VLDB Journal manuscript No.
(will be inserted by the editor)

A Unified Approach to Ranking in Probabilistic Databases

Jian Li · Barna Saha · Amol Deshpande

the date of receipt and acceptance should be inserted later

Abstract Ranking is a fundamental operation in data analysis and decision support, and plays an even more crucial role if the dataset being explored exhibits uncertainty. This has led to much work in understanding how to rank the tuples in a probabilistic dataset in recent years. In this article, we present a unified approach to ranking and top-k query processing in probabilistic databases by viewing it as a multi-criterion optimization problem, and by deriving a set of *features* that capture the key properties of a probabilistic dataset that dictate the ranked result. We contend that a single, specific ranking function may not suffice for probabilistic databases, and we instead propose two *parameterized ranking functions*, called PRF$^\omega$ and PRF^e, that generalize or can approximate many of the previously proposed ranking functions. We present novel *generating functions*-based algorithms for efficiently ranking large datasets according to these ranking functions, even if the datasets exhibit complex correlations modeled using *probabilistic and/xor trees* or *Markov networks*. We further propose that the parameters of the ranking function be *learned* from user preferences, and we develop an approach to learn those parameters. Finally, we present a comprehensive experimental study that illustrates the effectiveness of our parameterized ranking functions, especially PRF^e, at approximating other ranking functions and the scalability of our proposed algorithms for exact or approximate ranking.

1 Introduction

Recent years have seen a dramatic increase in the number of applications domains that naturally generate uncertain data and that demand support for executing complex decision support queries over them. These include information retrieval [20], data integration and cleaning [2, 17], text analytics [24, 30], social network analysis [1], sensor data management [11, 16], financial applications, biological and scientific data management, etc. Uncertainty arises in these environments for a variety of reasons. Sensor data typically contains noise and measurement errors and is often incomplete because of sensor faults or communication link failures. In social networks and scientific domains, the observed interaction or experimental data is often very noisy, and ubiquitous use of predictive models adds a further layer of uncertainty. Use of automated tools in data integration and information extraction introduce significant uncertainty in the output.

By their very nature, many of these applications require support for *ranking* or *top-k query processing* over large volumes of data. For instance, consider a *House Search* application where a user is searching for a house using a real estate sales dataset that lists the houses for sale. Such a dataset, which may be constructed by crawling and combining data from multiple sources, is inherently uncertain and noisy. In fact, the houses that the user prefers the most, are also the most likely to be sold by now. We may denote such uncertainty by associating with each advertisement a *probability* that it is still valid. Incorporating such uncertainties into the returned answers is, however, a challenge considering the complex interplay between the relevance of a house by itself, and the probability that the advertisement is still valid.

Many other application domains also exhibit resource constraints of some form, and we must somehow rank the entities or tuples under consideration to select the most relevant objects to focus our attention on. For example, in financial applications, we may want to choose the best stocks in which to invest, given their expected performance in the future (which is uncertain at best). In learning or classification tasks, we often need to choose the best "k" features to use [55]. In sensor networks or scientific databases, we may

Computer Science Department, University of Maryland, College Park 20742, MD, USA

not know the "true" values of the physical properties being measured because of measurement noises or failures [16], but we may still need to choose a set of sensors or entities in response to a user query.

Ranking in presence of uncertainty is non-trivial even if the relevance scores can be computed easily (the main challenge in the deterministic case), mainly because of the complex trade-offs introduced by the score distributions and the tuple uncertainties. This has led to many ranking functions being proposed for combining the scores and the probabilities in recent years, all of which appear quite natural at the first glance (we review several of them in detail later). We begin with a systematic exploration of these issues by recognizing that ranking in probabilistic databases is inherently a multi-criterion optimization problem, and by deriving a set of *features*, the key properties of a probabilistic dataset that influence the ranked result. We empirically illustrate the diverse and conflicting behavior of several natural ranking functions, and argue that a single specific ranking function may not be appropriate to rank different uncertain databases that we may encounter in practice. Furthermore, different users may weigh the features differently, resulting in different rankings over the same dataset. We then define a general and powerful ranking function, called PRF, that allows us to explore the space of possible ranking functions. We discuss its relationship to previously proposed ranking functions, and also identify two specific parameterized ranking functions, called PRF^ω and PRF^e, as being interesting. The PRF^ω ranking function is essentially a linear, weighted ranking function that resembles the scoring functions typically used in information retrieval, web search, data integration, keyword query answering etc. [8, 15, 26, 33, 51]. We observe that PRF^ω may not be suitable for ranking large datasets due to its high running time, and instead propose PRF^e, which uses a single parameter, and can effectively approximate previously proposed ranking functions for probabilistic databases very well.

We then develop novel algorithms based on *generating functions* to efficiently rank the tuples in a probabilistic dataset using any PRF ranking function. Our algorithm can handle a probabilistic dataset with arbitrary correlations; however, it is particularly efficient when the probabilistic database contains only *mutual exclusivity* and/or *co-existence* correlations (called *probabilistic and/xor trees* [41]). Our main contributions can be summarized as follows:

- We develop a framework for *learning* ranking functions over probabilistic databases by identifying a set of key *features*, by proposing several parameterized ranking functions over those features, and by choosing the parameters based on user preferences or feedback.
- We present novel algorithms based on *generating functions* for efficiently ranking very large datasets. Our key algorithm is an $O(n \log n)$ algorithm for evaluating a PRF^e function over datasets with low correlations (specifically, constant height probabilistic and/xor trees). The algorithm runs in $O(n)$ time if the dataset is pre-sorted by score.
- We present a polynomial time algorithm for ranking a correlated dataset when the correlations are captured using a bounded-treewidth graphical model. The algorithm we present is actually for computing the probability that a given tuple is ranked at a given position across all the possible worlds, and is of independent interest.
- We develop a novel, DFT-based algorithm for approximating an arbitrary weighted ranking function using a linear combination of PRF^e functions.
- We show that a PRF^ω ranked result can be seen as a *consensus* answer under a suitably defined distance function – a consensus answer is defined to be the answer that is closest in expectation to the answers over the possible worlds.
- We present a comprehensive experimental study over several real and synthetic datasets, comparing the behavior of the ranking functions and the effectiveness of our proposed algorithms.

Outline: We begin with a brief discussion of the related work (Section 2). In Section 3, we review our probabilistic database model and the prior work on ranking in probabilistic databases, and propose two parameterized ranking functions. In Section 4, we present our generating functions-based algorithms for ranking. We then present an approach to approximate different ranking functions using our parameterized ranking functions, and to learn a ranking function from user preferences (Section 5). In Section 6, we explore the connection between PRF^ω and consensus top-k query results. In Section 7, we observe an interesting property of the PRF^e function that helps us gain better insight into its behavior. We then present a comprehensive experiment study in Section 8. Finally, in Section 9, we develop an algorithm for handling correlated datasets where the correlations are captured using bounded-treewidth graphical models.

2 Related Work

There has been much work on managing probabilistic, uncertain, incomplete, and/or fuzzy data in database systems (see, e.g., [11, 13, 20, 23, 37, 40, 52]). The work in this area has spanned a range of issues from theoretical development of data models and data languages to practical implementation issues such as indexing techniques; several research efforts are underway to build systems to manage uncertain data (e.g., MYSTIQ [13], Trio [52], ORION [11], MayBMS [37], PrDB [48]). The approaches can be differentiated based on whether they support *tuple-level uncertainty* where "existence" probabilities are attached to the tuples

of the database, or *attribute-level uncertainty* where (possibly continuous) probability distributions are attached to the attributes, or both. The proposed approaches differ further based on whether they consider correlations or not. Most work in probabilistic databases has either assumed independence [13, 20] or has restricted the correlations that can be modeled [2, 40, 47]. More recently, several approaches have been presented that allow representation of arbitrary correlations and querying over correlated databases [23, 38, 48].

The area of ranking and top-k query processing has also seen much work in databases (see, e.g., Ilyas et al.'s survey [28]). More recently, several researchers have considered top-k query processing in probabilistic databases. Soliman et al. [49] defined the problem of ranking over probabilistic databases, and proposed two ranking functions to combine tuple scores and probabilities. Yi et al. [53] present improved algorithms for the same ranking functions. Zhang et al. [54] present a desiderata for ranking functions, and propose the notion of *Global Top-k* answers. Ming Hua et al. [27] propose *probabilistic threshold ranking*, which is quite similar to Global Top-k. Cormode et al. [12] also present a semantics of ranking functions and a new ranking function called *expected rank*. Liu et al. [44] propose the notion of k-selection queries; unlike most of the above definitions, the result here is sensitive to the actual tuple scores. We will review these ranking functions in detail in next section. Ge et al. [21] propose the notion of *typical answers*, where they propose returning a collection of typical answers instead of just one answer. This can be seen as complementary to our approach here; one could show the typical answers to the user to understand the user preferences during an exploratory phase, and then learn a single ranking function to rank using the techniques developed in this article.

There is also work on top-k query processing in probabilistic databases where the ranking is by the result tuple *probabilities* (i.e., probability and score are identical) [45]. The main challenge in that work is efficient computation of the probabilities, whereas we assume that the probability and score are either given or can be computed easily.

The aforementioned work has focused mainly on tuple uncertainty and discrete attribute uncertainty. Soliman and Ilyas [50] were the first to consider the problem of handling continuous distributions. Recently, in a followup work [42], we extended the algorithm for PRF to arbitrary continuous distributions. We were able to obtain exact polynomial time algorithms for some continuous probability distribution classes, and efficient approximation schemes with provable guarantees for arbitrary probability distributions. One important ingredient of those algorithms is an extension of the generating function used in this article.

Recently, there has also been much work on nearest neighbor-style queries over uncertain datasets [5, 9, 10, 39]. In fact, a nearest neighbor query (or a k-nearest neighbor query) can be seen as a ranking query where the score of a point is the distance of that point to the given query point. Thus, our new ranking semantics and algorithms can be directly used for nearest neighbor queries over uncertain points with discrete probability distributions.

There is a tremendous body of work on ranking documents in information retrieval, and learning how to rank documents given user preferences (see Liu [43] for a comprehensive survey). That work has considered aspects such as different ranking models, loss functions, different scoring techniques etc. The techniques developed there tend to be specific to document retrieval (focusing on keywords, terms, and relevance), and usually do not deal with *existence* uncertainty (although they often do model document relevance as a random variable). Furthermore, our work here primarily focuses on highly efficient algorithms for ranking using a spectrum of different ranking functions. Exploring and understanding the connections between the two research areas is a fruitful direction for further research.

Finally, we note that one PRF function is only able to model preferences of one user. There is an increasing interest in finding a ranking that satisfies multiple users having diverse preferences and intents (see, e.g., [3, 4]). However, those models typically assume *certain* inputs. Incorporating uncertainty into those models or introducing the notion of diversity into our model is an interesting research direction.

3 Problem Formulation

We begin with defining our model of a probabilistic database, called *probabilistic and/xor tree* [41], that captures several common types of correlations. We then review the prior work on top-k query processing in probabilistic databases, and argue that a single specific ranking function may not capture the intricacies of ranking with uncertainty. We then present our parameterized ranking functions, PRF$^\omega$ and PRF^e.

3.1 Probabilistic Database Model

We use the prevalent *possible worlds semantics* for probabilistic databases [13]. We denote a probabilistic relation with tuple uncertainty by D_T, where T denotes the set of tuples (in Section 4.4, we present extensions to handle attribute uncertainty). The set of all possible worlds is denoted by $PW = \{pw_1, pw_2, ..., pw_n\}$. Each tuple $t_i \in T$ is associated with an existence probability $\Pr(t_i)$ and a score $\text{score}(t_i)$, computed based on a scoring function $\text{score} : T \to \mathbb{R}$. Usually $\text{score}(t)$ is computed based on the tuple attribute values and measures the relative user preference for different tuples. In a deterministic database, tuples with higher scores should be ranked higher. We use

Time	Car Loc	Plate No	Speed	...	Prob	Tuple Id
11:40	L1	X-123	120	...	0.4	t_1
11:55	L2	Y-245	130	...	0.7	t_2
11:35	L3	Y-245	80	...	0.3	t_3
12:10	L4	Z-541	95	...	0.4	t_4
12:25	L5	Z-541	110	...	0.6	t_5
12:15	L6	L-110	105	...	1.0	t_6

Possible Worlds	Prob
$pw_1 = \{t_2, t_1, t_6, t_4\}$	.112
$pw_2 = \{t_2, t_1, t_5, t_6\}$	.168
$pw_3 = \{t_1, t_6, t_4, t_3\}$	.048
$pw_4 = \{t_1, t_5, t_6, t_3\}$	.072
$pw_5 = \{t_2, t_6, t_4\}$	.168
$pw_6 = \{t_2, t_5, t_6\}$	.252
$pw_7 = \{t_6, t_4, t_3\}$	.072
$pw_8 = \{t_5, t_6, t_3\}$	.108

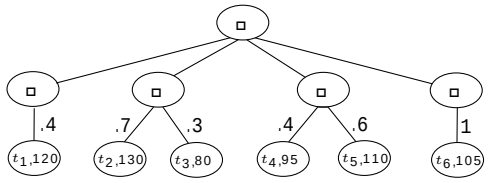

Fig. 1 Example of a probabilistic database which contains automatically captured information about speeding cars – here the *Plate No.* is the possible worlds key and the *speed* is the score attribute that we will use for ranking. Tuples t_2 and t_3 (similarly, t_4 and t_5) are mutually exclusive. The second table lists all possible worlds. Note that the tuples are sorted according to their speeds in each possible world. The corresponding and/xor tree compactly encodes these correlations.

$r_{pw} : T \to \{1, \ldots, n\} \cup \{\infty\}$ to denote the rank of the tuple t in a possible world pw according to score. If t does not appear in the possible world pw, we let $r_{pw}(t) = \infty$. We say t_1 *ranks higher* than t_2 in the possible world pw if $r_{pw}(t_1) < r_{pw}(t_2)$. For each tuple t, we define a random variable $r(t)$ that denotes the rank of t in D_T.

Definition 1 The *positional probability* of a tuple t being ranked at position k, denoted $\Pr(r(t) = k)$, is the total probability of the possible worlds where t is ranked at position k. The *rank distribution* of a tuple t, denoted $\Pr(r(t))$, is simply the probability distribution of the random variable $r(t)$.

Probabilistic And/Xor Tree Model: Our algorithms can handle arbitrarily correlated datasets with correlations modeled using Markov networks (Section 9). However, in most of this article, we focus on the *probabilistic and/xor tree model*, introduced in our prior work [41], that can capture only a more restricted set of correlations, but admits highly efficient query processing algorithms. More specifically, an and/xor tree captures two types of correlations: (1) *mutual exclusivity* (denoted $\oplus$ (*xor*)) and (2) *mutual co-existence* ($\wedge$ (*and*)). Two events satisfy the mutual co-existence correlation if, in any possible world, either both events occur or neither occurs. Similarly two events are mutually exclusive if there is no possible world where both happen.

Formally, in tree T, we denote the set of children of node v by $Ch_T(v)$ and the least common ancestor of two leaves l_1 and l_2 by $LCA_T(l_1, l_2)$. We omit the subscript if the context is clear. For simplicity, we separate the attributes of the relation into two groups: (1) a possible worlds *key*, denoted K, which is unique in any possible world (i.e., two tuples that agree on K are mutually exclusive), and (2) the value attributes, denoted A. If the relation does not have any key attributes, $K = \phi$.

Definition 2 A probabilistic and/xor tree T represents the mutual exclusion and co-existence correlations in a probabilistic relation $R^P(K; A)$, where K is the possible worlds key, and A denotes the value attributes. In T, each leaf denotes a tuple, and each inner node has a mark, $\oplus$ or $\wedge$. For each $\oplus$ node u and each of its children $v \in Ch(u)$, there is a nonnegative value $p_{(u,v)}$ associated with the edge (u, v). Moreover, we require:

Possible Worlds	Prob
$pw_1 = \{(t_3; 6); (t_2; 5); (t_1 1)\}$	.3
$pw_2 = \{(t_3; 9); (t_1; 7)\}$	.3
$pw_3 = \{(t_2; 8); (t_4; 4); (t_5; 3)\}$	.4

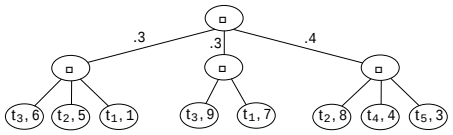

Fig. 2 Example of a highly correlated probabilistic database with 3 possible worlds and the and/xor tree that captures the correlation.

(Probability Constraint) $\sum_{v: v \in Ch(u)} \Pr(u, v) \leq 1$.

(Key Constraint) For any two different leaves l_1, l_2 holding the same key, $LCA(l_1, l_2)$ is a $\oplus$ node.[1]

Let T_v be the subtree rooted at v and $Ch(v) = \{v_1, \ldots, v_\ell\}$. The subtree T_v inductively defines a random subs]TJ/F14 9.9626 Tf 17626

finite set of possible worlds. This can be done by listing all possible worlds, creating one ^ node for each world, and using an _ node as the root to capture that these worlds are mutually exclusive. Figure 2 shows an example of this.

The correlations captured by an and/xor tree can be represented by probabilistic c-tables [23] and provenance semirings [22]. However, that does not directly imply an efficient algorithm for ranking. We remark that Markov or Bayesian networks are able to capture more general correlations in a compact way [48]; however, probability computations on them (*inference*) is typically exponential in the treewidth of the model. The treewidth of an and/xor tree (viewing it as a Markov network) is not bounded, and hence the techniques developed for those models can not be used to obtain polynomial time algorithms for and/xor trees. And/xor trees also exhibit superficial similarities to ws-trees [38], which can also capture mutual exclusivity and coexistence between tuples. We note that no prior work on ranking in probabilistic databases has considered more complex correlations than x-tuples.

3.2 Ranking over Probabilistic Data: Definitions and Prior Work

The interplay between probabilities and scores complicates the semantics of ranking in probabilistic databases. This was observed by Soliman et al. [49], who first considered this problem and presented two definitions of top-k queries in probabilistic databases. Several other definitions of ranking have been proposed since then. We briefly review the ranking functions we consider in this work.

- **Uncertain Top-k** (U-Top) [49]: Here the query returns the k-tuple set that appears as the top-k answer in most possible worlds (weighted by the probabilities of the worlds).
- **Uncertain Rank-k** (U-Rank) [49]: At each rank i, we return the tuple with the maximum probability of being at the i'th rank in all possible worlds. In other words, U-Rank returns:
  $\{t_i, i = 1, 2, .., k\}$, where $t_i = argmax_t(\Pr(r(t) = i))$. Note that, under these semantics, the same tuple may be ranked at multiple positions. In our experiments, we use a slightly modified version that enforces distinct tuples in the answer (by not choosing a tuple at a position if it is already chosen at a higher position).
- **Probabilistic Threshold Top-k** (PT(h)) [27][2]: The original definition of a probabilistic threshold query asks for all tuples with probability of being in top-h answer larger than a pre-specified threshold, i.e., all tuples t such that $\Pr(r(t) \quad h) > threshold$. For consistency with other ranking functions, we slightly modify the definition and instead ask for the k tuples with the largest $\Pr(r(t) \quad h)$ values.
- **Expected Ranks** (E-Rank) [12]: The tuples are ranked in the increasing order by the *expected* value of their ranks across the possible worlds, i.e., by:
  $$\sum_{pw \in PW} \Pr(pw) r_{pw}(t),$$
  where $r_{pw}(t)$ is defined to be $|pw|$ if $t \notin pw$.
- **Expected Score** (E-Score): Another natural ranking function, also considered by [12], is simply to rank the tuples by their expected score, $\Pr(t)\text{score}(t)$.
- **k-selection Query [44]:** A *k-selection* query returns the set of k tuples, such that the expected score of the best available tuple across the possible worlds is maximized.
- **Consensus Top-k** (Con-Topk): This is a semantics for top-k queries developed under the framework of *consensus answers* in probabilistic databases [41]. We defer its definition till Section 6 where we discuss in detail its relationship with the PRF function proposed in this article.

Normalized Kendall Distance: To compare different ranking functions or criteria, we need a distance measure to evaluate the closeness of two top-k answers. We use the prevalent *Kendall tau* distance defined for comparing top-k answers for this purpose [19]. It is also called *Kemeny distance* in the literature and is considered to have many advantages over other distance metrics [18]. Let K_1 and K_2 denote two top-k ranked lists. Then *Kendall tau distance* between K_1 and K_2 is defined to be:

$$\text{dis}(K_1, K_2) = \sum_{(i,j) \in P(K_1, K_2)} \hat{K}(i, j),$$

where $P(K_1, K_2)$ is the set of all unordered pairs of $K_1 \cup K_2$. Given such a pair (i, j), we use notation $i >_{K_1} j$ to denote that either (1) $i \in K_1$, $j \in K_1$ and i is ranked higher than j in K_1, or (2) $i \in K_1$, $j \notin K_1$. Then, $\hat{K}(i, j) = 1$ if either: (1) $i >_{K_1} j$ and $i <_{K_2} j$; or (2) $i <_{K_1} j$ and $i >_{K_2} j$. Otherwise, $\hat{K}(i, j) = 0$. Intuitively the Kendall distance measures the number of inversions between the two rankings. For ease of comparison, we divide the Kendall distance by k^2 to obtain *normalized Kendall* distance, which always lies in $[0, 1]$.

Example 1 Suppose the set of tuples is $\{1, 2, 3, 4, 5\}$, and we have two top-3 lists $K_1 = \{1, 2, 3\}$ and $K_2 = \{3, 2, 4\}$. Then, for instance, $\hat{K}(1, 2) = 1$, $\hat{K}(1, 4) = 1$, and $\hat{K}(3, 4) = 0$. The normalized Kendall distance is $\text{dis}(K_1, K_2) = \frac{4}{9}$.

A higher value of the Kendall distance indicates a larger disagreement between the two top-k lists. It is easy to see that if the Kendall distance between two top-k answers is δ, then the two answers must share at least $1 - \sqrt{\delta}$ fraction of tuples (so if the distance is 0.09, then the top-k answers share at least 70%, and typically 90% or more tuples). The

[2] This is quite similar to the Global Top-k semantics [54].

	E-Score	PT(100)	U-Rank	E-Rank	U-Top
E-Score	–	0.1241	0.3027	0.7992	0.2760
PT(100)	0.1241	–	0.3324	0.9290	0.3674
U-Rank	0.3027	0.3324	–	0.9293	0.2046
E-Rank	0.7992	0.9290	0.9293	–	0.9456
U-Top	0.2760	0.3674	0.2046	0.9456	–

IIP-100,000 (k = 100)

	E-Score	PT(100)	U-Rank	E-Rank	U-Top
E-Score	–	0.8642	0.8902	0.0044	0.9258
PT(100)	0.8642	–	0.3950	0.8647	0.5791
U-Rank	0.8902	0.3950	–	0.8907	0.3160
E-Rank	0.0044	0.8647	0.8907	–	0.9263
U-Top	0.9258	0.5791	0.3160	0.9263	–

Syn-IND Dataset with 100,000 tuples (k = 100)

Table 1 Normalized Kendall distance between top-k answers according to various ranking functions for two datasets.

distance is 0 if two top-k answers are identical and 1 if they are disjoint.

Comparing Ranking Functions: We compared the top-100 answers returned by five of the ranking functions with each other using the normalized Kendall distance, for two datasets with 100,000 independent tuples each (see Section 8 for a description of the datasets). Table 1 shows the results of this experiment. As we can see, the five ranking functions return wildly different top-k answers for the two datasets, with no obvious trends. For the first dataset, E-Rank behaves very differently from all other functions, whereas for the second dataset, E-Rank happens to be quite close to E-Score. However both of them deviate largely from U-Top, PT(h), and U-Rank. The behavior of E-Score is very sensitive to the dataset, especially the score distribution: it is close to PT(h) and U-Rank for the first dataset, but far away from all of them in the second dataset (by looking into the results, it shares less than 15 tuples with the Top-100 answers of the others). We observed similar behavior for other datasets, and for datasets with correlations.

This simple experiment illustrates the issues with ranking in probabilistic databases – although several of these definitions seem natural, the wildly different answers they return indicate that none of them could be the "right" definition. We also observe that in large datasets, E-Rank tends to give very high priority to a tuple with a high probability even if it has a low score. In our synthetic dataset Syn-IND-100,000 with expected size 50000, t_2 (the tuple with 2nd highest score) has probability approximately 0.98 and t_{1000} (the tuple with 1000th highest score) has probability 0.99. The expected ranks of t_2 and t_{1000} are approximately 10000 and 6000 respectively, and hence t_{1000} is ranked above t_2 even though t_{1000} is only slightly more probable.

As mentioned above, the original U-Rank function may return the same tuple at different ranks (also observed by the authors [49]). This problem becomes even more severe when the dataset and k are both large. For example, in RD-100,000, the same tuple is ranked at positions 67895 to 100000. In the table, we show a slightly modified version of U-Rank to enforce distinct tuples in the answer.

3.3 Parameterized Ranking Functions

Ranking in uncertain databases is inherently a multi-criterion optimization problem, and it is not always clear how to rank two tuples that dominate each other along different axes. Consider a database with two tuples t_1 (score = 100, Pr(t_1) = 0.5), and t_2 (score = 50, Pr(t_2) = 1.0). Even in this simple case, it is not clear whether to rank t_1 above t_2 or vice versa. This is an instance of the classic risk-reward trade-off, and the choice between these two options largely depends on the application domain and/or user preferences.

We propose to follow the traditional approach to dealing with such tradeoffs, by identifying a set of *features*, by defining a parameterized ranking function over these features, and by learning the parameters (weights) themselves using user preferences [8, 15, 26, 33]. To achieve this, we propose a family of ranking functions, parameterized by one or more parameters, and design algorithms to efficiently find the top-k answer according to any ranking function from these families. Our general ranking function, PRF, directly subsumes some of the previously proposed ranking functions, and can also be used to approximate other ranking functions. Moreover, the parameters can be learned from user preferences, which allows us to adapt to different scenarios and different application domains.

Features: Although it is tempting to use the tuple probability and the tuple score as the features, a ranking function based on just those two will be highly sensitive to the actual values of the scores; further, such a ranking function will be insensitive to the correlations in the database, and hence cannot capture the rich interactions between ranking and possible worlds.

Instead we propose to use the positional probabilities as the features: for each tuple t, we have n features,

$$\Pr(r(t) = i), i = 1, \quad , n,$$

where n is the number of tuples in the database. This set of features succinctly captures the possible worlds. Further, correlations among tuples, if any, are naturally accounted for when computing the features. We note that in most cases, we do not explicitly compute all the features, and instead design algorithms that can directly compute the value of the overall ranking function.

Ranking Functions: Next we define a general ranking function which allows exploring the trade-offs discussed above.

Definition 3 Let $\omega : T \quad \mathbb{N} \; ! \; \mathbb{C}$ be a *weight function*, that maps a tuple-rank pair to a complex number. The *parameterized ranking function (*PRF*)*, $\Upsilon_\omega : T \; ! \; \mathbb{C}$ in its most

Pr(r(t$_i$) = j)	Positional prob. of t$_i$ being ranked at position j
Pr(r(t$_i$))	Rank distribution of t$_i$
PRF	Parameterized ranking function ω(t) = $\sum_{i>0}$! (t; i) Pr(r(t) = i)
PRF$^\omega$(h)	Special case of PRF: ! (t; i) = w$_i$, w$_i$ = 0; 8i > h
PRF^e( )	Special case of PRF$^\omega$: w$_i$ = ^{i}; 2 C
PRF$^\ell$	Special case of PRF$^\omega$: w$_i$ = i
(p)	Delta function: (p) = 1 if p is true, (p) = 0 o.w.

Table 2 Notation

general form is defined to be:

$$\begin{aligned}\Upsilon_\omega(t) &= \sum_{pw:t\,2\,pw} \omega(t, r_{pw}(t)) \quad \Pr(pw) \\ &= \sum_{pw:t\,2\,pw} \sum_{i>0} \omega(t,i) \Pr(pw \wedge r_{pw}(t) = i) \\ &= \sum_{i>0} \omega(t,i) \quad \Pr(r(t) = i).\end{aligned}$$

A top-k query returns k tuples with the highest jΥ_ωj values.

In most cases, ω is a real positive function and we just need to find the k tuples with highest Υ_ω values. However we allow ω to be a complex function in order to approximate other functions efficiently (see Section 5.1). Depending on the actual function ω, we get different ranking functions with diverse behaviors. Before discussing the relationship to prior ranking functions, we define two special cases.

PRF$^\omega(h)$: One important class of ranking functions is when $\omega(t,i) = \omega(i)$ (i.e., independent of t); $\omega(i) = 0$ $8i > h$ for some positive integer h (typically $h \quad n$). This forms one of prevalent classes of ranking functions used in domains such as information retrieval and machine learning, with the weights typically learned from user preferences [8, 15, 26, 33]. Also, the weight function $\omega(i) = \frac{\ln 2}{\ln(i+1)}$ (called *discount factor*) is often used in the context of ranking documents in information retrieval [29].

PRF$^e(\alpha)$: This is a special case of PRF$^\omega(h)$ where $\omega(i) = \alpha^i$, where α is a constant and may be a real or a complex number. Here $h = n$ (no weights are 0). Typically we expect jαj 1, otherwise we have the counterintuitive behavior that tuples with lower scores are preferred.

PRF$^\omega$ and PRF^e form the two parameterized ranking functions that we propose in this work. Although PRF$^\omega$ is the more natural ranking function and has been used elsewhere, PRF^e is more suitable for ranking in probabilistic databases for various reasons. First, the features as we have defined above are not completely arbitrary, and the features Pr($r(t) = i$) for small i are clearly more important than the ones for large i. Hence in most cases we would like the weight function, $\omega(i)$, to be monotonically non-increasing. PRF^e naturally captures this behavior (as long as jαj 1). More importantly, we can compute the PRF^e function in $O(n \log n)$ time ($O(n)$ time if the dataset is presorted by score) even for datasets with low degrees of correlations (i.e., modeled by and/xor trees with low heights). This makes it significantly more attractive for ranking over large datasets.

Furthermore, ranking by PRF$^e(\alpha)$, with suitably chosen α, can approximate rankings by many other functions reasonably well even with only real α. Finally, a linear combination of exponential functions, with complex bases, is known to be very expressive in representing other functions [6]. We make use of this fact to approximate many ranking functions by linear combinations of a small number of PRF^e functions, thus significantly speeding up the running time (Section 5.1).

Relationship to other ranking functions: We illustrate some of the choices of weight function, and relate them to prior ranking functions.[3] We omit the subscript ω if the context is clear. Let $\delta(p)$ denote a delta function where p is a boolean predicate: $\delta(p) = 1$ if $p = true$, and $\delta(p) = 0$ otherwise.

- **Ranking by probabilities:** If $\omega(t,i) = 1$, the result is the set of k tuples with the highest probabilities [45].
- **Expected Score:** By setting $\omega(t,i) = \text{score}(t)$, we get the E-Score:

$$\begin{aligned}\Upsilon(t) &= \sum_{pw:t\,2\,pw} \text{score}(t) \Pr(pw) = \text{score}(t) \Pr(t) \\ &= \text{E}[\text{score}(t)]\end{aligned}$$

- **Probabilistic Threshold Top-k** (PT(h))**:** If we choose $\omega(i) = \delta(i \quad h)$, i.e., $\omega(i) = 1$ for $i \quad h$, and = 0 otherwise, then we have exactly the answer for PT(h).
- **Uncertain Rank-k** (U-Rank)**:** Let $\omega_j(i) = \delta(i = j)$, for some $1 \quad j \quad k$. We can see the tuple with largest Υ_{ω_j} value is the rank-j answer in U-Rank query [49]. This allows us to compute the U-Rank answer by evaluating $\Upsilon_{\omega_j}(t)$ for all t 2 T and $j = 1, \ldots, k$.
- **Expected ranks** (E-Rank): Let PRF$^\ell$ (PRF linear) be another special case of the PRF$^\omega$ function, where $w_i = \omega(i) = i$. The PRF$^\ell$ function bears a close similarity to the notion of *expected ranks*. Recall that the expected rank of a tuple t is defined to be:

$$\text{E}[r_{pw}(t)] = \sum_{pw\,2\,PW} \Pr(pw) r_{pw}(t)$$

where $r_{pw}(t) =$ jpwj if $t_i \not 2 \; pw$. Let C denote the expected size of a possible world. It is easy to see that: $C = \sum_{i=1}^{n} p_i$ due to linearity of expectation. Then the expected rank of t can be seen to consist of two parts:

[3] The definition of the U-Top introduced by Soliman et al. [49] requires that the retrieved k tuples belong to a valid possible world. However, it is not required in our definition, and hence it is not possible to simulate U-Top using PRF.

(1) the contribution of possible worlds where t exists:

$$er_1(t) = \sum_{i>0} i \quad \Pr(r(t) = i) = \quad \Upsilon(t)$$

where $\Upsilon(t)$ is the PRF$^\ell$ value of tuple t.[4]

(2) the contribution of worlds where t does not exist:

$$er_2(t) = \sum_{pw: t \notin pw} \Pr(pw)|pw|$$
$$= (1 \quad p(t))(\sum_{t_i \neq t} \Pr(t_i \mid t \text{ does not exist}))$$

If the tuples are independent of each other, then we have:

$$\sum_{t_i \neq t} \Pr(t_i \mid t \text{ does not exist}) = (C \quad p(t))$$

Thus, the expected ranks can be computed in the same time as PRF$^\ell$ in tuple-independent datasets. This term can also be computed efficiently in many other cases, including in datasets where only mutual exclusion correlations are permitted. If the correlations are represented using a probabilistic and/xor tree (see Section 4.2) or a low-treewidth graphical model (see Section 9), then we can compute this term efficiently as well, thus generalizing the prior algorithms for computing expected ranks.

– **k-selection Query [44]:** It is easy to see that a k-selection query is equivalent to setting: $\omega(t, i) = \delta(i = 1)\text{score}(t)$.

As we can see, many different ranking functions can be seen as special cases of the general PRF ranking function, supporting our claim that PRF can effectively unify these different approaches to ranking uncertain datasets.

Finally, we note that, when choosing the positional probabilities as the only "features", we made an implicit assumption that the actual values (magnitudes) of the scores do not matter (this property was called "value invariance" by Cormode et al. [12]). However, that may not be true in some application domains. In its most general form, PRF allows tuple-specific and hence score-dependent weights, but such weight functions are not easy to learn, and significant domain expertise may be needed to design them. We can also explicitly use score as an additional feature; the contribution of this feature to the PRF values of the tuples can be computed in $O(n)$ time (our algorithms for computing PRF values have at least $O(n \log n)$ complexity). Further, except for the special algorithm for learning a single PRF^e function (Section 5.2), the other learning algorithms discussed in this article can be easily adapted to include the score feature. For simplicity, we omit the score feature in the rest of the article.

[4] Note that, in the expected rank approach, we pick the k tuples with the *lowest* expected rank, but in our approach, we choose the tuples with the *highest* PRF function values, hence the negation.

4 Ranking Algorithms

We next present an algorithm for efficiently ranking according to a PRF function. We first present the basic idea behind our algorithm assuming mutual independence, and then consider correlated tuples with correlations represented using an and/xor tree. We then present a very efficient algorithm for ranking using a PRF^e function, and then briefly discuss how to handle attribute uncertainty.

4.1 Assuming Tuple Independence

First we show how the PRF function can be computed in $O(n^2)$ time for a general weight function ω, and for a given set of tuples $T = \{t_1, \ldots, t_n\}$. In all our algorithms, we assume that $\omega(t, i)$ can be computed in $O(1)$ time.

Clearly it is sufficient to compute $\Pr(r(t) = j)$ for any tuple t and $1 \quad j \quad n$ in $O(n^2)$ time. Given these values, we can directly compute the values of $\Upsilon(t)$ in $O(n^2)$ time. (Later, we will present several algorithms which run in $O(n)$ or $O(n \log n)$ time which combine these two steps for some special ω functions).

We first sort the tuples in a non-increasing order by their scores; assume $t_1, \ldots, t_n$ indicates this sorted order. Suppose now we want to compute $\Pr(r(t_i) = j)$. Let $T_i = \{t_1, t_2, \ldots, t_i\}$ and σ_i be an indicator variable that takes value 1 if t_i is present in a possible world, and 0 otherwise. Further, let $\sigma = \langle\sigma_1, \ldots, \sigma_n\rangle$ denote a vector containing all the indicator variables. Then, we can write $\Pr(r(t_i) = j)$ as follows:

$$\Pr(r(t_i) = j)$$
$$= \Pr(t_i) \sum_{pw: |pw \setminus T_{i-1}| = j-1} \Pr(pw)$$
$$= \Pr(t_i) \sum_{\sigma: \sum_{l=1}^{i-1} \sigma_l = j-1} \prod_{l<i:\sigma_l=1} \Pr(t_l) \prod_{l<i:\sigma_l=0} (1 \quad \Pr(t_l))$$

The first equality says that tuple t_i ranks at the jth position if and only if t_i *and* exactly $j \quad 1$ tuples from $T_{i \quad 1}$ are present in the possible world. The second equality is obtained by rewriting the sum to be over the indicator vector (each assignment to the indicator vector corresponds to a possible world), and by exploiting the fact that the tuples are independent of each other. The naive method to evaluate the above formula by explicitly listing all possible worlds needs exponential time. Now, we present a polynomial time algorithm based on generating functions.

Consider the following generating function over x:

$$F(x) = \prod_{i=1}^{n} (a_i + b_i x)$$

The coefficient of x^k in $F(x)$ is:

$$\sum_{|\beta|=k} \prod_{i:\beta_i=0} a_i \prod_{i:\beta_i=1} b_i$$

where $\beta = \langle\beta_1, \ldots, \beta_n\rangle$ is a boolean vector, and $|\beta|$ denotes

Algorithm 1: IND-PRF-RANK(D_T)

```
F^0(x) = 1
for i = 1 to n do
    F^i(x) = (Pr(t_i) / Pr(t_{i-1})) F^{i-1}(x) (1 - Pr(t_{i-1}) + Pr(t_{i-1})x)
    Expand F^i(x) in the form of sum_j c_j x^j
    Υ(t_i) = sum_{j=1}^{n} ω(t_i, j) c_j
return k tuples with largest Υ values
```

the number of 1's in β. Now consider the following generating function:

$$F^i(x) = \left(\prod_{t \in T_{i-1}} (1 - \Pr(t) + \Pr(t)\, x)\right) \Pr(t_i)\, x$$
$$= \sum_j c_j x^j.$$

We can see that the coefficient c_j of x^j in the expansion of F^i is exactly the probability that t_i is at rank j, i.e., $c_j = \Pr(r(t_i) = j)$. We note F^i contains at most $i+1$ nonzero terms. We observe this both from the form of F^i above, and also from the fact that $\Pr(r(t_i) = j) = 0$ if $j > i$. Hence, we can expand F^i to compute the coefficients in $O(i^2)$ time. This allows us to compute $\Pr(r(t_i) = j)$ for t_i in $O(i^2)$ time; $\Upsilon(t_i)$, in turn, can be written as:

$$\Upsilon(t_i) = \sum_j \omega(t_i, j)\ \Pr(r(t_i) = j) = \sum_j \omega(t_i, j) c_j \qquad (1)$$

which can be computed in $O(i^2)$ time.

Example 2 Consider a relation with 3 independent tuples t_1, t_2, t_3 (already sorted according to the score function) with existence probabilities 0.5, 0.6, 0.4, respectively. The generating function for t_3 is:

$F^3(x) = (.5 + .5x)(.4 + .6x)(.4x) = .12x^3 + .2x^2 + .08x$

This gives us:

$\Pr(r(t_3) = 1) = .08; \Pr(r(t_3) = 2) = .2; \Pr(r(t_3) = 3) = .12$

If we expand each F^i for $1 \le i \le n$ from scratch, we need $O(n^2)$ time for each F^i and $O(n^3)$ time in total. However, the expansion of F^i can be obtained from the expansion of F^{i-1} in $O(i)$ time by observing that:

$$F^i(x) = \frac{\Pr(t_i)}{\Pr(t_{i-1})} F^{i-1}(x)\left(1 - \Pr(t_{i-1}) + \Pr(t_{i-1})x\right) \qquad (2)$$

This trick gives us a $O(n^2)$ time complexity for computing the values of the ranking function for all tuples. See Algorithm 1 for the pseudocode. Note that $O(n^2)$ time is asymptotically optimal in general since the computation involves at least $O(n^2)$ probabilities, namely $\Pr(r(t_i) = j)$ for all $1 \le i, j \le n$.

For some specific ω functions, we may be able to achieve faster running time. For PRF$^\omega(h)$ functions, we only need to expand all F^i's up to x^h term since $\omega(i) = 0$ for $i > h$. Then, the expansion from $F^{i-1}(x)$ to $F^i(x)$ only takes $O(h)$ time. This yields an $O(n(h + n\log n))$ time algorithm. We note the above technique also gives an $O(nk + n\log n)$ time algorithm for answering the U-Rank top-k query (all the needed probabilities can be computed in that time), thus matching the best known upper bound by Yi et al. [53] (the original algorithm in [49] runs in $O(n^2k)$ time).

We remark that the generating function technique can be seen as a variant of dynamic programming in some sense; however, using it explicitly in place of the obscure recursion formula gives us a much cleaner view and allows us to generalize it to handle more complicated tuple correlations. This also leads to an algorithm for extremely efficient evaluation of PRF^e functions (Section 4.3).

4.2 Probabilistic And/Xor Trees

Next we generalize our algorithm to handle a correlated database where the correlations can be captured using an *and/xor* tree. In fact, many types of probability computations on and/xor trees can be done efficiently and elegantly using generating functions. Here we first provide a general result and then specialize it for PRF computation.

As before, let $T = \{t_1, t_2, \ldots, t_n\}$ denote the tuples sorted in a non-increasing order of their score function, and let $T_i = \{t_1, t_2, \ldots, t_i\}$. Let $\mathcal{T}$ denote the and/xor tree. Suppose $X = \{x_1, x_2, \ldots\}$ is a set of variables. Define a mapping π which associates each leaf $l \in \mathcal{T}$ with a variable $\pi(l) \in X$. Let $\mathcal{T}_v$ denote the subtree rooted at v and let $v_1, \ldots, v_h$ be v's children. For each node $v \in \mathcal{T}$, we define a generating function $F_v(X) = F_v(x_1, x_2, \ldots)$ recursively:

If v is a leaf, $F_v(X) = \pi(v)$.

If v is a $_$ node,

$F_v(X) = (1 - \sum_{l=1}^{h} p_{(v,v_l)}) + \sum_{l=1}^{h} p_{(v,v_l)} F_{v_l}(X)$

If v is a $\wedge$ node, $F_v^i(X) = \prod_{l=1}^{h} F_{v_l}(X)$.

The generating function $F(X)$ for tree $\mathcal{T}$ is the one defined above for the root. It is easy to see, if we have a constant number of variables, the polynomial can be expanded in the form of $\sum_{i_1,i_2,\ldots} c_{i_1,i_2\ldots} x_1^{i_1} x_2^{i_2} \ldots$ in polynomial time. Now recall that each possible world pw contains a subset of the leaves of $\mathcal{T}$ (as dictated by the $_$ and $\wedge$ nodes). The following theorem characterizes the relationship between the coefficients of F and the probabilities we are interested in.

Theorem 1 *The coefficient of the term $\prod_j x_j^{i_j}$ in $F(X)$ is the total probability of the possible worlds for which, for all j, there are exactly i_j leaves associated with variable x_j.*

See Appendix A for the proof. We first provide two simple examples to show how to use Theorem 1 to compute the

We can use a similar idea to speed up the computation if the tuples are correlated and the correlations are represented using an and/xor tree. Let T_i be the and/xor tree where $\pi(t_j) = x$ for 1 $j < i$, $\pi(t_i) = y$ and $\pi(t_j) = 1$ for $j > i$. Suppose the generating function for T_i is $F^i(x, y) = \sum_j c_j^0 x^j + (\sum_j c_j x^{j-1})y$ and $\Upsilon(t_i) = \sum_{j=1}^n \alpha^j c_j$. We observe an intriguing relationship between the PRF^e value and the generating function:

$$\Upsilon(t_i) = \sum_j c_j \alpha^j = \Big(\sum_j c_j^0 \alpha^j + (\sum_j c_j \alpha^{j-1})\alpha\Big) \quad \sum_j c_j^0 \alpha^j$$
$$= F^i(\alpha, \alpha) \quad F^i(\alpha, 0).$$

Given this, $\Upsilon(t_i)$ can be computed in linear time by bottom up evaluation of $F^i(\alpha, \alpha)$ and $F^i(\alpha, 0)$ in T^i. If we simply repeat it n times, once for each t_i, this gives us a $O(n^2)$ total running time.

By carefully sharing the intermediate results among computations of $\Upsilon(t_i)$, we can improve the running time to $O(n \log n + nd)$ where d is the height of the and/xor tree. This improved algorithm runs in iterations. Suppose the tuples are already pre-sorted by their scores. Initially, the label of all leaves, i.e., $\pi(t_i)$, is 1. In iteration i, we change the label of leaf t_{i-1} from y to x and the label of t_i from 1 to y. The algorithm maintains the following information in each inner node v: the numerical values of $F_v^i(\alpha, \alpha)$ and $F_v^i(\alpha, 0)$. The values on node v need to be updated when the value of one of its children changes. Therefore, in each iteration, the computation only happens on the two paths, one from t_{i-1} to the root and one from t_i to the root. Since we update at most $O(d)$ nodes for each iteration, the running time is $O(nd)$. Suppose we want to update the information on the path from t_{i-1} to the root. We first update the $F_v^i(.,.)$ values for the leaf t_{i-1}. Since $F^i_{t_{i-1}} = \pi(t_{i-1}) = x$, we have $F^i_{t_{i-1}}(\alpha, \alpha) = \alpha$ and $F^i_{t_{i-1}}(\alpha, 0) = \alpha$. We assume v's child, say u, just had its values changed. The updating rule for $F_v^i(.,.)$(both $F_v^i(\alpha, \alpha)$ and $F_v^i(\alpha, 0)$) in node v is as follows.

1. v is a $\wedge$ node, $F_v^i(.,.) \quad F_v^{i-1}(.,.)F_u^i(.,.)/F_u^{i-1}(.,.)$
2. v is a $\vee$ node, then:

   $F_v^i(.,.) \quad F_v^{i-1}(.,.) + p_{(v,u)}F_u^i(.,.) \quad p_{(v,u)}F_u^{i-1}(.,.)$

The values on other nodes are not affected. The updating rule for the path from t_i to the root is the same except that for the leaf t_i, we have $F_{t_i}^i(\alpha, \alpha) = \alpha$ and $F_{t_i}^i(\alpha, 0) = 0$ since $F_{t_i}^i(x, y) = \pi(t_i) = y$. See Algorithm 3 for the psuedocode.

We note that, for the case of *x-tuples*, which can be represented using a two-level tree, this gives us an $O(n \log n)$ algorithm for ranking according to PRF^e.

Algorithm 3: ANDXOR-PRF^e-RANK(T)

```
F_{t_i}( ; ) = 1; F_{t_i}( ;0) = 1; ∀i
for i = 1 to n do
    if i ≠ 1 then
        F_{t_{i-1}}( ; ) = ; F_{t_{i-1}}( ;0) =
        UPDATE(T ; t_{i-1})
    F_{t_i}( ; ) = ; F_{t_i}( ;0) = 0
    UPDATE(T ; t_i)
    (t_i) = F_r( ; )   F_r( ;0)
return k tuples with largest   values
Subroutine: UPDATE(T ; v)
while v is not the root do
    u   v
    v   parent(v)
    if v is a ∧ node then
        F_v(:;:)   F_v(:;:)F_u^i(:;:)=F_u(:;:)
    if v is a ∨ node then
        F_v(:;:)   F_v(:;:) + p_{(v,u)}F_u(:;:)   p_{(v,u)}F_u(:;:)
```

4.4 Attribute Uncertainty or Uncertain Scores

We briefly describe how we can do ranking over tuples with discrete attribute uncertainty where the uncertain attributes are part of the tuple scoring function (if the uncertain attributes do not affect the tuple score, then they can be ignored for the ranking purposes). More generally, this approach can handle the case when there is a discrete probability distribution over the score of the tuple.

Assume $\sum_j p_{i,j}$ 1 for all i. The score score_i of tuple t_i takes value $v_{i,j}$ with probability $p_{i,j}$ and t_i does not appear in the database with probability 1 $\sum_j p_{i,j}$. It is easy to see the PRF value of t_i is

$$\Upsilon(t_i) = \sum_{k>0} \omega(t_i, k)\, \Pr(r(t_i) = k)$$
$$= \sum_{k>0} \omega(t_i, k) \sum_j \Pr(r(t_i) = k \wedge \text{score}_i = v_{i,j})$$
$$= \sum_j \Big(\sum_{k>0} \omega(t_i, k)\, \Pr(r(t_i) = k \wedge \text{score}_i = v_{i,j})\Big)$$

The algorithm works by treating the alternatives of the tuples (with a separate alternative for each different possible score for the tuple) as different tuples. In other words, we create a new tuple $t_{i,j}$ for each $v_{i,j}$ value. $t_{i,j}$ has existence probability $p_{i,j}$. Then, we add an *xor* constraint over the alternatives $\{t_{i,j}\}_j$ of each tuple t_i. We can then use the algorithm for the probabilistic and/xor tree model to find the values of the PRF function for each $t_{i,j}$ separately. Note that $\Pr(r(t_i) = k \wedge \text{score}_i = v_{i,j})$ is exactly the probability that $r(t_{i,j}) = k$ in the and/xor tree. Thus, by the above equation, we have that $\Upsilon(t_{i,j}) = \sum_{k>0} \omega(t_i, k)\, \Pr(r(t_i) = k \wedge \text{score}_i = v_{i,j})$ and $\Upsilon(t_i) = \sum_j \Upsilon(t_{i,j})$. Therefore, in a final step, we calculate the Υ score for each original tuple t_i by adding the Υ scores of its alternatives $\{t_{i,j}\}_j$. If the original tuples were independent, the complexity of this algorithm is $O(n^2)$ for

	PRF	PRF$^\omega$(h)	PRF^e
Independent tuples	O(n^2)	O(nh + n log n)	O(n log n)
And/Xor tree (height=d)	O(n^3) or O(n^2(log n)^{2}d)	O(n^3) or O(n^2(log n)^{2}d)	O(nd + n log n)
And/Xor tree	O(n^3)	O(n^3)	O($\sum_i$ d$_i$ + n log n)

Table 3 Summary of the running times. n is the number of tuples. d$_i$ is the depth of tuple t$_i$ in the and/xor tree.

computing the PRF function, and $O(n \log n)$ for computing the PRF^e function where n is the size of the input, i.e., the total number of different possible scores.

4.5 Summary

We summarize the complexities of the algorithms for different models in Table 3. Now, we explain some entries in the table which has not been discussed. The first is the PRF computation over an and/xor tree with height d. We have two choices here. One is just to use the algorithm for arbitrary and/xor trees, i.e., to use the algorithm in Appendix B to expand $F^i(x, y)$ for each i, which runs in $O(n^2)$ time. The overall running time is $O(n^3)$. The other one is to use the divide-and-conquer algorithm in Appendix B to expand the polynomial for each ^ node in $F^i(x, y)$. We can easily see that expanding nodes for each level of the tree requires at most $O(n(\log n)^2)$ time. Therefore, the running time for expanding $F^i(x, y)$ is at most $O(n(\log n)^2)d)$ and the overall running time is $O(n^2(\log n)^2 d)$ which is much better than $O(n^3)$ if d n. For PRF$^\omega(h)$ computation over and/xor trees, we do not know how to achieve a better bound as in the tuple-independent datasets. We leave it as an interesting open problem.

For PRF^e computation on and/xor trees, we use ANDXOR-PRF^e-RANK. Now, the procedure UPDATE(T , t_i) runs in $O(d_i)$ time where d_i is the depth of tuple t_i in the and/xor tree, i.e., the length of path from the root to t_i. Therefore, the total running time is $O(\sum_i d_i + n \log n)$. If the height of the and/xor tree is bounded by d, the running time is simply $O(nd + n \log n)$.

5 Approximating and Learning Ranking Functions

In this section, we discuss how to choose the PRF functions and their parameters. Depending on the application domain and the scenarios, there are two approaches to this:

If we know the ranking function we would like to use (say PT(h)), then we can either simulate or approximate it using appropriate PRF functions.

If we are instead provided user preferences data, we can learn the parameters from them.

Clearly, we would prefer to use a PRF^e function, if possible, since it admits highly efficient ranking algorithms. For this purpose, we begin with presenting an algorithm to find an approximation to an arbitrary PRF$^\omega$ function using a linear combination of PRF^e functions. We then discuss how to learn a PRF$^\omega$ function from user preferences, and finally present an algorithm for learning a single PRF^e function.

5.1 Approximating PRF$^\omega$ using PRF^e Functions

A linear combination of complex exponential functions is known to be very expressive, and can approximate many other functions very well [6]. Specifically, given a PRF$^\omega$ function, if we can write $\omega(i)$ as: $\omega(i)$ $\sum_{l=1}^{L} u_l \alpha_l^i$, then we have that:

$$\Upsilon(t) = \sum_i \omega(i) \Pr(r(t) = i) \quad \sum_{l=1}^{L} u_l \left(\sum_i \alpha_l^i \Pr(r(t) = i) \right)$$

This reduces the computation of $\Upsilon(t)$ to L individual PRF^e function computations, each of which only takes linear time. This gives us an $O(n \log n + nL)$ time algorithm for approximately ranking using PRF$^\omega$ function for independent tuples (as opposed to $O(n^2)$ for exact ranking).

Several techniques have been proposed for finding such approximations using complex exponentials [6, 25]. Those techniques are however computationally inefficient, involving computation of the inverses of large matrices and the roots of polynomials of high orders.

In this section, we present a clean and efficient algorithm, based on Discrete Fourier Transforms (DFT), for approximating a function $\omega(i)$, that approaches zero for large values of i (in other words, $\omega(i)$ $\omega(i+1)8i, \omega(i) = 0, i > h$). As we noted earlier, this captures the typical behavior of the $\omega(i)$ function. An example of such a function is the step function ($\omega(i) = 18i$ $h, = 08i > h$) which corresponds to the ranking function PT(h). At a high level, our algorithm starts with a DFT approximation of $\omega(i)$ and then adapts it by adding several damping, scaling and shifting factors.

Discrete Fourier transformation (DFT) is a well known technique for representing a function as a linear combination of complex exponentials (also called *frequency domain representation*). More specifically, a discrete function $\omega(i)$ defined on a finite domain $[0, N$ 1] can be decomposed into exactly N exponentials as:

$$\omega(i) = \frac{1}{N} \sum_{k=0}^{N} \psi(k) e^{\frac{2\pi \jmath}{N} ki} \qquad i = 0, \ldots, N \quad 1. \tag{4}$$

where $\jmath$ is the imaginary unit and $\psi(0),$ $, \psi(N$ 1) denotes the DFT transform of $\omega(0),$ $, \omega(N$ 1). If we want to approximate ω by fewer, say L, exponentials, we

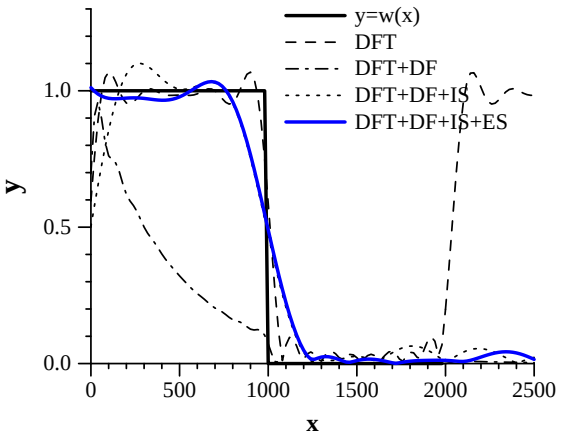

Fig. 4 Illustrating the effect of the approximation steps: w(i) = step function with N = 1000, L = 20

can instead use the L DFT coefficients with maximum absolute value. Assume that $\psi(0), \ldots, \psi(L \quad 1)$ are those coefficients. Then our approximation ϖ_L^{DFT} of ω by L exponentials is given by:

$$\varpi_L^{DFT}(i) = \frac{1}{N} \sum_{k=0}^{L \quad 1} \psi(k) e^{\frac{2\pi \jmath}{N} ki} \qquad i = 0, \ldots, N \quad 1. \quad (5)$$

However, DFT utilizes only complex exponentials of unit norm, i.e., $e^{\jmath r}$ (where r is a real), which makes this approximation periodic (with a period of N). This is not suitable for approximating an ω function used in PRF, which is typically a monotonically non-increasing function. If we make N sufficiently large, say larger than the total number of tuples, then we usually need a large number of exponentials (L) to get a reasonable approximation. Moreover, computing DFT for very large N is computationally non-trivial. Furthermore, the number of tuples n may not be known in advance.

We next present a set of nontrivial tricks to adapt the base DFT approximation to overcome these shortcomings. We assume $\omega(i)$ takes non-zero values within interval $[0, N \quad 1]$ and the absolute values of both $\omega(i)$ and $\omega_L^{DFT}(i)$ are bounded by B. To illustrate our method, we use the step function:

$$\omega(i) = \begin{cases} 1, i < N \\ 0, i \quad N \end{cases}$$

with N = 1000 as our running example to show our method and the specific shortcomings it addresses. Figure 4 illustrates the effect of each of these adaptations.

1. **(DFT)** We perform pure DFT on the domain $[1, aN]$, where a is a small integer constant (typically < 10). As we can see in Figure 4 (where N = 1000 and a = 2), this results in a periodic approximation with a period of 2000. Although the approximation is reasonable for $x < 2000$, the periodicity is unacceptable if the number of tuples is larger than 2000 (since the positions between 2000 and 3000 (similarly, between 4000 and 5000) would be given high weights).

2. **(Damping Factor (DF))** To address this issue, we introduce a damping factor η 1 such that $B\eta^{aN}$ ϵ where ϵ is a small positive real (for example, 10 ^{5}). Our new approximation becomes:

$$\varpi_L^{DFT+DF}(i) = \eta^i \ \varpi_L^{DFT}(i) = \frac{1}{N} \sum_{k=0}^{L \quad 1} \psi(k)(\eta e^{\frac{2\pi \jmath}{N} k})^i \quad (6)$$

By incorporating this damping factor, the periodicity is mitigated, since we have: $\lim_{i \; ! \; +1} \varpi_L^{DFT+DF}(i) = 0$. Especially, $\varpi_L^{DFT+DF}(i)$ ϵ for $i > \alpha N$.

3. **(Initial Scaling (IS))** However the use of damping factor introduces another problem: it gives a biased approximation when i is small (see Figure 4). Taking the step function as an example, $\varpi_L^{DFT+DF}(i)$ is approximately η^i for 0 $i < N$ instead of 1. To rectify this, we initially perform DFT on a different sequence $\hat{\omega}(i) = \eta^{\ i}\omega(i)$ (rather than $\omega(i)$) on domain 2 $[0, aN]$. Therefore, ϖ^{DFT+IS} is a reasonable approximation of $\hat{\omega}$. Then, if we apply the damping factor, it will give us an unbiased approximation of ω, which we denote by $\varpi^{DFT+DF+IS}$.

4. **(Extending and Shifting (ES))** This step is in particular tailored for optimizing the approximation performance for ranking functions. DFT does not perform well at discontinuous points, specifically at i = 0 (the left boundary), which can significantly affect the ranking approximation. To handle this, we extrapolate ω to make it continuous around 0. Let the resulting function be ω which is defined on $[\ bN, +1]$ for small $b > 0$. Again, taking the step function for example, we let $\omega(i) = \begin{cases} 1, & bN \quad i < N; \\ 0, i & N. \end{cases}$ Then, we shift $\omega(i)$ rightwards by bN to make its domain lie entirely in positive axis, do initial scaling and perform DFT on the resulting sequence. We denote the approximation of the resulting sequence by $\varpi^0(i)$(by performing (6)). For the approximation of original $\omega(i)$ values, we only need to do corresponding leftward shifting , namely $\varpi^{DFT+DF+IS+ES}(i) = \varpi^0(i + bN)$. Figure 4 shows that DFT+DF+IS+ES gives a much better approximation than others around i = 0.

Figures 4 and 5(i) illustrate the efficacy of our approximation technique for the step function. As we can see, we are able to approximate that function very well with just 20 or

Fig. 5 Approximating functions using linear combinations of complex exponentials: effect of increasing the number of coef cients

30 coef cients. Figure 5(ii) and (iii) show the approximations for a piecewise linear function and an arbitrarily generated continuous function respectively, both of which are much easier to approximate than the step function.

5.2 Learning a PRF^l or PRF^e Function

Next we address the question of how to learn the weights of a PRF^l function or the for a single PRF^e function from user preferences. To learn a linear combination of PRF^e functions, we rst learn a PRF^l function and approximate it as above.

Prior work on learning ranking functions (e.g., [8, 15, 26, 33]) assumes that the user preferences are provided in the form of a set of pairs of tuples, and for each pair, we are told which tuple is ranked higher. Our problem differs slightly from this prior work in that, the features that we use to rank the tuples (i.e., $Pr(r(t) = i); i = 1; : : : ; n$) cannot be computed for each tuple individually, but must be computed for the entire dataset (since the values of the features for a tuple depend on the other tuples in the dataset). Hence we assume that we are instead given a small sample of the tuples, and the user ranking for all those tuples. We compute the features assuming this sample constitutes the entire relation, and learn a ranking function accordingly, with the goal to nd the parameters (the weights w_i for PRF^l or the parameter for PRF^e) that minimize the number of disagreements with the provided ranking over the samples.

Given this, the problem of learning PRF^l is identical to the problem addressed in the prior work, and we utilize the algorithm based on support vector machines (SVM) [33] in our experiments.

On the other hand, we are not aware of any work that has addressed learning a ranking function like PRF^e. We use a simple binary search-like heuristic to nd the optimal real value of that minimizes the Kendall distance between the user-speci ed ranking and the ranking according to PRF^e( ). In other words, we try to nd $\arg\min_{\in[0;1]}(dis(; ( )))$ where $dis()$ is the Kendall distance between two rankings, is the ranking for the given sample and ( ) is the one obtained by using PRF^e( ) function. Suppose we want to nd the optimal within the interval $[L; U]$ now. We rst compute $dis(; (L + i \frac{U-L}{10}))$ for $i = 1; : : : ; 9$ and nd i for which the distance is the smallest. Then we reduce our search range to $[\max(L; L + (i-1)\frac{U-L}{10}); \min(U; L + (i+1)\frac{U-L}{10})]$ and repeat the above recursively. Although this algorithm can only converge to a local minimum, in our experimental study, we observed that all of the prior ranking functions exhibit a uni-valley behavior (Section 8), and in such cases, this algorithm nds the global optimal.

6 PRF as a Consensus Top-k Answer

In this section, we show there is a close connection between PRF^l and the notion of consensus top-k answer (Con-Top$k$) proposed in [41]. We rst review the de nition of a consensus top-k ranking.

De nition 4 Let $dis()$ denote a distance function between two top-k rankings. Then the most consensus answer is de ned to be the top-k ranking for which the expected distance to the answer $_{pw}$ of the (random) world pw is minimized, i.e.,

$$= \arg\min_{\in} \{E[dis(; _{pw})]\}:$$

$dis()$ may be any distance function de ned on pairs of top-k answers. In [41], we discussed how to compute or approximate Con-Top$k$ under a number of distance functions, such as Spearman's rho, Kendall's tau and intersection metric [19].

Example 7 Consider the example in Figure 1. Assume $k = 2$ and the distance function is the symmetric difference metric $dis = |(_1 \cup _2) - (_2 \cap _1)|$. The most consensus top-2 answer is $\{t_2; t_5\}$ and the expected distance is $E[dis(; _{pw})] = :112 \cdot 2 + :168 \cdot 2 + :048 \cdot 4 + :072 \cdot 4 + :168 \cdot 2 + :252 \cdot 0 + :072 \cdot 4 + :108 \cdot 2$.

We rst show that a Con-Top$k$ answer under symmetric difference is equivalent to PT(h)(k), a special case of PRF^l. Then, we generalize the result and show that any

lations, and by ignoring the correlations, and then compute the Kendall distance between these two (e.g., for PRF^e, we compute the rankings using **PROB-ANDOR-PRF-RANK** and **IND-PRF-RANK** algorithms). Figure 7(iii) shows the results for the PRF^e ranking function for varying α, whereas in Figure 7(iv), we plot the results for PRF^e($\alpha = 0.9$), PT(100), and U-Rank. As we can see, on highly correlated datasets, ignoring the correlations can result in significantly inaccurate top-k answers. This is not as pronounced for the Syn-XOR dataset. This is because, in any group of tuples that are mutually exclusive, there are typically only a few tuples that may have sufficiently high probabilities to be part of the top-k answer; the rest of the tuples may be ignored for ranking purposes. Because of this, assuming tuples to be independent of each other does not result in significant errors. As α approaches 1, PRF^e tends to sort the tuples by probabilities, so all four curves in Figure 7(iii) become close to 0. Ranking by E-Score is invariant to the correlations, which is a significant drawback of that function.

8.4 Execution Times

Figure 8(i) shows the execution times for four ranking functions: PRF^e, PT(h), U-Rank and E-Rank, for the IIP-datasets, for different dataset sizes and k. We note that the running time for PRF$^\omega$ is similar to that of PT(h). As expected, ranking by PRF^e or E-Rank is very efficient (1000000 tuples can be ranked within 1 or 2 seconds). Indeed, after sorting the dataset in an non-decreasing score order, PRF^e needs only a single scan of the dataset, and E-Rank needs to scan the dataset twice. Execution times for $PT(h)$ and U-Rank-k increase linearly with h and k respectively and the algorithms become very slow for high h and k. The running times of both PRF^e and E-Rank are not significantly affected by k.

Figure 8(ii) compares the execution time for PT(h) and its approximation using a linear combination of PRF^e functions (see Figure 6(iii)), for two different values of k. The notation $w50$ indicates that 50 exponentials were used in the approximation (note that the approximate ranking, based on PRF^e, is insensitive to the value of k). As we can see, for large datasets and for higher values of k, exact computation takes several orders of magnitude more time to compute than the approximation. For example, the exact algorithm takes nearly 1 hour for $n = 500,000$ and $h = 10,000$ while the approximate answer obtained using $L = 50$ PRF^e functions takes only 24 seconds and achieves a Kendall distance 0.09.

For correlated datasets, the effect is even more pronounced. In Figure 8(iii), we plot the results of a similar experiment, but using two correlated datasets: Syn-XOR and Syn-HIGH. Note that the number of tuples in these datasets is smaller by a factor of 10. As we can see, our generating functions-based algorithms for computing PRF^e are highly efficient, even for datasets with high degrees of correlation. As above, approximation of the PT(h) ranking function using a linear combination of PRF^e functions is significantly cheaper to compute than using the exact algorithm.

Combined with the previous results illustrating that a linear combination of PRF^e functions can approximate other ranking functions very well, this validates the unified ranking approach that we propose in this article.

9 PRF Computation for Arbitrary Correlations

Among many models for capturing the correlations in a probabilistic database, graphical models (Markov or Bayesian networks) perhaps represent the most systematic approach [48]. The appeal of graphical models stems both from the pictorial representation of the dependencies, and a rich literature on doing inference over them. In this section, we present an algorithm for computing the PRF function values for all tuples of a correlated dataset when the correlations are represented using a graphical model. The resulting algorithm is a non-trivial dynamic program over the *junction tree* of the graphical model. Our main result is that we can compute the PRF function in polynomial time if the junction tree of the graphical model has bounded treewidth. Further, although this exact algorithm may not be feasible for large datasets or for high-treewidth graphical models (in which case approximations should be pursued), it is quite practical for small-treewidth graphical models (e.g., it runs in $O(n^3)$ time over a Markov sequence [34, 36, 46]). It is worth noting that this result can not subsume our algorithm for and/xor trees (Section 4.2) since the treewidth of the moralized graph of a probabilistic and/xor tree may not be bounded. In some sense, this is close to *instance-optimal* since the complexity of the underlying inference problem is itself exponential in the treewidth of the graphical model (this however does not preclude the possibility that the ranking itself could be done more efficiently without computing the PRF function explicitly – however, such an algorithm is unlikely to exist).

9.1 Definitions

We start with briefly reviewing some notations and definitions related to graphical models and junction trees. Let $T = \{t_1, t_2, \ldots, t_n\}$ be the set of tuples in D_T, sorted in an non-increasing order of their score values. For each tuple t in T, we associate an indicator random variable X_t, which is 1 if t is present, and 0 otherwise. Let $\mathbf{X} = \{X_{t_1}, \ldots, X_{t_n}\}$ and $\mathbf{X}_i = \{X_{t_1}, \ldots, X_{t_i}\}$. For a set of variables S, we use $\Pr(S)$ to denote the joint probability distribution over those variables. So $\Pr(\mathbf{X})$ denotes the joint probability distribution that we are trying to reason about. This joint distribution captures all the correlations in the dataset. However, directly

trying to represent it would take $O(2^n)$ space, and hence is clearly infeasible.

Probabilistic graphical models allow us to represent this joint distribution compactly by exploiting the conditional independences present among the variables. Given three disjoint sets of random variables A, B, C, we say that A is conditionally independent of B *given* C if and only if:

$$\Pr(A, B|C) = \Pr(A|C)\Pr(B|C)$$

We assume that we are provided with a *junction tree* over the variables X that captures the correlations among them. A junction tree can be constructed from a graphical model using standard algorithms [31]. Recently junction trees have also been used as a internal representation for probabilistic databases, and have been shown to be quite effective at handling lightly correlated probabilistic databases [35]. We describe the key properties of junction trees next.

Junction tree: Let T be a tree with each node v associated with a subset C_v X. We say T is a *junction tree* if any intersection $C_u \setminus C_v$ for any u, v 2 T is contained in C_w for every node w on the unique path between u and v in T (this is called the *running intersection property*). The treewidth tw of a junction tree is defined to be $\max_{v2T}$ jC_vj 1.

Denote $S_{u,v} = C_v \setminus C_u$ for each edge (u, v) 2 T. We call $S_{u,v}$ a *separator* since removal of $S_{u,v}$ disconnects the graphical model. The set of conditional independences embodied by a junction tree can be found using the Markov property:

(Markov Property) Given variable sets A, B, C, if C separates A and B (i.e., removal of variables in C disconnects the variables in A from variables in B in the junction tree), then A is conditionally independent of B given C.

Example 8 Let $T = \{t_1, t_2, t_3, t_4, t_5\}$. Figure 9 (i) and (ii) show the (undirected) graphical model and the corresponding junction tree T. T has four nodes: $C_1 = \{X_{t_4}, X_{t_5}\}$, $C_2 = \{X_{t_4}, X_{t_3}\}$, $C_3 = \{X_{t_3}, X_{t_1}\}$ and $C_4 = \{X_{t_3}, X_{t_2}\}$. The treewidth of T is 1. We have, $S_{1,2} = \{X_4\}$, $S_{2,3} = \{X_3\}$ and $S_{2,4} = \{X_3\}$. Using the Markov property, we observe that X_5 is independent of X_1, X_2, X_3 given X_4.

Clique and Separator Potentials: With each clique C_v in the junction tree, we associate a *potential* $\pi_v(C_v)$, which is a function over all variables X_{t_i} 2 C_v and captures the correlations among those variables. Similarly, with each separator $S_{u,v}$, we associate a *potential* $\mu_{u,v}(S_{u,v})$. Without loss of generality, we assume that the potentials are *calibrated*, i.e., the potential corresponding to a clique (or a separator) is exactly the joint probability distribution over the variables in that clique (separator). Given a junction tree with arbitrary potentials, calibrated potentials can be computed using a standard *message passing algorithm* [31]. The complexity

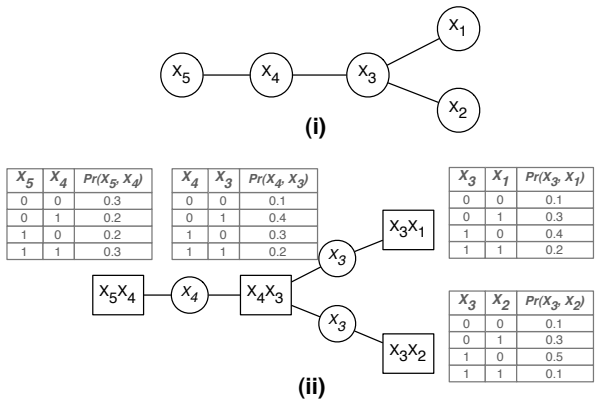

Fig. 9 (i) A graphical model; (ii) A junction tree for the model along with the (calibrated) potentials.

of this algorithm is $O(n2^{tw})$. Then the joint probability distribution of X, whose correlations can be captured using a calibrated junction tree T, can be written as:

$$\Pr(X) = \frac{\prod_{v2T}\ \pi_v(C_v)}{\prod_{(u,v)2T}\ \mu_{u,v}(S_{u,v})} = \frac{\prod_{v2T}\ \Pr(C_v)}{\prod_{(u,v)2T}\ \Pr(S_{u,v})}$$

9.2 Problem Simplification

We begin with describing the first step of our algorithm, and defining a reduced and simpler to state problem.

Recall that our goal is to rank the tuples according to $\Upsilon(t_i) = \sum_{j>0} \omega(j) \Pr(r(t_i) = j)$. For this purpose, we first compute the positional probabilities, $\Pr(r(t_i) = j)$ $8j$ $8t_i$, using the algorithms presented in the next two subsections. Given those, the values of $\Upsilon(t_i)$ can be computed in $O(n^2)$ time for all tuples, and the ranking itself can be done in $O(n \log n)$ time (by sorting). The positional probabilities ($Pr(r(t_i) = j)$) may also be of interest by themselves.

For each tuple t_i, we compute $\Pr(r(t_i) = j)$ $8j$ at once. Recall that $\Pr(r(t_i) = j)$ is the probability that t_i exists (i.e., $X_i = 1$) and exactly j 1 tuples with scores higher than t_i are present (i.e., $\sum_{l=1}^{i\ 1} X_l = j$ 1). In other words:

$$\Pr(r(t_i) = j) = \Pr(X_i = 1 \wedge \sum_{l=1}^{i\ 1} X_l = j \quad 1)$$

$$= \Pr((\sum_{l=1}^{i\ 1} X_l = j \quad 1) | X_i = 1) \Pr(X_i = 1)$$

Hence, we begin with first conditioning the junction tree by setting $X_i = 1$, and re-calibrating. This is done by identifying all cliques and separators which contain X_i, and by updating the corresponding probability distributions by removing the values corresponding to $X_i = 0$. More precisely, we replace a probability distribution $\Pr(X_{i_1}, \ldots, X_{i_k}, X_i)$,

by a potential $(X_{i_1}, \ldots, X_{i_k})$ computed as:

$$(X_{i_1} = v_1, \ldots, X_{i_k} = v_k) = \Pr(X_{i_1} = v_1, \ldots, X_{i_k} = v_k, X_i = 1)$$

is not a probability distribution since the entries in it may not sum up to 1. Further, the potentials may not be consistent with each other. Hence, we need to recalibrate this junction tree using message passing [31]. As mentioned earlier, this takes $O(n2^{tw})$ time. Figure 10 shows the resulting (uncalibrated) junction tree after conditioning on $X_3 = 1$.

Fig. 10 Conditioning on $X_5 = 1$ results in a smaller junction tree, with uncalibrated potentials, that captures the distribution over X_1, X_2, X_3, X_4 given $X_5 = 1$.

If X_i is a separator in the junction tree, then we get more than one junction tree after conditioning on $X_i = 1$. Figure 11 shows the two junction trees we would get after conditioning on $X_4 = 1$. The variables in these junction trees are independent of each other (this follows from the Markov property), and the junction trees can be processed separately from each other.

Since the resulting junction tree or junction trees capture the probability distribution conditioned on the event $X_i = 1$, our problem now reduces to nding the probability distribution of $P = \sum_{l=1}^{i-1} X_l$ in those junction trees. For cleaner description of the algorithm, we associate an indicator variable X_l with each variable X_l in the junction tree. X_l is set to 1 if $l \leq i-1$, and is 0 otherwise. This allows us to state the key problem to be solved as follows:

Rede ned Problem[8]: Given a junction tree over m binary variables $Y_1, \ldots, Y_m$, where each variable Y_j is associated with an indicator variable $Y_j \in \{0, 1\}$, nd the probability distribution of the random variable $P = \sum_{l=1}^{m} Y_l$.

If the result of the conditioning was a single junction tree (over $m = n-1$ variables), we multiply the resulting probabilities by $\Pr(X_i = 1)$ to get the rank distribution of t_i.

However, if we get $k > 1$ junction trees, then we need one additional step. Let $P_1, \ldots, P_k$ be the random variables denoting the partial sums for each of junction trees. We need to combine the probability distributions over these partial sums, $\Pr(P_1), \ldots, \Pr(P_k)$, into a single probability distribution over $\Pr(P_1 + \cdots + P_k)$. This can be done by repeatedly applying the following general formula:

$$\Pr(P_1 + P_2 = a) = \sum_{j=0}^{a} \Pr(P_1 = j) \Pr(P_2 = a - j)$$

A naive implementation of the above takes time $O(n^2)$. Although this can be improved using the ideas presented in Appendix B, the complexity of computing $\Pr(P_i)$ is much higher and dominates the overall complexity.

Fig. 11 Conditioning on $X_4 = 1$ results in two junction trees.

9.3 Algorithm for Markov Sequences

We rst describe an algorithm for Markov chains, a special, yet important, case of the graphical models. Markov chains appear naturally in many settings, and have been studied in probabilistic database literature as well [34, 36, 46]. Any nite-length Markov chain is a Markov network whose underlying graph is simply a path: each variable is directly dependent on only its predecessor and successor. The junction tree for a Markov chain is also a path in which each node corresponds to an edge of the Markov chain. The treewidth of such a junction tree is one. Without loss of generality, we assume that the Markov chain is $Y_1, \ldots, Y_m$ (Figure 12(i)). The corresponding junction tree T is a path with cliques $C_j = \{Y_j, Y_{j+1}\}$ as shown in the gure.

We compute the distribution $\Pr(\sum_{l=1}^{m} Y_l)$ recursively. Let $P_j = \sum_{l=1}^{j} Y_l$ denote the partial sum over the rst j variables $Y_1, \ldots, Y_j$.

At the clique $\{Y_{j-1}, Y_j\}$, $j \geq 1$, we recursively compute the joint probability distribution $\Pr(Y_j, P_{j-1})$. The initial distribution $\Pr(Y_2, P_1)$, $P_1 = Y_1$, is computed directly:

$$\Pr(Y_2, P_1 = 0) = \Pr(Y_2, Y_1 = 0) + (1 - \alpha_i) \Pr(Y_2, Y_1 = 1)$$
$$\Pr(Y_2, P_1 = 1) = \alpha_i \Pr(Y_2, Y_1 = 1).$$

Given $\Pr(Y_j, P_{j-1})$, we compute $\Pr(Y_{j+1}, P_j)$ as follows. Observe that P_{j-1} and Y_{j+1} are conditionally independent given the value of Y_j (by Markov property). Thus we have:

$$\Pr(Y_{j+1}, Y_j, P_{j-1}) = \frac{\Pr(Y_{j+1}, Y_j) \Pr(Y_j, P_{j-1})}{\Pr(Y_j)}$$

Using $\Pr(Y_{j+1}, Y_j, P_{j-1})$, we can compute:

$$\Pr(Y_{j+1}, P_j = a) = \Pr(Y_{j+1}, Y_j = 0, P_{j-1} = a) + \Pr(Y_{j+1}, Y_j = 1, P_{j-1} = a - \alpha_j)$$

[8] We rename the variables to avoid confusion.

$O(n)$ such probability distributions, and each of those computations takes $O(n^2 2^{|C|})$ time. Since there are at most n cliques, and since we have to repeat this process for every tuple, the overall complexity of ranking the dataset can be seen to be: $O(n^4 2^{tw})$, where tw denotes the treewidth of the junction tree, i.e., the size of the maximum clique - 1.

10 Conclusions

In this article we presented a unified framework for ranking over probabilistic databases, and presented several novel and highly efficient algorithms for answering top-k queries. Considering the complex interplay between probabilities and scores, instead of proposing a specific ranking function, we propose using two parameterized ranking functions, called PRF^ω and PRF^e, which allow the user to control the tuples that appear in the top-k answers. We developed novel algorithms for evaluating these ranking functions over large, possibly correlated, probabilistic datasets. We also developed an approach for approximating a ranking function using a linear combination of PRF^e functions thus enabling highly efficient, albeit approximate computation, and also for learning a ranking function from user preferences.

Our work opens up many avenues for further research. There may be other non-trivial subclasses of PRF functions, aside from PRF^e, that can be computed efficiently. Understanding the behavior of various ranking functions and their relationships across probabilistic databases with diverse uncertainties and correlation structures also remains an important open problem in this area. Finally, the issues of ranking have been studied for many years in disciplines ranging from economics to information retrieval; better understanding the connections between that work and ranking in probabilistic databases remains a fruitful direction for further research.

References

1. E. Adar and C. Re. Managing uncertainty in social networks. *IEEE Data Eng. Bull.*, 2007.
2. P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases. In *ICDE*, 2006.
3. Y. Azar, I. Gamzu, and X. Yin. Multiple intents reranking. In *STOC*, pages 669–678, 2009.
4. N. Bansal, K. Jain, A. Kazeykina, and J. Naor. Approximation Algorithms for Diversified Search Ranking. *ICALP*, pages 273–284, 2010.
5. G. Beskales, M. Soliman, and I. Ilyas. Efficient search for the top-k probable nearest neighbors in uncertain databases. *VLDB*, 2008.
6. G. Beylkin and L. Monzon. On approximation of functions by exponential sums. *Applied and Computational Harmonic Analysis*, 19:17–48, 2005.
7. A. Bjorck and V. Pereyra. Solution of vandermonde systems of equations. *Mathematics of Computation*, 24(112):893–903, 1970.
8. C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *ICML*, pages 89–96, 2005.
9. R. Cheng, J. Chen, M. Mokbel, and C. Chow. Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data. In *ICDE*, 2008.
10. R. Cheng, L. Chen, J. Chen, and X. Xie. Evaluating probability threshold k-nearest-neighbor queries over uncertain data. In *EDBT*, 2009.
11. R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, 2003.
12. G. Cormode, F. Li, and K. Yi. Semantics of ranking queries for probabilistic data and expected ranks. In *ICDE*, 2009.
13. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.
14. N. Dalvi and D. Suciu. Management of probabilistic data: Foundations and challenges. In *PODS*, 2007.
15. O. Dekel, C. Manning, and Y. Singer. Log-linear models for label-ranking. In *NIPS 16*, 2004.
16. A. Deshpande, C. Guestrin, and S. Madden. Using probabilistic models for data management in acquisitional environments. In *CIDR*, 2005.
17. X. L. Dong, A. Halevy, and C. Yu. Data integration with uncertainty. In *VLDB*, 2007.
18. C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW*, 2001.
19. R. Fagin, R. Kumar, and D. Sivakumar. Comparing top-k lists. In *SODA*, 2003.
20. N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. on Info. Syst.*, 1997.
21. T. Ge, S. Zdonik, and S. Madden. Top-k queries on uncertain data: On score distribution and typical answers. In *SIGMOD*, pages 375–388, 2009.
22. T. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.
23. T. Green and V. Tannen. Models for incomplete and probabilistic information. In *EDBT*, 2006.
24. R. Gupta, S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, 2006.
25. J. F. Hauer, C. J. Demeure, and L. L. Scharf. Initial results in prony analysis of power system response signals. *IEEE Trans. on Power Systems*, 5(1):80–89, 1990.
26. R. Herbrich, T. Graepel, P. Bollmann-Sdorra, and K. Obermayer. Learning preference relations for information retrieval. In *ICML-98 Workshop: Text Categorization and Machine Learning*, page 8084, 1998.

27. M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: A probabilistic threshold approach. In *SIGMOD*, 2008.
28. I. Ilyas, G. Beskales, and M. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 2008.
29. K. Järvelin, J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 2002.
30. T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. *IEEE Data Eng. Bull.*, 29(1), 2006.
31. F. Jensen and F. Jensen. Optimal junction trees. In *UAI*, pages 360–366, 1994.
32. C. Jin, K. Yi, L. Chen, J. Xu Yu, X. Lin. Sliding-window top-k queries on uncertain streams. In *VLDB*, 2008.
33. T. Joachims. Optimizing search engines using clickthrough data. In *Proc. SIGKDD*, pages 133–142, 2002.
34. B. Kanagal and A. Deshpande. Efficient query evaluation over temporally correlated probabilistic streams. In *ICDE*, 2009.
35. B. Kanagal and A. Deshpande. Indexing correlated probabilistic databases. In *SIGMOD*, 2009.
36. B. Kimelfeld and C. Ré. Transducing markov sequences. In *PODS*, pages 15–26, 2010.
37. C. Koch. *MayBMS: A System for Managing Large Uncertain and Probabilistic Databases*. Managing and Mining Uncertain Data. Charu Aggarwal ed., 2009.
38. C. Koch and D. Olteanu. Conditioning probabilistic databases. *PVLDB*, 1(1):313–325, 2008.
39. H.P. Kriegel, P. Kunath, M. Renz. Probabilistic nearest-neighbor query on uncertain objects. In *DASFAA*, 2007.
40. L. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *TODS*, 1997.
41. J. Li and A. Deshpande. Consensus answers for queries over probabilistic databases. PODS, 2009.
42. J. Li and A. Deshpande. Ranking continuous probabilistic datasets. In *VLDB*, 2010.
43. T. Y. Liu. Learning to Rank for Information Retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.
44. X. Liu, M. Ye, J. Xu, Y. Tian, and W. Lee. k-selection query over uncertain data. In *DASFAA (1)*, pages 444–459, 2010.
45. C. Ré, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, 2007.
46. C. Ré, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *SIGMOD Conference*, 2008.
47. A. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, 2006.
48. P. Sen, A. Deshpande, and L. Getoor. PrDB: managing and exploiting rich correlations in probabilistic databases. *VLDB J.*, 18(5):1065–1090, 2009.
49. M. Soliman, I. Ilyas, and K. C. Chang. Top-k query processing in uncertain databases. In *ICDE*, 2007.
50. M. Soliman and I. Ilyas. Ranking with uncertain scores. In *ICDE*, pages 317–328, 2009.
51. P. Talukdar, M. Jacob, M. Mehmood, K. Crammer, Z. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. *PVLDB*, 1(1):785–796, 2008.
52. J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.
53. K. Yi, F. Li, D. Srivastava, G. Kollios. Efficient processing of top-k queries in uncertain databases. *ICDE*, 2008.
54. X. Zhang and J. Chomicki. On the semantics and evaluation of top-k queries in probabilistic databases. In *DBRank*, 2008.
55. O. Zuk, L. Ein-Dor, and E. Domany. Ranking under uncertainty. In *UAI*, pages 466–473, 2007.

A Proofs

Proof (**Theorem 1**) Suppose T is rooted at r, $r_1;:::;r_h$ are r's children, and T_l is the subtree rooted at r_l. We denote by S (or S_l) the random set of leaves generated according to model T (or T_l). We let F (or F_l) be the generating function corresponding to T (or T_l). For ease of notation, we use $\mathbf{i}$ to denote index vector $hi_1; i_2; :::i$, I to denote the set of all such $\mathbf{i}$s and $X^{\mathbf{i}}$ to denote $\prod_j x_j^{i_j}$. Therefore, we can write $F(X) = \sum_{i_1,i_2,...} c_{i_1,i_2...} x_1^{i_1} x_2^{i_2} ::: = \sum_{\mathbf{i} 2 I} c_{\mathbf{i}} X^{\mathbf{i}}$: We use the notation $S = \mathbf{i}$ for some $\mathbf{i} = hi_1; i_2; :::i\ 2$ I to denote the event that S contains i_j leaves associated with variable x_j for all j. Given the notations, we need to show $c_{\mathbf{i}} = Pr(S = \mathbf{i})$.

We shall prove by induction on the height of the and/xor tree. We consider two cases. If r is a ^ node, we know from Definition 2 that $S = \bigcup_{l=1}^{h} S_l$. First, it is not hard to see that given $S_l = \mathbf{i}_l$ for 1 l h, the event $S = \mathbf{i}$ happens if and only if $\sum_l \mathbf{i}_l = \mathbf{i}$. Therefore,

$$Pr(S = \mathbf{i}) = \sum_{\sum_l \mathbf{i}_l = \mathbf{i}} \prod_{l=1}^{h} Pr(S_l = \mathbf{i}_l): \tag{7}$$

Assume F_l can be written as $\sum_{\mathbf{i}_l} c_{l,\mathbf{i}_l} X^{\mathbf{i}_l}$. From the construction of the generating function, we know that

$$F(X) = \prod_{l=1}^{h} F_l = \prod_{l=1}^{h} \sum_{\mathbf{i}_l 2 I} c_{l,\mathbf{i}_l} X^{\mathbf{i}_l} = \sum_{\mathbf{i} 2 I} \Big(\sum_{\sum_l \mathbf{i}_l = \mathbf{i}} \prod_{l=1}^{h} c_{l,\mathbf{i}_l} X^{\mathbf{i}_l} \Big)$$

$$= \sum_{\mathbf{i} 2 I} \Big(\sum_{\sum_l \mathbf{i}_l = \mathbf{i}} \prod_{l=1}^{h} c_{l,\mathbf{i}_l} \Big) X^{\mathbf{i}} \tag{8}$$

By induction hypothesis, we have $Pr(S_l = \mathbf{i}_l) = c_{l,\mathbf{i}_l}$ for any l and $\mathbf{i}_l$. Therefore, we can conclude from (7) and (8) that $F(X) = \sum_{\mathbf{i}} Pr(S = \mathbf{i}) X^{\mathbf{i}}$.

Now let us consider the other case where r is a _ node. From Definition 2, it is not hard to see that

$$Pr(S = \mathbf{i}) = \sum_{l=1}^{h} Pr(S_l = \mathbf{i}) p_{(r,r_l)} \tag{9}$$

Moreover, we have

$$F(X) = \sum_{l=1}^{h} p_{(r,r_l)} F_l(X) = \sum_{l=1}^{h} p_{(r,r_l)} \sum_{\mathbf{i}_l} c_{l,\mathbf{i}_l} X^{\mathbf{i}_l}$$

$$= \sum_{\mathbf{i}} \Big(\sum_{l=1}^{h} p_{(r,r_l)} c_{l,\mathbf{i}} \Big) X^{\mathbf{i}} = \sum_{\mathbf{i}} \Pr(S = \mathbf{i}) X^{\mathbf{i}}$$

where the last equality follows from (9) and induction hypothesis. This completes the proof.

B Expanding Polynomials

This section is devoted to several algorithms for expanding polynomials into standard forms.

Multiplication of a Set of Polynomials: Given a set of polynomials in the form of $P_i = \sum_{j=0} c_{ij} x^j$ for $1 \quad i \quad k$, we want to compute the multiplication $P = \prod_{i=1}^{k} P_i$ written in the standard form $P = \sum_{j=0} c_j x^j$, i.e., we need to compute the coefficients c_j. Let $d(P_i)$ be the degree of the polynomial P_i. Let $n = \sum_i^k d(P_i)$ be the degree of P.

Naive Method: First we note that the naive method (multiply P_is one by one) gives us an $O(n^2)$ time algorithm by simple counting argument. Let $P_i = \prod_{j=1}^{i} P_j$. It is easy to see $d(P_i) = \sum_{j=1}^{i} d(P_i)$. So the time to multiply P_i and P_{i+1} is $O(d(P_i) \quad d(P_{i+1}))$. Then, we can see the total time complexity is:

$$\sum_{i=1}^{k-1} O(d(P_i) \quad d(P_{i+1})) = O(n) \quad \sum_{i=1}^{k-1} d(P_{i+1}) = O(n^2).$$

Divide-and-Conquer: Now, we show how to use divide-and-conquer and FFT (Fast Fourier Transformation) to achieve an $O(n \log^2 n)$ time algorithm. It is well known that the multiplication of two polynomials of degree $O(n)$ can be done in $O(n \log n)$ time using FFT. The divide-and-conquer algorithm is as follows: If there exists any P_i such that $d(P_i) \quad \frac{1}{3}d(P)$, we evaluate $\prod_{j:j \in i} P_i$ recursively and then multiply it with P_i using FFT. If not, we partition all P_is into two sets S_1 and S_2 such that $\frac{1}{3}d(P) \quad d(\prod_{i \in S_j} P_i) \quad \frac{2}{3}d(P)$. Then we evaluate S_1 and S_2 separately and multiply them together using FFT. It is easy to see the time complexity of the algorithm running on input size n satisfies

$$T(n) \quad \max\{T(\tfrac{2}{3}n) + O(n \log n); T(n_1) + T(n_2) + O(n \log n)\}$$

where $n_1 + n_2 = n$ and $\frac{1}{3}n \quad n_1 \quad n_2 \quad \frac{2}{3}n$. By solving the above recursive formula, we know $T(n) = O(n \log^2 n)$.

Expanding a Nested Formula: We consider a more general problem of expanding a nested expression of uni-variable polynomial (with variable x) into its standard form $\sum c_i x^i$. Here a nested expression refers to a formula that only involves constants, the variable x, addition +, multiplication , and parenthesis (and), for example, $f(x) = ((1 + x + x^2)(x^2 + 2x^3) + x^3(2 + 3x^4))(1 + 2x)$. Formally, we define recursively an *expression* to be either

1. A constant or the variable x, or
2. The sum of two *expressions*, or
3. The product of two *expressions*.

We assume the degree of the polynomial and the length of the expression are of sizes $O(n)$. The naive method runs in time $O(n^3)$ (each inner node needs $O(n^2)$ time as shown in the last subsection). If we use the previous divide-and-conquer method for expanding each inner node, you can easily get $O(n^2 \log^2 n)$. Now we sketch two improved algorithms with running time $O(n^2)$. The first is conceptual simpler while the second is much easier to implement.

Algorithm 1:

1. Choose $n + 1$ different numbers $x_0; :::; x_n$.
2. Evaluate the polynomial at these points, i.e., compute $f(x_i)$. It is easy to see that each evaluation takes linear time (bottom-up over the tree). So this step takes $O(n^2)$ time in total.
3. Use any $O(n^2)$ polynomial interpolation algorithm to find the coefficient. In fact, the interpolation reduces to finding a solution for the following linear system:

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & ::: & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & ::: & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & ::: & x_n & 1 \end{bmatrix} \begin{bmatrix} c_n \\ c_{n-1} \\ \vdots \\ c_0 \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix}.$$

The commonly used Gaussian elimination for inverting a matrix requires $O(n^3)$ operations. The matrix we used is a special type of matrix and is commonly referred to as a Vandermonde matrix. There exists numerical algorithms that can invert a Vandermonde matrix in $O(n^2)$ time, for example [7].

A small drawback of the above algorithm is that the algorithms used to invert a Vandermonde matrix is nontrivial to implement. The next algorithm does not need to invert a matrix, is much simpler to implement and has the same running time of $O(n^2)$.

Algorithm 2: We need some notation first. Suppose the polynomial is $f(x) = \sum_{j=0}^{n} c_j x^j$ (c_js are unknown yet). Let $\mathbf{e}_i$ be the $(n + 1)$-dimensional zero vector except that the i^{th} entry is 1, i.e., $\mathbf{e}_i = (0; 0; ::; 1; :::; 0; 0)$. Let $\mathbf{d}_i = (1; e^{\frac{2\pi j}{n+1} i}; e^{\frac{2\pi j}{n+1} 2i}; ::: : i)$ be the $n + 1$-dimensional vector which is the DFT (Discrete Fourier Transformation) of $\mathbf{e}_i$. Let $u = e^{\frac{2\pi j}{n+1}}$ be the $n + 1^{\text{th}}$ root of unit. Let $\mathbf{u} = (1; u; u^2; :::; u^n)$ and $\mathbf{u}^k = (1; u^k; u^{2k}; u^{3k}; :::: :)$.

By definition, $\mathbf{e}_i = \frac{1}{n+1} \sum_k \mathbf{d}_{ik} \mathbf{u}^k$ where $\mathbf{d}_{ik}$ is the k^{th} entry of $\mathbf{d}_i$. Let $\mathbf{c} = (c_0; ::: ; c_n)$ be the coefficient vector of f. It is trivial to see $c_i = \mathbf{c} \quad \mathbf{e}_i$ (the inner product). Therefore, we have that:

$$c_i = \mathbf{c} \quad \mathbf{e}_i = \frac{1}{n+1} \sum_k \mathbf{d}_{ik} (\mathbf{c} \quad \mathbf{u}^k) = \frac{1}{n+1} \sum_k \mathbf{d}_{ik} f(u^k) \qquad (10)$$

The last equality holds by the definition of $f(x)$. If we use $\mathbf{f}$ to denote the vector $(f(u^0); ::: ; f(u^n))$ and $\mathbf{D}$ to denote the matrix $\{\mathbf{d}_{ij}\}_{0 \quad i,j \quad n}$, the above equation can be simply written as

$$\mathbf{c} = \frac{1}{n+1} \mathbf{D} \mathbf{f}.$$

Now, we are ready describe our algorithm:

1. Compute $f(u^k)$ for all k. This consists of evaluating $f(x)$ over complex x n times, which takes $O(n^2)$ time.
2. Use (10) to compute the coefficients. This again takes $O(n^2)$ time.

In fact, the above algorithm can be seen as a specialization of the first algorithm. Instead of picking arbitrary $n + 1$ real points $x_0; ::: x_n$ to evaluate the polynomial, we pick $n + 1$ complex points $1; u; u^2; ::: ; u^n$. The Vandermonde matrix formed by these points, i.e.,

$$\mathbf{F} = \begin{bmatrix} u^{0\,0} & u^{0\,1} & ::: & u^{0\,n} \\ u^{1\,0} & u^{1\,1} & ::: & u^{1\,n} \\ \vdots & \vdots & \ddots & \vdots \\ u^{n\,0} & u^{n\,1} & ::: & u^{n\,n} \end{bmatrix}$$

has a very nice property that

$$\mathbf{F}^{-1} = \frac{1}{n+1} \mathbf{F}^*$$

where $\mathbf{F}^*$ is the conjugate of $\mathbf{F}$ (This can be verified easily). Therefore, we can obtain $\mathbf{F}^{-1}$ for free. Actually, it is easy to see that $\mathbf{F}^*$ is exactly $\mathbf{D}$.