k-Approximate Search in Probabilistic Strings

Exposé for Diploma Thesis

Tobias Mühl

HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT II
INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR WISSENSMANAGEMENT IN DER BIOINFORMATIK

Academic advisors: Prof. Dr. Ulf Leser
Dr. Sebastian Wandelt

1 Motivation and background

Strings over the alphabet $\Sigma = \{A, C, G, N, T\}$ are used to represent DNA sequences in bioinformatics. These sequences can be very long, for instance, a genome of a human consists of more than three billion base pairs, which means that naive string representations are often very long. Indexing and searching such long DNA sequences is an essential problem in bioinformatics and very memory intensive. During read mapping, for instance, small sequences obtained from shotgun-sequencing, so-called reads, are searched in a reference genome.

Exact search for small strings in large strings has been widely explored in computer science. In general, two scenarios can be distinguished: 1) search using an index structure (offline) and 2) search without an index structure (online). In offline search algorithms, the long string is preprocessed, in order to reduce search time during the actual search phase. There exist many indexes for exact substring search in long strings, such as suffix trees [4], suffix arrays [8], n-gram indexes [16], or prefix trees [15]. Online search algorithms on the other hand are applied if only few queries exists or the index structure is too large to fit into (main) memory.

Often, exact search for strings is not sufficient. For instance, sequencing errors are introduced during shotgun sequencing, i.e. some bases of sequences are wrongly identified, missed out, or accidentally inserted. In addition, DNA changes over time due to mutation and recombination. To assess the similarity of two strings, many similarity/distance functions were proposed in the past: Examples are edit-distance (also called Levenshtein distance as introduced by Levenshtein in 1965), Jaccard distance, Haming distance. In this thesis, we focus on the edit-distance.

In order to find strings within a given edit-distance inside a reference string, one needs methods for approximate string search. Most of these methods are based on the seed-and-extend approach [1] (also called filter-and-verification). The idea of seed-and-extend is to break up query strings into smaller parts, locate all exact matches for the smaller parts in the reference string, and then extend all candidate matches to find out, whether they become full approximate matches.

Sequence data in bioinformatics is often uncertain and probabilistic. For instance, reads in shotgun sequencing are annotated with quality scores for each base. These quality scores can be understood as how certain a sequencing machine is about a base. Probabilities over long strings are also used to represent the distribution of SNPs or InDels (insertions and deletions) in the population of a species. These frequency distributions are usually obtained by multiple sequence alignment methods [2, 13].

Indexing and searching such long probabilistic strings was investigated only recently. In [7], Jestes et al. present two models for probabilistic strings: a character-level and a string-level model. The character-level model annotates probabilities for each character of the string whereas in the string-level model probabilities are given for each possible instantiation of a whole string (called *possible world*). Jestes et al. focus on approximative matching of two probabilistic strings. Based upon the character-level model, Ge et al. contribute a further algorithm to enable substring search on uncertain text in [6] and develop filter and pruning techniques.

2 Goal

The primary goal of this thesis is to design and implement a k-approximate substring search algorithm for probabilistic strings. The existing string-level model is appropriate for very short strings only, due to combinatorial explosion when combining probabilistic parts. The character-level model is less expressive than the string-level model, since conditional probabilities between characters cannot be modeled. We will develop a new model for probabilistic strings: the so-called *chunk-level* model. It is a hybrid form of character-level and string-level models. In a chunk-level model a string consists of a sequence of chunks that are themselves string-level probabilistic strings.

During the first part of this thesis, we develop and implement a transformation method to convert probabilistic strings into the chunk-level model. In a second step, we will design an algorithm for searching probabilistic strings represented in the chunk-level model. We hope to benefit from the fact that individual chunks can be searched independently.

Finally, we implement our string transformation and search algorithm in C++. The results will be compared to the performance of the gold-standard given by Ge et al in [6]. The resulting search algorithm should be correct, i.e. find all k-approximate matches. The main criteria of evaluation will be search time. For the evaluation we use several test cases with different types of data and different degrees of probability.

3 Technical Preliminaries

A *string* s is a finite sequence over an alphabet Σ. The length of a string s is denoted by $|s|$ and the substring starting at position i with length n is denoted by $s(i, n)$. We write $s(i)$ as an abbreviation for $s(i, 1)$. All positions in a sequence are zero-based, i.e., the first character is accessed by $s(0)$. The concatenation of two strings s_1 and s_2 is denoted $s_1 \circ s_2$. The length of a string s is given by $|s|$.

Definition 1 (edit-distance)**.** *Given strings s and t, s is called k-*approximately similar to *t, denoted $d(s, t) \leq k$, if s can be transformed into t by at most k edit operations. The edit operations are: replacing one symbol in s, deleting one symbol from s, and inserting one symbol into s.*

In [7], Jestes et al. present two models for probabilistic strings: a character-level and a string-level model. We repeat both definitions here for convenience of the reader.

Definition 2 (string-level probabilistic string [7])**.** *A* string-level probabilistic string *is a collection $S = \{(\sigma_1, p_1), ...(\sigma_m, p_m)\}$, where $\sigma_i \in \Sigma^*$, $p_i \in (0, 1]$, and $\sum_{i=1}^{m} p_i = 1$. S instantiates into σ_i with probability p_i. $Pr(i) = p_i$ returns the probability of the possible world (σ_i, p_i).*

Definition 3 (character-level probabilistic string [7])**.** *A* character-level probabilistic string *is $S = S[1]...S[n]$. For $i = 1, ..., n$, $S[i] = \{(c_{i,1}, p_{i,1}), ...(c_{i,m_i}, p_{i,m_i})\}$ where $c_{i,j} \in \Sigma$, $p_{i,j} \in (0, 1]$, and $\sum_{i=1}^{m_i} p_{i,j} = 1$. Each $S[i]$ instantiates into $c_{i,j}$ with probability $p_{i,j}$ independently.*

Definition 4 (possible world [7])**.** *A* possible world *of a probabilistic string S given in any model is a possible instantiation of S. In the string-level model $|S| = m$ denotes the number of* possible worlds *for a string S.*

Example 1 (probabilistic strings). $S_1 = \{(ATTG, 0.15), (ATTC, 0.35), (CTTG, 0.15), (CTTC, 0.15)\}$ *is a string-level probabilistic string with four possible worlds. The world* $ATTG$ *has a probability of 0.15. A character-level probabilistic string is* $S_2 = S_2[1]...S_2[4]$*, with* $S_2[1] = \{(A, 0.5), (C, 0.5)\}$*,* $S_2[2] = \{(T, 1.0)\}$*,* $S_2[3] = \{(T, 1.0)\}$*, and* $S_2[4] = \{(G, 0.3), (C, 0.7)\}$.

The two probabilistic strings S_1 and S_2 from Example 1 are equivalent. It is easy to see that each character-level string can be transformed into a string-level string by exhaustively enumerating all possible worlds from the character-level string (similar task to computing the cross product). However, the transformation from string-level probabilistic strings to character-level probabilistic strings is not always possible. For instance, the string $S_1 = \{(AC, 0.5), (CA, 0.5)\}$ cannot be expressed as a character-level string. Therefore, the string-level model is clearly more expressive than the character level model. Below, when we introduce our new model for probabilistic strings, we only describe how to process string-level probabilistic strings. Character-level probabilistic strings are handled by transformation into string-level probabilistic strings first.

Definition 5 (Approximate query/matches for deterministic strings). *A* k-approximate query *is a pair* (q, k)*, where* q *is a query string and* k *an edit distance threshold. Given a deterministic string* s *and* (q, k)*, the set of* k-approximate matches *is defined as* $MP((q, k), s) = \{(u, pos) \mid \exists v, w : v \circ u \circ w \equiv s \wedge d(q, u) \leq k \wedge pos = |v| + 1\}$ *with the matching positions function. Furthermore we write* $q \sim_k s$ *if and only if the query* (q, k) *has at least one match in* s*:*

$$q \sim_k s \Leftrightarrow \exists u, v, w : v \circ u \circ w \equiv s \wedge d(q, u) \leq k$$

Definition 6 and 7 describe two different types of approximate queries for string-level probabilistic strings: The deterministic one (Definition 6) works on the possible worlds of S and returns all positions where the search pattern occures. The probabilistic query (Definition 7) is boolean and answers whether a string matches the query with a given probability or not.

Definition 6 (Approximate query/matches for probabilistic strings). *Given a string-level probabilistic string* $S = \{(\sigma_1, p_1), ...(\sigma_m, p_m)\}$ *and a query* (q, k)*, the set of* k-approximate matches *is defined as the union of all* k*-approximate matches for each possible world of* S*:* $MP((q, k), S) = \{(pwid, u, pos) \mid \exists(\sigma_{pwid}, p) \in S, v, w : v \circ u \circ w \equiv \sigma_{pwid} \wedge d(q, u) \leq k \wedge pos = |v| + 1\}$.

Definition 7 (Approximate probabilistic queries/matches). *A* k-approximate probabilistic query *is a pair* (q, k, τ)*, where* q *is a query string,* k *an edit distance threshold, and* τ *a probability threshold. We define a function for the matching possible worlds* $MPW((q, k, \tau), S) = \{pwid \mid \exists(\sigma_{pwid}, p) \in S, u, v, w : v \circ u \circ w \equiv \sigma_{pwid} \wedge d(q, u) \leq k\}$*. Given a string-level probabilistic string* $S = \{(\sigma_1, p_1), ...(\sigma_m, p_m)\}$*,* S matches (q, k, τ)*, if and only if the sum of the probabilities of the possible worlds that matches is* $\geq \tau$*:*

$$(q, k, \tau) \sim S \Leftrightarrow \tau \leq \sum_{pwid \in MPW} Pr(pwid)$$

Example 2 (queries/matches for probabilistic strings). *Assume that we a re given a string-level probabilistic string* $S = \{(ATTG, 0.15), (ATTC, 0.35), (CTTG, 0.15), (CTTC, 0.15)\}$*. Given the approximate query* $(AT, 0)$*, possible worlds* $(ATTG, 0.15)$ *and* $(ATTC, 0.35)$ *have a match at position* 1 *and thus* $MP((AT, 0), S) = \{(1, AT, 1), (2, AT, 1)\}$*). Given the* k*-approximate probabilistic query* $(AT, 0, 0.5)$*,* S *matches* $(AT, 0, 0.5)$*, since* $MPW((AT, 0, 0.5), S) = \{1, 2\}$ *and* $Pr(1) + Pr(2) = 0.15 + 0.35 = 0.5 \geq \tau$.

4 Related Work for approximate Search in probabilistic strings

Approximate search using different indexing methods is well described in [17], [1], [9], [11] and [12]. The often used strategy is to partition the search string into $k + 1$ pieces, where k is the query's edit-distance. At least one of those pieces must have an exact match in the index. How the search string is partitioned in an optimal way is, for instance, described in [12] and [10].

For probabilistic strings, Jestes et al. define the strings-level model (among the character-level model) in [7]. They also provide an algorithm for matching two string-level probabilistic strings with respect to similarity. Both strings are indexed with q-grams, so that every possible world of the strings is indexed in a deterministic way. To determine the similarity of two uncertain strings S_1 and S_2, they define the expected edit-distance (EED). The EED is given by the sum of the distances of all possible worlds of S_1 and S_2 multiplied with their probabilities. Two strings S_1 and S_2 matches if their EED is below a threshold parameter τ. Because calculating the exact EED is expensive, [7] introduces pruning techniques based on the strings' length and intersections of the sets of q-grams of both strings.

Furthermore, Jestes et al. also provide an algorithm to match two probabilistic strings given in the character-level model [7]. Therefore two character-level probabilistic strings S_1 and S_2 are indexed with q-grams. In difference to the string-level model, every index entry has a dedicated probability and several index entries can exist for one position in the string. The EED is used again as a measurement for probable similarity. Since it is still expensive to compute the EED for two probabilistic strings, Jestes et al. present further pruning techniques. The first lower bound calculates a minimum value for the EED based on the similarity of S_1's and S_2's probabilistic and deterministic q-grams. The second lower bound is given by a modification of the dynamic programming (DP) algorithm to compute the edit-distance. Thereafter two upper bounds are applied, corresponding to the lower bound strategies. The first one compares the q-grams of both strings while the second one computes the worst case of a modified DP algorithm for computing the edit-distance. In summary Jestes et al. proof whether two strings match with a degree of similarity and probability τ. They foremost compare both strings with the more relaxed pruning methods. If one method decides that S_1 and S_2 are a clear mismatch or match they can stop at this point. Only in a few cases they have to fall back to the expensive exact calculation of the EED.

While [7] focus on matching whole strings only, in [6] Ge et al. provide an algorithm to also perform substring searches on probabilistic data. Their search query is the tupel (k, τ) and defines two threshold parameters k and τ, where k denotes the maximum edit-distance allowed and τ is the minimum probability. The probabilistic string is indexed with q-grams as in [7]. When performing an k-approximate search the number of potential matches to be verified grows rapidly, since $k + 1$ substrings are searched in a lot of possible worlds. To shrink the verification effort, a filter is already applied when searching the index: Every q-gram in the index is extended with a left and a right signature that represent the left and right continuation of the q-gram in the string. Uncertain positions are mapped to a special wildcard character in the signature. After potential matches are found by the index, Ge et al. force additional pruning techniques to further reduce the number of matches to be verified. The first pruning technique bases on the matrix of the DP algorithm to calculate the edit-distance of two strings. The second one calculates two paths through S beginning from the position of the potential match. These paths describe the best and worst matching substring for p at this position and gives a lower and upper bound.

Example 3 (searching a character-level probabilistic string in [6])**.** *Given a search pattern $CCGTTT$ for a (k, τ)-query, where $k = 1$ and $\tau = 0.2$. Let us search in the string $AACCATTTAA$. First, p is split into $k + 1$ pieces, e.g. $g_1 = CCG$ and $g_2 = TTT$. A hash function $h : \Sigma \mapsto \{0, 1\}$ maps $\{A, C\} \mapsto 0$ and $\{G, T\} \mapsto 1$. So g_1's right signature of length 2 is 11 while g_2's left signature of length 2 is 01. For an exact matching index-entry the signatures' edit-distance is compared to not exceed k. Say a potential match is the index entry TT at position 6 with the left signature 00 (CA). Obviously, the edit-distance of the signatures 00 and 01 is $1 \leq k$. So this match needs to be verified, which means the lower and upper bound pruning methods are applied. If only the lower bound methods succeed, the potential match is verified in the string to become a full approximate match.*

Similarity of probabilistic sets is investigated by Gao et al. in [5]. They provide two new similarity functions for sets called expected similarity (ES) and confidence-based similarity (CS), both based on the Jaccard similarity.

5 New chunk-level model for string-level probabilistic strings

String-level probabilistic strings have the major shortcoming that shared substrings between possible worlds are represented multiple times. Our new idea for chunk-level probabilistic strings is to extract common substrings between possible worlds and represent them only one time. We will foremost explain what chunk-level strings are and how they can be obtained from string-level probabilistic strings. Later, in Section 5.1, we explain how to search probabilistic strings given in our model. But before formally introducing chunk-level probabilistic strings, we start with an example.

Example 4. *Given the string-level probabilistic string* $S = \{(CCTTTGA, 0.5), (AATTT, 0.5)\}$*, both possible worlds share the substring* TTT*. This can be seen when performing a multiple string alignment (MSA). The optimal MSA (match score 1, mismatch score -1) of the two possible worlds is as follows:*

```
Possible World 1:  CC TTT GA
Possible World 2:  AA TTT --
```

The MSA yields three obvious blocks: one block consisting of $\{CC, AA\}$*, one block consisting of* $\{TTT, TTT\}$*, and one block* $\{GA, --\}$*. Each of these blocks can be seen as a string-level probabilistic string on its own:* $S_1 = \{(CC, 0.5), (AA, 0.5)\}$*,* $S_2 = \{(TTT, 1.0)\}$ *and* $S_3 = \{(GA : 0.5), (--, 0.5)\}$*.*

However, if we combine the three string-level strings S_1, S_2, and S_3 from Example 4 in a naive way, we have the same problem as the character-level model: we lose conditional probabilities and allow, for instance, the string $AATTGA$ with a probability of 0.25. In order to avoid this loss of information, we need to keep track of all possible paths through chunks, e.g. that GA of S_3 cannot be combined with AA from S_1. We store for each choice of each chunk the IDs of the possible worlds of S it occurs in. For instance, for GA in S_3 we will record that it only occurs in Possible World 1. We call this set of possible worlds PW-tag. The PW-tag is stored as a bitstring where each bit position stands for one possible world of the origin string. For readability, we write down PW-tags as sets, e.g. $\{PW1, PW2\}$.

Definition 8 (chunk-level probabilistic string)**.** *A* chunk-level probabilistic string *is* $S = C[1]...C[n]$*. For* $i = 1, ..., n$*,* $C[i] = C_i = (\{(\sigma_{i,1}, p_{i,1}, pw_{i,1}), ...(\sigma_{i,m_i}, p_{i,m_i}, pw_{i,m_i})\}, pos_i)$ *where* $\sigma_{i,j} \in \Sigma^*$*,* $p_{i,j} \in (0, 1]$*,* $\sum_{j=1}^{m_i} p_{i,j} = 1$*, and each* $pw_{i,j}$ *is a set of possible worlds.* pos_i *is the position where* C_i *starts in the MSA of the corresponding string-level probabilistic string. Each* C_i *instantiates into* $\sigma_{i,j}$ *with probability* $p_{i,j}$ *independently.*

In order to reconstruct a given possible world X from a chunk-level probabilistic string, we extract each string labeled with PWX from each chunk and concatenate the result. To reconstruct the probability of the possible world X we need an extra map, that stores the probabilities for all possible worlds.

Example 5 (chunk-level probabilistic strings)**.** *Given the chunk-level probabilistic string from Example 4 as defined in 8, so that* $S = C_1...C_3$*, with* $C_1 = (\{(CC, 0.5, \{PW1\}), (AA, 0.5, \{PW2\})\}, 1)$*,* $C_2 = (\{(TTT, 1.0, \{PW1, PW2\})\}, 3)$*, and* $C_3 = (\{(GA, 0.5\,\{PW1\}), (--, 1.0, \{PW2\})\}, 6)$*. Extracting possible world 1 yields* $CC \circ TTT \circ GA$*. The probability of possible world 1 must be looked up in a map and is found to be* 0.5*.*

In the following we sketch our ideas how to identify obvious chunks from a MSA of the possible worlds of a string-level probabilistic string. Please note that we will compute an approximate MSA for long strings. Our aim is to extract equal columns of the MSA and put them into the chunk. We will design and implement an algorithm that processes a MSA from left to right (column by

column) and greedily extracts common rows. The first phase of the algorithm proceeds as long as all characters in the current column are equivalent. These characters will be added to the current (initially empty) chunk. Once we find a column with mismatches or gaps, we close the current chunk and assign a probability of 1.0 to the single string. Remember that all rows coincided for all previously processed columns. In its second phase, the algorithm moves right on the MSA until all columns agree again. After finishing the second phase, the algorithm returns to phase one and greedily aggregates equivalent columns into one chunk. The formal definition of the transformation algorithm will be done at a later stage of this thesis.

id	chunk	pos_i
C_1	$\{(CC, 0.5, \{PW1\}),$ $(AA, 0.5, \{PW2\})\}$	1
C_2	$(TTT, 1.0, \{PW1, PW2\})\}$	3
C_3	$\{(GA, 0.5, \{PW1\}),$ $(--, 0.5, \{PW2\})\}$	6

Table 5.1: Chunks from Example 4 with probability and PW-Tag

5.1 Searching chunk-level strings

Searching in a chunk-level probabilistic string S in our terms means that for a given search string q (also called search pattern) all probable substrings in S are found, that are approximatly equal to q. *Probable* means that the matching substring occurs in S with a probability higher than a given threshold τ, while *approximatly equal* aims that the match and the search pattern haven not to be equal but similar with a certain degree called k. Hence, we have three input parameters that determine our search: q, k and τ. We propose a method for a k-approximate string search over a chunk-level probabilistic string as follows:

1. Find all k-approximate matches inside each chunk (and map these matches to the original possible worlds) and
2. Find all k-approximate matches in strings overlapping at least two chunks.

To find not only exact matches but also k-approximate ones, the search pattern is split into $k + 1$ pieces (called partitions). Instead of searching for the whole pattern, these partitions are searched for in the following. The first step can be performed by using an index structure on each chunk. In our case, we plan to use suffix trees. The second step, finding all matches in strings overlapping chunks, needs more thought. For instance, searching the chunk-level string from Example 4, we want to be able to find the pattern $CCTT$ (which is across C_1 and C_2). To tackle this problem, we will try the following two approaches.

Signatures: Every indexed chunk is extended by an additional multilevel signature as described in [6]. The left signature of a chunk represents the last characters of the previous chunk while the right signature corresponds to the beginning of the following chunk. If the neighbors have more than one choice it is possible that a chunk has more than one left or right signature. Precisely, it can have as many signatures as the left or right hand chunk has number of choices. When searching for a substring, the beginning of the substring must be found in at least one of these chunks. For a small alphabet, e.g. $\Sigma = \{A, C, G, T\}$, we possibly end up with a lot of potential matches in a first step as A, C, G and T which produce a lot of hits. In this case the signatures will be used very extensively. As this is a remaining problem, we have to find out if this thwarts our search algorithm or find an appropriate solution.

Example 6. *In Table 5.2 the chunks from Example 4 are extended with a right and a left signature of length 2. For signature calculation a hash function $h : \Sigma \mapsto 0, 1$ was used, with $h(X) = 0$, for $X \in \{A, C\}$ und*

$h(X) = 1$, for $X \in \{G, T\}$. C_2 has got only one left signature, because $h(CC) = h(AA)$. Given the search pattern $ATTTG$ for an 1-approximate search could lead to the search partitions ATT and TG. Considering the index for chunk C_1 also contains $A\$$, it should match for ATT. Before verifying the rest of the pattern (TT) in the neighbour chunk, we first calculate the right signature of length 2 of A in ATT and match it against C_1's right signature: $h(TT) = 11 = rSig(C_1)$.

id	chunk	pos_i	lSig	rSig
C_1	$\{(CC\$, 0.5, \{PW1\}),$ $(AA\$, 0.5, \{PW2\})\}$	1	-	*11*
C_2	$\{(TTT\$, 1.0, \{PW1, PW2\})\}$	3	*00*	*10*
C_3	$\{(GA, 0.5, \{PW1\}),$ $(--, 0.5, \{PW2\})\}$	6	*11*	-

Table 5.2: Level 1 signatures of length 2 for chunks from Example 4

Additional pseudochunks: If we fix a maximum query length q_{max}, we can solve the problem of overlapping matches by introducing redundant chunks, so-called pseudochunks. For all positions where the to-be-searched string is split into two chunks, we form a new pseudochunk. It contains the $(q_{max} - 1 + k)$-rightmost characters of every choice of the left hand chunk and the $(q_{max} - 1 + k)$-leftmost characters of every choice of the right hand chunk, following the ideas in [18]. This ensures that every possible search pattern is entirely covered by at least one chunk. Storing some data redundantly in the additional pseudochunks may lead to redundant search results that must be filtered out. This can be done by comparing the PW-tag and the origin-string-position. If both are equal for two potential matches they are redundant and one can be removed.

Example 7. *See Table 5.3. The pseudochunks C_{12} and C_{23} are added to Table 5.1 for a maximum query length $q_{max} = 3$ and $k \leq 1$. The chunk C_2 in the middle has only one choice and with a probability of 1.0 and occurs in every possible world of S. That is why the pseudochunks are simple combinations of C_1 and C_3 with C_2. Searching now for TTG would have a match in the pseudochunk C_{23}. In this case the length of C_1, C_2 and C_3 is less than q_{max}, that is why they could even be omitted from search, having the pseudochunks.*

C_1	$\{(CC, 0.5, \{PW1\}),$ $(AA, 0.5, \{PW2\})\}$	1
C_{12}	$\{(CCTTT, 0.5, \{PW1\}),$ $(AATTT, 0.5, \{PW2\})\}$	1
C_2	$(TTT, 1.0, \{PW1, PW2\})\}$	3
C_{23}	$\{(TTTGA, 0.5, \{PW1\}),$ $(TTT, 0.5, \{PW2\})\}$	3
C_3	$\{(GA, 0.5, \{PW1\}),$ $(--, 0.5, \{PW2\})\}$	6

Table 5.3: Chunks from Example 4 with additional *pseudochunks* (C_{12}, C_{23}) with $q_{max} = 3$ and $k \leq 1$.

No matter which approach is chosen for dealing with overlapping matches, each possible world of each individual chunk is indexed as a deterministic string. To sum up, running a search query will follow the procedure below:

1. Splitting the search query in $k + 1$ partitions.
2. Commit every partition to the indexes of every chunk. Possible filtering methods will be applied locally to the intermediate matches.
3. The results returned by the indexes are verified and merged.

In Definition 6 and 7 we describe two types of approximate queries for string-level probabilistic strings. Both can be applied to chunk-level probabilistic strings, too. The deterministic approximate

query (q, k) returns all positions of the search pattern q in the possible worlds of S. Note that the positions are related to the MSA that was used to transform the string-level string into the chunk-level model, since the positions stored in the individual chunks are related to the MSA. The probabilistic approximate search query (q, k, τ) matches S if and only if the accumulated probability of all possible worlds of S that matches is $\geq \tau$. In the chunk-level model we need to calculate this accumulated probability in a way different to the string level model. Therefore we define the function MPW on a chunk-level probabilistic string S as follows:

$$MPW((q, k, \tau), S) = \{pwid \mid \exists\, C \in S, (\sigma, p, pw) \in C : q \sim_k \sigma \wedge pwid \in pw\}$$

After obtaining all possible worlds that matches we look up and accumulate their probabilities. The results returned by the indexes hence depend on the query type.

6 Implementation and Evaluation

The implementation will be written in C++. We anticipate the following four major parts of the implementation:

- Transformation into Chunks: We use an external program to create an approximate MSA for given string-level probabilistic strings. The outcome MSA is processed by our tranformation algorithm.
- Indexing Chunks: Every chunk owns one index per choice. Its choices are indexed in a deterministic manner. We plan to look for existing implementations of index algorithms to run our algorithm with different libraries and different indexing strategies, for instance CST++[14] and ESA/FM-Index in Seqan [3]. This is to determine which one scales best with our model.
- Searching indexed chunks: Since the chunks contain multiple indexes we have to design a class that holds and manages the different indexes of a chunk. It has to delegate search request and coordinate index filter, for instance as used in approach 1 for dealing with overlapping matches. The matches for one chunk are aggregated to results and are given to the tiers above.
- Global search: The results from the individual chunks and chunk-level strings have to be merged at this point. Verification with possible pruning methods on the origin string is done here.

As a result of this architecture we want to become able to easily exchange underlying low level algorithm, especially the indexing methods. As mentioned we start with using suffix trees but plan to probe other techniques and various libraries, too.

Another goal is to optimize the index for parallelisation, so that different chunks can be processed on different nodes. In Figure 6.1 a system with three nodes containing six chunks is shown. Although in the figure only one index is shown per chunk, each chunk will have multiple indexes, as the chunks are *string-level* probabilistic strings with several possible worlds. Every search on an index will span an individual thread.

The implementation will be compared to the related work [6] and [7]. As data, we use the datasets available at the Institute for Knowledge Management in Bioinformatics at the Humboldt University of Berlin: 1000 human genomes, 180 genomes of Arabidopsis thaliana. and 38 genomes of yeast strains. For our testcases see Table 6.1. More testcases might be added if required.

For every testcase several genomes (or chromosomes) of one species are initially aligned and treated as the possible worlds of one *string-level* probabilistic string with equal probabilities. Next our transformation algorithm is used to build the *chunk-level* model of the string. Afterwards the *chunk-level*

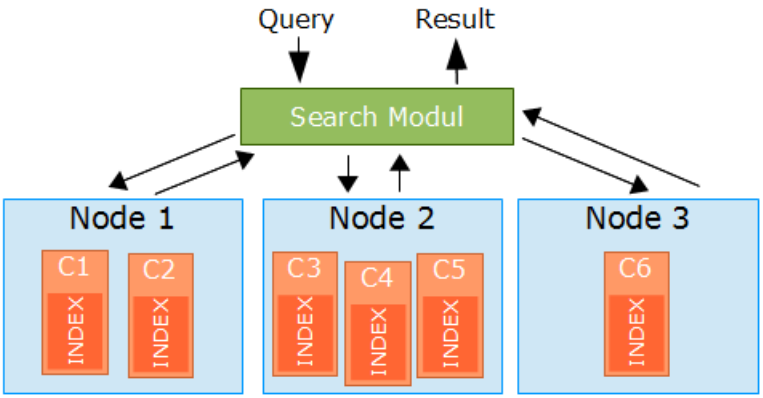

Figure 6.1: Architecture of the implementation.

testcase	data source	notes
T1	synthetic	random strings (varying alphabet size, length, and SNPs%)
T2	Human	1-100 Chromosome 1
T3	Arabidopsis thaliana	1-100 Chromosome 1
T4	Yeast	1-38 genomes

Table 6.1: Testcases

probabilistic string is given to our system for indexing and preprocessing. Finally different search queries are executed on the system. We run the testcases both on one single machine with one node and with three nodes distributed over three machines. The results will be compared to the results of the implementation by Ge et al.

References

[1] R. A. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. In *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, CPM '92, pages 185–192, London, UK, UK, 1992. Springer.

[2] X. Chen and M. Tompa. Comparative assessment of methods for aligning multiple genome sequences. *Nat Biotech*, 28(6):567–572, June 2010.

[3] A. Döring, D. Weese, T. Rausch, and K. Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics*, 9, 2008.

[4] J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proceedings of 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 152–165, 2008.

[5] M. Gao, C. Jin, W. Wang, X. Lin, and A. Zhou. Similarity query processing for probabilistic sets. *Information Sciences: an International Journal*, 196:97–117, 2012.

[6] T. Ge and Z. Li. Approximate substring matching over uncertain strings. In *Proceedings of the VLDB Endowment, Vol. 4, No. 11*, pages 772–782. VLDB Endowment, 2011.

[7] J. Jestes, F. Li, Z. Yan, and K. Yi. Probabilistic string similarity joins. In *SIGMOD*, pages 327–338. Springer Verlag, 2010.

[8] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[9] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33:2001, 1999.

[10] G. Navarro and R. Baeza-yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1:http://www.clei.cl., 1998.

[11] G. Navarro, R. Baeza-yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24:2001, 2000.

[12] G. Navarro and L. Salmela. Indexing variable length substrings for exact and approximate matching. In *String Processing and Information Retrieval*, pages 214–221. Springer Verlag, 2009.

[13] C. Notredame. Recent Evolutions of Multiple Sequence Alignment Algorithms. *PLoS Computational Biology*, 3(8):e123+, Aug. 2007.

[14] E. Ohlebusch, J. Fischer, and S. Gog. Cst++. In *SPIRE'10*, pages 322–333, 2010.

[15] A. Rheinländer, M. Knobloch, N. Hochmuth, and U. Leser. Prefix tree indexing for similarity search and similarity joins on genomic data. In *Proceedings of the 22nd SSDBM*, pages 519–536, Berlin, Heidelberg, 2010. Springer.

[16] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Pro-*

ceedings of the Third Annual European Symposium on Algorithms, ESA '95, pages 327–340, London, UK, 1995. Springer.

[17] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[18] S. Wandelt, J. Starlinger, M. Bux, and U. Leser. Scalable similarity search in thousand(s) of genomes. *Proceedings VLDB Endowment*, 2013.