| | |
|---|---|
| Title | ProbTree: a query-efficient representation of probabilistic graphs |
| Author(s) | Maniu, S; Cheng, R; Senellart, P |
| Citation | The 1st International Workshop on Big Uncertain Data (BUDA 2014) in conjunction with SIGMOD/PODS 2014, Snowbird, UT., 22 June 2014. |
| Issued Date | 2014 |
| URL | http://hdl.handle.net/10722/203649 |
| Rights | Creative Commons: Attribution 3.0 Hong Kong License |

# ProbTree: A Query-Efficient Representation of Probabilistic Graphs

Technical Paper

Silviu Maniu Reynold Cheng
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
{smaniu,ckcheng}@cs.hku.hk

Pierre Senellart
Institut Mines-Télécom; Télécom ParisTech
CNRS LTCI
75634 Parix Cedex 13, France
pierre.senellart@telecom-paristech.fr

## ABSTRACT

Information in many applications, such as mobile wireless systems, social networks, and road networks, is captured by graphs, in many cases uncertain. We study the problem of querying a probabilistic graph; in particular, we examine "source-to-target" queries, such as computing the shortest path between two vertices. Evaluating ST-queries over probabilistic graphs is #P-hard, as it requires examining an exponential number of "possible worlds". 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 *ProbTree*, a data structure that stores a succinct representation of the probabilistic graph. Existing ST-query solutions are executed on top of this structure, with the number of samples and possible world sizes reduced.

## 1. INTRODUCTION

*Probabilistic graphs.* 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 [9]. Mobile devices form ad-hoc networks through WiFi technologies [11]. In a road network, cities are connected by roads [1]. In biological networks, proteins interact with each other in a complex manner [3]. Substantial research has been devoted to the effective processing of graph queries, including reachability [5], shortest paths [8], frequent subgraphs [16], and graph patterns [4].

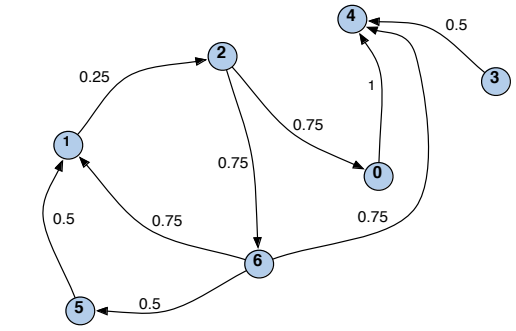
In the applications above, data uncertainty is inherent. A natural way to capture graph uncertainty is to represent them as *probabilistic graphs* [20, 10]. There exist two main representations of edge uncertainty in probabilistic graphs. In the *edge-existential model*, each edge is augmented with a probability value, which indicates the chance that the edge exists (Figure 1a). This graph captures the reliability and failure in computer network connections [10]. It can also represent the uncertainty in social and biological networks [3]. In the *weight-distribution model*, each edge is associated with a probability distribution of weight values [14]. 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 in large probabilistic graphs has been considered only recently. Some representative works include finding shortest paths and reliability estimation [10], searching nearest neighbors [17], and mining frequent subgraphs [21]. In this paper, we study the evaluation of an important query class, known as the *source-to-target* query, or *ST-query*, which are defined over source vertex $s$ and target vertex $t$ in a probabilistic graph, such as reachability queries (RQ) and shortest distance queries (SDQ).

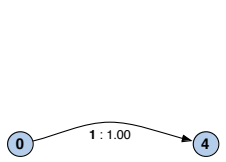
To query a probabilistic graph $\mathcal{G}$, the *possible world semantics* (PWS) is often used [6]. Conceptually, $\mathcal{G}$ is interpreted as a set of possible worlds, each of which is a definite (non-probabilistic) graph itself. Each possible world is given a probability of its existence derived from edge probabilities. The semantics of evaluating a query $q$ (e.g., an SDQ) on $\mathcal{G}$ involves 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 is #P-hard [20].

To improve ST-query efficiency, researchers have proposed *sampling solutions* [10, 17], where possible worlds with high existential probabilities are extracted. These algorithms, which examine fewer possible worlds than the PWS, are more efficient. However, these solutions suffer from two major downsides, which can hamper query efficiency significantly: (i) a possible world can be very large, directly affecting query efficiency; and (ii) to achieve high accuracy, a lot of possible world samples may need to be generated.

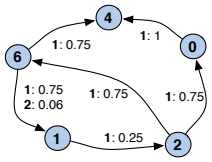
*Our contributions.* Our goal is to tackle these issues, so that an ST-query can be efficiently answered. Our main idea is to evaluate the query on $\mathcal{G}(q)$, a weight-distribution probabilistic graph derived from $\mathcal{G}$. Let $q(s,t)$ be an ST-query with source vertex $s$ and target vertex $t$. The result of running $q(s,t)$ on $\mathcal{G}(q)$ should be similar (ideally, identical) to that of $q(s,t)$ executed on $\mathcal{G}$. Consider an RQ, $q(0,4)$, executed on the graph in Figure 1a. There is only one path of probability 1 between vertex 0 and 4. Correspondingly, $\mathcal{G}(q)$ is a directed edge $0 \rightarrow 4$, with $\{1 : 1.00\}$ denoting a unit-length path between vertex 0 and 4 of probability 1. Answering $q(0,4)$ on $\mathcal{G}(q)$ is the same as evaluating $q(0,4)$ on $\mathcal{G}$ – both tell us that vertex 0 reaches vertex 4 with probability 1. Figure 1c illustrates $\mathcal{G}(q)$ for $q(1,4)$. Here, edge $3 \rightarrow 4$ is not included, since it does not affect the result




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




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




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

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

of $q(1,4)$. Also, the subgraph containing vertex 1, 5, and 6 is abstracted by a directed edge $6 \to 1$, which means that there are two paths from vertex 6 to vertex 1, with lengths 1 and 2 and respective probabilities of 0.75 and 0.0625.

In these examples, $\mathcal{G}(q)$ is smaller than $\mathcal{G}$. Hence, the possible world graphs sampled from $\mathcal{G}(q)$ are smaller than those generated from $\mathcal{G}$, increasing both efficiency and accuracy.

How can a small $\mathcal{G}(q)$ be obtained? We propose to use a *ProbTree*, a structure derived from $\mathcal{G}$. Given a query $q(s,t)$, the ProbTree is decomposed to yield $\mathcal{G}(q)$. We require the ProbTree to be of size comparable to $\mathcal{G}$. Moreover, the time for indexing and retrieving a ProbTree should be small. To achieve these goals, we show that the ProbTree must be in the form of a tree. In the following we examine structures called *SPQR trees* [12], and implement ProbTrees by incorporating probabilistic graph information into SPQR trees.

## 2. FORMAL MODEL

*Probabilistic graphs.* We begin by giving the definition of a probabilistic graph, modeled as a directed graph with (finite) distance probability distributions on edges, and the notion of possible world induced by such a graph.

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 commonly denote* $V(\mathcal{G})$*,* $E(\mathcal{G})$*,* $p_{\mathcal{G}}$ *the vertices, edges, and probability assignment function of a given probabilistic graph* $\mathcal{G}$*.*

A given edge $e$ is considered non-existing in the graph with probability $1 - \sum_{w \in \mathrm{supp}(p(e))} p(e)(w)$. Probability distributions on different edges are considered independent.

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 called a possible world of* $\mathcal{G}$ *if* $E_G \subseteq E$ *and* $\omega$ *is such that, 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 defined by:*

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

*ST-queries on probabilistic graphs.* In this paper, we focus on *source-target* distance query types (or ST-query for short), which can be answered on the discrete *distance distribution* of the input pair of vertices. The distance distribution $p(s \to t)$ between 2 vertices $s, t \in V$ is a set of tuples of the form $(d_i, p_i)$, where $p_i$ is the probability that the shortest distance between $s$ and $t$ is $d_i$.

Examples of ST-queries include:

**Reachability:** probability that $t$ is reachable from vertex $s$.

**Distance-constraint reachability:** probability that $t$ is reachable from vertex $s$ within distance $d$.

**Expected shortest distance:** expected value of the distance distribution from $s$ to $t$.

All query types above are computationally hard on probabilistic graphs as shown in [20].

*Indexes on probabilistic graphs.* To define indexes on probabilistic graphs, we use the notion of *transformation system.*

DEFINITION 3. *A* probabilistic graph transformation system *is a pair* $(\mathsf{index}, \mathsf{retrieve})$ *where: (i)* $\mathsf{index}$ *is a function that takes as input a probabilistic graph* $\mathcal{G}$ *and produces as output some arbitrary object* $\mathcal{I} = \mathsf{index}(\mathcal{G})$*; (ii)* $\mathsf{retrieve}$ *is an operator that, given an arbitrary* $(s,t)$ *ST-query* $q$ *in* $\mathcal{G}$ *(where* $s$ *and* $t$ *are the source and target of the query), and the index* $\mathcal{I}$ *obtained by applying* $\mathsf{index}$ *on* $\mathcal{G}$*, produces a probabilistic graph* $\mathcal{G}(q) = \mathsf{retrieve}_q(\mathcal{I})$ *such that* $s$ *and* $t$ *are vertices of* $\mathcal{G}(q)$*.*

In other words, a transformation encodes a probabilistic graph into an structure that can be used to build specific probabilistic graphs for pairs of vertices. Since the pair of vertices can be found in the target probabilistic graph, ST-queries on this pair can be run on top of this target graph.

For a transformed graph, there are two classes of important properties to be taken into account: (i) the *loss*, quantified as the difference between the probabilities returned by the transformed graph and those of the original graph, and (ii) the *efficiency*, quantified as the time and space cost of evaluation on the transformed graph. We detail the formalization of each of the two types below.

We are especially interested in *lossless* translations, such that a ST-query produces the same result on the transformation as on the original probabilistic graph; for generality, we use a common quantitative notion of loss for a transformation, the *mean squared error* (MSE). We consider a transformation *lossless* if, for all possible queries, its MSE is equal to 0.

A transformation system is called an efficient representation if it is efficient for answering a given kind of query. Formally:

DEFINITION 4. *A transformation system* (index, retrieve) *is said to be an* efficient representation *for query class $\mathcal{Q}$ if the following properties are satisfied: (i)* index *is a polynomial-time function; (ii) for every probabilistic graph $\mathcal{G}$, $|\mathsf{index}(\mathcal{G})| = O(|\mathcal{G}|)$ (i.e., the space occupied by the index is 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.*

In addition, we look for representations 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 as little as possible, and especially smaller than query evaluation over the original graph.

# 3. INDEPENDENCE AND PROBTREE

We now turn to answering an important question: is it possible to achieve an efficient representation of probabilistic graphs (with no or limited loss)? We argue next that this is possible by reducing the *independent subgraphs* of a probabilistic graph, thus obtaining a *tree decomposition* of the graph, called a ProbTree.

***Independent subgraphs.*** By definition, each edge in a probabilistic graph – along with its associated probability distribution – is independent of probability distributions of the other edges. The principle behind a lossless compression is to collapse larger subgraphs to edges, such that independence is maintained:

DEFINITION 5. *We define an* independent subgraph *of a probabilistic graph $\mathcal{G}$ as 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) in $\mathcal{G}$, the internal vertices are connected, in an undirected sense, only to other internal vertices of $S$ or to the endpoint vertices; (ii) the endpoints can have links to other vertices in the graph $\mathcal{G}$, 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 $v_1$ and $v_2$ the two endpoints and associated probability distributions $p(v_1 \to v_2)$, $p(v_2 \to v_1)$ computed from $S$. To understand why this is possible, we need to introduce the notion of joint distance probability distributions:

DEFINITION 6. *Given $\mathcal{G} = (V, E, p)$ 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 $\{d_{11}, \dots, d_{ij}, \dots, d_{nn}\}$ the probability:*

$$\Pr\Big[\bigwedge_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant n}} p(v_i \to v_j) = d_{ij}\Big].$$

The above basically 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. 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 1. *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 that, for each edge $e \in E$:*

$$\sum_{w \in \mathrm{supp}(p(e))} p(e)(w) < 1.$$

*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 $\mathcal{G}$.*

In other words, Theorem 1 states that the independent graph approach is the unique manner in which a lossless compression can be obtained for a probabilistic graph.

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

# 4. SPQR TREES

We introduce in this section one method for obtaining a ProbTree: SPQR trees. 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.

Tutte [19] studied the structure of the triconnected components of a graph, and Hopcroft and Tarjan [13] gave optimal algorithms for decomposition. They showed that the triconnected components of a graph are unique and form a tree in an unique manner. By using the refined triconnected component decomposition algorithms – SPQR trees [7] – as "black boxes", we can construct the ProbTree.

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

There are three types of internal graphs in an SPQR tree, and by extension in $\mathcal{T}$ [19]: (i) a cycle of at least three

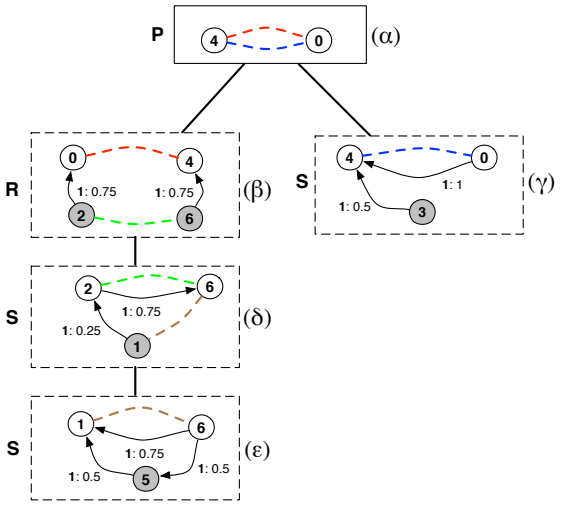



Figure 2: SPQR tree of the graph in Figure 1.

edges; the corresponding tree bags are called *serial* or *S-bags*, (ii) two vertices having parallel edges; the corresponding bags are called *parallel* or *P-bags*, and (iii) a triconnected graph not containing any of the above two structures; the corresponding bags are called *rigid* or *R-bags*.

Example 1. *We present in Figure 2 the SPQR ProbTree 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 ProbTree is composed of three S-bags, one P-bag and one R-bag. Each bag contains the union of the induced subgraph of $\mathcal{G}$ and the skeleton edges. Moreover, each bag contains a triconnected component.*

*Take bag ($\delta$) 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 ($\varepsilon$), summarizing paths from 6 to 1 in node ($\varepsilon$) (there is no path from 1 to 6 in node ($\varepsilon$)). Vertices 2 and 6 are a separation pair for the subgraph induced by the vertices in bags ($\delta$) and ($\varepsilon$), i.e., vertices $1, 2, 5, 6$.*

*Bag ($\beta$) is an R-bag, and the bag is a P-node, containing two parallel undirected skeleton edges, corresponding to the two branches of the SPQR tree.*

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

The first step is the application of the SPQR tree algorithms from [12], which creates a tree $\mathcal{T}$ and a mapping $\mathcal{B}$ from bags of $\mathcal{T}$ to sets of vertices of $\mathcal{G}$. We omit here the details of the SPQR algorithm, as it is not our focus, and we direct the reader to [12] 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 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 ($\beta$). Edge $0 \to 4$ can either be computed as coming from the independent subgraph defined by bags ($\alpha$) and ($\gamma$), or by the independent subgraph defined by bags ($\beta$), ($\delta$), and ($\varepsilon$).

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

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

```
1  G ← undirected, unweighted graph of G;
2  (B, T) ← compute-spqr(G);
3  for n node of T do
4  |  copy the edges of G to B(n);
5  for l, leaf of T do
6  |  root T at l; for h ← height(T) to 0 do
7  |  |  for node n of T s.t. level(n) = h do
8  |  |  |  precompute-propagate^SPQR(B(n), T);
9  root the tree at the node with largest bag;
10 return (T, B);
```

This bi-directional computation is very useful for the retrieve step, 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.

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. In the case of a P-bag, the distance distributions between endpoints can be computed using a MIN convolution – denoted in the following as $\odot$ – of all the parallel edges in the bag. This computation is linear in the maximum distance of the input distributions. In the case of an S-bag, the endpoint distribution can be computed by applying a SUM convolution of the path between $v_1$ and $v_2$ passing through the other vertices in the bag – denoted as $\oplus$ – followed by a MIN convolution with the direct edge distribution. The SUM convolution is quadratic in the maximum distance of the input distributions. For more details on the computation of convolutions of probability distributions, we refer the reader to [2].

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 $(\varepsilon, \delta)$ multiplicative guarantee. We can then use the per-bag guarantees to compute the overall guarantees on the distributions in the root bag, in the spirit of [18].

Algorithm 2 shows 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.

***Retrieval.*** When answering $(s, t)$ ST-queries on the ProbTree we have two main cases. First, when both $s$ and $t$ are present in the root node, we only need to query the root bag without the 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.

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

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

**for** *distribution* $p^c(u \to v)$ *in children of* $B$ **do**
  $p(u \to v) \leftarrow p(u \to v) \odot p^c(u \to v)$ ;
**for** *edge* $v_1 \to v_2$ *between endpoints* $v_1, v_2$ **do**
  **if** $p^c(v_1 \to v_2) \notin \mathsf{computed}(B)$ **then**
    **if** $\mathsf{type}(B) = R$ **then**
      $p^c(v_1 \to v_2) \leftarrow \mathtt{sample}(v_1, v_2, B)$ ;
    **else if** $\mathsf{type}(B) = S$ **then**
      $p'(v_1 \to v_2) \leftarrow p(v_1 \to u_1) \oplus \cdots \oplus p(u_j \to v_2)$ ;
      $p^c(v_1 \to v_2) \leftarrow p(v_1 \to v_2) \odot p'(v_1 \to v_2)$ ;
    add $p^c(v_1 \to v_2)$ to $\mathsf{computed}(B)$;

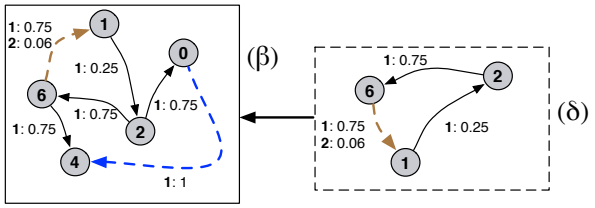



Figure 3: Retrieval for the pair $(1, 4)$.

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. Which node is chosen is not important – 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 retrieve. Algorithm 3 presents this operation in detail.

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

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

root the tree at one of the bags containing $t$;
**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** $\mathcal{B}(\text{root}(\mathcal{T}))$

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

*First, since* 1 *and* 4 *are on the same branch of* $\mathcal{T}$*, we can root the tree at bag* $(\beta)$*. Moreover, one can notice that there is no need to recompute endpoint distributions on bags* $(\alpha)$*,* $(\gamma)$*, and* $(\varepsilon)$*. Hence, the computed edge* $6 \to 1$ *is used from bag* $(\varepsilon)$ *and* $0 \to 4$ *from* $(\alpha)$*. However, the computed edges* $6 \to 2$ *and* $2 \to 6$ *are not propagated from bag* $(\delta)$ *to bag* $(\beta)$*, as their computation involves a query vertex, in this case vertex 1. Hence, all vertices and edges from bag* $(\delta)$ *are propagated to bag* $(\beta)$*, and joined by the original edge in* $(\alpha)$*,* $0 \to 4$*. The resulting graph in the new root – bag* $(\beta)$ *– is a graph which outputs equivalent results for the query on* $(1, 4)$ *as the original graph in Figure 1a.*

*Properties.* It is easy to check that the above operators define an efficient representation where queries run faster on the decompressed graph than on the original graph. Theorem 1 ensures the validity of the approach. The implementation of SPQR trees of [12] 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 4. Moreover, it is known that the number of skeleton edges added in the triconnected components tree is $\mathsf{O}(E)$ (more precisely, it is upper-bounded by $3|E| - 6$, as shown in [19]), thus verifying Property (ii).

Each retrieve outputs a graph that is at most as big as the original graph, and hence the standard shortest-path algorithms [8] would execute in less time for each sample. Moreover, retrieve is linear in the number of tree bags, which is itself linear in the size of $\mathcal{G}$, verifying Property (iii). Hence $(\mathsf{index}^{\text{SPQR}}, \mathsf{retrieve}^{\text{SPQR}})$ is an efficient representation.

**Note.** SPQR trees are not the only way to decompose a graph to obtain a ProbTree. We have experimented with *partial fixed-width decompositions* of the graph, and found they can be useful (and indeed, sometimes more efficient than SPQR trees) when one desires lossless representations, but with a "weaker" decomposition. Due to space reasons, we omit them here.

## 5. EXPERIMENTAL EVALUATION

We now report on our experimental evaluation showing the efficiency of SPQR trees for indexing probabilistic graphs.

*Datasets and setup.* We use two probabilistic graph datasets, from different application domains:

1. The Wiki dataset, representing Wikipedia[1] 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

[1] `http://en.wikipedia.org/`

Table 1: ProbTree properties ($\mathcal{R}$ is the root bag)

| Graph | Type | $\mathcal{R}$ vertices | $\mathcal{R}$ edges | $\mathcal{T}$ height |
|---|---|---|---|---|
| Wiki | orig. | 109,694 | 1,568,754 | 0 |
| | SPQR | 41,268 | 296,714 | 536 |
| NH | orig. | 66,627 | 159,694 | 0 |
| | SPQR | 45,777 | 112,676 | 9 |

two authors agree on a topic. The graph has 109,694 vertices and 1,568,754 edges.

2. The United States road network graphs[2], 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 the NH road network of 66,627 vertices and 159,694 edges.

Our ProbTree framework was implemented in C++, and all experiments were run on a Linux machine with a quad-core 3.6 GHz CPU and 48 GB of RAM. The deterministic part of the SPQR decomposition was done using the implementation in the Open Graph Drawing Framework library[3].

***ProbTree properties.*** 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.

Table 1 shows the properties of applying ProbTree on the four graphs, containing the number of vertices and edges in the root bag. It can be noted that the best decomposition for SPQR is achieved in the Wiki graph, which is also the densest graph. The index operator is very efficient, running in the order of seconds even on large graphs. In NH the running time is 23 seconds, while on Wiki it is 40 seconds. Moreover, the space overhead of $\mathcal{I}$ reasonable. Generally, ProbTree only incurs between roughly 10% (Wiki, 32MB from 30MB for the original graph) and quadruple (NH, 16MB from 4.5MB for the original graph) space overhead compared to the space cost of the original graph.

***Running time.*** For evaluating the execution time and query accuracy, we used the following experimental setup. For each dataset, a 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. For each query pair we generated the actual distance distribution between the vertices, by applying Dijsktra'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 4 shows, the efficiency gains are important when queries are executed on ProbTree decompositions. The gains range from a factor of 2 in the case of NH to 5 for Wiki. ProbTree based on SPQR performs extremely well in the denser graph, Wiki.

The retrieve time does not influence significantly the execution of the queries. In the worst case, SPQR for NH, it is roughly 2% of the execution time for 1,000 samples, and under 0.2% for Wiki. Also, the average number of bags needing re-computation is very small, again under 3% for

[2] http://www.dis.uniroma.it/challenge9/data/tiger
[3] http://ogdf.net/

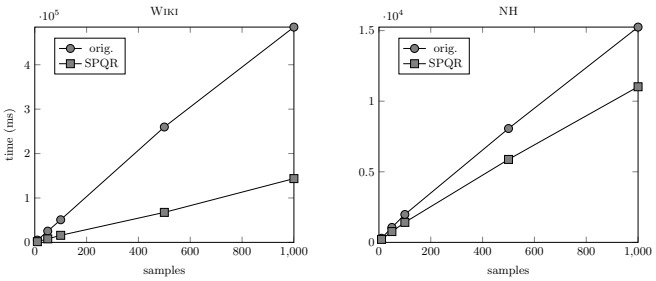



Figure 4: Running time for ProbTree versus rounds of samples

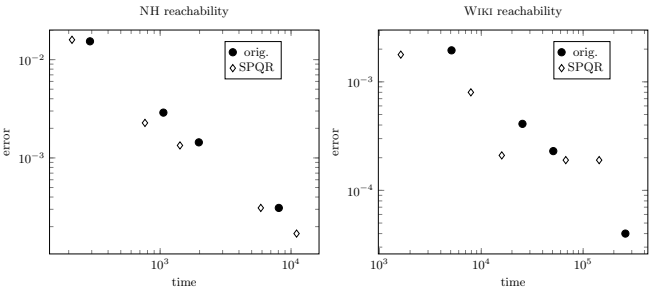



Figure 5: Relative error vs. time (log-log axis)

NH and under 0.05% for Wiki – out of the total number of bags in the tree decomposition.

***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 enough to justify using indexing, and not simply more sampling rounds? To check this, we have plotted the running time of applying sampling on ProbTree versus its error – expressed in terms of the mean squared error as compared to the ground truth results. For brevity, we only track the results for reachability – or 2-terminal reliability – queries. As query answers are derived directly from the distance distribution, results for other types of queries are equivalent.

Figure 5 presents the results for the NH and Wiki graphs (note the log-log axes). The black dots represent the results on sampling the original graph, for sample rounds between 10 and 1,000. Intuitively, we want the points corresponding to ProbTree (drawn for the same amount of samples) to lie "below" the line induced by the black points, meaning that they yield a better time-accuracy trade-off. As seen before, the gains in execution time when using the decompositions are important. The results also show that the relative error can be even slightly improved when using ProbTree. For instance, note that the white dots in the NH graph are slightly lower than the corresponding black dots, suggesting an increase in accuracy. In the case of the Wiki graph, the errors are lower for SPQR when the number of samples is less than 500. Note that the error is relatively constant for SPQR after a point, suggesting a lower bound of error due to the sampling in the R-bags of the SPQR tree. This suggests that SPQR in Wiki is best to be used in conjunction with a lower number of samples, and that its appeal is mainly directed at denser graphs.

Table 2: Distance-constraint reachability running time (sec) and error ratios (between parentheses) for three estimators in [15]

| **Decomp.** | RHH | RHT | Dagger |
|---|---|---|---|
| orig. | 0.095 (0.122) | 0.123 (0.109) | 0.631 (0.225) |
| SPQR | 0.050 (0.071) | 0.061 (0.073) | 0.338 (0.129) |

*Comparison with other algorithms.* One of our arguments in using ProbTree as a pre-computed index is that it can be applied directly to existing solutions. To check this, we apply the distance-constraint reachability (DCR) estimators studied in [15] to the SPQR decomposition 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 $E = |\hat{R} - R|/R$, where $\hat{R}$ is the result of an estimator and $R$ is the result of the exact computation.

Table 2 summarizes the results. First of all, it can be easily noted that, indeed, applying ProbTree decompositions directly affects the running time of any of the three estimators, with a two-fold increase in efficiency. Applying ProbTree also increases accuracy – in terms of the error ratio – for all three estimators. These results complement the results in Figure 5 and suggest that the fact some edges are already pre-computed minimizes the chance of sampling error.

# 6. CONCLUSIONS

In this paper, we studied efficient ST-query evaluation in probabilistic graphs. We formally define the notion of transformation on such graphs, and propose the ProbTree. We design an SPQR tree variant of ProbTree, and, through extensive experiments on real datasets, we show that it enhances query processing, is easy to compute, and has low space costs. The graphs produced by ProbTree can also easily be used by existing querying algorithms.

## Acknowledgments

This work has been partly funded by Télécom ParisTech's Chair on Big Data and Market Insights.

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

[3] 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.

[4] P. Barceló, L. Libkin, and J. L. Reutter. Querying graph patterns. In *PODS*. ACM, 2011.

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

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

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

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

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

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

[11] 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 *INFOCOMM*, 2007.

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

[13] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6), 1973.

[14] M. Hua and J. Pei. Probabilistic path queries in road networks. In *EDBT*, 2010.

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

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

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

[18] A. Souihli and P. Senellart. Optimizing approximations of DNF query lineage in probabilistic XML. In *ICDE*, 2013.

[19] W. T. Tutte. Connectivity in graphs. *Mathematical Expositions*, 15, 1966.

[20] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3), 1979.

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