Approximate Substring Matching over Uncertain Strings

Tingjian Ge, Zheng Li
University of Kentucky
ge@cs.uky.edu, zheng.li@uky.edu

ABSTRACT

Text data is prevalent in life. Some of this data is uncertain and is best modeled by probability distributions. Examples include biological sequence data and automatic ECG annotations, among others. Approximate *substring* matching over *uncertain* texts is largely an unexplored problem in data management. In this paper, we study this intriguing question. We propose a semantics called (k, τ)-matching queries and argue that it is more suitable in this context than a related semantics that has been proposed previously. Since uncertainty incurs considerable overhead on indexing as well as the final verification for a match, we devise techniques for both. For indexing, we propose a multilevel filtering technique based on measuring signature distance; for verification, we design two algorithms that give upper and lower bounds and significantly reduce the costs. We validate our algorithms with a systematic evaluation on two real-world datasets and some synthetic datasets.

1. INTRODUCTION

Text data is prevalent in life. Approximate substring matching over *deterministic* data is well studied in computer science. It has many applications, including computational biology, signal processing, and text retrieval. There is an excellent survey by Navarro [10] on the algorithms and applications of this topic. As the amount of text data increases in an unprecedented rate due to factors such as large genomic projects (e.g., [1]) and the growth of the Internet, managing the sheer amount of (often noisy) text data has become more challenging than ever. Specifically, as a consequence of the burgeoning growth of data and the cost/technology constraints against producing completely clean data, there is much uncertainty in the data itself.

In the Holter monitor application, for example, sensors attached to heart-disease patients send out ECG signals continuously to a computer through a wireless network [3]. For each heartbeat, the annotation software gives a symbol such as N (Normal beat), L (Left bundle branch block beat), and R, etc. However, quite often, the ECG signal of each beat may have ambiguity, and a probability distribution on a few possibilities can be given [3]. A doctor might be interested in locating a pattern such as "NNAV" indicating two normal beats followed by an atrial premature beat and then a premature ventricular contraction, in order to verify a specific diagnosis.

As another example, a single DNA sequence can be a few million to a few hundred million characters (DNA can be seen as texts over an alphabet of size $4 - \{A, C, G, T\}$). It has uncertainty due to a number of factors in the high-throughput sequencing technologies [8, 15, 4]. Indeed, the NC-IUB committee standardized incompletely specified bases in DNA [13] to address this common presence of uncertainty, by adding to the alphabet the letters 'R' for A or G, and 'Y' for T or C, among others.

While *approximate* substring matching itself (e.g., through the edit distance metric [10, 16]) addresses the uncertainty issue to some degree, it is still necessary to model the uncertainty as probability distributions. This is because we do have partial knowledge about the uncertain characters. For example, from high throughput sequencing technologies, we usually can narrow down an uncertain character to two or three alternatives, and can assign probabilities to them based on their frequencies of occurrence in sequencing runs [17]. In essence, we differentiate between *complete mismatches* and *probable matches with some uncertainty*, in order to make the best informed decisions.

Related Work. Previous work on managing large-scale sequence data includes [14] by Tata et al. who proposed a declarative querying framework for biological sequences, and [7] by Kandhan et al. who studied the multi-pattern matching problem.

In addition, there is some work on deterministic *string joins*, e.g., [2, 5, 9], where the applications include data cleaning, data integration, and fuzzy keyword search. Recently, it is extended into the probabilistic setting [6]. The matching there is at the *whole* string level on both sides. Thus, it is only applicable to finding similar strings (that are of similar lengths). Our work, however, targets many applications in which text strings can be of arbitrary length (often very long) and are uncertain in places; we need to search for some patterns within them, i.e., *substring* matches over *uncertain* texts.

Other closely related work (e.g., indexing) will be cited inline as appropriate.

Our Contributions. Jestes et al. [6] propose the notion of *expected edit distance* (EED) over all possible worlds of two uncertain strings. We show that EED is inappropriate for our problem. Instead, with our newly proposed (k, τ)-matching query, we can easily extend previous query processing techniques. More importantly, we can find matches without having too many false positives. This is illustrated in Figure 1, an experimental result on a real dataset – the E. coli 536 DNA sequence [17] (details will be described in Section 6). The figure shows that, in order to catch a *true* match, EED must use a big distance threshold which will incur significantly (about three orders of magnitude) more false positives than our (k, τ)-queries.

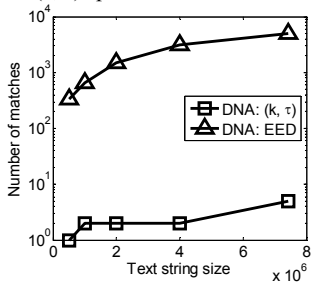

Figure 1. Comparison of the number of matches.

Proceedings of the VLDB Endowment, Vol. 4, No. 11

We then study how to efficiently answer a (k, τ)- query. An efficient indexing method is needed to speed up the search on long texts. The problem becomes more crucial with uncertain texts. We propose a novel technique called *multilevel signature filtering* for indexing. Our index is highly selective, saving both I/O (random seeking) and CPU costs. After the index scan, the final step is to verify whether each position in text given by the index actually contains a real match. We devise two efficient bounding algorithms, one based on CDF and the other on local perturbations. To summarize, the contributions of this work include:

- Proposing the semantics of (k, τ)-matching queries (Sec. 3).
- Devising techniques for indexing uncertain texts (Sec. 4).
- Devising efficient verification algorithms (Sec. 5).
- Systematic evaluation on real and synthetic datasets (Sec. 6).

2. PRELIMINARIES

Terminology and Notations. A *string* s contains a sequence of *characters* chosen from a finite *alphabet* Σ. Thus, the domain of a string is often written as Σ^*. The *length* of a string s, denoted as $|s|$, is the number of characters in s. The *edit distance* of two strings s_1 and s_2, denoted as $d(s_1, s_2)$, is the minimal number of character *insertions*, *deletions*, or *substitutions* (called *edit operations*) transforming one string into the other.

Deterministic Substring Matching Problem. Given as input a *pattern string* p, a set of *text strings* $\{x_i \mid 1 \le i \le r\}$, and an edit distance threshold k, the problem is to find all substrings s of x_i's such that $d(p, s) \le k$. These r text strings might correspond to a string column of a table that has r records. For instance, the column could be the whole human genome and each record is a gene [14]. When we only consider one text string (at a time), we often drop the subscript and simply use x to denote a text string.

Indexing. [10] is a good survey of online algorithms for this problem. When the texts are very large, indexing is needed, which includes suffix-tree approaches and q-gram based approaches as the most popular ones [11, 12]. The most frequently used indexing method for *approximate* matching is the q-gram indexes (and their variations, e.g., the q-sample indexes) [11]. In such an index, for each q-gram (i.e., substring of length q), all its positions in the texts (called *occurrences*) are stored in increasing text order.

There are variations on how to use the index to search for approximate matches of a pattern p. But most of them use the idea of partitioning p [12]. We can partition p into $k + 1$ *pieces*, where k is the edit distance threshold. Since the number of errors is no more than k, it must be true that at least one piece must have an *exact* match in the text. We thus search for each of the $k + 1$ pieces in the q-grams of an index. For each match, we go over its associated position list, and *verify* the areas in the text for an approximate match. There are additional details on handling *pieces* that are longer or shorter than q, which we omit here (see, e.g., [11]). For verification, any online algorithm will do [10]; many of them are based on an approach as follows.

Verification Algorithm. We need to verify the two substrings in p on both sides of the matching piece (which we call g). We can use *two runs* of a variant of the well-known dynamic programming (DP) algorithm for computing an edit distance. In run 1, we find a *best* match of the substring to the left of g in p with the corresponding substring to the left of g in the text. Likewise, in run 2, we find a *best* match of the substring to the right of g. Let the best match in run 1 have an edit distance d_l and the best match in run 2 have a distance d_r. Then clearly we require $d_l + d_r \le k$. Note that in run 1, we actually reverse the substring (to the left of g), so that it starts from a known, fixed position in the text (i.e., immediately to the left of g), but may end at any position in a range (depending on which one gives the minimum edit distance). The DP algorithm to compute an edit distance is based on:

$$d[i, j] = \min\{d[i, j-1] + 1,\ d[i-1, j] + 1,\ d[i-1, j-1] + c(p[i], x[j])\} \quad (1)$$

where

$$c(p[i], x[j]) = \begin{cases} 0, & \text{if } p[i] = x[j] \\ 1, & \text{if } p[i] \ne x[j] \end{cases}.$$

This essentially fills in a two-dimensional table, which we call a *DP table*. We say that one dimension is the *pattern dimension* and the other is the *text dimension*. Note that we use a longer text substring $|x| = |p| + k$, and the best match is the minimum value amongst the $2k$ cells at the text dimension index from $|p| - k + 1$ to $|p| + k$ and the pattern dimension index $|p|$ (i.e., the shaded region in the last row in Figure 5(a)).

Moreover, the DP algorithm always gives us a "path" on how to reach the minimum distance value. Whenever a cell of the DP table is filled in a value using (1), we also record which of the three neighbors (i.e., $d[i, j-1]$, $d[i-1, j]$, or $d[i-1, j-1]$) is used (i.e., selected by the *min* in (1)). Thus, in the end, when tracing back, we get a path, which we call an *optimal path*.

3. QUERY SEMANTICS

3.1 (k, τ)-Matching Semantics

We now look at the matching problem over *uncertain* texts. With minor changes, our algorithms can also apply to uncertain patterns. But it is more common that only texts are uncertain in the applications that we are aware of.

Uncertainty Model. In recent work [6], Jestes et al. proposed two models of uncertain strings, namely the *string-level* model and the *character-level* model. A string-level model enumerates all possible worlds and their probabilities at the whole-string level, while a character-level model describes distributions for each uncertain character. We focus on the *character-level* model, since this model is both realistic and concise in representing the uncertainty in *long* text strings, whereas the string-level model is unnecessary for our applications and is prohibitively expensive for long strings. For example, the high sequencing technology for DNA gives uncertainty at the nucleotide (i.e., character) level [8, 15]. In a character-level model, a string is a random variable $X = X[1]\ldots X[n]$, where $X[i]$ $(1 \le i \le n)$ in general is a random variable with a discrete distribution over Σ. We have the following new semantics for pattern matching queries over uncertain texts.

Definition 1. *A* **(k, τ)-matching query** *is based on a pattern string p, a set of uncertain text strings* $\{X_i\}$ $(1 \le i \le r)$*, and threshold parameters k, τ, and asks for all substrings X of* X_i*'s such that* $\Pr[d(p, X) \le k] > \tau$.

The parameter k here is based on specific domain knowledge. For example, in a DNA sequence, if we allow at most two mutations, we can set $k = 2$. The parameter τ is chosen to exclude (low probability) random matches by chance. For instance, for a pattern of length 5, we may set $\tau = (1/2)^5$, requiring that, on average, each character has a probability of at least 1/2 leading to a match (which is higher than pure chance).

3.2 Comparison with EED

Jestes et al. [6] propose the notion of EED for comparing uncertain strings, which is the *expected edit distance*. One can imagine

that EED could also be used in our problem by retrieving all substrings X such that EED(p, X) ≤ k. However, there are two major reasons why EED is not suitable in this context.

Reason 1. EED does not implement the possible-world semantics *completely* at the query level. As a result, many algorithms developed for the deterministic case are inapplicable. The deterministic counterpart of our predicate is $d(p, X) \le k$. Following the possible world semantics, it should be that, in each possible world ω, we use *exactly the same* predicate $d(p, x) \le k$, where $X = x$ in ω. In the end, we get some aggregate property over all worlds, e.g., $\Pr[d(p, X) \le k]$. A (k, τ)-query is based on this.

By contrast, EED averages the distance between p and X over all (weighted) possible worlds in the first place, and then impose a threshold on this average. That is, it summarizes the possible worlds early and changes the predicate to have a threshold on the EED. This makes existing algorithms that work for the deterministic case no longer applicable to the possible worlds.

An example is the indexing work in Section 2, where, by partitioning p into $k + 1$ pieces, we know that at least one of the pieces must have an exact match. We can still use this mechanism with (k, τ)-queries since only those worlds with $d(p, x) \le k$ will be relevant. With EED, however, we cannot cleanly use this method because a relevant world potentially could have an arbitrary edit distance, although the average of all worlds is within a threshold.

Reason 2. The second and more important reason is that EED may either miss *real* matches or have an unduly big threshold so that many false positives may mix in. Let us look at an example.

Example 1. *In the DNA sequence of E. coli that has a few million characters, suppose we look for close matches of a pattern p of length* 20. *Consider two substrings X_1 and X_2 in the DNA, both of length* 20. *X_1 is a perfect match* (*i.e., edit distance* 0), *but we have some uncertainty in it:* 9 *characters are uncertain, each with probability* 1/3 *being the correct one. Thus, it can be shown* (*from the linearity of expectation*) *that when the* 9 *characters are about evenly spaced in X_1,* EED(p, X_1) *is about* 9 × (2/3) = 6. *On the other hand, X_2 contains* 2 *uncertain characters, each with probability* 0.5 *being the correct one; but it also contains* 5 *mismatched characters*. *Suppose that purely from the life science, an interesting match requires an edit distance of no more than* 2 (*say, allowing mutations at two places at most*), *which will clearly exclude X_2. However, when the mismatched and uncertain characters are evenly spaced, it can be shown that* EED(p, X_2) *is about* 5+2×0.5 = 6, *which is the same as* EED(p, X_1). *Therefore, the threshold of an* EED *predicate needs to be at least* 6 *in order to include the real match X_1, in which case many false positives such as X_2 also qualify the predicate. We will further verify this point in our experiments on real datasets, Fig.* 1 *and Sec.* 6.

The root cause here is that EED does not distinguish between *true errors* and *uncertainty*. A big EED threshold in order to accommodate uncertainty could also mix in a large amount of noise, i.e., strings with many true errors. A (k, τ)-query, on the other hand, sets the allowed errors (parameter k) and the uncertainty threshold (parameter τ) separately. A (2, 0)-matching query, for example, will select X_1, but not X_2 in Example 1.

Reflecting back, the edit distance of uncertain strings is essentially a distribution. Some strings (such as X_1) produce *long-tailed* distributions while others (such as X_2) produce *short-tailed* ones. EED only looks at the expectations. However, we do care about certain long tails at the small distance end; the probability there, albeit small, can still be significant compared to mere chance. Thus, a (k, τ)-query can select real matches with little noise mixed into the results. The relatively low probability in this case is simply because of the accumulated uncertainty. Some additional (slightly more formal) analysis of this appears in Appendix A.

It is well known that uncertainty in a DNA or protein sequence can be quite uneven [1]. Specifically, the uncertain character ratio has a significant variance in the DNA/protein sequence of our real datasets, and we can easily spot places in the strings where uncertainty is well above average. However, in spite of the uncertainty, we should be able to discern true patterns.

4. INDEXING UNCERTAIN STRINGS

We generalize a q-gram index. The basic idea is that for each q-gram in the index, each position in the list becomes a probabilistic occurrence of the pattern. Since one position in the text can possibly produce multiple q-grams (with total probability summing to 1), a position could appear multiple times in the index and the index would be bigger than the deterministic case.

The key reason for using a q-gram based index (rather than, say, a suffix tree based index) is that q is a system parameter and is usually small [11], which entails that the increase in size of an index due to uncertainty can be small and tunable. More precisely, let the fraction of uncertain characters in the strings be θ, and let the average number of alternatives of an uncertain character be α. Then the uncertain index is approximately $(\alpha\theta + 1 - \theta)^q$ times the size of a deterministic one. In all practical applications, θ and α are typically small. Specifically, suppose the ratio of uncertain characters is about 10%, and α is close to 2 on average. If we use q = 4, then the index size is about 1.46 times the deterministic one, an acceptable storage cost for handling uncertainty.

4.1 Improved Indexing for Uncertain Strings

Let us tackle the problem of a longer position list due to *uncertainty*. Note that we cannot simply increase q to scale up. This is because as q increases, the number of q-grams that we need to store grows exponentially. Moreover, recall that, during index search, we first partition the pattern p into $k + 1$ pieces and use each piece to probe the index. Clearly, $|p|$ is well bounded and as q increases, each piece of p will become shorter than the q-grams. This implies that we have to scan many q-gram entries (that have the piece string as a *prefix*, since they are all potential matches) and their position lists. Thus, we still end up verifying many positions. Another reason for not increasing q is that for *uncertain* strings, as discussed earlier, an increase of q causes an exponential growth of the index size, and we may need to verify more positions. Therefore, we need a different solution.

4.1.1 Multilevel Signature Filtering in an Index

We use concise signatures to further filter candidate positions. A signature is formed by left or right extensions of the q-gram at a candidate position. We first look at the data structure of our index for deterministic strings.

Definition 2 (signatures). *Consider a string x from Σ^* and an occurrence of a q-gram g at position pos in x. Suppose there is a hash function $h: \Sigma \to [0, 2^b - 1]$ that maps a character from Σ to a b-bit value. Then, the bit sequence $h(x[pos-1])\bullet\ h(x[pos-2])\bullet\ \ldots\bullet h(x[pos-l])$ is called a* left signature of length l *of the occurrence of g. Similarly, the bit sequence $h(x[pos+q])\bullet\ h(x[pos+q+1])\bullet\ \ldots\bullet h(x[pos+q+l-1])$ is called a* right signature of length l. *Here, $\bullet$ denotes the bit sequence concatenation.*

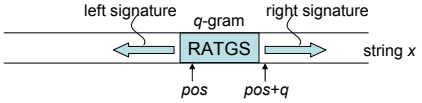

Figure 2. Illustrating the left and right signatures of a string.

A signature of length l has $b \cdot l$ bits. In this paper, unless otherwise specified, we use $b = 2$. The hash function h maps a character in Σ to a two-bit value. Figure 2 illustrates the concept.

We now discuss how we use this signature for indexing. Recall that a q-gram index contains a position list for each q-gram. We further organize these candidate positions according to their signatures. Unless specified otherwise, we use a left signature of length 8 (i.e., 16 bits) and a right signature of length 8 (i.e., 16 bits), totaling 32 bits. We call this 32-bit value a *tag*. We store a table of <*tag*, *pointer*> entries, where a *pointer* either points to a list of positions having that tag value (if the list is short, say, no more than three positions) or points to a next level of directory (if the list is long). This is illustrated in Figure 3.

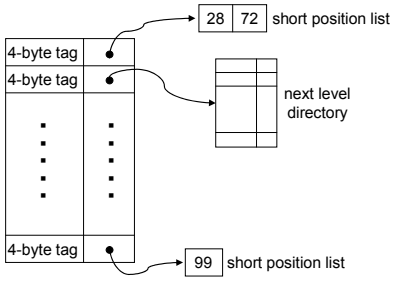

Figure 3. Illustrating multilevel signature filtering structure.

The next level directory has the same structure, except that a tag consists of left and right signatures that are the *continuations* of the previous level (i.e., the next 8 characters on both sides); thus the tag is again 4 bytes. Recursively, this directory structure can continue to deeper levels, forming a tree.

Note that the signature tree structure is very compact. Most practical datasets do not require many levels. Moreover, most search queries use relatively short patterns, where deep levels will not be helpful. Therefore, we can keep the maximum number of levels small (e.g., 3) and allow the position list at the deepest level to be of any length. A long position list at the deepest level implies that there is a long and very common pattern in the text; if a search ever gets there, it would probably need to access those positions anyway, since there are many true matches. A final remark is that this technique can be applied to any q-gram based indexing scheme; for simplicity, we illustrate it with the most basic q-gram index.

4.1.2 Using the Index

We first discuss how to use the signatures in an index for filtering. The intuition is as follows. The indexing mechanism (Section 2) partitions p into $k + 1$ pieces and at least one of the pieces g is required to have an exact match in the text x (called an *occurrence*). We can do a quick verification by examining g's left and right signatures in x (as found in the index). These signatures should not differ much from g's left and right signatures in p; otherwise, $d(p, x) \leq k$ would be violated. The precise requirement is described in Theorem 1 that follows. We start with some definitions and lemmas.

Definition 3 (prefix distance). *A* best matching prefix (BMP) *of a string x for a string p is a prefix x_0 of x such that the edit distance between p and x_0 is minimum* (*among all prefixes of x*). *We say that $d(p, x_0)$ is string x's* prefix distance *from p.*

BMP and prefix distance can be computed through the same dynamic programming between p and x. The only extra step is that, in the end, we pick the smallest distance value from the row (or column) corresponding to the last character of p in the DP table. Note that the verification algorithm in Section 2 essentially computes prefix distances. Let us look at an example.

Example 2. *The last row of the DP table in Figure* 4 *shows that* "1" *is the smallest. Hence,* 1 *is x's prefix distance from p, and GGAP is the best matching prefix of x for p.*

p \ x		G	G	A	P	P
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	1	1	2	3
P	3	2	2	2	1	2

Figure 4. Illustrating best matching prefix and prefix distance.

Definition 4 (edit distance of signatures). *We treat a signature in Definition* 2 *as a string with each b-bit hash value as a character. That is, the alphabet of the string is* $[0, 2^b - 1]$. *The* edit distance of two signatures σ_1 *and* σ_2 *is their edit distance when we treat them as two such strings.*

Lemma 1. *The edit distance of two signatures* (*Definition* 4) *is no more than the edit distance of their* original *strings.*

The proofs of all lemmas and theorems in the paper appear in Appendix B.

It is a simple extension to handle uncertain characters. We reserve a hash value (in the signature function), e.g., 0, for all uncertain characters. Let us call it a *universal* value. A universal value can match any characters in a DP algorithm for edit distance. Note that this is a conservative decision; but the signature is still *very selective* since the uncertain ratio is not too high in practice. We will verify this in the experiments. Moreover, what we gain from this simple treatment is that we can keep the index very concise, without listing out all alternatives of an uncertain character or the probabilities. This makes it efficient when scanning the index.

Lemma 2. *Consider the dynamic programming algorithm that is used to compute the edit distance, as shown in Equation* (1). *It must be true that:*

$$d[i][j-1] - 1 \leq d[i][j] \leq d[i][j-1] + 1 \quad (2)$$

$$d[i-1][j] - 1 \leq d[i][j] \leq d[i-1][j] + 1 \quad (3)$$

$$d[i-1][j-1] \leq d[i][j]. \quad (4)$$

Intuitively, Inequalities (2) and (3) say that the difference between two horizontal or vertical neighbor cells of a DP table can only be -1, 0, or 1, while Inequality (4) says that values on each diagonal form a nondecreasing sequence. Lemma 2 is based on Lemmas 1 and 2 in [16]. The following theorem establishes the correctness of our index search algorithm.

Theorem 1. *Suppose a q-gram g of the pattern p has an occurrence in the text x* (*which we try to verify with the signatures in an index*). *At this occurrence, suppose g has a left signature x_l that has length l_1, and a right signature x_r that has length l_2. Accordingly in p, we compute g's left* (*right*) *signature p_l (p_r) that has length $l_1 + k$ ($l_2 + k$), where k is the distance threshold. Let d_l be p_l's prefix distance from x_l and d_r be p_r's prefix distance from x_r. Then, the verification requires $d_l + d_r \leq k$.*

Example 3. *Fig. 5*(*a*) *shows a DP table, where the shaded region in the bottom row may correspond to the last three cells of the last row of the DP table in Figure* 4 (*i.e., values* 2, 1, 2). *As we project these three cells diagonal-wise, they can map to the first three cells in the third column* (*i.e., values* 2, 1, 1), *which correspond back to the vertical shaded region in Figure* 5(a). *Due to Lemma* 2, *the second set of three values cannot be bigger than the first*

set. For a signature of length l, the minimum value in the vertical shaded region is the prefix distance d_l or d_r in Theorem 1.

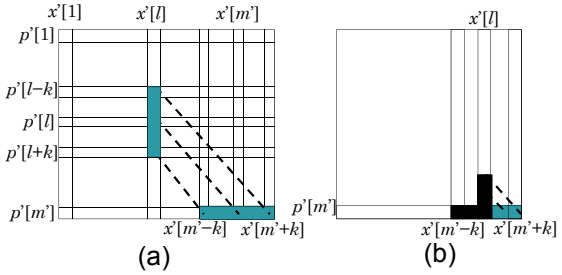

Figure 5. Dynamic programming verification and signatures.

We now discuss the index search. The initial part of breaking a pattern into $k+1$ pieces and performing an exact match on each piece is the same as before. After we use one of the pieces to reach a directory tree T as shown in Figure 3, we have the following algorithm PUSH-POP-SIGNATURE-CHECK.

Algorithm PUSH-POP-SIGNATURE-CHECK (p, k, T)

Input: Pattern string p, distance threshold k, signature directory tree T corresponding to a q-gram g.

Output: A set of positions in the text strings to verify.

1: Do a depth-first traversal (DFT) of T, with the following extra steps during the DFT.
2: Initialize 2 DP tables DP_l and DP_r (for left/right verifications) to be empty and they grow and shrink as follows.
3: Upon reaching $< t, c >$, where t is a tag in a node of T that leads to a child node c:
4: Before following t to c in the DFT: **(PUSH steps)**
5: Extend the *text* dimension of DP_l (and DP_r) with the left (right) signature in t.
6: Extend the *pattern* dimension of DP_l (DP_r) (using the signature of p) to make it k characters longer than the text dimension.
7: Populate the new cells in DP_l and DP_r.
8: Let d_l (d_r) be the *prefix distance* of the pattern from the text in DP_l (DP_r).
9: **if** $d_l + d_r > k$ **then** skip the whole subtree rooted at c and continue the DFT **end if**
10: Upon finishing the subtree rooted at c and backtracking: **(POP steps)**
11: Truncate text and pattern dimensions to the length before the extensions in lines 5 and 6.
12: Upon reaching a leaf node (a position list) during DFT, add the list to the set of positions to be returned.

Doing PUSH/POP's on a DP table enables efficient signature checking. For clarity of presentation, in line 6, we omit the details on the case in which p is exhausted during the extension (i.e., $l > m' - k$ in Fig 5b, where m' is pattern length); this is discussed in Appendix C. The correctness of the algorithm directly follows from Theorem 1. We now look at a simplified example.

Example 4. *Suppose p contains a q-gram "BEST", and this q-gram has an occurrence in a string x:*

x: ……[*DC*][*CB*][*BEST*][*RO*][*DO*]*……*
p: [*DC*][*AB*][*BEST*][*NR*][*OB*]

As shown above, the index locates an occurrence of the q-gram in the middle of x (*surrounded by brackets*)*. Suppose a tag contains two characters of the left and right signatures each. Then, tag t_1 in the root node contains the signatures of BC* (*left signature, in reverse order*) *and RO, while tag t_2 in a child c of the root contains the signatures of CD* (*left*) *and DO* (*right*)*. We also show the signatures at corresponding positions in p. From now on, we drop the phrase "the signatures of" for brevity.*

Let k = 2. The algorithm starts with empty DP_l and DP_r. It first pushes BC (*RO, respectively*) *into the text dimension of DP_l (DP_r). That is, $l_1 = l_2 = 2$ in Theorem* 1*. It also extends DP_l (DP_r)'s pattern dimension to be BACD* (*NROB*)*, i.e., k characters longer than the text ($l_1 + k = l_2 + k = 4$ in Theorem* 1*). The prefix distances of these patterns from the text strings are both* 1 *in DP_l and DP_r, with a total of* 2*, which is ≤ k. Note that if we did not extend the pattern dimension of DP_r with the extra k characters* (*but it only has NR*)*, then the distance in DP_r would be* 2*, causing a violation of the total distance constraint. Similarly, we go to the child node c* (*PUSH*) *and continue the checking.*

5. VERIFICATION ALGORITHMS

The goal of verification is to conclude whether a candidate position selected by an index is a true match. A straightforward solution is: (1) For each possible world, calculate the smallest edit distance using the verification algorithm (Sec. 2); (2) Sum up the probabilities of all worlds in which $d(p, x) \le k$, and compare the sum with τ. This clearly can be very costly. An *efficient* exact algorithm is extremely difficult to obtain (we conjecture that this problem is #P-complete). We present two algorithms, each of which gives an upper and a lower bound of the probability. By comparing them with τ, we can either early accept (if a lower bound is > τ) or early reject (if an upper bound is < τ) a candidate position. Only when we can do neither, do we resort to the (slow) exact verification.

5.1 Bounds Based on CDF

The basic verification (Sec. 2) consists of two symmetric runs of a DP algorithm. We describe how we change such a DP algorithm to accommodate uncertain characters. For each cell of the DP table, ideally we wish to get a *distribution* $\{p[j] | 0 \le j \le k, p[j] = \Pr(D = j)\}$, where D is the edit distance at the cell. It is truncated to $0 \le j \le k$ because once $D > k$, we do not need the exact probability to answer a (k, τ)-matching query. This would give an exact algorithm. Unfortunately, this distribution is extremely hard to get because neighboring cells' distributions are correlated.

Instead, our key idea is to compute (at most) $k+1$ pairs of values in each cell, i.e., $\{(F_l[j], F_u[j]) \mid 0 \le j \le k\}$, where $F_l[j]$ and $F_u[j]$ are the lower and upper bounds of $\Pr[D \le j]$ at the cell, respectively. That is, we bound the cumulative distribution functions (CDF). Then for a (k, τ)-query, we check the bounds $(F_l[k], F_u[k])$.

To fill in the DP table, we only need to consider a basic step, i.e., how to get the cumulative probability bounds of a cell D from those of its three neighbor cells D_1 (upper left), D_2 (upper), and D_3 (left). This is shown in the ELEMENTARY-STEP algorithm below. The intuition is that, at cell D, when the text character C matches the pattern character c (with probability p_1), it is always optimal to take the distribution from the diagonal upper left neighbor D_1 (Theorem 2), and thus we can take the whole distribution from there with weight p_1. When there is a mismatch at D, we need some relaxation.

Algorithm ELEMENTARY-STEP (D_1, D_2, D_3, c, C)

Input: $D_i = \{(F_l^{(i)}[j],\ F_u^{(i)}[j]) \mid 0 \le j \le k\}, for\ 1 \le i \le 3$,
pattern character c, text character random variable C
Output: $D = \{(F_l[j],\ F_u[j]) \mid 0 \le j \le k\}$

1: $p_1 \leftarrow \Pr[C = c]$ *// probability of a match at cell D*
2: $p_2 \leftarrow 1 - p_1$
// calculate the $(F_l[j], F_u[j])$ *pairs at cell D*
3: **for each** $j \leftarrow 0$ *to* k **do**
4: $F_l[j] \leftarrow p_1 F_l^{(1)}[j] + p_2 F_l^{(argmin_i D_i)}[j-1]$
5: $F_u[j] \leftarrow p_1 F_u^{(1)}[j] + p_2 \min(\sum_{i=1}^{3} F_u^{(i)}[j-1], 1)$
6: **end for**

In line 4, $argmin_i D_i$ returns the index value i that *minimizes* D_i; the minimization is defined as the D_i ($1 \le i \le 3$) that has the greatest $F_l^{(i)}[0]$; a tie is broken by selecting the greatest $F_l^{(i)}[1]$ (and then $F_l^{(i)}[2]$, etc., i.e., the one with highest probability of being *small*). In lines 4 and 5, $F_l^{(i)}[-1] = F_u^{(i)}[-1] = 0$. The complexity of the whole algorithm is $O(k^2|p|)$. We omit the detailed analysis; the main point is that we do not fill in the whole table, but only cells with distance $\le k$. Theorem 2 shows our result on the bounds, followed by an example on the algorithm.

Theorem 2. *At each cell of the DP table,* $F_l[j] \le \Pr[D \le j] \le F_u[j]$, *for* $0 \le j \le k$, *where D denotes the edit distance.*

p \ X	C	$G_{.1}A_{.4}T_{.5}$	$G_{.1}A_{.4}T_{.5}$	$G_{.1}A_{.4}T_{.5}$	$G_{.1}A_{.4}T_{.5}$
C	(1, 1)	(0, 0) (1, 1)	(0, 0) (0, 0) (1, 1)		
A	(0, 0) (1, 1)	(.4, .4) (1, 1)	(0, 0) (.64, .64) (1, 1)	(0, 0) (0, 0) (.784, .784)	
T	(0, 0) (0, 0) (1, 1)	(0, 0) (.7, .7) (1, 1)	(.2, .2) (.7, .7) (1, 1)	(0, 0) (.42, .42) (.85, 1)	(0, 0) (0, 0) (.602, .602)

Figure 6 Computing bounds based on CDF.

Example 5. *Consider p = "CAT" and X is "C" followed by four characters, each of which has the same distribution* $G_{.1}A_{.4}T_{.5}$, *denoting that it is G* (*A, T*) *with probability* 0.1 (0.4, 0.5). *The DP table that computes the bounds is shown in Figure* 6 (k =2). *The three empty cells have distances above* 2. *The j'th pair in a cell is* $(F_l[j], F_u[j])$, *for* $0 \le j \le 2$. *When there are fewer than three pairs in a cell, the omitted ones are* (1, 1). *For a deterministic distance value d in a cell* (*e.g., any cell in the 1st row or 1st column*), *as a CDF, we have d pairs of* (0, 0) *followed by a* (1, 1) *as our bounds. E.g., the 3rd cell in the 1st row has a deterministic distance* 2.

Take the cell at the 3rd row and the 4th column as an example. It corresponds to T in p and $G_{.1}A_{.4}T_{.5}$ *in X. The probability that these two characters match is* 0.5 ($p_1 = p_2 = 0.5$ *in ELEMENTARY-STEP*). *First consider* $F_l[j]$. *The* $\text{argmax}_i D_i$ *is* 3 *because its left neighbor* (D_3) *has the highest* $F_l[0]$ (*i.e.,* 0.2). *Therefore,* $F_l[j]$ *is a mixture of the lower bounds in* D_1 *and those in* D_3 (0.5 *weight each*), *giving* 0, 0.42, 0.85 *as the first values of the three pairs. We omit the details on* $F_u[j]$. *Then based on Theorem* 2, *the lower and upper bounds of the edit distance at this cell being no more than* 0, 1, *and* 2 *are* (0, 0), (0.42, 0.42) *and* (0.85, 1), *respectively.*

5.2 Bounds Based on Local Perturbation

We present a different approach for finding the bounds.

Definition 5 (adjacent and remote possible worlds). *Given a* (k, τ)*-matching query on a pattern p and an uncertain text X, we say that a possible world w of X, in which X has a value x, is* adjacent to *p if* $d(p, x) \le k$. *We say that w is* remote to *p if* $d(p, x) > k$. *Finally, we say that w is the* closest (farthest, *respectively*) *if* $d(p, x)$ *is a minimum* (*maximum*) *one amongst all possible worlds.*

Key Idea. Definition 5 implies that answering a (k, τ)-matching query is equivalent to determining if the probability sum of all adjacent worlds is greater than τ. Suppose we start from *one* adjacent world, which has a particular set of *bindings* for all uncertain characters. A *binding* is a commitment of an uncertain character to a fixed value in a possible world. We then apply various *local perturbations*, which are defined as changing a subset of the existing bindings on uncertain characters. After these perturbations, if the resulting possible worlds are still adjacent to p, the sum of their probabilities is a lower bound of $\Pr[d(p, X) \le k]$. In the same vein, if we start from a remote possible world, we can get a probability sum p_r (of remote worlds). Then, $1-p_r$ is an upper bound of $\Pr[d(p, X) \le k]$. Let us first consider how to get an initial adjacent and a remote world to bootstrap the bounds.

Lemma 3. *In Equation* (1) *in Section* 2, *when*

$$c(p[i], X[j]) = \begin{cases} 0, & if\ \Pr(X[j] = p[i]) > 0 \\ 1, & if\ \Pr(X[j] = p[i]) = 0 \end{cases}$$

is used in the DP algorithm, we get a distance value of the closest possible world. For each cell in the optimal path, we bind $X[j]$ *to* $p[i]$ *if* $\Pr(X[j] = p[i]) > 0$; *otherwise we bind* $X[j]$ *arbitrarily. This binding gives us a closest possible world.*

Obtaining an initial remote world. Lemma 3 gives us a closest world, which is certainly an adjacent world (if there is one). We now discuss how to get a remote world. This can be done through a randomized algorithm. We bind each uncertain character *randomly* based on its distribution, and then compute the edit distance. This is repeated until we get a remote world. If we do not get any remote world after it is repeated t times (a parameter discussed below), we return $p_u = 1$ as the probability upper bound.

Let p_0 be the true probability that the distance is no more than k. With probability $(p_0)^t$, we do not find any remote world and return $p_u = 1$, which is $1 - p_0$ higher than the true probability. When p_0 is close to 1, $1 - p_0$ is small; as we decrease p_0, the probability $(p_0)^t$ drops exponentially and quickly diminishes. For example, if $t = 10$ and $p_0 = 0.6$, $(p_0)^t$ is only about 0.006. In other words, if p_0 is very close to 1, returning $p_u = 1$ is acceptable any way; otherwise, with a high probability, $1 - (p_0)^t$, we can reach a remote world within a small number (t) of trials.

The following definition and theorems lead to the bounds.

Definition 7 (crucial variables). *In a DP table of a possible world w, consider a cell D that is on an optimal path* (*to the minimum distance*). *If the path goes from D's upper-left neighbor cell* D_1 *to D and the two cells contain the same distance value, then the uncertain text character that D corresponds to is called a* **crucial variable** *of w.*

Theorem 3. *Let* δ (≥ 0) *be the difference between k and the edit distance in an adjacent world, which has c crucial variables. Let* $p_l = \Pr(at\ least\ c - \delta$ *crucial variables have the same values as in the optimal path*). *Then* $Pr[d(p, X) \le k] \ge p_l$.

Theorem 4. *Consider a remote world that has a distance k' > k and let* $\delta = k' - k - 1$. *If there are u uncertain characters in X, and* $p_r = \Pr(at\ least\ u - \delta$ *uncertain characters have the same values as in the optimal path*), *then* $Pr[d(p, X) \le k] \le 1 - p_r$.

We can further improve the bounds obtained by Theorems 3 and 4 using randomly chosen many initial adjacent and remote worlds; this is presented in Appendix D.

Obtaining p_l and p_r in Theorems 3 and 4. We observe that computing these two probabilities boils down to the following problem: *Suppose there are r events E_i $(i = 1, \ldots, r)$, and we are given* $\Pr[E_i] = p_i$. *What is the probability that at least s events* (*among those r events*) *happen?*

Our solution is as follows. Let $P(i, j)$ denote the probability that, within the last i events, at least j of them happen. Our goal is thus $P(r, s)$. We then have the following recursive equation:

$$P(i, j) = p_{r-i+1} \cdot P(i-1, j-1) + (1 - p_{r-i+1}) \cdot P(i-1, j)$$

This recursion gives us an efficient (dynamic programming) algorithm that can compute $P(r, s)$ in time $O(r \cdot s)$.

Example 6. *Let us continue on Example* 5. *Consider matching p with the first four characters of X and k* = 2 (*the cell at the 3rd row and the 4th column in Fig.* 6). *A chameleon binding CATT gives a distance* 1, *and there are c* = 2 *crucial variables. Applying Theorem* 3, *we get a lower bound* $p_l = 0.7$. *For an upper bound, we first get an initial remote world using our randomized algorithm, say, CTGA, which has a distance* 3. *Applying Theorem 4, we get an upper bound* $p_u = 0.98$. *Comparing with the bounds we get in Example* 5, p_l *is looser while* p_u *is tighter.*

6. EXPERIMENTS

We use two real datasets and some synthetic datasets, which are described in Appendix E, along with our experimental setup. By default, we use: $|p| = 18$, $k = 2$, and $\tau = (1/2)^{|p|}$.

6.1 Query Semantics

We compare (k, τ) queries with EED in [6]. As discussed in Section 3.2, it is well known that the uncertainty ratio has a large variance in biological sequences; we can easily spot places in our datasets where uncertainty is well above average. For the DNA dataset, we arbitrarily pick three patterns of length 15, where the occurrence in the text has 6 uncertain characters. Then we compare a (k, τ) and an EED query in finding these three patterns. In Fig. 1 of Section 1, we have shown the average number of matches for the three patterns over the whole text string (about 7MB) and its prefixes of various sizes. For the (k, τ)-query, we use $k = 1$. As expected, the (k, τ)-query finds the corresponding pattern. For the EED query, we find that we need the distance threshold to be at least 4 to pick up the pattern in the text. With this threshold 4, we report the average number of matches of EED (for the three patterns). From Fig. 1 we see that EED selects about *three orders of magnitude more false positives* than the (k, τ) query. This is because EED has to use a large threshold in order to overcome the uncertainty and catch the real pattern. We observe similar result with the protein dataset, as shown in Appendix F.

6.2 Signatures and Verification Bounds

We verify the effectiveness of our signature filtering and verification bounds by comparing a few approaches: (1) the base method without using our work, (2) a method that does not use signatures, but uses verification bounds based on CDF, (3) no signatures, but verification bounds based on local perturbation (denoted as "bounds 2" in the figures), (4) a method that only uses signatures, but not bounds, and finally (5) a method that uses both signatures and bounds based on CDF.

Fig. 7 shows the result over the whole DNA string and its various prefixes. We can clearly see the differences among various methods. Using signature filtering can improve the performance by about four times, while the effect of using verification bounds is more dramatic − almost 30 times of an improvement. Using signatures saves by doing fast filtering as the index is scanned; we end up with significantly fewer positions to retrieve from the text string and perform the verification. Indeed, we show in Fig. 11 the number of positions in the text strings that remain to be verified before and after the signature filtering is applied, and before and after the verification bounds are applied.

For the bounds based on local perturbation, we try different numbers of repeated random instances to get initial possible worlds. We find that using about t = 11 initial worlds gives the best result. Using more worlds only marginally improves the bounds due to the overlaps of variable settings between two possible worlds (details in Appendix D). We always incrementally find a good t value to use (until a bigger t does not improve the bounds significantly). Moreover, Fig 7 indicates that using CDF based bounds and using perturbation based bounds result in similar performance improvements (with the CDF based bounds being slightly better). Thus, in the sequel, with the exception of Fig. 8, we only show the results of CDF based bounds. Our additional experiments on the protein data are reported in Appendix F.

We then observe the behaviors as we vary θ, the ratio of uncertain characters. In the original dataset, θ is about 0.16. Starting from the original data, we synthetically add or reduce uncertainty by converting random deterministic characters to probabilistic ones or the other way around. We show the results in Fig. 8 for different θ (using the same legend as Fig. 7). First, we notice that perturbation based bounds get better than the CDF based bounds as θ increases. This is because as the number of uncertain characters increases, variance increases, and we are more likely to get a better-quality initial possible world (i.e., with an edit distance far from k), which results in better bounds. Thus, perturbation based bounds should be used when the uncertainty level is high.

Furthermore, with all methods, we see an increase of execution time as θ gets bigger due to the extra cost of handling uncertain characters. In particular, the increase is more dramatic when we do not use verification bounds. This is because the verification has to go through all possible worlds, the number of which increases exponentially with the number of uncertain characters. In these experiments, the base methods (no signatures, no bounds) are very expensive, with running time up to an hour.

In the next experiment, we synthetically generate larger text files (up to 6 GB) by extending the original DNA dataset with deterministic and uncertain characters that follow the same distributions. For this experiment, we only compare the running times of the two methods that use verification bounds (with and without using signatures), as the methods that do not use bounds take a very long time. By comparing these two methods, we can see the effects of signature filtering in terms of I/O and CPU costs. Fig 9 shows the elapsed times of both methods, and we provide a breakdown of total costs into I/O and CPU costs in Fig. 10 for 2 GB and 4 GB texts. Overall we see similar trends as the case with smaller text strings, and the CPU cost is close to the I/O cost as a result of handling uncertain characters.

We then vary additional parameters. Using the real datasets, we search with patterns of different lengths ($|p|$) from 10 to 50. Fig 12 shows the result for the DNA dataset and Fig 13 for the protein dataset (they share the legend in Fig 13). In Fig 12, the execution times increase as $|p|$ gets bigger because more uncertain characters in the text strings need to be verified. We have a sharper increase for the two methods that do not use bounds because they have to

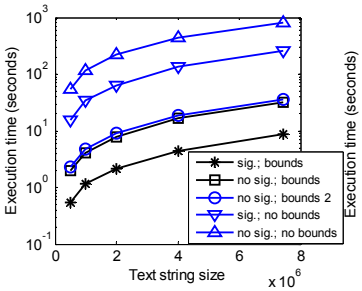

Fig. 7 Running time for various settings.

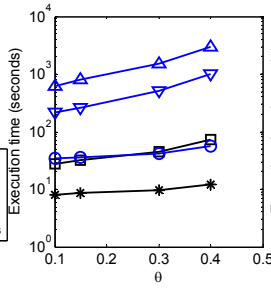

Fig. 8 Varying θ.

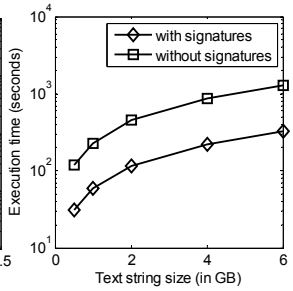

Fig. 9 Using larger synthetic data.

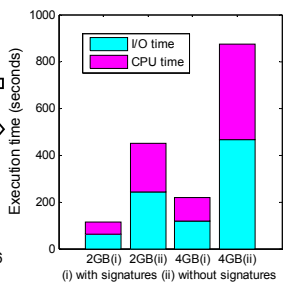

Fig. 10 Breakdown of I/O & CPU costs.

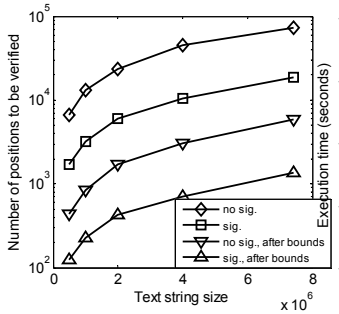

Fig. 11 # of positions to be verified.

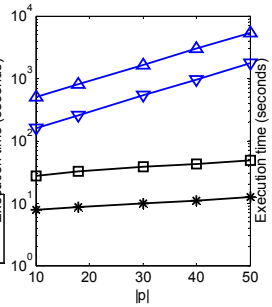

Fig. 12 Varying $|p|$ (DNA).

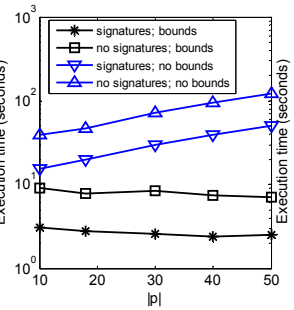

Fig. 13 Varying $|p|$ (protein).

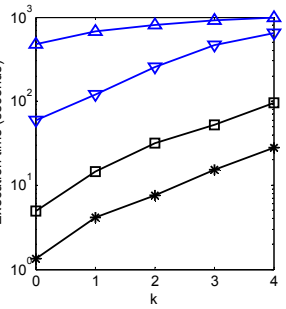

Fig. 14 Varying threshold k.

iterate over all possible worlds, the number of which exponentially increase with more uncertain characters.

An interesting phenomenon in Fig 13 (protein) is that, while the growth of elapsed times grows slower than in the DNA dataset for the two methods that do not use bounds, the execution times actually *decrease* (as $|p|$ increases) for the other two methods. The reason is as follows. Recall that we break p into $k + 1$ pieces and uses each piece to first find an exact match. As $|p|$ increases, each piece also increases, which can filter out some more positions (but after retrieving the substrings in the text at corresponding occurrences). This indeed somewhat offsets the increased costs due to more uncertain characters to verify in the text. For the protein dataset, this extra filtering effect is more significant due to its larger alphabet size (i.e., more selective per character). As a result, it actually becomes faster with a bigger $|p|$. Additionally, by comparing Figures 13 and 12, we see that the searches are in general faster with the protein dataset for the same reason.

Finally, we vary k, the edit distance threshold, whose results are shown in Fig 14 for the DNA dataset, using the same legend as Fig 13 (similar trend appears in the result of proteins in Fig A.5 of Appendix). We see that all methods run slower with bigger k values. This is because, as k increases, the matching is more relaxed and we need to examine more positions and uncertain characters. One might wonder why the execution time also increases (although less dramatically) with the base method, i.e., the case of no signatures and no bounds. Don't we have to iterate through all possible worlds regardless of k values? The reason for the increase is that, as k grows, each of the $k + 1$ pieces (that we use to first perform an exact match) becomes shorter. Consequently, more positions in the text string will need to be examined.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we study a real and unsolved problem of approximate pattern matching over uncertain texts. We propose a novel semantics and demonstrate its advantages over an alternative one introduced by previous work. To answer such a query, we enhance q-gram based indexing to handle uncertain texts and propose efficient verification algorithms. As future work, we plan to study the matching problem under correlated uncertainty to address a wider range of applications.

Acknowledgements: This work was supported in part by the NSF, under the grant IIS-1017452. We wish to thank the anonymous referees for their helpful comments.

8. REFERENCES

[1] C. Cantor, C. Smith. *Genomics: The Science and Technology Behind the Human Genome Project*. Wiley, 1999.
[2] S. Chaudhuri, V. Ganti, R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5-16, 2006.
[3] S. Dash, K. Chon, S. Lu, E. Raeder. Automatic Real Time Detection of Atrial Fibrillation. In *Ann Biomed Eng.*, 37(9), pp. 1701-9, 2009.
[4] B. Ewing et al.. Base-calling of automated sequencer traces using phred I accuracy assessment. In *Genome Res.*, 8(3):175-85, 1998.
[5] L. Gravano et al. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491-500, 2001.
[6] J. Jestes, F. Li, Z. Yan, K. Yi. Probabilistic String Similarity Joins. In *SIGMOD*, pages 327-338, 2010.
[7] R. Kandhan, N. Teletia and J. M. Patel. SigMatch: Fast and Scalable Multi-Pattern Matching. In *PVLDB*, 3(1), pages 1173-1184, 2010.
[8] M. Kircher, J. Kelso. High-throughput DNA sequencing – concepts and limitations. In *BioEssays*, 32, pages 524-536, 2010.
[9] C. Li, J. Lu, Y. Lu. Efficient Merging and Filtering Algorithms for Approximate String Searches. In *ICDE*, pages 257-266, 2008.
[10] G. Navarro. A Guided Tour to Approximate String Matching. In *ACM Computing Surveys*, Vol. 33, No. 1, pages 31–88, 2001.
[11] G. Navarro et al. Indexing Methods for Approximate String Matching. In *IEEE Data Eng. Bulletin*, Vol. 24, No. 4, pages 19-27, 2001.
[12] G. Navarro, L. Salmela. Indexing Variable Length Substrings for Exact and Approximate Matching. In *SPIRE*, pages 214-221, 2009.
[13] Nomenclature Committee of the International Union of Biochemistry (NC-IUB). Nomenclature for incompletely specified bases in nucleic acid sequences. In *Biochem J.*, 229(2), pages 281-6, 1985.
[14] S. Tata, J. M. Patel, J. Friedman, A. Swaroop. Declarative Querying for Biological Sequences. In *ICDE*, pages 87-98, 2006.
[15] The U. of Michigan DNA Sequencing Core. *Interpretation of Sequencing Chromatograms*. http://seqcore.brcf.med.umich.edu/
[16] E. Ukkonen. Finding approximate patterns in strings. In *Journal of Algorithms*, 6, pages 132-137, 1985.
[17] http://www.ncbi.nlm.nih.gov/Traces/home/.

APPENDIX

A. ADDITIONAL COMPARISON BETWEEN (k, τ) AND EED

Define a random variable D as the edit distance between a pattern p and a target substring X in text, i.e., $D = d(p, X)$. Suppose that X's value in the *true possible world* (i.e., without uncertainty) is x, and that x is an approximate match that the user is interested in, i.e., $d(p, x) = k$, for a small k. Let $\Pr(D = k) = \alpha$. Then, for any $0 < \tau < \alpha$, a (k, τ)-matching query will find X. On the other hand, the expectation $E[D]$ could be very large (certainly much larger than k), for an arbitrary distribution of D. This can be seen from Example 1 in Section 3.

However, conversely, if $E[D] < \delta_1|p|$, for a small $0 < \delta_1 < 1$, we show that a (k, τ)-matching query will always catch the match, where $k = \delta_2|p|$, $\delta_2 = \left[1 + {}^{1}/_{(2^{|p|} - 1)}\right]\delta_1$, and $\tau = 2^{-|p|}$. Recall that, for $|\Sigma| > 2$, such a τ value is greater than pure chance. For a practical $|p|$ value, we have $\delta_2 \approx \delta_1$. For instance, if $|p| = 10$, then $\delta_2 < 1.001\delta_1$. To prove this, from Markov's inequality, we have:

$$\Pr[D > \delta_2|p|] < \frac{E[D]}{\delta_2|p|} < \frac{\delta_1|p|}{\delta_2|p|} = \frac{\delta_1}{\delta_2}$$

Hence:

$$\Pr[D \le \delta_2|p|] > 1 - \frac{\delta_1}{\delta_2}$$

Letting $k = \delta_2|p|$ and $\tau = 2^{-|p|}$, we have that, for $\delta_2 = \left[1 + {}^{1}/_{(2^{|p|} - 1)}\right]\delta_1$, a (k, τ)-matching query must be satisfied.

B. PROOFS OF LEMMAS AND THEOREMS

Proof of Lemma 1: Let two strings be s_1 and s_2, and their signatures be σ_1 and σ_2. Let $d(s_1, s_2) = d$. By definition, using d edit operations, we can transform s_1 to s_2. If characters c_1 and c_2 are equal in the original alphabet, then $h(c_1) = h(c_2)$, where h is the hash function used for signatures. Therefore, using the same sequence of edit operations (that transform s_1 to s_2) on σ_1, except that all characters in the original alphabet are replaced by the corresponding hash values, we get σ_2. This implies that $d(\sigma_1, \sigma_2) \le d$. □

Proof of Theorem 1: Recall that the verification of an occurrence in x consists of two symmetric runs of DP; it suffices to only consider one of them. Suppose one run tries to match substring p' from p with substring x' from x where $|x'| = |p'| + k$. We are interested in a best matching prefix of x' for p'. The DP proceeds as shown in Figure 5(a), where we denote $|p'|$ as m'. For verification, we need the minimum distance value within the shaded region at the bottom row of Figure 5(a), i.e., the cells at the row for $p'[m']$ and between the columns for $x'[m'-k]$ and $x'[m'+k]$. Let the minimum value in this region be d_1 for run 1 and d_2 for run 2. Clearly, the verification requires $d_1 + d_2 \le k$.

Now, Inequality (4) of Lemma 2 states that moving up in the diagonal direction (as illustrated by the dashed lines in Figure 5(a)) cannot increase a distance value. Thus, there exists a value that is no more than d_1 (or d_2 for run 2) in the shaded region at the column for $x'[l]$, between the rows for $p'[l-k]$ and $p'[l+k]$ in Figure 5(a). Moreover, Lemma 1 indicates that the distances cannot get greater with signatures. Therefore, replacing l by l_1 for run 1 and by l_2 for run 2, the success of verification implies $d_l + d_r \le k$. □

Proof of Theorem 2. First consider a possible world ω_1 in which $C = c$. Let the distance values at cells D and D_i $(1 \le i \le 3)$ be v and v_i, respectively. We claim that $v = v_1$. This is because Lemma 2 shows that $v_2, v_3 \ge v_1 - 1$; thus, $v = \min(v_1, v_2 + 1, v_3 + 1) = v_1$. Next, consider a possible world ω_2 in which $C \ne c$. From Equation (1), $v = \min_i(v_i) + 1$. In line 4 of the algorithm, by using $\text{argmin}_i D_i$, we essentially pick one *fixed* neighbor cell (i.e., the one that has a small distance value with the highest probability) to use for *all* possible worlds in which $C \ne c$; and hence the true v value could be smaller than this one in some possible worlds. Thus, the $F_l[j]$ in line 4 is a *lower* bound of $\Pr[D \le j]$ (i.e., we may have a smaller distance with a higher probability).

For $F_u[j]$, in line 5, first consider the possible worlds in which $C \ne c$. By Equation (1), D should be the minimum of the three neighbors' distances plus one. Thus, $\Pr[D \le j]$ is the probability that at least one of its three neighbors has distance no more than $j - 1$. Then from *union bound*, this is at most $\sum_{i=1}^{3} F_u^{(i)}[j-1]$. For the case of $C = c$, the reasoning is the same as in $F_l[j]$. Therefore, $F_u[j]$ in line 5 gives an upper bound of $Pr(D \le j)$. □

Proof of Lemma 3. The binding described in Lemma 3 is equivalent to the following: we bind $X[j]$ to $p[i]$ whenever one of its alternatives is $p[i]$; otherwise we bind $X[j]$ arbitrarily. We call it a *chameleon binding*. We say it is an *inverse chameleon binding* if we bind $X[j]$ to any value other than $p[i]$ whenever possible.

In addition to the result in this lemma, we also prove that an inverse chameleon binding, however, merely gives an upper bound of the distance in the farthest world at each cell of the DP table.

First, we can prove that a chameleon binding (inverse chameleon binding) gives a lower (upper) bound for the distance value at each cell of the DP table in a closest (farthest) possible world by induction, similar to the proof of Theorem 5 in [6]. Thus we omit the details. We next show that a chameleon binding actually gives the *exact values* of a closest world. For this, we only need to show that for *any given cell* of the DP table, there exists a *consistent* binding (i.e., binding each uncertain character to only one value) that produces the value for that cell.

We prove this by induction. We show that if the optimal paths to a cell D's three neighbors D_1 (upper left), D_2 (upper), and D_3 (left) are all consistent, so is the optimal path to D (the starting boundary condition is trivially true). There are two cases: (1) If the text and pattern character at D have a non-zero probability of a match, we bind the uncertain text character C to that pattern character (according to chameleon binding) and mark the path going from D_1 to D. This is valid because of Lemma 2, and the argument is the same as that in the proof of Theorem 2. The path to D is a consistent binding because C has not been bound to anything in the path to D_1. (2) If the text character C and the pattern character have no chance to match at D, C is free and can be bound to anything in its distribution. This clearly also produces a consistent path to D. □

Proof of Theorem 3. Consider the optimal path π in the DP table. π encodes a sequence of edit operations that transforms one string to the other. Changing the value of one crucial variable V increases the cost of π by one (by changing a match of cost 0 to a substitution of cost 1). As a result, changing δ crucial variables increases the cost of π by δ. Moreover, changing a non-crucial variable does not change the cost of π. Therefore, if at least $c - \delta$ crucial variables maintain their values in the crucial path, the cost of π stays within k. Since there is at least one path (namely, π) that has a cost within k, the edit distance must be within k. Thus,

aggregating the probabilities of such possible worlds gives a lower bound. □

Example A.1. *Figure* A.1(a) *shows chameleon binding* (*the first values in each cell*) *and inverse chameleon binding* (*the second values*) *using the same strings as Example* 5. *The chameleon binding gives values in the closest world while the inverse chameleon binding only gives upper bounds on the distances of the farthest world. Observe that the results here are consistent with the bounds in Example* 5 (*Figure* 6). *While the bounds given by the inverse chameleon binding are achievable in Figure* A.1(*a*), *this is not always the case. A counterexample is given in Figure* A.1(*b*). *The final value,* 3 (*in the solid circle*), *is not achievable. The value* 3 *is derived based on the condition that the uncertain character C in X is bound to "T". However, when C is bound to "T", the value in the cell above* (2 *in the dashed circle*) *can only be* 1, *due to a match, which in turn makes the final value* 2. *It can be easily verified that the farthest world only has distance* 2.

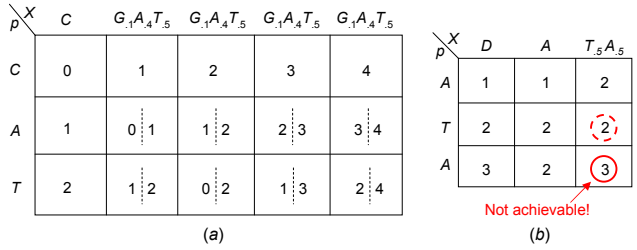

Figure A.1 Illustrating chameleon bindings.

Proof of Theorem 4. We first prove the following lemma: Changing the binding of one character in string X can at most change its edit distance from p by one. Let the two instances of X be x_1 and x_2 that only differ in one character. Let $d(p, x_1) = d$. Without loss of generality, suppose $d(p, x_2) \geq d + 2$. By definition, we can use d edit operations to transform p to x_1. However, this implies that we can use $d + 1$ edit operations to transform p to x_2 by first transforming p to x_1, and then with an extra substitution to further transform it to x_2. Thus, $d(p, x_2) \leq d + 1$, a contradiction.

From the above lemma, after changing the values of δ uncertain characters, the edit distance is still above k. Thus, p_r is a lower bound of the probability of all remote worlds and $1 - p_r$ is an upper bound of the probability of an approximate match. □

C. ADDITIONAL DETAILS ON PATTERN EXHAUSTION

As shown in Figure 5(b), when the text signature length l extends beyond $m'-k$ (where m' is the pattern length), we cannot extend the pattern dimension to be k characters longer. Instead, as indicated by the dark region in Figure 5(b), we check the *prefix distance of the text from the pattern* (d_1) and the *prefix distance of the pattern from the text* (d_2), and use $\min(d_1, d_2)$ as d_l (d_r) in line 8 of the PUSH-POP-SIGNATURE-CHECK algorithm. We stop extending the text when l reaches $m'+k$.

D. ADDITIONAL IMPROVEMENT ON PERTURBATION BASED BOUNDS

We can further improve the bounds by perturbing a different initial world (either adjacent or remote), and by summing up the probabilities of any *new* worlds discovered this way.

Imagine that we run multiple instances of the procedure described in Theorems 3 and 4, except that we start from a different world for each instance. This can be achieved by simply binding each uncertain character randomly based on its distribution, and putting it either in a set of adjacent worlds W_1 or a set of remote worlds W_2 based on its distance. Repeating this w times and using a probabilistic argument similar to the one in Section 5.2, it is unlikely for W_1 or W_2 to be empty unless the actual probability is really close to 0 or 1, in which case our bounds are already very tight.

After having W_1 and W_2, we bootstrap the worlds around them, as in Theorems 3 and 4. However, we cannot simply add up the probabilities produced by each instance in W_1 (or W_2) because the possible worlds generated from two instances may have an overlap. We now solve this problem.

Theorems 3 and 4 can both be abstracted to the following common stochastic model, which we call $\boldsymbol{M_1}$:

(1) We start with a set of random variables $\{C_i\}$ $(1 \leq i \leq r)$;

(2) Each random variable C_i has probability p_i to be a chosen value v_i $(1 \leq i \leq r)$;

(3) At most δ of the random variables can be free variables, while the rest must be their corresponding v_i's as in (2).

We are interested in the probability of the event in (3). Consider that we start from two initial worlds and get two instances I_1 and I_2, both of which follow model M_1. We thus have two sets of random variables in (1). When each of those variables (C_i) has value v_i as in (2), let d be the number of common random variables that have different values in I_1 and I_2. Let I_1 (I_2) have the parameter value δ_1 (δ_2) in step (3) of M_1. Clearly, if $\delta_1 < d$ and $\delta_2 < d$, there must be at least one random variable that has different values between any possible world that I_1 generates and any one that I_2 generates. Therefore, when this condition is met, the two sets of possible worlds must be distinct, and we can simply add up their probabilities of event (3) towards the final bounds.

Example A.2. *Continuing on Example* 6, *for the upper bound, suppose we start another instance from an initial world CGGA, which also has a distance of* 3 *from p. Recall that the first instance in Example* 6 *is CTGA. Thus, the aforementioned parameter* $d = 1$ *and* $\delta_1 = \delta_2 = 0 < d$. *Therefore, we can add up the probabilities from the two instances and get an improved upper bound* 0.976.

We can further generalize this idea across more than two instances. We iterate over them one by one and accumulate a set of mutually exclusive instances. We check if the above condition (i.e., $\delta_1 < d$ and $\delta_2 < d$) is satisfied between the current instance and each one already in the set. If so, we add the current instance to the set. In the end, we add up the probabilities of all instances in the set and get the lower/upper bound.

E. DATASETS AND SETUP OF EXPERIMENT

We implement all algorithms described in the paper, as well as the EED bounding algorithms and the basic EED algorithm presented in [6] for comparisons. The experiments are performed on a machine with an Intel Core 2 Duo 1.66Ghz processor and a 1GB RAM. All experiments start with a cold buffer cache. We use two real datasets and a few synthetic datasets as described below.

The DNA dataset. We download the raw datasets of *sequencing runs* of Escherichia coli 536 from the NCBI SRA (Sequence Read Archive) database [17]. Then we use Bowtie [20] to align the short DNA sequences with the complete Escherichia coli genome reference, which is downloaded from the NCBI Genome database [21]. The mapping reports output by Bowtie show positions with-

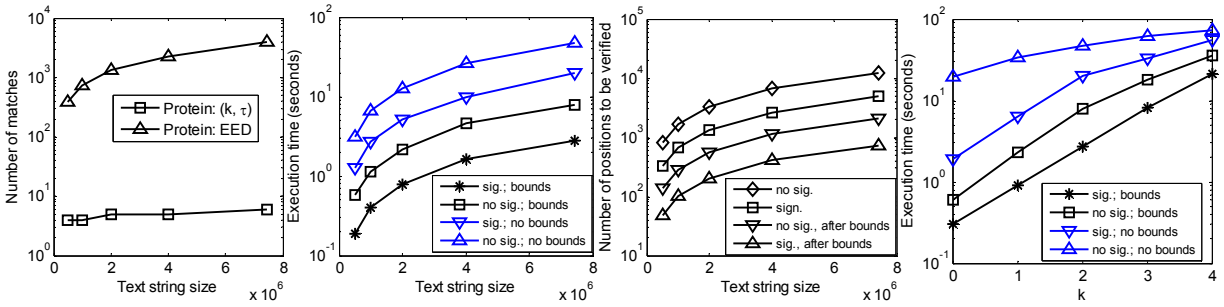

Fig. A.2 Matches of (k, τ) vs. EED. Fig. A.3 Running times for proteins. Fig. A.4 #positions to verify (protein). Fig. A.5 Varying k (protein).

in the DNA that have more than one possible value. As studied in [19], the nucleotide variation observed in the sequencing reads comes from errors introduced by sequencing procedures. From the Bowtie reports, we naturally model uncertain characters with a discrete distribution based on the frequencies of alternative values among the sequencing runs.

The protein dataset. The protein dataset is a collection of protein sequences downloaded from Uniprot [22]. Like DNA, protein data has a great deal of uncertainty [18]. Uniprot provides descriptions of uncertain characters in the proteins, from which we can learn the alternative values of an uncertain character and their frequencies. For example, at [23], the "Amino acid modifications" and "Natural variations" sections contain the uncertainty information for one particular protein, Insulin. We incorporate such uncertainty information into the strings.

Synthetic datasets. We also generate a few synthetic datasets based on the two real datasets above. The reason for using synthetic datasets is that, in some experiments, we would like to vary the parameter values of data (such as the uncertainty ratio θ) or the size of the data. We will describe how we generate specific datasets when we describe the specific experiments.

In addition, unless otherwise specified, we examine the behaviors of a (k, τ)-pattern matching query using a pattern string p, with the following default parameter values: $|p| = 18$, $k = 2$, and $\tau = (1/2)^{|p|}$. As each internal node of an index tree is an array with fixed size entries, we dense-pack the internal nodes in level order. All the leaf nodes are stored contiguously. The overhead of incorporating signatures and uncertain characters when building an index for text strings is proportional to the increase in index size, as discussed in Section 4. However, this is a one-time cost, as indexes in these applications are designed to be built once and queried many times; thus query performance is of primary concern. Typically these large text datasets are rarely updated.

F. ADDITIONAL EXPERIMENTS

Comparison of false positives between *EED* and (k, τ)-queries. In Fig. A.2, we also show the result of this comparison from the protein dataset. We use three patterns of length 9, each having 3 uncertain characters in the text. We report the average number of matches of a (k, τ) query and an EED query over the text strings of various sizes. We need the distance threshold of EED to be at least 3 in order to recognize the three patterns. Fig. A.2 again shows that EED selects about three orders of magnitude more false positives than the number of matches of the (k, τ) query.

As an aside, comparing the two experiments (Figures 1 and A.2), we notice that, to get roughly the same number of matches from EED, we need a shorter pattern (and a smaller threshold) for the protein dataset. This is because the protein strings have a larger alphabet size (20) than DNA strings (4), which makes a pattern more selective.

Experiments with different settings on protein dataset. Using the same parameters as on DNA, we experiment on the protein dataset, the result of which is shown in Fig. A.3. We can see that, while the trend is about the same as the DNA dataset (Fig. 7), the execution times are on a smaller scale (about an order of magnitude faster). This is because proteins have a larger alphabet size. As a result, patterns with the same length (and with the same k parameter − the distance threshold) are more selective, which leaves fewer positions to be verified. We show the numbers of positions to be verified for various methods on the protein dataset in Fig. A.4, which are considerably fewer than in the DNA dataset − Fig. 11.

Executions times when varying parameter k on proteins. The experimental results are shown in Figure A.5. Compared to the results on DNA dataset in Figure 14, we observe that the running time increase as k grows becomes more pronounced with the protein dataset due to its larger alphabet size.

G. ADDITIONAL REFERENCES

[18] N. Bandeira, K. Clauser, P. Pevzner. Shotgun Protein Sequencing: Assembly of MS/MS Mass Spectra from Mixtures of Modified Proteins. In *Molecular and Cellular Proteomics*, 6:1123-34, 2007.

[19] W. Huang, G. Marth. EagleView: A genome assembly viewer for next-generation sequencing technologies. In *Genome Res.*, 18(9):1538-43, 2008.

[20] http://bowtie-bio.sourceforge.net/index.shtml.

[21] http://www.ncbi.nlm.nih.gov/sites/entrez?db=genome.

[22] http://www.uniprot.org/.

[23] http://www.uniprot.org/P01308/.