lossless and lossy manners.

3. INDEXING PROBABILISTIC GRAPHS

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

DEFINITION 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 vertice set, the edge set, and the probability assignment function of $\mathcal{G}$ respectively.

Note that for an edge e, the probability that it does not exist in $\mathcal{G}$ is $1 - \sum_{w \in \mathrm{supp}(p(e))} p(e)(w)$. Definition 1 is essentially the *weight-distribution model* [25], where each edge is associated with a finite probability distribution of weights. This definition also captures the *edge-existential model* [19, 26], 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. It would be interesting to study how to extend our solution to support edge correlations.

DEFINITION 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:*

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

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

PROPOSITION 1. *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* $\sum_{G \in \mathrm{PW}(\mathcal{G})} \Pr[G] = 1$.

ST-queries. In this paper, we study the *source-target distance* query (or *ST-query*), 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) [12]. Probability that t is reachable from s.

Distance-constraint reachability (d-RQ) [26]. Probability that t is reachable from s within distance d.

Expected shortest distance (SDQ) [8]. 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 $p(s \to t)$. Then, any of these queries can be derived from (d_i, p_i). Unfortunately, these queries are hard to evaluate, as stated below:

THEOREM 1 ([38, 8]). *Evaluating RQ, d-RQ (for $d \geqslant 2$), and SDQ is* $\mathsf{FP}^{\#\mathsf{P}}$*-complete.*

Without loss of generality, in the rest of the paper we assume that the answer of a 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. *A* probabilistic graph transformation system *is a pair* $(\mathsf{index}, \mathsf{retrieve})$ *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*;*
- $\mathsf{retrieve}$ *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). 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 time and space cost of evaluating q on $\mathcal{G}(q)$. We formalize these properties below.

DEFINITION 4. *Let* $(\mathsf{index}, \mathsf{retrieve})$ *be a probabilistic graph transformation system. Given a probabilistic graph $\mathcal{G} = (V, E, p)$,*

and a class of ST-queries $\mathcal{Q}$ (e.g., RQs), the transformation loss of $(\mathsf{index}, \mathsf{retrieve})$ *on $\mathcal{G}$ for $\mathcal{Q}$ is:*

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

where $q^{\mathcal{G}}$ is the result of q evaluated on $\mathcal{G}$. A transformation system $(\mathsf{index}, \mathsf{retrieve})$ *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 above definition 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 5. *A transformation system* $(\mathsf{index}, \mathsf{retrieve})$ *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 $\mathcal{G}$, $|\mathsf{index}(\mathcal{G})| = \mathsf{O}(|\mathcal{G}|)$ (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 precomputes all pairwise results, i.e., the index operator. This system satisfies Property (iii), since the $\mathsf{retrieve}_q$ builds a trivial two-vertex graph. Evaluating q over the resulting graph is very efficient, since this just involves looking up the distance probability distribution on the edges of this graph, in $\mathsf{O}(1)$ time. However, neither Property (i) nor (ii) hold, since indexing is intractable unless $\#\mathsf{P}$ is tractable, which would imply $\mathsf{P} = \mathsf{NP}$; 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 $\mathsf{retrieve}_q$ on $\mathsf{index}(\mathcal{G})$, together with the running time of q on $\mathcal{G}(q)$, should be faster than evaluating q on $\mathcal{G}$. One such indexing system is the PTree, as presented next.

4. INDEPENDENCE AND PTREE

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 *PTree*. The PTree 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 6. *An* independent subgraph *of a probabilistic graph $\mathcal{G}$ is a connected induced subgraph $S \subseteq \mathcal{G}$ with arbitrarily many* internal vertices *and at most two* endpoint vertices *v_1, v_2 such that:*

(i) Each internal vertex is connected only to other internal vertices of S, or to the endpoint vertices, in an undirected manner;

(ii) the endpoint vertices can be linked to other vertices in $\mathcal{G}$, to internal vertices, and to themselves.

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 7. *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 integers that gives for every tuple (d_{ij}), where d_{ij} is a real valued distance, the probability* $\Pr\left[\bigwedge_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant n}} v_i \to v_j \text{ has length } d_{ij}\right]$.

The above characterizes the semantic 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. A fundamental result is the following: Independent subgraphs are exactly those that can be removed from the graph while preserving joint distance probability distributions for non-removed vertices.

THEOREM 2. *Let $\mathcal{G} = (V, E, p)$ be a probabilistic graph and V' a non-empty subset of vertices of V that are connected in $\mathcal{G}$. We assume for each $e \in E$, $\sum_{w \in \mathrm{supp}(p(e))} p(e)(w) < 1$.*

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 $\mathcal{G}$.*

The proof of the theorem is presented in Appendix B.

In other words, Theorem 2 states that the independent subgraph approach is the unique manner in which a lossless indexing system can be obtained for a probabilistic graph.

PTree. 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 *PTree*.

DEFINITION 8. *Let $\mathcal{G}$ be a probabilistic graph. A* PTree *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 $\mathcal{T}$ to a* probabilistic graph $\mathcal{B}$ *(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 1. *To understand what a PTree looks like, consider the tree depicted in Figure 2 which is a PTree for the graph of Figure 1. 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 of the PTree and focus on vertices. The vertices in bags of any subtree of $\mathcal{T}$ induce an independent subgraph in the graph of Figure* 1*: for example, the subtree rooted at node (δ) contains vertices 1, 2, 5, and 6, which is indeed 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.*

How can we efficiently find independent subgraphs, build a PTree, and use this PTree 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 for which index is intractable; tree decompositions (Section 6) that yield weaker decompositions, but in which index is fully tractable, even linear in the size of the original graph.

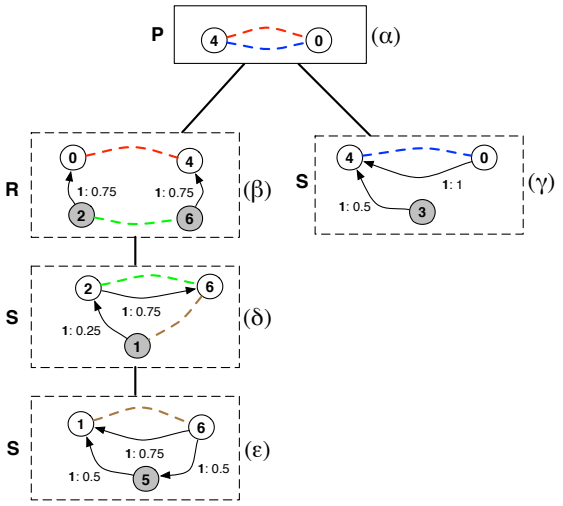


Figure 2: SPQR tree of the graph in Figure 1.

5. SPQR TREES

We introduce in this section a first method for indexing probabilistic graphs into a PTree: SPQR trees [14]. First, we need some graph theory basics on k-connectedness [15].

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 graph 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.[2]

Tutte [37] studied the structure of the triconnected components of a graph, and Hopcroft and Tarjan [24, 23] gave optimal algorithms for decomposition. They showed that the triconnected components of a graph are unique:

THEOREM 3 ([37, 24, 23]). *The biconnected and triconnected components of a graph G are unique and the inclusion relationships among them forms a tree.*

Using Theorem 2 and Theorem 3, 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 graph – since biconnected and triconnected components are independent subgraphs and they are unique.

Hopcroft and Tarjan's algorithms were refined, by using SPQR trees [14] and Gutwenger's linear implementation of them [21]. 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 distributions on the interface edges can be computed, and how they can be propagated inside a PTree. 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 PTree $\mathcal{T}$ consists of nodes corresponding to the triconnected components of the graph. Two types of edges are present in the bags of the PTree: *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 graphs in an SPQR tree, and by extension in $\mathcal{T}$ [37]:

1. a cycle of at least three edges; 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 not containing any of the above two structures; the corresponding bags are called *rigid* or *R-bags*.

EXAMPLE 2. *We present in Figure 2 the SPQR PTree resulting from the graph in Figure 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 PTree 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.

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 [21] 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 PTree $(\mathcal{T}, \mathcal{B})$.

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

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

```
/* 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 first step is the application of the SPQR tree algorithms from [14, 21], 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

[2] In practice, decomposition of independent subgraphs with 0 endpoints (that is, connected components) is not of much interest, so we will mainly consider independent subgraphs with 1 or 2 endpoints.

algorithm, as it is not our focus, and we direct the reader to [21] 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 the endpoints $\{v_1, v_2\}$, i.e., $p(v_1 \to v_2)$ and $p(v_2 \to v_1)$, 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 convolutions of distance distributions.

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. Figure 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 [6].

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 [33]. We can then use the per-bag guarantees to compute the overall guarantees on the distributions in the root bag, in the spirit of [36].

Finally, to increase 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 3. *Returning to the SPQR tree in Figure 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 $p(5 \to 1)$:*

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

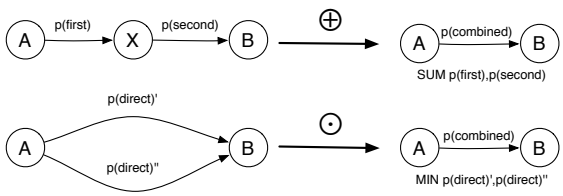


Figure 3: Probability compositions of simple paths.

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

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

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 PTree 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 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.

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

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

input : PTree $(\mathcal{T},\mathcal{B})$, 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 $h \leftarrow \mathsf{height}(\mathcal{T})$ **to** 0 **do**
  for *node* n *of* $\mathcal{T}$ *s.t.* $\mathsf{level}(n) = h$ **do**
    $B \leftarrow \mathcal{B}(n)$;
    if $V(B) \cap \{s\} \neq \emptyset$ **then**
      delete p^c in $\mathsf{parent}(B)$ resulting from B;
      $E(\mathsf{parent}(B)) \leftarrow E(\mathsf{parent}(B)) \cup E(B)$;
      $V(\mathsf{parent}(B)) \leftarrow V(\mathsf{parent}(B)) \cup V(B)$;
return $\mathcal{B}(\mathrm{root}(\mathcal{T}))$

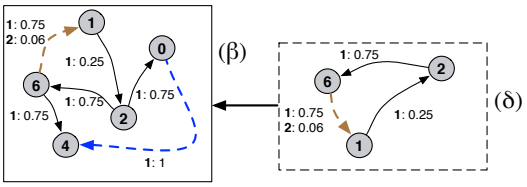


Figure 4: Retrieval for the pair $(1,4)$.

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 Figure 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 2 ensures the validity of the approach. The implementation of SPQR trees of [21] 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 5. 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 [37]), 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 [16] would execute in less time for each sample[3]. 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 PTrees 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. Yet, as we shall explore in the next section, we can achieve lossless indexing by applying another classical graph decomposition technique, namely *fixed-width tree decompositions* (FWDs).

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

6. FIXED-WIDTH DECOMPOSITIONS

Tree decomposition [34] of graphs is a classic technique to solve NP-hard problems in linear time [5], on graphs of bounded treewidth.

Tree decompositions. Following the original definitions in [34], we start by defining a tree decomposition:

DEFINITION 9 (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*:

DEFINITION 10 (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 [10]. However, determining the treewidth of a given graph is NP-complete [4]. 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 [5]. 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 [31]. 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* [39, 3] (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, we adapt the approaches presented in these works to the setting of probabilistic graphs, building a FWD PTree.

6.1 General Approach

Indexing. The adaptation of existing FWD algorithms to probabilistic graphs is straightforward. We transform the probabilistic graph $\mathcal{G}$ into its undirected, deterministic, variant G. Then we apply FWD to generate the list of bags and the resulting tree. When each

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

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

```
   /* decompose the graph into bags of size ⩽ w                    */
1  G ← undirected, unweighted graph of G;
2  S = ∅, T = ∅;
3  for d ← 1 to w do
4      while there exists a vertex v with degree d in G do
5          create new bag B;
6          V(B) ← v and all its neighbors;
7          for all unmarked edge e in G between vertices of V(B) do
8              E(B) ← E(B) ∪ {e}; mark e;
9          covered (B) ← {v};
10         remove v from G and add to G a (d − 1)-clique between v's
           neighbors;
11         S ← S ∪ {B};
   /* create the root graph and the bag tree                       */
12 V(R) ← all vertices in G not in covered(B);
13 E(R) ← all unmarked edges in G;
14 for bag B in S do
15     mark B;
16     if ∃ an unmarked bag B' s.t. V(B)\covered(B) ⊆ B' then
17         update (T, B) so that B' is parent of B;
18     else update (T, B) so that R is parent of B
   /* compute edges between uncovered vertices and
      propagate up                                                 */
19 for h ← height(T) to 0 do
20     for bag B s.t. level(B) = h do
21         precompute-propagate^FWD(B);
22 root T at R;
23 return (T, B);
```

bag is generated, we copy the probabilistic edges from $\mathcal{G}$ to it, in a manner similar to SPQR. After the decomposition process has ended, the remaining edges and vertices are copied to the root graph $\mathcal{R}$ along with new edges computed from the tree decomposition. The resulting index $\mathsf{index}^{\mathrm{FWD}}(\mathcal{G})$ will be a PTree $(\mathcal{T}, \mathcal{B})$.

Algorithm 4 presents the index operator. It consists of three stages: the main decomposition, the building of the FWD PTree 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 [39, 3], 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 which includes completely in their vertex set the 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 $p(v_1 \to v_2)$ 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: $p(v_1 \to v_2) = p(v_1 \to v_2) \odot (p(v_1 \to v) \oplus p(v \to v_2))$. 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 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

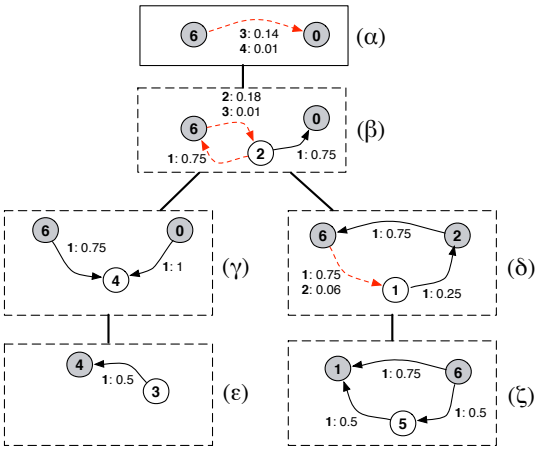


Figure 5: The $w = 2$ decomposition of the example graph. Vertices in white are the vertices covered by each bag, and dashed red edges are edges which are computed from children. Each edge has a distribution of distance probabilities associated to it.

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 $\mathtt{precompute\text{-}propagate}^{\mathrm{FWD}}$ is similar to the SPQR version, and we omit it here.

EXAMPLE 5. *We give in Figure 5 the result of applying Algorithm 4 on the graph in Figure 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 edges resulting from the bottom-up propagation. In the figure, the dashed red edges represent the edges which have been computed from the children.*

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 is the case of bag (ε). On the right-hand side, bag (ζ) will provide a $6 \to 1$ edge to bag (δ). Bag (δ) also propagates edges $6 \to 2$ and $2 \to 6$ to bag (β). Finally, bag (β) propagates edge $6 \to 0$ to the root bag (α).

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 Figure 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 Figure 1.

Properties. We can show that FWD for $w \leqslant 2$ are lossless, but, 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. It is sufficient for us to imagine a bag resulting from a $w > 2$ decomposition having covered vertex v and neighbor vertices $v_1, v_2, v_3, \ldots, v_w$ with 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 SPQR.

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

In terms of time complexity, we know that the FWD itself is linear in the number of vertices in the graph [39, 3]; however the computation of pairwise probability distributions is quadratic in w for each bag.

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 $d = \mathsf{O}(E)$, this does not occur in practice. Moreover, for $w \leqslant 2$, i.e, the lossless cases, there are only 2 pairs to generate, and each bag is visited only once by Algorithm 4. Hence, in practical settings, the complexity of propagating computations is linear in the number of vertices in the graph.

The complexity of the retrieval for tree decompositions is the same as in the case of SPQR PTrees. Hence $(\mathsf{index}^{\mathrm{FWD}}, \mathsf{retrieve}^{\mathrm{FWD}})$ is also an indexing system, lossless for $w \leqslant 2$.

The proofs of these properties can be found in Appendix C.

For $w \leqslant 2$ the decompositions are lossless – albeit not optimal – and all pre-computed edges can be efficiently evaluated. As we shall see, their gains in efficiency approach those of the SPQR indexes. In some cases – such as denser networks – their efficiency is still not fully satisfactory. As we shall show in the next section, we can improve on time efficiency at the cost of an increase in space requirements, by devising extensions to FWDs dealing with the correlations introduced with higher treewidths.

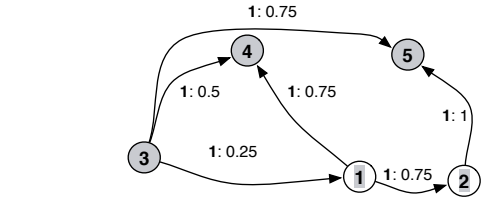
6.2 Handling Higher Treewidths

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 sampling directly 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.

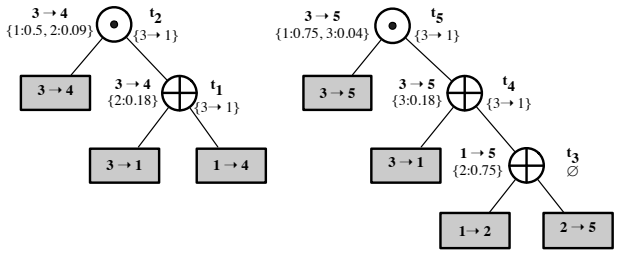
Instead, we can compute the full *lineage* of the distance distributions between endpoints at pre-processing, and leave the handling of the correlations at query time. For this, we compute the lineage at tree decomposition time and build a *lineage tree*, i.e., a parse tree of the path between endpoints.

A *lineage tree* is a binary tree where the intermediary nodes are either $\oplus$ or $\odot$, depending on the convolution of the subtrees, and the leaves are edges from $\mathcal{G}$. Such lineage trees 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, $T.edge$. Both $T.edge$ and $T.dst$ can be computed directly at decomposition time, just as in the previous decompositions, SPQR and FWD.

The $\mathsf{dependent}(T)$ computation goes on as follows. We first



(a) Original graph



(b) Lineage trees

Figure 6: Example of dependent path lineages and annotated lineage trees

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

EXAMPLE 6. *Let us take the graph in Figure 6a. Say we cover nodes* 1 *and* 2 *and wish to pre-compute* $3 \to 4$ *to* $3 \to 5$. *Figure 6b shows the resulting lineage trees for edges* $3 \to 4$ *(left) and* $3 \to 5$ *(right).* $3 \to 4$ *is rooted at* t_2, *and* $3 \to 5$ *at* t_5. *Edge* $3 \to 1$ *introduces correlations between* t_2 *and* t_5, *but also between the subtrees* t_1 *and* t_4. *For* t_5, $\mathsf{dependent}(t_5) = \emptyset$ *so the pre-computed distance for edge* $1 \to 5$, $\{2 : 0.75\}$, *does not have any correlations.*

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 else d ← dl ⊙ dr
13 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

Table 2: Dataset summary

graph	vertices	edges
WIKIPEDIA	252,335	2,544,312
COMMUNICATION	62,651	147,878
NH	115,055	260,394
CA	1,595,577	3,919,162

FWDs with $w \leqslant 2$, as bags are always independent of other parts in the graph. Then, at sampling time, each time a computed edge is encountered we evaluate the lineage tree corresponding to its tree.

Given such a lineage tree, 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. *Let us return to the tree t_5 in Figure 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_2. When t_2 needs to be evaluated, we need to use this sampled distance for $3 \to 1$. Edge $3 \to 5$ and t_3 can use their distributions without sampling, as they do not have correlations anywhere else.*

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, breaking the linear size requirement for it to be an indexing system. Note that the computation of LIN is still linear in the graph size, so it is still extremely efficient to compute time-wise.

As we shall show in experiments, the number of computed edges added to $\mathcal{R}$ has a direct influence on query evaluation. Yet, LIN can achieve considerable increases in efficiency, especially for dense graphs, where SPQR and FWD fare relatively poorly, meaning it still has good practical applicability.

7. 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 shown in Table 2. The details of their construction and of the PTree implementation are presented in Appendix D. The implementation code and the datasets used in this paper are available at `http://bit.ly/1vL7QQo`.

We first present the results for the approaches that are indexing systems, SPQR and FWD. Then we proceed to evaluate the lossless extension for higher w, LIN, which has more practical appeal for cases where space is not an issue.

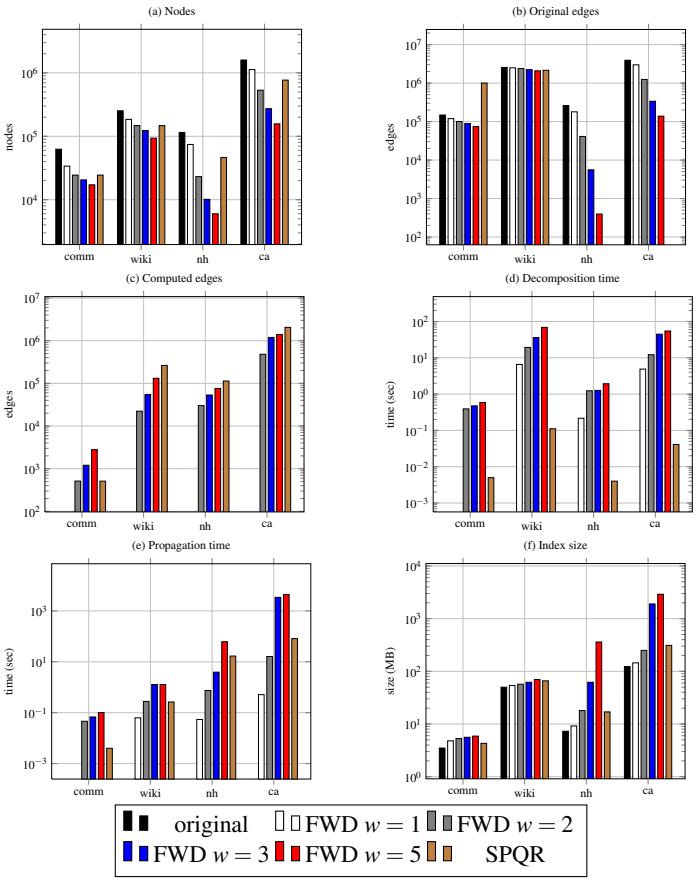


Figure 7: PTree properties (log y scale)

PTree properties. For each dataset, we have generated both the lossless FWD, i.e., $w \in \{1,2\}$, the SPQR PTrees, and, for comparison purposes, the lossy FWD for $w \in \{3,5\}$. 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 PTrees 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 100 samples.

Figure 7a-c illustrates the sizes of the resulting $\mathcal{R}$ graph, from applying PTree on the four graphs. We show here the number of nodes, the number of original edges retained and the number of computed edges added to the root. Ideally, we wish that the sum of the original edges and the computed edges is less than the original graph size, since the sampling procedures directly depend on the number of edges. It can be seen that 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 WIKIPEDIA graph. This is also expected since even removing the long tail of the degrees retains the high-degree nodes, where most of the edges are concentrated. They, however, are always significantly lower than the original graph size. For the SPQR decomposition, it can be seen that is is better than the largest lossless FWD decomposition, $w = 2$. In all cases, $w = 0$ consists only of the root bag, containing the original graph. Note that in the SPQR case we define the root as the largest bag in the tree. Interestingly – for the road graphs, NH and CA – in the case of SPQR, the root does not retain any original edges and it is fully populated by computed edges. We conjecture that this is due to the relative sparseness of road networks as compared to social or

Table 3: PTree Δ_{PW}

Graph	FWD, $w = 1$	FWD, $w = 2$	SPQR
WIKIPEDIA	2^{20710}	2^{72440}	2^{881506}
COMMUNICATION	$2^{19922.4}$	$2^{32621.1}$	$2^{12856.1}$
NH	2^{720}	$2^{65006.7}$	$2^{31104.1}$
CA	2^{9930}	2^{762300}	2^{497820}

communication networks.

Figure 7d-f shows the preprocessing execution time and space overhead properties of PTree. As can be noticed, the index is very efficient, running in the order of seconds even on large graphs. The same observations holds 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 taking only on the order of second on the larges graphs. The exception to this behaviour are the lossy decompositions of the road networks, where the distance propagation can take a few hours.

The space overhead of $\mathcal{I}$ is also reasonable. Generally, PTree FWD $w \leqslant 2$ and SPQR only incur between 10% (WIKIPEDIA) and double (NH) space overhead compared to the space cost of the original graph. Again the lossy FWDs for the road networks increase significantly in size compared to their original graph size, even reaching a few gigabytes in the case of CA. Since the higher widths of the tree decompositions no longer retain the linear size increase property, this is indeed theoretically possible to happen. Nevertheless, the query time savings of the lossless decompositions of NH and CA are important enough for even $w = 2$, as we shall see next.

Possible worlds. The size of the $\mathcal{R}$ graph can directly result into a significant decrease in possible worlds, as shown in Table 3. We track the coefficient $\Delta_{\text{PW}} = |\text{PW}(\mathcal{G})|/|\text{PW}(\mathcal{G}(q))|$, where $\mathcal{G}$ is the original graph and $\mathcal{G}(q)$ is the result of applying retrieve on $\mathcal{I}$ (FWD or SPQR) for a query q. The results were aggregated by averaging over a workload of 1,000 random pairs. As can be seen, the size of the root $\mathcal{R}$ directly affects the number of possible worlds saved. Moreover, it is a clear indication that pre-computing distance distribution on edges has an effect of "collapsing" the possible worlds. Note that LIN has the same potential of collapsing the possible world as FWD does.

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 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 on every sampling round. For testing, we executed the workloads for a number of samples between 10 and 1,000.

As Figure 8 shows, the efficiency gains are important when queries are executed on PTree indexes. The gains on the lossless decompositions are up to 5 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 WIKIPEDIA, do not have such an important increase in efficiency, and we will show later how this can be alleviated by using LIN for higher treewidths.

We turn to showing how the retrieve operator time influences the execution of the queries, in Figure 9 (log scale). In terms of execution time, the retrieval of equivalent graphs does not take a significant time out of the total execution time. In the worst case, SPQR for WIKIPEDIA, it is roughly 1% of the execution time for 1,000 samples. Hence, the sampling time greatly dominates the

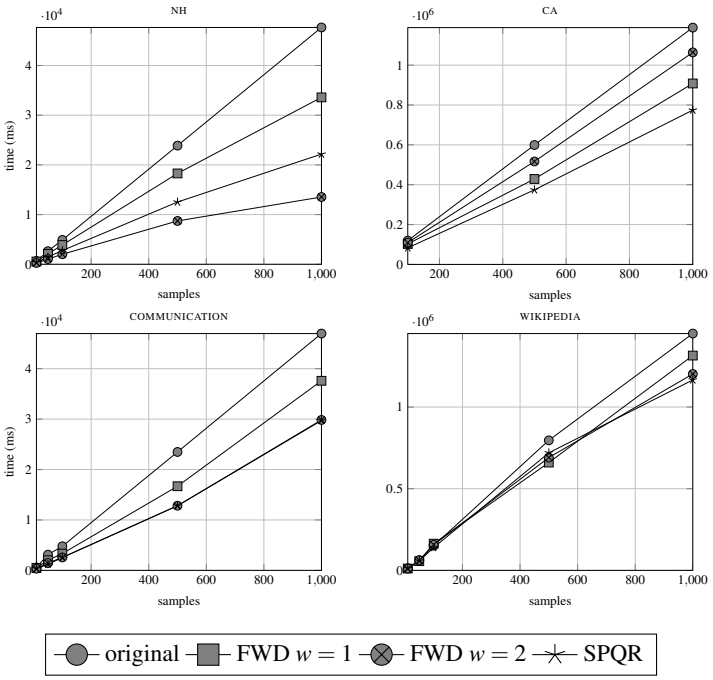


Figure 8: Running time versus number of samples

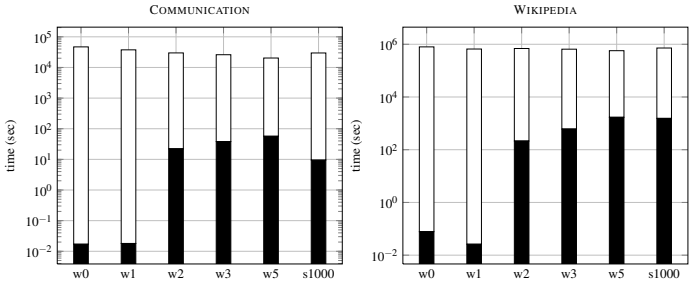


Figure 9: Proportion of retrieve (black) out of the total query time (white) (log y scale)

query time and applying retrieve is highly efficient.

Error vs. time. The question we wish to answer now is the following: Can such approaches beat sampling algorithms? That is, is the error vs. time trade-off – especially for SPQR and the lossy FWD, $w > 2$ – enough to justify using our algorithms, and not simply more sampling rounds? To check this, we have plotted the running time of applying sampling on PTree 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 equivalent relative error results.

Figure 10 presents the results for the COMMUNICATION and WIKIPEDIA graphs (note the log-log axes). The black dots represent the results on sampling the original graph, for a number of sample rounds between 10 and 1,000. Intuitively, we want the points corresponding to PTree 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 PTree. For instance, note that the white dots in the COMMUNICATION graph are slightly lower than the corresponding black dots, suggesting an increase in accuracy. The results

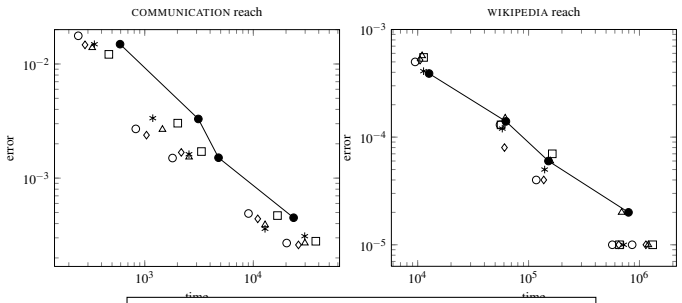


Figure 10: Relative error vs. time (log-log axis)

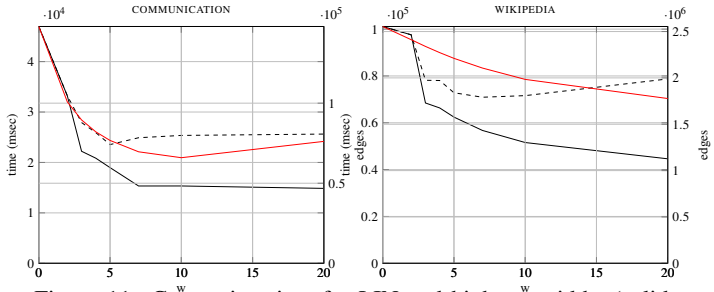


Figure 11: Computing time for LIN and high treewidths (solid: lineage, dashed: baseline) and number of edges in $\mathcal{R}$ (red)

are replicated for the case of the WIKIPEDIA graph.

It can also be seen that the lossy FWD variants have error rates comparable to the error rates of the lossless variants, for a greater increase in running time. This suggests that the number of correlations introduced by the distribution pre-computation is not as large in practice, and that this kind of decompositions can be effectively used for query answering. However, we have no bound of the error introduced and hence we cannot control the error. This can be alleviated by the lineage sampling we presented in the previous sections, as we shall show next.

Using lineage for higher widths. We show, in Figure 11, how the LIN decompositions fare for higher treewidths. Note that in this case, all query processing is lossless as we take care of the correlations at sample time. We compare the query processing time (black lines) with the number of edges (red line) in the root bag $\mathcal{R}$, for widths $w \in \{3,4,5,7,10,20\}$ and for the COMMUNICATION and WIKIPEDIA graphs.

In terms of running time, we compare with a baseline in which every edge in the lineage tree is sampled, even if it is not a dependency edge. This can mean that potentially we will sample the same number of edges as in the original $\mathcal{G}$. This can still lead to reasonable gains in efficiency, as the Dijsktra algorithms – even implemented with optimized Fibonacci heaps – can still cause significant overhead due to the processing and updating of the vertex heaps. The dashed black line shows the execution time for this baseline. It can be seen that this baseline can achieve a two-fold increase in efficiency in COMMUNICATION, and around 20% for WIKIPEDIA, and that its minimal running time is achieved for decompositions under $w = 10$. In COMMUNICATION, this is the moment in which we begin adding more edges than removing in $\mathcal{R}$, but not in WIKIPEDIA. The optimized processing of lineage trees described in Algorithm 5 – illustrated by the black line – performs even better. In COMMUNICATION, it achieves a three-fold increase in efficiency, but the execution time plateaus for $w > 10$, as the number of edges in $\mathcal{R}$ begins to increase. As the number of edges in $\mathcal{R}$ still decreases even for $w = 20$ for WIKIPEDIA, the best efficiency is achieved at $w = 20$, a two-fold time reduction. This is much more efficient than either SPQR or FWD. Interestingly, the increase in efficiency is greater than the relative number of edges in $\mathcal{R}$, suggesting that the structure of the graph is just as important as its size. In terms of space, the decomposition is still efficient for these datasets. For WIKIPEDIA LIN $w = 20$, the size of the index is around 1 GB, which can be easily handled in-memory on any modern desktop computer.

Table 4: Distance-constraint reachability running time (sec) and error ratios (in parentheses) for three d-RQ estimators

Decomp.	RHH	RHT	Dagger
original	0.095 (0.122)	0.123 (0.109)	0.631 (0.225)
FWD $w=2$	0.069 (0.056)	0.113 (0.057)	0.185 (0.146)
SPQR	0.050 (0.071)	0.061 (0.073)	0.338 (0.129)

Comparison with other algorithms. One of our arguments in using PTree as a pre-computed index is that it can be applied directly to existing solutions. To check this, we apply the distance-constraint reachability (d-RQ) estimators studied in [26] to the FWD and SPQR versions of the NH graph. We use the RHH, RHT and the Dagger sampling estimator and apply directly the authors' implementation. We also track the error ratio, defined as $E = \left|\hat{R} - R\right| / R$, where $\hat{R}$ is the result of an estimator and R is the result of the exact computation. We use the same experimental setup as [26], 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.

Table 4 summarizes the results. First of all, it can be easily noted that, indeed, applying PTree decompositions directly affects the running time of any of the three estimators, up to a 3-times increase in efficiency. Applying PTree also increases accuracy – in terms of the error ratio – for all three estimators. The best accuracy increase for RHH and RHT is achieved by the FWDs, while for Dagger sampling the SPQR decomposition performs best. These results complement the results in Figure 10 and suggest that the fact some edges are already pre-computed minimizes the chance of sampling error. The difference in behaviour of FWD and SPQR between the baseline Dagger and RHT/RHH suggests that the denser graph structure resulting from SPQR helps in its execution on advanced estimators. Hence, our decompositions can be readily used for state-of-the art estimators in a pre-computations step, since the retrieve operator is very efficient and they can potentially increase the accuracy (or at the very least, keep it at the same level) over using the original graphs.

8. 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 PTree, with two variants: the SPQR tree and the FWD. SPQR trees have the advantage of an optimal decomposition, and therefore more potential for 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 PTree 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 [33] and frequent subgraph discovery [41]).

APPENDIX

A. REFERENCES

[1] J. Añez, T. De La Barra, and B. Pérez. Dual graph representation of transport networks. *Transportation Research Part B: Methodological*, 30(3), 1996.

[2] E. Adar and C. Re. Managing uncertainty in social networks. *IEEE Data Eng. Bull.*, 30(2), 2007.

[3] T. Akiba, C. Sommer, and K. Kawarabayashi. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *EDBT*, 2012.

[4] S. Arnborg, D. G. Corneil, and A. Proskuworski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2), 1987.

[5] S. Arnborg and A. Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1), 1989.

[6] R. B. Ash and C. A. Doléans. *Probability & Measure Theory*. Academic Press, 2nd edition, 1999.

[7] S. Asthana, O. D. King, F. D. Gibbons, and F. P. Roth. Predicting protein complex membership using probabilistic network reliability. *Genome Research*, 14(6), 2004.

[8] M. O. Ball. Computational complexity of network reliability analysis: An overview. *IEEE Trans. Reliability*, 35(3), 1986.

[9] P. Barceló, L. Libkin, and J. L. Reutter. Querying regular graph patterns. *J. ACM*, 61(1), 2014.

[10] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6), 1996.

[11] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, 2009.

[12] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5), 2003.

[13] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4), 2007.

[14] G. Di Battista and R. Tamassia. On-line graph algorithms with spqr-trees. In *ICALP*, 1990.

[15] R. Diestel. *Graph Theory*. Springer, Berlin, Germany, 3rd edition, 2005.

[16] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 1959.

[17] P. Domingos and M. Richardson. Mining the network value of customers. In *KDD*, 2001.

[18] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD Conference*, 2012.

[19] G. S. Fishman. A comparison of four Monte Carlo methods for estimating the probability of s-t connectedness. *IEEE Trans. Reliability*, 35(2), 1986.

[20] J. Ghosh, H. Q. Ngo, S. Yoon, and C. Qiao. On a routing problem within probabilistic graphs and its application to intermittently connected networks. In *INFOCOM*, 2007.

[21] C. Gutwenger and P. Mutzel. A linear time implementation of spqr-trees. In *Graph Drawing*, 2000.

[22] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *CIKM*, 2005.

[23] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3), 1973.

[24] J. E. Hopcroft and R. E. Tarjan. Efficient algorithms for graph manipulation [h] (algorithm 447). *Commun. ACM*, 16(6), 1973.

[25] M. Hua and J. Pei. Probabilistic path queries in road networks: traffic uncertainty aware path selection. In *EDBT*, 2010.

[26] R. Jin, L. Liu, B. Ding, and H. Wang. Distance-constraint reachability computation in uncertain graphs. *PVLDB*, 4(9), 2011.

[27] B. Kanagal and A. Deshpande. Lineage processing over correlated probabilistic databases. In *SIGMOD Conference*, 2010.

[28] A. Khan, F. Bonchi, A. Gionis, and F. Gullo. Fast reliability search in uncertain graphs. In *EDBT*, 2014.

[29] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, 2001.

[30] D. Liben-Nowell and J. M. Kleinberg. The link-prediction problem for social networks. *JASIST*, 58(7), 2007.

[31] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev. E*, 64, Jul 2001.

[32] O. Papapetrou, E. Ioannou, and D. Skoutas. Efficient discovery of frequent subgraph patterns in uncertain graph databases. In *EDBT*, 2011.

[33] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. k-nearest neighbors in uncertain graphs. *PVLDB*, 3(1), 2010.

[34] N. Robertson and P. D. Seymour. Graph minors. iii. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1), 1984.

[35] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.

[36] A. Souihli and P. Senellart. Optimizing approximations of dnf query lineage in probabilistic xml. In *ICDE*, 2013.

[37] W. T. Tutte. *Connectivity in graphs*, volume 15 of *Mathematical Expositions*. University of Toronto Press, 1966.

[38] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3), 1979.

[39] F. Wei. Tedi: efficient shortest path query answering on graphs. In *SIGMOD Conference*, 2010.

[40] Y. Yuan, G. Wang, H. Wang, and L. Chen. Efficient subgraph search over large uncertain graphs. *PVLDB*, 4(11), 2011.

[41] Z. Zou, H. Gao, and J. Li. Discovering frequent subgraphs over uncertain graph databases under probabilistic semantics. In *KDD*, 2010.

B. PROOF OF THEOREM 2

PROOF. Let us first show that such a $\mathcal{G}'$ exists if V' is the set of internal vertices of an independent set 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:

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

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. □

C. PROPERTIES OF FWD

PROPOSITION 2. *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. □

PROPOSITION 3. *Let* $(\mathcal{T}, \mathcal{B})$ *be a* FWD *PTree 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. □

PROPOSITION 4. *The complexity of* *precompute-propagate*$^{\text{FWD}}$ *is* $\mathsf{O}(w^2 d)$, *where* d *is the maximum distance having non-zero probability in the graph.*

PROOF. The number of endpoint pairs in a bag is $\mathsf{O}(w^2)$. 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. □

D. DATASETS AND IMPLEMENTATIONS

We used four probabilistic graphs datasets, from different application domains:

1. The WIKIPEDIA dataset, representing Wikipedia[4] text interactions between contributors. Each probabilistic edge has distance 1 and the probability 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. The graph has 252,335 vertices and 2,544,312 edges.

2. The COMMUNICATION dataset, obtained from the SNAP website[5] 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[6], in which the edges represent roads between geographic locations, and have weights representing the average driving time. We have attached to each edge the probability of driving occurring without incident, chosen uniformly in the interval $[0.95, 1]$. 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.

Our PTree 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 was implemented by us, while the deterministic part of the SPQR decomposition was done using the implementation in the Open Graph Drawing Framework library[7]. This implementation is an efficient implementation of the linear decomposition algorithm presented in [21].

[4] `http://en.wikipedia.org/`
[5] `http://snap.stanford.edu/data/p2p-Gnutella31.html`
[6] `http://www.dis.uniroma.it/challenge9/data/tiger`
[7] `http://ogdf.net/`