Subgraph Pattern Matching over Uncertain Graphs with Identity Linkage Uncertainty

Walaa Eldin Moustafa[1], Angelika Kimmig[2], Amol Deshpande[1], Lise Getoor[1]

[1] *Department of Computer Science, University of Maryland, College Park, USA*
[2] *Department of Computer Science, KU Leuven, Belgium*
{walaa,amol,getoor}@cs.umd.edu, angelika.kimmig@cs.kuleuven.be

Abstract— **There is a growing need for methods that can represent and query uncertain graphs. These uncertain graphs are often the result of an information extraction and integration system that attempts to extract an entity graph or a knowledge graph from multiple unstructured sources [25], [7]. Such an integration typically leads to *identity uncertainty*, as different data sources may use different *references* to the same underlying real-world *entities*. Integration usually also introduces additional uncertainty on node attributes and edge existence. In this paper, we propose the notion of a *probabilistic entity graph* (PEG), a formal model that uniformly and systematically addresses these three types of uncertainty. A PEG is a probabilistic graph model that defines a distribution over possible graphs at the entity level. We introduce a general framework for constructing a PEG given uncertain data at the reference level and develop efficient algorithms to answer *subgraph pattern matching* queries in this setting. Our algorithms are based on two novel ideas: *context-aware path indexing* and *reduction by join-candidates*, which drastically reduce the query search space. A comprehensive experimental evaluation shows that our approach outperforms baseline implementations by orders of magnitude.**

I. Introduction

Querying relational data that has been extracted from various sources is beneficial in a variety of application domains, such as online social networks, the Web, communication networks, bioinformatics and financial data management. The relational structure of such data can conveniently be represented in the form of graphs or networks, which allows one to express queries for groups of objects with a given attribute and link structure as *subgraph pattern matching* tasks. However, to obtain high quality answers, it is crucial to explicitly take into account the uncertainty inherent in the data when querying.

The reasons for uncertainty are manyfold. As different data sources often use different *references* to the same real world object or *entity*, one needs to rely on automatic approaches, such as entity resolution techniques [3], [1], [10], to identify entities during integration; use of such techniques introduces *identity uncertainty* in the data. Identity uncertainty in turn leads to other types of uncertainty, including uncertainty about attribute values and edge existence. The latter types of uncertainty may also come directly from the data, for instance, from an information extraction system that associates a confidence with an extraction [26], [4]. Despite the many scalable algorithms and indexing techniques for analyzing and querying graphs that have been developed recently, e.g., [29], [6], [11], [35], [9], and that also address uncertain data, e.g., [5], [16], [19], [22], [30], methods that *jointly* address these different types of uncertainties are still lacking.

In this work, we propose the notion of a *probabilistic entity graph*, PEG, a general abstract probabilistic graph model that combines three common types of uncertainty. Specifically, we consider: 1) *identity uncertainty*, that is, uncertainty about whether each real world entity is represented by one or multiple objects or references in the data, 2) uncertainty about the attribute values of nodes (i.e., *attribute value uncertainty*), and 3) uncertainty about whether particular edges exist (i.e., *edge existence uncertainty*). In addition, we develop techniques for efficiently answering subgraph pattern queries over such uncertain graphs. We show that our model defines a probability distribution over possible graphs describing entities, their attributes and relationships among them. We then introduce techniques to find all matches of a subgraph pattern that have probability above a given threshold. Answering subgraph pattern matching queries is NP-hard on non-probabilistic graphs, and becomes #P-complete when including identity uncertainty. Nonetheless, we propose and systematically explore a range of novel techniques to prune the search space and effectively perform subgraph pattern matching over large-scale uncertain graphs.

To summarize, we make the following contributions:

- We introduce *probabilistic entity graphs*, a general uncertain graph model that captures identity, attribute and edge uncertainties.
- We define the semantics of a probabilistic entity graph as a probability distribution over possible entity graphs.
- We develop scalable algorithms to answer subgraph pattern matching queries over such uncertain graph data, based on *query path decomposition*.
- We present a novel graph indexing method, *context-aware path indexing*, to capture information about the graph paths, their surrounding structures, and their probabilities, enabling efficient retrieval of candidate matches.
- We propose *reduction by join-candidates*, an algorithm that efficiently prunes candidate answers by progressively propagating structural and probabilistic information between the candidates.
- We demonstrate that our approaches can evaluate complex queries over graphs with millions of nodes and edges in seconds, outperforming a baseline implementation by orders of magnitude.

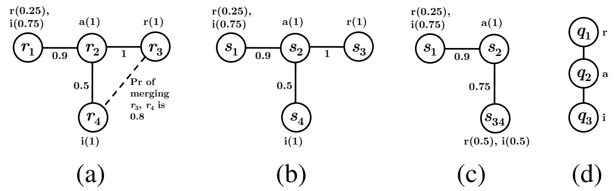


Fig. 1. (a) Reference-level network, (b), (c) the two possible entity graphs, (d) a query graph

II. Motivating Example

As a simple motivating example, consider a system to help organizations find experts in different domains. The system integrates information about experts and their affiliations from multiple sources. Assume three sources: an online professional network (e.g., LinkedIn), an online social network (e.g., Facebook), and personal webpages or blogs. The system uses the experts' names, their affiliations (`Academia` (a), `Research Lab` (r), or `Industry` (i)), and relationships between experts. Figure 1 illustrates a small example, where we omit names for clarity. We use the term *reference* to denote the *observed objects*, which in this example are strings encoding names, while we use the term *entity* to refer to *real-world objects*, that is, the experts in our case. A real-world object may thus correspond to a collection of references, as names may be abbreviated, misspelled, etc. In Figure 1(a), nodes represent references, letters inside nodes represent reference IDs, and letters outside nodes represent *labels*, that is, affiliations, along with their probabilities in parentheses, representing node attribute uncertainty. Consider node r_1, extracted from a personal webpage. Suppose that a text analysis method suggests that the name is “Gerald Maya” and the affiliation is `Industry` with probability 0.75 and `Research Lab` with probability 0.25 (an instance of *attribute value uncertainty*). Nodes r_2 and r_3 are extracted from an online professional network, with name “Becky Castor” and an affiliation with `Academia`, and the name “Christopher Tucker” and an affiliation with a `Research Lab`, respectively. Finally, node r_4 is extracted from an online social network, with the name “Chris Tucker” and an `Industry` affiliation. Relationships, such as co-worker, friend, or class-mate, between the individuals are extracted (represented as edges in the figure) and are associated with probabilities that reflect the likelihood of the relationship's existence (i.e., *edge existence uncertainty*). These probabilities can be calculated based on whatever information is available from these online resources, such as the number of common connections or shared attributes between them. Finally, to encode that “Christopher Tucker” and “Chris Tucker” may refer to the same person, we put them together in the same *reference set* (depicted as a dashed line in the figure). To quantify such *identity uncertainty*, i.e., the uncertainty about whether multiple references refer to the same real-world entity, we assign this reference set a probability of 0.8, denoting the likelihood that the elements in the set correspond to a single real-world entity.

Figures 1(b) and (c) illustrate the two possible sets of entities with their labels and relations for the example reference network shown in Figure 1(a), where the letters inside the nodes represent entity IDs. Figure 1(b) depicts the entity graph in which r_3 and r_4 remain unmerged, i.e., are assumed to be separate real-world entities ($pr = 0.2$); Figure 1(c) depicts the one where they are assumed to refer to the same person ($pr = 0.8$), and thus merged into a new node s_{34} with its own label and edge probability distributions. Going from a set of references to an entity requires merging the information associated with the references, i.e., their labels and the relationships they participate in. In this example, we average the probability distributions. Since r_3 has label r and r_4 has label i, we assign a label distribution of $\{r(0.5), i(0.5)\}$ to entity s_{34}. Similarly, s_{34} has an edge to s_2 with $pr = 0.75$ (average of r_3's edge with $pr = 1$ and r_4's edge with $pr = 0.5$).

Clearly, we want to specify queries to our information system at the level of entities rather than references. In this work, we focus on subgraph pattern matching queries, perhaps the most widely used and studied class of queries over graphs. Figure 1(d) depicts a query which asks for all paths of length 2 over nodes labeled (r, a, i). A query also specifies a minimum threshold α ($\alpha = 0.25$ in the example), to indicate that only matches with probability larger than α should be returned. In this simple case, we can answer our query by examining all possible matches. In the entity graph in Figure 1(b), with r_3 and r_4 unmerged, the nodes (s_3, s_2, s_4) form a path with the required labels. The probability of that path is computed by multiplying the three node label probabilities (1, 1, 1), the two edge probabilities (1, 0.5), and the probability that the nodes r_3 and r_4 are *not* merged (0.2); resulting in a match probability of 0.1, which is below our cutoff of 0.25. The other two potential matches, (s_1, s_2, s_4) and (s_3, s_2, s_1), do not satisfy the minimum threshold constraint either. The second entity graph in Figure 1(c) contains two potential matches for the query: (s_1, s_2, s_{34}) with probability 0.084, and (s_{34}, s_2, s_1) with probability 0.253. Therefore, (s_{34}, s_2, s_1) is the only answer to our query. Clearly, such an exhaustive approach is infeasible for larger graphs. In this work, we therefore develop a scalable approach to answer subgraph pattern matching queries in this setting.

III. Uncertain Graph Modeling

We now discuss our formal model for the types of uncertainties arising in the example above, where we are given information about *references*, or mentions of objects, but are interested in queries about *entities*, or the objects themselves. We introduce *probabilistic entity graphs*, which define a probability distribution over graphs, where nodes correspond to entities, node labels to entities' attributes, and edges to the relations between them. The key challenge here is that references induce constraints on which entity nodes can co-occur in the same graph, as each graph structure corresponds to one possible way of assigning references to existing entities. To deal with these dependencies, we represent our probability distribution as a *probabilistic graphical model* (PGM) [18]. After a quick summary of the necessary basics, we introduce the notion of a *probabilistic graph description* (PGD), and show how to derive the actual PGM, our probabilistic entity graph, from the PGD. We first focus on the basic case, where distributions over labels and edges are all independent, and then show how additional dependencies can directly be introduced.

A PGM $\mathcal{P} = \langle \mathcal{V}, \mathcal{F} \rangle$ defines a joint probability distribution over its random variables $\mathcal{V}$ via its set of factors $\mathcal{F}$. Each factor $f \in \mathcal{F}$ is defined over a subset $\mathcal{V}_f$ of $\mathcal{V}$ and represents a dependency between those random variables. Given a complete joint assignment $\boldsymbol{v} \in Dom(\mathcal{V})$ to the variables in $\mathcal{V}$, the joint distribution is defined by $Pr(\boldsymbol{v}) = \frac{1}{\mathcal{Z}} \prod_{f \in \mathcal{F}} f(\boldsymbol{v}_f)$, where $\boldsymbol{v}_f$ denotes the assignments restricted to the arguments $\mathcal{V}_f$ of f and $\mathcal{Z} = \sum_{\boldsymbol{v}' \in Dom(\mathcal{V})} \prod_{f \in \mathcal{F}} f(\boldsymbol{v}'_f)$ is a normalization constant referred to as the *partition function*. The independencies in the distribution defined by a PGM are represented graphically in its *Markov network*, which contains one node for each random variable, and an edge between a pair of random variables if and only if the two variables co-occur in some factor. Each connected component in the Markov network corresponds to a part of the model that is *independent* from the rest. We can thus compute the normalized probability for each connected component separately and multiply them together to obtain the full joint distribution.

As a first step towards our probabilistic model, we introduce random variables for labels of references ($r.x$), existence of edges between pairs of references ($(r_1, r_2).x$), and existence of an entity corresponding to a set of references ($s.x$), together with a probability distribution for each of them.

Definition 1: **Probabilistic Graph Description:** A probabilistic graph description (PGD) is a tuple $D = (R, S, \Sigma, P, m^{\Sigma}, m^{\{T,F\}})$, where R is a set of references, S is a set of subsets of R (each of them a potential entity) including at least all singleton subsets, Σ is a set of labels, and:

- P is a set of probability distributions containing (1) for each $r \in R$, a probability distribution $p^r(r.\boldsymbol{x})$ over a random variable $r.x$ with values from Σ, (2) for each $(r_1, r_2) \in R \times R$, a probability distribution $p^{(r_1,r_2)}((r_1, r_2).\boldsymbol{x})$ over a random variable $(r_1, r_2).x$ with values from $\{T, F\}$, and (3) for each $s \in S$, a probability distribution $p^s(s.\boldsymbol{x})$ over a random variable $s.x$ with values from $\{T, F\}$.
- The merge functions m^{Σ} and $m^{\{T,F\}}$ transform a set of probability distributions over random variables with values in Σ and $\{T, F\}$, respectively, into a single distribution.

Note that a PGD is not a graphical model, but simply specifies the objects and observations as well as a set of basic, independent probability distributions over those. This includes the set of observed references R ($\{r_1, \ldots, r_4\}$ in Figure 1(a)) together with their possible labels Σ ($\{a, r, i\}$ in the example); it also includes the probabilities for the existence of edges between two references, and the set S of potential entities, where each entity is represented by a subset of the references ($S = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}, \{r_3, r_4\}\}$ in the example). The PGD specifies independent probability distributions for the existence of such entities, as well as merge functions that specify how to combine distributions of labels and edges if references are merged into an entity, for instance, by averaging, or, in the case of Boolean domains, by forming the disjunction of the input distributions.

In the next step of our model construction, the probabilistic entity graph combines these independent probability distributions into a coherent graphical model that encodes the dependencies between entities induced by shared references and combines the distributions over labels and edges using the merge functions provided by the PGD.

Definition 2: **Probabilistic Entity Graph:** For a given PGD D, the probabilistic entity graph (PEG) U is the graphical model $\langle \mathcal{V}, \mathcal{F} \rangle$ whose set of random variables $\mathcal{V}$ contains, for each $s \in S$, a node existence variable $s.n$ and a label variable $s.l$, and for each pair $(s_1, s_2) \in S \times S$, an edge existence variable $(s_1, s_2).e$, and whose set of factors $\mathcal{F}$ is defined as follows. For each $r \in R$ with $S^r = \{s_1, \ldots, s_k\} = \{s \in S | r \in s\}$, $\mathcal{F}$ contains a node existence factor

$$f^N(s_1.n = v_1, \ldots, s_k.n = v_k) =$$
$$\begin{cases} p^s(s_i.x = T) & \text{if } v_i = T \text{ and, for all } j \neq i, v_j = F \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

For each $s \in S$, $\mathcal{F}$ contains a node label factor

$$Pr(s.\boldsymbol{l}) = \left[m^{\Sigma}(\{p^r | r \in s\})\right](s.\boldsymbol{l}) \quad (2)$$

For each $(s_1, s_2) \in S \times S$, $\mathcal{F}$ contains an edge existence factor

$$Pr((s_1, s_2).\boldsymbol{e}) = \left[m^{\{T,F\}}(\{p^{(r_1,r_2)} | r_i \in s_i\})\right]((s_1, s_2).\boldsymbol{e}) \quad (3)$$

Identity uncertainty is modeled by the node existence factors ($f^N(s_1.n = v_1, \ldots, s_k.n = v_k)$), which ensure that all assignments where two entity nodes share a reference have zero probability. The node label factors ($Pr(s.\boldsymbol{l})$) are probability distributions obtained by aggregating the label probability distributions of all references in the underlying set s via the node label merge function. In the same way, the edge existence factors ($Pr((s_1, s_2).\boldsymbol{e})$) are probability distributions obtained by aggregating the edge existence probability distributions of all pairs of references from the underlying sets via the edge existence merge function.

Example: In Figure 1(a), we choose to use *pointwise average* as the merge function. Since $p^r(r_3.x = \text{r}) = 1$ and $p^r(r_4.x = \text{i}) = 1$, the node s_{34} (formed by merging $\{r_3, r_4\}$) has label distribution $Pr(s_{34}.l = \text{r}) = Pr(s_{34}.l = \text{i}) = 0.5$. Similarly, $Pr((s_{34}, s_2).e = T) = 0.75$, as the reference level edges (r_3, r_2) and (r_4, r_2) have probabilities 1 and 0.5 respectively.

Further, in the example, $p^s(s_3.x = T) = p^s(s_4.x = T) = 0.25$, and $p^s(s_{34}.x = T) = 0.5$ (Figure 1 only shows the final normalized probabilities, computed by following the process described below). Hence, the node existence factor for r_3 is: $f^N(s_3.n = T, s_{34}.n = F) = 0.25, f^N(s_3.n = F, s_{34}.n = T) = 0.5$, and 0 otherwise (note that the factors are not pdfs, and hence do not have to sum up to 1). Similarly, the node existence factor for r_4 is: $f^N(s_4.n = T, s_{34}.n = F) = 0.25, f^N(s_4.n = F, s_{34}.n = T) = 0.5$, and 0 otherwise. The factor for r_1 is: $f^N(s_1.n = T) = 1, f^N(s_1.n = F) = 0$, and the factor for r_2 is: $f^N(s_2.n = T) = 1, f^N(s_2.n = F) = 0$.

Exploiting Independence: Writing out the probability distribution defined by the PEG, we have

$$Pr(S.\boldsymbol{n}, S.\boldsymbol{l}, (S \times S).\boldsymbol{e}) = \frac{1}{\mathcal{Z}} \cdot \prod_{r \in R} f^N(S^r.\boldsymbol{n}) \cdot \prod_{s \in S} Pr(s.\boldsymbol{l}) \cdot \prod_{(s_1,s_2) \in S \times S} Pr((s_1, s_2).\boldsymbol{e}) \tag{4}$$

We use shorthand notation for assignments to sets of random variables, e.g., $S.\boldsymbol{n}$ for $s_1.n = n_1, \ldots, s_{|S|}.n = n_{|S|}$. The partition function $\mathcal{Z}$ is the sum of the factor product over all variable assignments. As all node label and edge existence factors are probability distributions independent of all other factors, Eq. 4 is equivalent to

$$Pr(S.\boldsymbol{n}, S.\boldsymbol{l}, (S \times S).\boldsymbol{e}) = Pr(S.\boldsymbol{n}) \cdot \prod_{s \in S} Pr(s.\boldsymbol{l}) \cdot \prod_{(s_1,s_2) \in S \times S} Pr((s_1, s_2).\boldsymbol{e}) \tag{5}$$

where $Pr(S.\boldsymbol{n})$ is the normalized product of all node existence factors, that is, the partition function $\mathcal{Z}_n$ is with respect to those factors only:

$$Pr(S.\boldsymbol{n}) = \frac{1}{\mathcal{Z}_n} \prod_{r \in R} f^N(S^r.\boldsymbol{n}) \tag{6}$$

This function can often be decomposed further, based on the independencies encoded in the Markov network. Let $\mathcal{C}(S.n)$ be the partitioning of the set of random variables $S.n$ induced by the connected components of the Markov network, that is, each element of $\mathcal{C}(S.n)$ contains all random variables participating in one such component. We then obtain

$$\begin{aligned} Pr(S.\boldsymbol{n}) &= \prod_{S_i.n \in \mathcal{C}(S.n)} \frac{1}{\mathcal{Z}_{n_i}} \prod_{r \in R \wedge S^r \subseteq S_i} f^N(S^r.\boldsymbol{n}) \\ &= \prod_{S_i.n \in \mathcal{C}(S.n)} Pr(S_i.\boldsymbol{n}) \end{aligned} \tag{7}$$

where the partition function $\mathcal{Z}_{n_i}$ normalizes over all assignments for random variables in $S_i.n$.

Example: In our example, there are three connected components, i.e., $\mathcal{C}(S.n) = \{\{s_1.n\}, \{s_2.n\}, \{s_3.n, s_4.n, s_{34}.n\}\}$. For the first two, f^N is already a probability distribution, and thus $\mathcal{Z}_{n_1} = \mathcal{Z}_{n_2} = 1$. For the third, the product of the two corresponding factors has non-zero value for two assignments: $f^N(s_3.n = T, s_4.n = T, s_{34}.n = F) = f^N(s_3.n = T, s_{34}.n = F) \times f^N(s_4.n = T, s_{34}.n = F) = 0.0625$, and $f^N(s_3.n = F, s_4.n = F, s_{34}.n = T) = f^N(s_3.n = F, s_{34}.n = T) \times f^N(s_4.n = F, s_{34}.n = T) = 0.25$.
Hence, $\mathcal{Z}_{n_3} = 0.25 + 0.0625 = 0.3125$,
$Pr(s_3.n = T, s_4.n = T, s_{34}.n = F) = 0.0625/0.3125 = 0.2$,
$Pr(s_3.n = F, s_4.n = F, s_{34}.n = T) = 0.25/0.3125 = 0.8$; the last two are the final probabilities of r_3 and r_4 not being merged (Figure 1(b)), and merged (Figure 1(c)), respectively.

Distribution over Graphs: Clearly, not all assignments to random variables in the model above directly correspond to legal graphs. We now show how to obtain the final distribution over labeled graphs. The set $PW(U)$ of possible world graphs of a PEG U consists of those graphs $W = (V, E, l(.))$ where V is a set of entity nodes corresponding to reference sets from S (merged into a single entity), $E \subseteq V \times V$ is a set of edges between them, and the label function $l : V \to \Sigma$ labels these nodes with elements of Σ. Slightly abusing notation, we identify a graph node $v \in V$ with the corresponding set of references $s \in S$ (thus treating V as a subset of S), and use both notations interchangeably. Each possible world graph W induces a partial value assignment $(S.\boldsymbol{n}^W, V.\boldsymbol{l}^W, (V \times V).\boldsymbol{e}^W)$ to the random variables in the graphical model as follows. For each $s \in V$, we have $s.n^W = T$, and for each $s \in S \setminus V$, we have $s.n^W = F$, that is, values of node existence variables mirror the (non-)existence of nodes in W. For each $s \in V$, we have $s.l^W = l(s)$, that is, for all existing nodes, values of node label random variables mirror the labels in W, and all other node label random variables remain unassigned. For all $(s_1, s_2) \in E$, we have $(s_1, s_2).e^W = T$, and for all $(s_1, s_2) \in (V \times V) \setminus E$, we have $(s_1, s_2).e^W = F$, that is, for all pairs of existing nodes, edge existence variables mirror the (non-)existence of edges in the graph, and all other edge existence random variables remain unassigned. The probability of W is now obtained based on Equation 5 by marginalizing over all unassigned variables. As those all appear in independent factors only, we get

$$Pr((V, E, l(.))) = Pr(S.\boldsymbol{n}^W) \cdot \prod_{v \in V} Pr(v.l = l(v)) \cdot \prod_{(s_1,s_2) \in E} Pr((s_1, s_2).e = T) \cdot \prod_{\substack{(s_1,s_2) \in \\ (V \times V) \setminus E}} Pr((s_1, s_2).e = F) \tag{8}$$

As every full assignment to the variables in the graphical model contributes to exactly one graph's probability, this defines a probability distribution over possible world graphs.

Introducing Correlations: Our model can easily be adapted to introduce additional correlations, such as edge existence probabilities depending on node labels. This is done by replacing the edge existence probabilities $p^{(r_1,r_2)}((r_1, r_2).\boldsymbol{x})$ in the PGD by conditional probabilities $p^{(r_1,r_2)}((r_1, r_2).\boldsymbol{x_0}|r_1.\boldsymbol{x_1}, r_2.\boldsymbol{x_2})$, The edge existence factors in the PEG now include three random variables, with values $Pr((s_1, s_2).\boldsymbol{e}|s_1.\boldsymbol{l_1}, s_2.\boldsymbol{l_2})$ obtained by applying the merge function in Equation 3 to the conditional distributions:

$$\begin{aligned} &Pr((s_1, s_2).\boldsymbol{e}|s_1.\boldsymbol{l_1}, s_2.\boldsymbol{l_2}) = \\ &\left[m^{\{T,F\}}(\{p^{(r_1,r_2)}|r_i \in s_i\})\right]((s_1, s_2).\boldsymbol{e}|s_1.\boldsymbol{l_1}, s_2.\boldsymbol{l_2}) \end{aligned} \tag{9}$$

Since there are no cyclic dependencies between random variables, the product of these factors still is a normalized probability distribution, which allows us to use the new edge existence factors instead of the previous ones in the full joint distribution (Equation 4), its factorization (Equation 5), and the probability of possible world graphs (Equation 8).

IV. Subgraph Pattern Matching

We now define the task of subgraph pattern matching over uncertain graphs based on the notion of match in graphs without uncertainty. We assume undirected graphs, but our approaches are equally applicable to directed graphs. A query graph $Q = (V_Q, E_Q)$ is a graph where each node $v \in V_Q$ is labeled with a label $l_Q(v) \in \Sigma$.

Definition 3: **Match:** Given a labeled graph $G = (V_G, E_G, l_G(.))$ and a query graph $Q = (V_Q, E_Q, l_Q(.))$, a subgraph $M = (V_M, E_M)$ of G is a match of Q in G if and only if there is a bijective mapping $\psi : V_Q \to V_M$ such that (i) $\forall u \in V_Q : l_Q(u) = l_G(\psi(u))$ and (ii) $(\psi(u), \psi(v)) \in E_M$ if and only if $(u, v) \in E_Q$.

Definition 4: **Probabilistic Match:** A graph M is a probabilistic match of a query graph Q in a PEG U if and only if M is a match of Q in at least one legal possible world graph G of U, that is, one where no two nodes share a reference. The probability of the match M is the sum of the probabilities of all possible world graphs of U where M is a match:

$$Pr(M) = \sum_{G \in PW(U) \wedge M \subseteq G} Pr(G) \tag{10}$$

Definition 5: **Probabilistic Subgraph Pattern Matching:** Given a PEG U, a query graph Q, and a probability threshold α, find all matches of Q in U such that $Pr(M) \geq \alpha$.

Naively, this problem could be solved by performing subgraph pattern matching over each possible world graph and for each match found, summing the probabilities of possible worlds it appears in. Clearly, this approach is computationally infeasible. In the remainder of this section, we show how to (a) find all matches by performing subgraph matching on a single graph only, and (b) calculate the probability of a given match directly, without need to explicitly consider all possible worlds it appears in. This provides the basis for the algorithms discussed in Section V, which further speed up probabilistic subgraph pattern matching.

Finding Matches: For a given PEG U, let G_U be the graph that has a node for each $s \in S$, labeled with the set of labels $L(s)$ that are associated with s with non-zero probability, that is, $L(s) = \{l' | l' \in \Sigma \wedge Pr(s.l = l') > 0\}$, and an edge between two nodes s_1 and s_2 if and only if $Pr((s_1, s_2).e = T) > 0$. To obtain a one-to-one correspondence between matches on G_U and probabilistic matches on U, we generalize the notion of match on G_U to (a) require the query node label to be in the set of labels of the matched node, and (b) only return matches where no two nodes share a reference. For the discussions to follow, we use the term probabilistic entity graph to denote G_U as well, as it is the structure that our algorithms operate on.

Calculating Probabilities: The probability of a match M on G_U (Equation 10) is the sum of the probabilities of a set of possible world graphs (Equation 8). As the graphs in this set are exactly those containing all nodes in V_M with correct labels as well as all edges in E_M, and arbitrary sets of additional nodes and edges, the probability of M equals the marginal

$$Pr(M) = Pr_n(M) \cdot Pr_{le}(M) \tag{11}$$

$$Pr_n(M) = Pr(V_M.n = T) \tag{12}$$

$$Pr_{le}(M) = \prod_{v \in V_M} Pr(v.l = l(v)) \cdot \prod_{e \in E_M} Pr(e.e = T) \tag{13}$$

where $Pr(V_M.n = T)$ is the corresponding marginal of $Pr(S.n)$ that sums out values of all node existence variables whose nodes are not part of M. In practice, as in Equation 7, we further exploit independencies in the underlying graphical model. Recall that $\mathcal{C}(S.n)$ partitions the set of node existence random variables $S.n$ based on the connected components of the Markov network. As each node in a match corresponds to one such random variable, we can use the same partitioning, restricted to the set of nodes V_M in the match, to calculate $Pr(V_M.n = T)$ as $\prod_{C.n \in \mathcal{C}(S.n)} Pr((V_M.n \cap C.n) = T)$. Note that $Pr_{le}(M)$ is *subgraph decomposable*, i.e., for two disjoint subgraphs M_1 and M_2, $Pr_{le}(M_1) \times Pr_{le}(M_2) = Pr_{le}(M_1 \cup M_2)$, but $Pr_n(M)$ is not.

V. Algorithms

The problem of probabilistic subgraph pattern matching with identity uncertainty is #P-complete. To increase efficiency, we propose a new *path-based* solution to this problem, which decomposes the query into a set of paths, finds matches of individual paths, and exploits probabilistic information to prune the space of possible matches. We focus on paths rather than nodes only when finding candidate matches, as this takes into account more probabilistic information, thus resulting in tighter bounds and increased pruning capabilities, especially when used in association with *path context information* and further reduction techniques as outlined below.

To enable efficient and scalable online processing, our approach combines an offline phase (Section V-A) and an online phase (Section V-B), summarized in Figure 2. We refer the reader to the extended version of the paper for detailed examples of the various steps [20].

A. Offline Phase

The offline phase precomputes entity-level probability information (specifically, *component probabilities*) and builds a novel disk-based *context-aware path index* on the probabilistic entity graph, indexing not only all the paths in the PEG up to a given length, but also other context information that captures different properties of the paths' local neighborhoods.

Component Probabilities: To reduce calls to the PGM engine during online inference, we precompute and store components of match probabilities (Equation 11). As $Pr_{le}(.)$ is decomposable, we only precompute its parts, i.e., node label and edge existence probabilities based on Equations 2 and 3, respectively. Since $Pr_n(.)$ is not decomposable, we precompute node existence marginals for all possible valid configurations of every connected component, i.e., those consisting of entities not sharing a reference. In general, the connected components are expected to be small enough in practice for this to be feasible. If not, we could instead either employ an approximate inference technique to compute the marginals, or compute them on demand using the PGM engine.

Path Index: The path index contains all paths in the probabilistic entity graph of length at most L and probability at least β that do not contain two nodes sharing an underlying reference[1]. Each entry has two components:

[1] Paths with smaller probability are computed on demand.

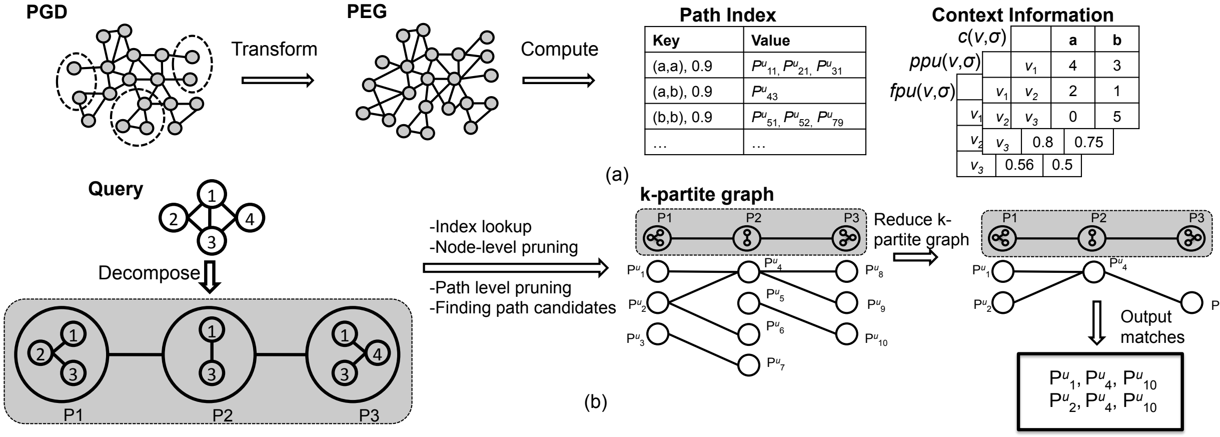
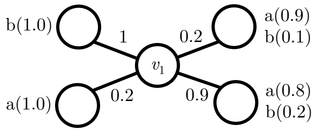


Fig. 2. (a) Offline and (b) online phase schematic diagrams. Path index keys consist of a node label sequence and a probability threshold, and the value is the set of paths corresponding to the key. There are three types of context information: $c(v,\sigma), ppu(v,\sigma), fpu(v,\sigma)$, where $v \in \{v_1, v_2, v_3, \ldots\}, \sigma \in \{a, b\}$.

- **Key:** the entry's key is a pair $\langle \mathbf{X}, \pi \rangle$, where $\mathbf{X} \in \Sigma^{l+1}$ is a sequence of node labels of length $l+1$, and $\pi \in \{\beta, \beta+\gamma, \beta+2\gamma, \ldots, 1\}$ is a probability value. The parameter γ defines the resolution of the index and provides a tradeoff between accuracy and response times.
- **Value:** the entry's value is the set of paths $\mathcal{P}^u$ of length l with probability under the node label assignment $\mathbf{X}$ between π and $\pi+\gamma$; the paths must also satisfy the reference constraint. For every $P^u \in \mathcal{P}^u$, we store the path itself as well as its two probability components $Pr_{le}(P^u)$ and $Pr_n(P^u)$.

Example: Say the graph contains two paths P^u_1, P^u_2 of length 1 such that $Pr_{le}(P^u_1) = 0.85, Pr_{le}(P^u_2) = 0.95, Pr_n(P^u_1) = Pr_n(P^u_2) = 1$ under the assignment (a,a) for their node labels; then we have $Pr(P^u_1) = 0.85, Pr(P^u_2) = 0.95$. If $\beta = 0.7$, and $\gamma = 0.1$, then the value of the key $\langle(a,a), 0.9\rangle$ will contain P^u_2 (along with its corresponding $Pr_{le}(.), Pr_n(.)$ probabilities), and the value of $\langle(a,a), 0.8\rangle$ will contain P^u_1, while the value of $\langle(a,a), 0.7\rangle$ will contain neither of them.

To increase efficiency, we build a *two-level index*, where the first level, accessing $\mathbf{X}$ via equality predicates, is a hash index, and the second level, accessing π via range predicates, is a B+-tree index. Paths of increasing length are built incrementally, exploiting the fact that all paths with probability at least β must consist of sub-paths with probability at least β as well. We use multiple threads and a synchronization barrier to build entries for different label sequences *in parallel*. To increase I/O performance, we accumulate a group of records in a *memory buffer* before writing the buffer to disk. Finally, for undirected graphs, entries for labels $\mathbf{X} = \{X_1, X_2, \ldots, X_{l-1}, X_l\}$ are identical to those for **X'** $= \{X_l, X_{l-1}, \ldots, X_2, X_1\}$ because of *symmetry*, and we therefore only store one direction for each such case and derive the other one as needed.



σ	a	b
$c(v_1,\sigma)$	3	3
$ppu(v_1,\sigma)$	0.9	1.0
$fpu(v_1,\sigma)$	0.72	1.0

Fig. 3. Context information example

Context Information: To enable further pruning (Section V-B2), we precompute context information for nodes. For a node $v \in G_U$ and a label $\sigma \in \Sigma$, let $N(v,\sigma)$ be the set of neighbors of v that have σ in their set of possible labels, i.e., $N(v,\sigma) = \{v' | v' \in \Gamma(v), \sigma \in L(v), refs(v) \cap refs(v') = \emptyset\}$, where $\Gamma(v)$ is the set of neighbors of node v, and $refs(v)$ is the set of underlying references of node v. For each node $v \in V_U$ and label $\sigma \in \Sigma$, we compute the following values, capturing different aspects of node/path neighborhoods:

- **Cardinality:** $c(v,\sigma) = |N(v,\sigma)|$, i.e., the size of $N(v,\sigma)$.
- **Partial Probability Upperbound:** $ppu(v,\sigma)$, which is an upperbound for the probabilities in the neighborhood of v considering only the edges between v and $N(v,\sigma)$.
$$ppu(v,\sigma) = max_{v' \in N(v,\sigma)} Pr((v,v').e = T)$$
- **Full Probability Upperbound:** $fpu(v,\sigma)$, which is an upperbound for the probabilities in the neighborhood of v also taking into account the neighbors' labels.
$$fpu(v,\sigma) = max_{v' \in N(v,\sigma)} Pr(v'.l = \sigma) \cdot Pr((v,v').e = T)$$

Example: In Figure 3, $c(v_1,a) = c(v_1,b) = 3$. $ppu(v_1,a) = 0.9$ because the highest edge probability that connects v_1 to a node with label a is 0.9. Similarly, $ppu(v_1,b) = 1.0$. Finally, $fpu(v_1,a) = 0.72$, as this is the highest product of an edge connecting v_1 to another node (with 0.9), and that node having label a (with 0.8). Similarly, $fpu(v_1,b) = 1.0$.

B. Online Phase

Our online query processing technique consists of five main steps (shown in Figure 2(b)): (1) decompose the query into a set of paths, (2) obtain a set of candidates for every path in the decomposition using the path index, (3) obtain *join-candidate paths* for every candidate path (i.e., candidate paths whose query paths share a node with the given candidate and can thus extend it to form a partial match), (4) jointly reduce the candidate search space by *reduction by join-candidates* which performs message passing in a k-partite graph, (5) find matches to the full query.

1) Path Decomposition: The task of *path decomposition* is to split the query into a set of possibly overlapping paths, each of length L or less, that cover the entire query, and whose matches can be obtained from the path index. To preserve the structural information of the query, intersection points between the paths are expressed as join predicates, which have to be satisfied when combining path matches into a full query match. For example, Figure 2(b) shows a query and its decomposition into three paths P_1, P_2, and P_3. In order to preserve the structural information of the query, any paths (P_1^u, P_2^u, P_3^u) that match (P_1, P_2, P_3) must satisfy the predicates $P_1^u.1 = P_2^u.1$, $P_1^u.3 = P_2^u.3$, $P_3^u.1 = P_2^u.1$, and $P_3^u.3 = P_2^u.3$ (we use $P_1^u.1$ to denote the vertex in path P_1^u that matches the vertex 1 in path P_1). Query path decomposition thus decomposes a query Q into a set of node/edge overlapping paths $\mathcal{P}$. For every pair of overlapping paths P_1 and P_2, the decomposition defines a set of *join predicates* $JP(P_1, P_2)$. Further, we denote the set of paths joining with a path P by $J(P)$.

As finding a least-cost path decomposition based on the number of operations involved in producing the final result is too costly, we instead use an estimate of the initial query search space size SS_0. We would thus like to find $argmin_{\mathcal{P} \subseteq \mathbb{P}(Q), \mathcal{P} \text{ covers } Q} SS_0(\mathcal{P})$, where $\mathbb{P}(Q)$ is the set of all possible paths of length at most L in Q. For each path P in the decomposition, we estimate the number of matches, or its cardinality $C(P, \alpha)$. The cardinality is based on the number of database paths matching the query path P with probability at least α, but also takes into account the fact that those matches have to be extended to neighboring query paths. We hence express $C(P, \alpha)$ in terms of the following quantities.

1) **Number of candidates** $|PIndex(l_Q(V_P), \alpha)|$ matching P's labels $l_Q(V_P)$ with probability at least α in the path index.
2) **Path degree** $degree(P)$: sum of path node degrees, not counting edges on the path, that is,
$$degree(P) = \textstyle\sum_{n \in V_P} degree(n) - 2 \times length(P)$$
3) **Path density** $density(P)$: this measures how close the nodes on P are to forming a clique. Let K be the number of edges between the nodes of P, and M the number of nodes on the path, then $density(P) = (2K)/(M(M-1))$.

Taking into account the direction of influence of these components on the true number of matches, we approximate $|P|$ as:

$$C(P, \alpha) \propto \frac{|PIndex(l_Q(V_P), \alpha)|}{degree(P) \cdot density(P)}$$

We then estimate the search space size as the product of all such path cardinalities. Therefore, our goal is to find:

$$argmin_{\substack{\mathcal{P} \subseteq \mathbb{P}(Q), \\ \mathcal{P} \text{ covers } Q}} \prod_{P \in \mathcal{P}} \frac{|PIndex(l_Q(V_P), \alpha)|}{degree(P) \cdot density(P)}$$

Since it is not practical to query the index for an arbitrary α and $l_Q(V_P)$ at query time, we build a histogram for every possible label sequence **X** during the offline phase at selected probability points $(\alpha_0, \ldots, 1)$. At runtime, we use exponential curve fitting to estimate $|PIndex(l_Q(V_P), \alpha)|$ given $hist(l_Q(V_P), \alpha_i)$ and $hist(l_Q(V_P), \alpha_{i+1})$ where $\alpha_i < \alpha < \alpha_{i+1}$.

We reduce the problem of optimizing the cost function to that of SET COVER, where the set of query edges corresponds to the universal set (in the SET COVER instance), and each path P in the query with length at most L is a candidate set. Note that we allow paths with shared edges, as this can reduce the cost of several paths at once (e.g., in the case of a very selective edge connected to multiple non-selective paths). The cost of the cover is the product of the individual costs of the participating paths. Since SET COVER is NP-complete, we use the standard greedy approximation to solve the problem, using the length of a path divided by its cost as score.

2) Finding Path Candidates: Given a path decomposition $\mathcal{P}$, we find candidate matches for every query path. For every path $P \in \mathcal{P}$, we retrieve its matches $PIndex(l_Q(V_P), \alpha)$ from the path index, but only keep those paths that satisfy certain context criteria. We denote the resulting set of matches by $cn(P)$ $(\subseteq PIndex(l_Q(V_P), \alpha))$. This second step relies on the following query statistics:

- **Node-level statistics:** For every node $n \in V_Q$, we calculate its neighborhood label count for every label $\sigma \in \Sigma$,
$$c(n, \sigma) = |\{m | m \in \Gamma(n), l_Q(m) = \sigma\}|$$
- **Path-level statistics:** A path match can only contribute to a full match if it can be extended to at least the neighboring nodes in both the query and the graph, and we can safely prune other path matches. For every path $P \in \mathcal{P}$, we therefore collect the following information:
  1) Path neighbors $\Gamma(P)$: the set of nodes that are not on P but are neighbors of at least one node on P.
  2) Reverse path neighbors: for every $m \in \Gamma(P)$, $rv(P, m)$ is the set of nodes on P that are neighbors of m.
  3) Path cycles: for every $n \in V_P$, path cycles, $cyc(P, n)$, is the set of nodes on P that are also connected to n by a query edge outside the path, and thus appear together with n in a cycle. To avoid information duplication, each such edge only contributes to the path cycles of one of its endpoints.

Node-level pruning: Using the node-level statistics, we obtain the set of candidates $cn(n)$ for every node $n \in V_Q$ as follows:

1) For every label $\sigma \in \Sigma$, v must have a number of neighbors that is greater than or equal to the number of neighbors of n with label σ, i.e., $c(v, \sigma) \geq c(n, \sigma), \forall \sigma \in \Sigma$.
2) For every label $\sigma \in \Sigma$, the probability of v having the correct label and at least the number of neighbors labeled σ required by the query has to exceed the query threshold α. Using precomputed full probability upperbounds as approximation and taking into account multiple occurrences of the same label, we thus only keep candidates v for n satisfying $Pr(v.l = l_Q(n)) \times fpu(v, \sigma)^{c(v,\sigma)} \geq \alpha, \forall \sigma \in \Sigma$.

Path-level pruning: Next, we prune the set of candidate paths using path-level statistics. For each path $P^u \in PIndex(l_Q(V_P), \alpha)$, we perform the following tests:

1) For every node $v \in V_{P^u}$, v must be a candidate for the corresponding node n in P, i.e., $v \in cn(n)$.
2) The probability of a path together with its neighboring nodes and cycles must be greater than or equal to α, which we test using $(Pr_{le}(P^u) \times Pr_n(P^u)) \times pu(P^u) \times cpr(P^u) \geq \alpha$, with $pu(P^u)$ and $cpr(P^u)$ defined as follows.

The *path-neighborhood probability upperbound* $pu(P^u)$ of a candidate path P^u matching a query path P is an upperbound for the probability of all nodes matching $\Gamma(P)$ and their edges. Let $m \in \Gamma(P)$ be a path P neighbor, and $n \in rv(P, m)$ a node on P. We compute a probability upperbound $pu(n, m, P^u)$ on the neighborhood of m as:

$$fpu(\psi(n), l_Q(m)) = \prod_{n' \in rv(P,m), n' \neq n} ppu(\psi(n'), l_Q(m))$$

where we use the full probability upperbound fpu for the edge between the match of m and the selected neighbor n, and partial probability upperbounds for all other neighbors of m's match, thus ensuring that information on m is only considered once. Choosing the tightest upperbound over all reverse path neighbors $rv(P, m)$ and aggregating over all $m \in \Gamma(P)$, we get the overall path P^u neighborhood probability upperbound:

$$pu(P^u) = \prod_{m \in \Gamma(P)} min_{n \in rv(P,m)} pu(n, m, P^u)$$

The *path-cycles probability* $cpr(P^u)$ is the probability of edges not on the path P^u but connecting path nodes:

$$cpr(P^u) = \prod_{\substack{n \in V_P, \\ m \in cyc(P,n)}} Pr((\psi(n), \psi(m)).e = T)$$

For every path P in the decomposition, the final list of candidates $cn(P)$ contains exactly those paths from the initial set $PIndex(l_Q(V_P), \alpha)$ that pass the above tests.

3) Finding Join-Candidates: In this step, for every candidate path $P^u \in cn(P)$ of every query path P, we find a set of paths that are candidates to be joined with P^u. Recall that every query path $P_1 \in \mathcal{P}$ can be joined with a set of paths $J(P_1) \subseteq \mathcal{P}$, and there is a set of join predicates $JP(P_1, P_2)$ between P_1 and every path $P_2 \in J(P_1)$. For a query path $P_1 \in \mathcal{P}$, and a candidate path $P_1^u \in cn(P_1)$, we define its join-candidate paths of type $P_2 \in J(P_1)$ as:

$$\begin{aligned} & cn(P_1, P_1^u, P_2) \\ & \quad = \{P_2^u | P_2^u \in cn(P_2) \wedge jp(P_1^u, P_2^u) = T, \forall jp \in JP(P_1, P_2) \\ & \quad \wedge Pr(P_1^u \circ P_2^u) \geq \alpha \ \wedge refs(V_{P_1^u}) \cap refs(V_{P_2^u}) = \emptyset\} \end{aligned}$$

where $jp(P_1^u, P_2^u)$ is the instantiation of the predicate $jp \in JP(P_1, P_2)$ using paths P_1^u and P_2^u, and $P_1^u \circ P_2^u$ is the subgraph consisting of the two joined paths. Intuitively, $cn(P_1, P_1^u, P_2)$ refers to the set of paths in $cn(P_2)$ that are candidates to be joined with $P_1^u \in cn(P_1)$.

To facilitate finding join-candidate paths, for each $P \in \mathcal{P}$, while finding $cn(P)$, we build a lookup table $T(P, P_i)$ for each query path $P_i \in J(P)$. For every table $T(P, P_i)$, the set of positions $\langle p_{i1}, \ldots, p_{ik} \rangle$ indicates the nodes in P_i that participate in join predicates. The key for table $T(P, P_i)$ is a set of nodes $\langle n_1, \ldots, n_k \rangle$, and the values are paths in $cn(P)$ that have nodes $\langle n_1, \ldots, n_k \rangle$ at positions $\langle p_{i1}, \ldots, p_{ik} \rangle$. Given a path $P_i^u \in cn(P_i)$, paths in P which are joinable with P_i^u can now be obtained using a direct lookup operation from table $T(P, P_i)$, where the access key is obtained from P_i^u.

4) Joint Search Space Reduction: We next exploit the mutual relationship between the candidates and their join-candidates to reduce the size of all candidate lists based on the following two observations. First, for a candidate match of a path P to contribute to a full query match, we must be able to combine it with at least one candidate for all query paths joining P. Second, if we can obtain an upperbound on the probability of all full query matches a candidate path can appear in, we can prune candidate paths based on the query threshold α. We refer to these two principles as *reduction by structure* and *reduction by upperbounds*, respectively, and discuss their details below. As they influence each other, the overall algorithm for joint search space reduction iterates between them until no further changes occur.

The reduction is based on a *k-partite graph*, where each partition corresponds to a query path, each vertex to a candidate path match, and each link to a join between two candidate paths.[2] Pruning a candidate thus corresponds to deleting a vertex and its outgoing links from the k-partite graph.

Definition 6: **Candidate k-partite Graph:** A candidate k-partite graph is a k-partite graph that has a partition for each $P \in \mathcal{P}$, where the set of vertices of each partition P are $cn(P)$. There is a link between P_1^u in partition P_1 and P_2^u in partition P_2 iff $P_2^u \in cn(P_1, P_1^u, P_2)$.

Every match of the query in the PEG corresponds to a subgraph of the candidate k-partite graph with one vertex per partition (i.e., one match for each query path) and all join links between them. We can thus safely prune all vertices that have no links to a partition they should link to, as well as those that cannot participate in any match with probability above the query threshold.

Reduction by structure: If a vertex has no links to at least one partition its query path joins with, we remove the vertex and all of its links to vertices in all partitions. This is repeated until no further changes occur.

Reduction by upperbounds: In order to exploit probabilistic information during search space reduction, we now introduce two types of vertex weights, based on $Pr_{le}(.)$ and $Pr_n(.)$, respectively, and discuss a message passing scheme that exploits these weights to obtain bounds for reduction by upperbounds.

The first type of weights is assigned such that when a subgraph's weights are multiplied, we obtain the final $Pr_{le}(.)$ probability of the corresponding match. To avoid double contributions in cases of overlap between paths, we assign the overlapping elements' probability to exactly one partition, i.e., for every $v \in V_Q, e \in E_Q$, we choose exactly one partition to cover v's or e's probability. Let partition P (we use P to refer to both the path and its corresponding partition) exclusively cover nodes and edges $cv(P)$ and $ce(P)$, respectively, then a vertex's first weight is

$$w_1(P^u) = \prod_{n \in cv(P)} Pr(\psi(n).l = l_Q(n)) \prod_{e \in ce(P)} Pr(\psi(e).e = T)$$

where $\psi(n)$ is the PEG node matching the query node n. As identity probabilities $Pr_n(P^u)$ are not decomposable, we directly use the identity probability of a path as the second

[2] To avoid confusion, we use the terms (vertex/link) when referring to the k-partite graph, and (node/edge) when referring to the PEG.

weight of its corresponding vertex in the k-partite graph (however, we cannot multiply weights of this type together as it is the case with w_1 weights):

$$w_2(P^u) = Pr_n(P^u)$$

In addition to the two weights, each vertex P^u has an associated *perception vector* of length k, that is, with one entry per partition. Each entry is an upperbound on the w_1 weights of all vertices in that partition that can appear in a full match with P^u. Initially, we have $w_1(P^u)$ for the entry corresponding to P^u's own partition, and 1 for all other partitions. During message passing, each vertex first sends its current vector to each of its neighboring vertices (excluding the entry for the receiving neighbor's partition). Once all messages are received, each vertex P_1^u updates its own vector as follows. For each vector entry corresponding to a partition P and each partition P_2 containing neighboring nodes of P_1^u, we choose the maximum value for P sent by the neighbors in P_2. We then take the minimum of these over all such P_2 as the new value in the vector, and iterate the overall process. The upperbound used to prune a vertex (and thus a candidate path) based on the query threshold α then is the product of all entries in the vertex' vector and its weight w_2.

As discussed above, the final algorithm iterates between both types of reduction until no further changes take place. We further improve efficiency through two optimizations: (1) **incremental maintenance**, wherein we only recompute upperbounds for vertices for which a neighbor has been deleted or has reduced its perception, and only consider vertices connected to a newly deleted link for deletion, and (2) **parallelization** of the reduction algorithm, with one thread per partition, where we introduce appropriate locking protocols to avoid incorrect modifications of the k-partite graph by multiple threads at the same time. We note here that we also exploit parallelism in other parts of the system such as constructing node candidates, path candidates, and join-candidate sets.

5) Finding Full Query Matches: The final step finds the full query matches, starting from the matches of one path and progressively adding matches of joining paths, based on an initially determined join order.

Join order determination: To avoid solving an additional optimization problem, we add paths to the join order one at a time, based on the following heuristic:

1) Choose the path with the largest number of nodes overlapping with the paths that already exist in the order.
2) In case of ties, choose the path with the largest number of join predicates with the existing paths.
3) In case of ties, choose the path with smallest cardinality (estimated as in path decomposition).

Finding matches: Given the join order $\{P_1, \dots, P_{|\mathcal{P}|}\}$, we use the reduced candidate k-partite graph to construct matches incrementally. The initial matches are the vertices in the partition corresponding to P_1. Each match M_i up to path P_i is extended to matches up to P_{i+1} as follows. We first identify all paths P_j with $j \leq i$ that join with P_{i+1}. For each vertex in P_{i+1}'s partition that has a link to the corresponding vertex in M_i for each such P_j, we extend M_i by adding that vertex's candidate match. We discard M_i if there is no such vertex, and only produce those extended matches that have probability at least α and do not contain two nodes sharing a reference.

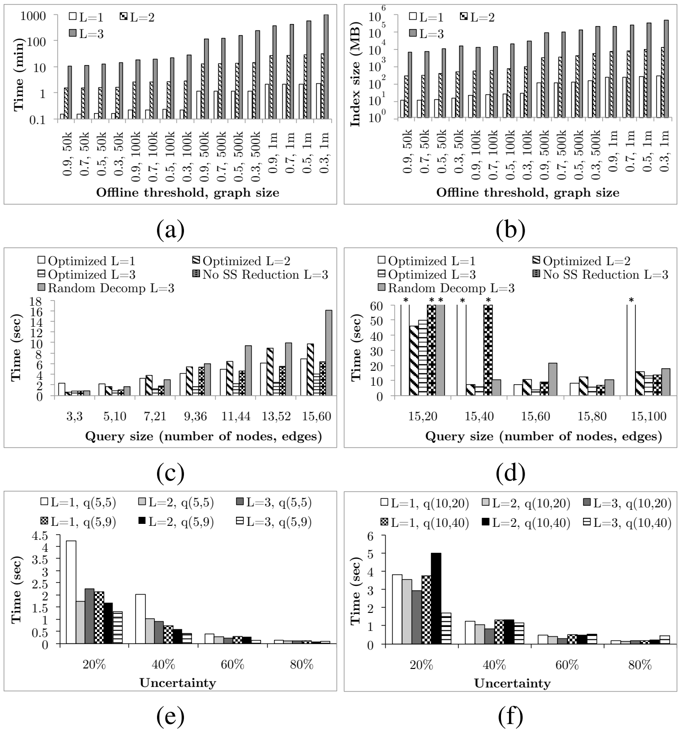
C. Handling Correlations

To handle edge existence correlations discussed in Section III, we replace the independent edge existence probabilities $Pr((s_1, s_2).e)$ in all the equations with their corresponding conditional probabilities $Pr((s_1, s_2).\boldsymbol{e}|s_1.\boldsymbol{l_1}, s_2.\boldsymbol{l_2})$. Since the end point node labels are required to match the labels of the corresponding nodes in the query, this conditional probability can be computed directly from the CPT in most of the equations. The only exceptions are the equations for $ppu(v, \sigma)$ and $fpu(v, \sigma)$ (Section V-A), where the existence probability of an edge is needed but one of the end point node labels is not known. Since those two functions are upper bounds, we simply modify the equations to find the maximum value over all possible labels of v. Although this reduces the pruning ability of the context information, we found it to have a negligible impact in our experimental evaluation.

VI. Experimental Evaluation

We have implemented our approach in Java, using the disk-based graph database engine Neo4j to store the probabilistic graph, and the key/value store KyotoCabinet to store the index as a B+-tree. We assess the index construction algorithm's performance in terms of both time and space, online query performance compared to various baselines, and the effect of our search space reduction methods.

Datasets: We use two real-world datasets from DBLP and IMDB, further details on which are provided in Section VI-C, as well as synthetic graphs whose structure is generated according to the *preferential attachment model* [2]. In the synthetic graphs, node label probabilities are based on a set of random probabilities $p_1, \dots, p_{|\Sigma|}$, which we weight by a zipf distribution, i.e., $p'_i = p_i/i$. We normalize those to obtain final probabilities $p''_i = p'_i/(\sum_j p'_j)$, which are assigned to node labels randomly. Edge probabilities are generated analogously. To generate reference sets corresponding to entities, we randomly choose k subsets of s nodes each from the graph, and randomly assign r pairs of nodes per group to the same reference set. That is, reference sets are of size 2, and the maximum size of a connected component is s. We assign random probabilities to reference sets. The merge functions average the underlying distributions for both node attributes and edge existence. We consider four settings with 50k, 100k, 500k, and 1m references, and a number of edges equal to five times the number of references in every setting. We set $k = \text{No. of references}/1000, s = r = 4$. We associate probability distributions with 20% of the references, relations, and reference sets unless otherwise stated. These settings result in probabilistic entity graphs of sizes (54k/292k), (108k/583k), (540k/ 2.95m) and (1.08m/5.88m) nodes/edges, respectively. Experiments are performed on an 8-core Linux Amazon EC2 instance, with 117GB of RAM and 1.8TB of instance storage.



(a) (b)

(c) (d)

(e) (f)

Fig. 4. (a-b) Offline phase performance; Online query times for (c) varying query sizes, (d) varying query densities, (e-f) varying degrees of uncertainty. A * indicates that the query did not finish in allotted time (15 minutes), or the process ran out of memory.

A. Offline Phase Performance

We first compare performance of the offline phase for maximum index path lengths $L = 1, 2, 3$. As the index size for the 500k network with $L = 4$ was larger than the instance space available, we do not report results for that path length.

Running Time: Figure 4(a) shows the offline phase running time (which includes calculating PEG component probabilities, building the index, and calculating context information) when varying both the graph size and the index lowerbound probability threshold β. The running time increases by a factor of 10 to 14 when going from $L = 1$ to $L = 2$, and 7 to 30 when going from $L = 2$ to $L = 3$. The running time increase is sub-linear in the size of the graph in most cases, which is due to higher memory buffer utilization for larger graphs.

Path Index Size: Figure 4(b) shows that index sizes at $L = 2$ are 32 times larger than those at $L = 1$ on average, and index sizes at $L = 3$ are 28 times larger than those at $L = 2$ on average. Index size increases at the same rate as the graph size at $L = 1$, and faster than the increase in the graph size at $L = 2$, e.g., the index size at 1m is 20 times larger than that of 50k on average at $L = 1$ and 25 times on average at $L = 2$. This is because indexes at $L = 1$ increase linearly with graph size, while at $L = 2$ the index size increases quadratically. The same trend applies at $L = 3$ as its size increases cubically.

B. Online Phase Performance

We compare the running time of our proposed algorithm with all optimizations (**Opt(L)**, with path lengths $L = 1, 2, 3$) against the following baselines.

1) **Random decomposition (RD):** Our proposed approach, but using random query decomposition instead of SET COVER, and a path join order based on the number of path index matches only (with path length $L = 3$).
2) **No search space reduction (NoSSR):** Our optimized method, but without the joint search space reduction on the k-partite graph, thus generating final results after constructing the candidate and relative candidate lists (with $L = 3$).
3) **SQL:** An SQL implementation of our queries run on top of MySQL database. This approach did not finish in a month on the 100k nodes dataset for a query with 5 nodes and 7 edges and a threshold of 0.7, which is answered in less than a second by our approach, and is therefore not used further.

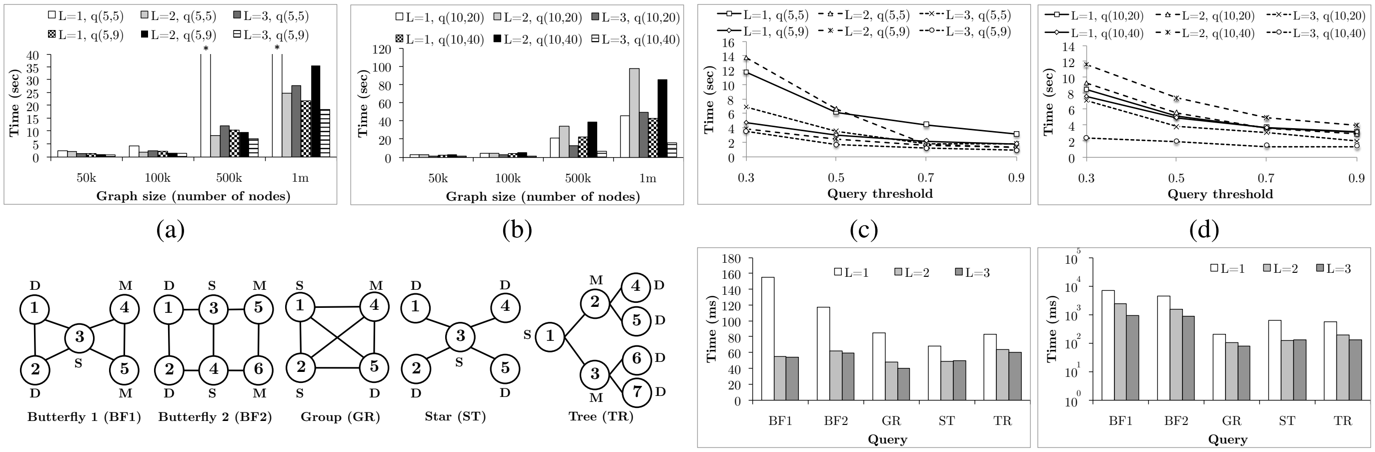
Unless stated otherwise, we use the 100k dataset and a query threshold of 0.7.

Varying input query size: Figure 4(c) shows running times for 7 different query sizes between q(3,3) and q(15,60), where $q(n, m)$ denotes a query with n nodes and m edges, averaged over five randomly generated queries per size. m is set to be $4n$, unless $n(n - 1)/2$ (the maximum number of edges permitted) is less than that, in which case, m is set to be $n(n - 1)/2$. Opt(3) always performs best. Opt(1) slightly outperforms Opt(2) for smaller queries, because the advantages of richer context information with $L = 2$ do not outweigh the overheads of processing the larger number of matches returned by the path index. However, in almost all other experiments we performed, Opt(2) outperforms Opt(1), by orders of magnitude in many cases. Opt(2) thus provides a compromise that does not take as much time and space as Opt(3) in building its index, and still has an acceptable performance, even in cases where Opt(1) may not succeed.

Varying input query density: Figure 4(d) shows running times for 5 different query densities, for queries with 15 nodes and between 20 and 100 edges. Each result is the average over five randomly generated queries of the corresponding size. Again, Opt(3) always outperforms Opt(1), Opt(2) (except for $q(15, 20)$), RD and NoSSR. Opt(1) runs out of memory for query $q(15, 20)$ due to the large number of matches returned by the index because of the query sparsity. Furthermore, for several queries sizes, at least one run of Opt(1), RD and NoSSR did not finish within the time limit of 15 minutes.

Varying input graph degree of uncertainty: We now focus on our approach, using query sizes q(5,5) and q(5,9) (in Figure 4(e)), and q(10,20) and q(10,40) (in Figure 4(f)), and varying the number of uncertain nodes and edges from 20% to 100%. Opt(3) always outperforms Opt(1) and Opt(2), while Opt(2) outperforms Opt(1) for all degrees of uncertainty larger than 20%.

Varying input graph size: We next vary the number of edges in the input graph between 300 thousand and 6 million, using query sizes q(5,5) and q(5,9) (in Figure 5(a)), and q(10,20) and q(10,40) (in Figure 5(b)). With query q(5,5), Opt(1) runs out of memory at both 500k and 1m due to the high number of matches. Opt(3) again performs best in most cases, and only slightly worse in the others.



(a)

(b)

(c)

(d)

(e) Pattern queries for real-world datasets

(f) DBLP Dataset

(g) IMDB Dataset

Fig. 5. (a-b) Online query times with varying input graph sizes, and (c-d) varying input query thresholds, for queries with 5 and 10 nodes. (e) Patterns used in real-world experiments; (f-g) Performance on the DBLP and IMDB real-world datasets, respectively.

Varying input query threshold: Varying the query threshold between 0.3 and 0.9 with queries of size q(5,5), q(5,9) (in Figure 5(c)) and q(10,20), q(10,40) (in Figure 5(d)), we see that performance generally improves with increasing threshold, where higher path lengths are less sensitive to the changes.

Finally, we performed experiments to study performance of our approach with respect to reducing the search space size. Experimental evaluation showed that our algorithms reduced the search space size by *more than ten orders of magnitude*. We refer the reader to [20] for a detailed discussion.

C. Performance on Real-world Data

We now provide experimental results on two real-world datasets, DBLP and IMDB. We use correlated edge and label probabilities with DBLP, and independent edge probabilities with IMDB. For the DBLP network, we extract the "author collaboration" graph, that is, nodes represent authors, and edges represent collaboration relationships. We annotate the collaboration graph with probabilistic data to capture different types of uncertainties. For every author, we assign a probability distribution over the areas that she/he is interested in, which can be Databases, Machine Learning, or Software Engineering. This information is based on the author's relative contribution to conferences of each area. To obtain the edge existence probability for a pair of authors, we first generate a base probability between 0.5 and 1 depending on the number of collaborations between them. If the authors' research interests as given by the node labels are the same, the conditional edge existence probability is the base probability p, else, it is $0.8 \cdot p$. We create a reference set for every pair of authors whose names have normalized string similarity score above 0.9 (to capture identity uncertainty). The resulting graph has 16.8k nodes and 40.3k edges. We use the collaboration patterns shown in Figure 5(e) with a query threshold of 0.1. Running times of the online phase are shown in Figure 5(f). Opt(3) outperforms Opt(2), which in turn outperforms Opt(1), for all queries except the star, whose maximum path length is 2. The IMDB network is a "co-starring" graph, that is, nodes are actors, and edges are co-starring relationships between actors. We use Drama, Comedy, Family and Action movies from the IMDB dataset, and create a co-starring edge between the two main stars of each movie. Standard statistical prediction methods are used to introduce probabilities to the network, where node attribute uncertainties are obtained from the distribution over movie genres an actor participates in, edge probabilities are obtained from the number of times two actors co-star together, and identity uncertainty is obtained from similarities in actor names, which may have occurred from duplicates or misspellings. The network has 91k nodes and 936k edges. We use the query structure depicted in Figure 5(e), with the same randomly generated label for all nodes in a query. The input probability threshold α is 0.1. Again, Figure 5(g) shows that Opt(3) outperforms Opt(2), which in turn outperforms Opt(1).

VII. Related Work

Although many research studies have addressed the problems of representing and querying uncertain and probabilistic data [27], the area of uncertain graph data processing is still new and gaining more interest recently. Research in uncertain graph databases has covered different topics such as finding shortest paths, reliable subgraphs, mining frequent patterns, and answering graph queries, e.g., [23], [17], [16], [12], [36], [22], [37], [5], [31], [30].

Udrea et al. define semantics for probabilistic RDF graphs formed by associating probabilities to triplets, calling them quadruples [28]. They propose algorithms for answering queries consisting of one quadruple with one variable at most. Huang el al. propose algorithms for query processing over probabilistic RDF graphs [14]. Lian et al. propose efficient algorithms for querying probabilistic RDF graphs with node attribute correlations [19]. Chen et al. propose algorithms for continuously searching for subgraph patterns over multiple streaming uncertain graphs [5]. Yuan et al. propose algorithms for retrieving graphs containing a query graph from an uncertain graph database [31]. In their more recent work, Yuan et al. consider the problem of graph similarity over uncertain graphs [30]. However, [14], [5], [31], [30] only consider edge uncertainty. Further none of these supports identity uncertainty.

Ioannou et al. propose query evaluation algorithms for uncertain data with identity uncertainty [15], but their methods are not designed for graph data. Furthermore, our semantics are more general, as we allow user-provided merge functions. Hua et al. propose a method for evaluating aggregate queries over data with identity uncertainty [13], but their methods are not designed for graph data either, and their model constrains the acceptable configurations of groups of references representing entities. Our PGM-based representation allows for arbitrary configurations. Dedupalog [1] is a system for declaratively resolving duplicate references using hard and soft constraints. GrDB [21] is a system for declarative cleaning of noisy graph data, including missing attributes and links, and resolving duplicate references. Neither of these considers the problem of querying uncertain graph data.

Subgraph pattern matching has received renewed interest in recent years, leading to new exact or approximate methods that search for patterns in graph databases consisting either of several relatively small graphs or a single large graph, e.g., [24], [29], [34], [6], [11], [32], [33], [9], [8], [35]. For path indexing, Zhao et al. use shortest path-based subgraph pattern matching [33], but they consider only certain graphs. Further, while we use context-aware path indexing, they utilize shortest paths calculated at query runtime to prune candidates. Although their use of shortest paths for subgraph pattern matching implies decomposing the query graph into paths as we do, they use different criteria for path decomposition and join order selection better suited for certain graphs. Our approach utilizes probabilistic information for pruning, and implements reduction by join-candidates to further reduce the search space. GraphGrep [24] uses path indexing for querying a database of multiple graphs. It does not handle probabilistic graphs, and it is designed to deal with small graph sizes in the order of tens to hundreds of nodes. For indexing, it indexes paths only without local information. Our approach can be used to query very large probabilistic graphs in the order of millions of nodes and edges.

VIII. Conclusions and Future Work

We presented a probabilistic approach for modeling uncertain entity graphs and answering queries over them. Our probabilistic entity graph captures node attribute uncertainty, edge existence uncertainty, and identity uncertainty. We presented efficient algorithms to solve subgraph pattern matching queries over such uncertain graphs, where queries are expressed and evaluated at the entity-level. Our approaches outperform an equivalent SQL implementation by orders of magnitude.

Acknowledgments Angelika Kimmig is supported by the Research Foundation Flanders (FWO Vlaanderen). This work is supported by NSF grants 0916736, 1319432, and 0746930.

References

[1] A. Arasu, C. Re, and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, 2009.

[2] A. L. Barabasi and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[3] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *The VLDB Journal*, 18(1):255–276, 2009.

[4] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.

[5] L. Chen and C. Wang. Continuous subgraph pattern search over certain and uncertain graph streams. *TKDE*, 22(8):1093–1109, 2010.

[6] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-index: towards verification-free query processing on graph databases. In *SIGMOD*, 2007.

[7] O. Etzioni, M. Banko, S. Soderland, and D. S. Weld. Open Information Extraction from the Web. *Communications of the ACM*, 51(12), 2008.

[8] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, 2011.

[9] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: from intractable to polynomial time. *PVLDB*, 3(1), 2010.

[10] L. Getoor and A. Machanavajjhala. Entity resolution: Theory, practice & open challenges. *PVLDB*, 5(12):2018–2019, 2012.

[11] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, 2008.

[12] P. Hintsanen and H. Toivonen. Finding reliable subgraphs from large probabilistic graphs. *DMKD*, 17(1):3–23, 2008.

[13] M. Hua and J. Pei. Aggregate queries on probabilistic record linkages. In *EDBT*, 2012.

[14] H. Huang and C. Liu. Query evaluation on probabilistic RDF databases. In *WISE*, 2009.

[15] E. Ioannou, W. Nejdl, C. Niederée, and Y. Velegrakis. On-the-fly entity-aware query processing in the presence of linkage. *PVLDB*, 3(1):429–438, 2010.

[16] R. Jin, L. Liu, and C. C. Aggarwal. Discovering highly reliable subgraphs in uncertain graphs. In *KDD*, 2011.

[17] R. Jin, L. Liu, B. Ding, and H. Wang. Distance-constraint reachability computation in uncertain graphs. *PVLDB*, 4(9):551–562, 2011.

[18] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[19] X. Lian and L. Chen. Efficient query answering in probabilistic RDF graphs. In *SIGMOD*, 2011.

[20] W. E. Moustafa, A. Kimmig, A. Deshpande, and L. Getoor. Subgraph pattern matching over uncertain graphs with identity linkage uncertainty. *CoRR*, abs/1305.7006, 2013.

[21] W. E. Moustafa, G. Namata, A. Deshpande, and L. Getoor. Declarative analysis of noisy information networks. In *ICDE Workshops*, 2011.

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

[23] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. k-nearest neighbors in uncertain graphs. *PVLDB*, 3(1):997–1008, 2010.

[24] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.

[25] A. Singhal. Introducing the Knowledge Graph: Things, Not Strings, 2012. Official Blog (of Google), see: http://goo.gl/zivFV.

[26] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[27] D. Suciu, D. Olteanu, R. Christopher, and C. Koch. *Probabilistic Databases*. Morgan & Claypool Publishers, 2011.

[28] O. Udrea, V. S. Subrahmanian, and Z. Majkic. Probabilistic RDF. In *IRI*, 2006.

[29] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.

[30] Y. Yuan, G. Wang, L. Chen, and H. Wang. Efficient subgraph similarity search on large probabilistic graph databases. *PVLDB*, 5(9), 2012.

[31] Y. Yuan, G. Wang, H. Wang, and L. Chen. Efficient subgraph search over large uncertain graphs. *PVLDB*, 4(11):876–886, 2011.

[32] S. Zhang, S. Li, and J. Yang. GADDI: distance index based subgraph matching in biological networks. In *EDBT*, 2009.

[33] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.

[34] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta <= graph. In *VLDB*, 2007.

[35] L. Zou, L. Chen, and M. T. Özsu. Distance-join: pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.

[36] Z. Zou, J. Li, H. Gao, and S. Zhang. Finding top-k maximal cliques in an uncertain graph. In *ICDE*, 2010.

[37] Z. Zou, J. Li, H. Gao, and S. Zhang. Mining frequent subgraph patterns from uncertain graph data. *TKDE*, 22(9):1203–1218, 2010.