Indexing a Dictionary for Subset Matching Queries

Gad M. Landau* Dekel Tsur† Oren Weimann‡

Abstract

We consider a subset matching variant of the *Dictionary Query* problem. Consider a dictionary D of n strings, where each string location contains a set of characters drawn from some alphabet $\Sigma = \{1, ..., |\Sigma|\}$. Our goal is to preprocess D so when given a query pattern p, where each location in p contains a single character from Σ, we answer if p matches to D. p is said to match to D if there is some $s \in D$ where $|p| = |s|$ and $p[i] \in s[i]$ for every $1 \leq i \leq |p|$.

To achieve a query time of $O(|p|)$, we construct a compressed trie of all possible patterns that appear in D. Assuming that for every $s \in D$ there are at most k locations where $|s[i]| > 1$, we present two constructions of the trie that yield a preprocessing time of $O(nm + |\Sigma|^k n \log(\min\{n, m\}))$, where n is the number of strings in D and m is the maximum length of a string in D. The first construction is based on divide and conquer and the second construction uses ideas introduced in [2] for text fingerprinting. Furthermore, we show how to obtain $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \log(\min\{n, m\}))$ preprocessing time and $O(|p| \log\log|\Sigma| + \min\{|p|, \log(|\Sigma|^k n)\} \log\log(|\Sigma|^k n))$ query time by cutting the dictionary strings and constructing two compressed tries.

Our problem is motivated by haplotype inference from a library of genotypes [13, 16]. There, D is a known library of genotypes ($|\Sigma| = 2$), and p is a haplotype. Indexing all possible haplotypes that can be inferred from D as well as gathering statistical information about them can be used to accelerate various haplotype inference algorithms.

1 Introduction

In the *Dictionary Query* problem, one is given a set D of strings $s_1, \ldots, s_n$ and subsequent queries ask whether a given query pattern p appears in D. In [5], this paradigm was broaden to allow a bounded number of mismatches, or allow a bounded number of "don't care" characters. We further extend dictionary queries to support a restricted version

*Department of Computer Science, University of Haifa, Haifa 31905, Israel, landau@cs.haifa.ac.il; Department of Computer Science and Engineering, NYU-Poly, Six MetroTech Center, Brooklyn, NY 11201-3840; partially supported by the National Science Foundation Award 0904 246, Israel Science Foundation grants 35/05 and 347/09, the Israel-Korea Scientific Research Cooperation, Yahoo, Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF) and DFG.

†Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel. dekelts@cs.bgu.ac.il

‡Faculty of Mathematics and Computer Science, Weizmann Institute of Science, Rehovot, Israel. oren.weimann@weizmann.ac.il

of *subset matching*. In subset matching, the characters are subsets of some alphabet Σ. A pattern p is said to match a string s of the same length if $p[i] \subseteq s[i]$ for every $1 \leq i \leq |p|$. The subset matching problem of finding all occurrences of a pattern string p in a text string t was solved in $O(N \log^2 N)$ deterministic time [6] and $(N \log N)$ randomized time [21], where N is the sum of sizes of the sets in p and t.

In this paper we consider the problem of indexing a dictionary for subset matching queries. We focus on a relaxed version of subset matching requiring that the query pattern is over single characters from Σ rather than subsets of Σ. Formally, the problem we consider is defined as follows. We are given a dictionary D of strings $s_1, \ldots, s_n$ where each string character is a subset of some alphabet Σ. A query p is a string over the alphabet Σ, and we say that p matches to s_i if $|p| = |s_i|$ and $p[j] \in s_i[j]$ for every $1 \leq j \leq |p|$. Our goal is to preprocess D for queries of the form "does p match to a string in D?".

Let m denote the length of the longest string in D and let D' be the set of all strings that match to a string in D. For example, if D contains two strings, $ab\{c,d\}$ and $ab\{c,d\}g\{a,b,c\}ad$, then $D' = \{abc, abd, abcgaad, abcgbad, abcgcad, abdgaad, abdgbad, abdgcad\}$. Notice that $|D'|$ is bounded by $O(|\Sigma|^k n)$. By storing the dictionary D' in a trie we can efficiently answer membership queries in $O(|p|)$ time for a pattern p. A *compressed trie* (i.e. a trie whose internal nodes all have more than one child and whose edges correspond to strings rather than single characters) can be naively constructed in $O(|\Sigma|^k nm)$ time and $O(|\Sigma||D'|)$ space, assuming every $s \in D$ has at most k locations in which $|s[i]| > 1$. The techniques of Cole et al. [5] can be used to solve the problem with $O(nm \log(nm) + n \log^k n/k!)$ preprocessing time, and $O(m + \log^k n \log\log n)$ query time. For small $|\Sigma|$, this approach is less efficient than the compressed trie approach.

In Sections 2 and 3 we present two faster constructions of the trie. The first construction is based on divide and conquer and requires $O(nm + |\Sigma|^k n \log n)$ preprocessing time. The second construction uses ideas introduced in [2] for text fingerprinting and requires $O(nm + |\Sigma|^k n \log m)$ preprocessing time. The space complexity is $O(|\Sigma||D'|)$, and it can be reduced to $O(|D'|)$ by using suffix tray [7] ideas. Intuitively, a suffix tray is a combination of a suffix tree and a suffix array where in some suffix tree nodes we store and array of length $|\Sigma|$ of children pointers and in some nodes we store two pointers to appropriate intervals in the suffix array. The save in space comes at the cost of $O(|p| + \log\log|\Sigma|)$ query time. In Section 4 we show that by cutting the dictionary strings and constructing two tries we can obtain $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \log(\min\{n, m\}))$ preprocessing time at the cost of $O(|p| \log\log|\Sigma| + \min\{|p|, \log|D'|\} \log\log|D'|) = O(|p| \log\log|\Sigma| + \min\{|p|, \log(|\Sigma|^k n)\} \log\log(|\Sigma|^k n))$ query time.

An important feature of our first two trie constructions is that they can calculate the number of appearances in D of each pattern in D' (i.e., which is most common? which is least common? etc.). This feature is useful in the application of *Haplotype Inference* that we next describe according to the presentation of Gusfield [12].

1.1 A haplotype trie from a genotype dictionary

In diploid organisms such as humans, there are two non-identical copies of each chromosome (except for the sex chromosome). A description of the data from a single copy is called a *haplotype* while a description of the conflated (mixed) data on the two copies is

called a *genotype*. The underlying data that forms a haplotype is either the full DNA sequence in the region, or more commonly the values of only DNA positions that are *Single Nucleotide Polymorphisms* (SNP's). A SNP is a position in the genome at which exactly two (of four) nucleotides occur in a large percentage of the population. If we consider only the SNP positions, each position can have one of two nucleotides and a haplotype can thus be represented as a 0/1 vector. A genotype can be represented as a 0/1/2 vector, where 0 means that both copies contain the first nucleotide, 1 means that both copies contain the second nucleotide and 2 means that the two copies contain different nucleotides (but we don't know which copy contains which nucleotide).

The next high-priority phase of human genomics will involve the development and use of a full *Haplotype Map* of the human genome [20]. Unfortunately, it is prohibitively expensive to directly determine the haplotypes of an individual. As a result, almost all population data consists of genotypes and the haplotypes are currently inferred from raw genotype data. The input to the haplotype inference problem consists of n genotypes (0/1/2 vectors), each of length m. A solution to the problem associates every genotype with a pair of haplotypes (binary vectors) as follows. For any genotype g, the associated binary vectors v_1, v_2 must both have value 0 (respectively 1) at any position where g has value 0 (respectively 1); but for any position where g has value 2, exactly one of v_1, v_2 must have value 0, while the other has value 1. The haplotypes inference problem has been studied extensively, e.g. [1, 4, 9, 10, 12, 15, 17, 19, 24, 26, 27, 32].

In our settings, the dictionary D corresponds to the library of genotypes, where every genotype location that has the value 2 is replaced by the set $\{0, 1\}$. This way, $|\Sigma| = 2$ and D' consists of all the possible haplotypes that can be part of a pair inferred from D. Our trie stores all haplotypes in D' and we can calculate the number of appearances in D of each such haplotype while constructing the trie. The trie can then be used to accelerate haplotype inference algorithms based on the "pure parsimony criteria", greedy heuristics such as "Clarks rule", and EM based algorithms.

2 An $O(nm + |\Sigma|^k n \log n)$ time construction

In this section we present an $O(nm + |\Sigma|^k n \log n)$ time construction for the compressed trie of D'. To simplify the presentation, for the rest of the paper we assume without loss of generality that all strings in D have the same length m. We say that string s is the *longest common prefix* (LCP) of strings x and y if s is the longest string that is a prefix of both x and y.

We first describe an algorithm for merging two compressed tries T_1 and T_2.

1. If one of the tries T_1 or T_2 has a single vertex, then return a copy of the other trie.

2. If both the roots of T_1 and T_2 have degree 1, and the labels of the edges leaving the roots of T_1 and T_2 have a common first letter, then find the longest common prefix (LCP) p of these labels. Remove the string p from T_1, that is, if the label of the edge e that leaves the root of T_1 is equal to p, remove the edge e and the root from T_1, and otherwise remove p from the label of e. Additionally, remove p from T_2.

   Next, recursively merge the two modified tries T_1 and T_2, and let T be the result

of the merge. Add a new root r to T and connect it by an edge to the old root of T, where the label of the edge is p.

3. If the two cases above do not occur, then split the trie T_1 as follows. For every edge $e = (r, v)$ that leaves the root r of T_1, create a new trie that contains r, v, and all the descendents of v in T_1. This trie will be denoted T_1^a, where a is the first letter in the label of e. Similarly, split the trie T_2 and create tries T_2^a.

   For each letter $a \in \Sigma$, recursively merge the tries T_1^a and T_2^a if these two tries exist. Finally, merge the roots of the merged tries.

If the LCP of two edge labels can be obtained in $O(1)$ time, then the time complexity of this algorithm is $O(|T_1|+|T_2|)$, where $|T|$ denotes the number of vertices in the compressed trie T. To perform such LCP queries in $O(1)$ time, we make use of *generalized suffix tree*.

Given a set X of n strings each of length bounded by m, a generalized suffix tree is a compressed trie containing all $O(nm)$ suffixes of the strings in X. strings in X. A generalized suffix tree can be built in $O(nm)$ time (e.g. [11, 22, 25, 29, 31]). By building a lowest common ancestor data structure (such as [18]) on the generalized suffix tree, we can support $O(1)$-time LCP queries between pairs of suffixes in X.

We now present the algorithm for building a compressed trie of D'.

1. For every string in D, replace every character that is a set of size greater than one with a new symbol ϕ.

2. Build a generalized suffix tree $\hat{T}$ for D.

3. Build compressed tries $T_1, \ldots, T_n$, where T_i is a compressed trie containing all the patterns that match s_i (recall that $D = \{s_1, \ldots, s_n\}$).

4. Repeat $\lceil \log n \rceil$ times:

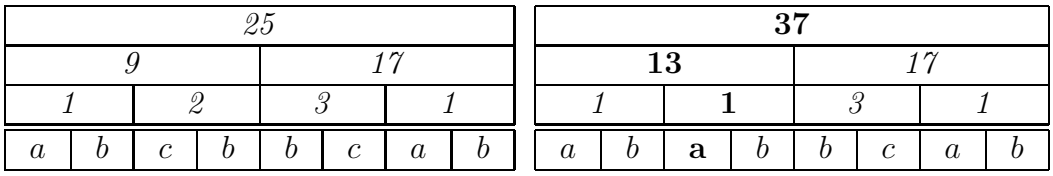
   (a) Partition the compressed tries into pairs, except at most one trie.

   (b) Merge each pair of tries into a single trie.

Constructing $\hat{T}$ requires $O(nm)$ time. Each edge label b in some trie that is built during the algorithm, matches a substring $s_i[j..j+|b|-1]$ of some string s_i in D. It is important to notice that $|s_i[l]| = 1$ for every $j+1 \leq l \leq j+|b|-1$. Using the generalized suffix tree $\hat{T}$, computing the LCP of two edge labels takes $O(1)$ time. Therefore, the merging of two compressed tries in the algorithm is performed in linear time. In each iteration of line 4, the total work is linear in the total sizes of the current tries, which is $O(|D'|) = O(|\Sigma|^k n)$. Thus, the overall time complexity of the algorithm is $O(nm + |\Sigma|^k n \log n)$.

3 An $O(nm + |\Sigma|^k n \log m)$ time construction

In this section we present an $O(nm + |\Sigma|^k n \log m)$ time construction for the compressed trie of D'. Consider the lexicographical ordering of all the strings in D'. Notice that if we knew this ordering and the length of the LCP of every adjacent strings in this ordering, then we could construct the trie in $O(|D'|) = O(|\Sigma|^k n)$ time by adding the strings in



(a) (b)

Figure 1: Figure (a) shows a possible naming table for the string $p = abcbbcab$. Note that the first and last cell in the third row have the same name as the names of the cells below these cells are the same (a and b). Figure (b) shows a possible naming table for the string $q = ababbcab$ that differs from p in one location. The cells of the naming table of q that differ from the corresponding cells of the naming table of p are marked in bold.

order. We next describe how to obtain the required ordering and LCP information in $O(nm + |\Sigma|^k n \log m)$ time.

We assign a unique integer name to every string in D' such that the names preserve the lexicographical order of D'. The names are assigned using a fingerprinting technique [2, 8, 23]. The idea behind fingerprinting is that the name of a string p can be computed fast from the name of a string q that differs from p only in one location.

A *naming table* of a string p is a table of $1 + \log |p|$ rows, where the i-th row contains 2^{i-1} cells (without loss of generality $|p|$ is a power of two, otherwise, we can extend p until $|p|$ is a power of two by concatenating to p a string of a repeated new character). Each cell in the table is assigned a name. First, the cells in the last row are named by the characters of p. Next, the cells of the second last row are named. The name of a cell depends on the names a_1 and a_2 assigned to the two cells below it. If there was other cell in the current row such that the blocks below it were also named a_1 and a_2, then the name used for that cell is also given for the current cell. Otherwise, a new name is used. This process is continued with the other rows in the table. See Figure 1(a) for an example.

The following property is what makes the naming technique appealing in our settings. Consider two strings p and q that differ only in one location. Then, the naming table of p differs from the naming table of q only in $1 + \log |p|$ cells (see Figure 1(b)).

Consider all the strings that match a specific string $s \in D$. It is possible to enumerate these strings in an order $s^{(1)}, s^{(2)}, \ldots, s^{(r)}$ in which two consecutive strings differ in exactly one location. This means that one can compute names for these strings in $O(m + r \log m)$ time as follows. First build the naming table of $s^{(1)}$ from bottom to top, using a two-dimensional table B to store the names given so far. More precisely, $B[a, b]$ is the name given for the pair (a, b), if the pair (a, b) was named. Since checking whether a pair of names appeared before takes constant time, the time it takes to build the naming table is linear in the number of cells in the table, which is $m + m/2 + m/4 + \cdots + 1 = 2m - 1$. Next, we build the naming table of $s^{(2)}$ by updating $1 + \log m$ cells in the table of $s^{(1)}$, which takes $O(\log m)$ time. Then, we build the naming table of $s^{(3)}$ using the naming table of $s^{(2)}$, and so on.

Applying the naming procedure to all strings in D takes $O(nm + |\Sigma|^k n \log m)$ time. The space complexity is $O((nm + |\Sigma|^k n \log m)^2)$ due to the table B. The space complexity

can be reduced to $O(nm + |\Sigma|^k n \log m)$ as shown in [8]. The algorithm of [8] uses a different order of filling the naming tables. In the first step, the algorithm computes the names in the second row from the bottom of the naming tables of all strings in D'. This is done by taking all pairs of names encountered in the first row of each naming table, lexicographically sorting these pairs, and then naming the pairs. In the second step, the algorithm computes the names in the third row from the bottom of the naming tables of all strings in D', and so on.

After naming all strings in D', we sort these strings according to their names. As noted above, this gives the lexicographical ordering of D'. Furthermore, the LCP of any two strings in D' can be computed in $O(\log m)$ time by comparing their naming tables top-down as noticed in [23]. Therefore, we can compute the length of the LCP of every two consecutive strings in the lexicographic ordering of D' in $O(|\Sigma|^k n \log m)$ time, and then construct the trie in $O(|D'|) = O(|\Sigma|^k n)$ time by adding the strings in lexicographical order.

4 An $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \log(\min\{n, m\}))$ time construction

In this section we present a different approach for solving the dictionary query problem. Instead of building one trie, we build two tries. This reduces the construction time, but gives a penalty in the query time.

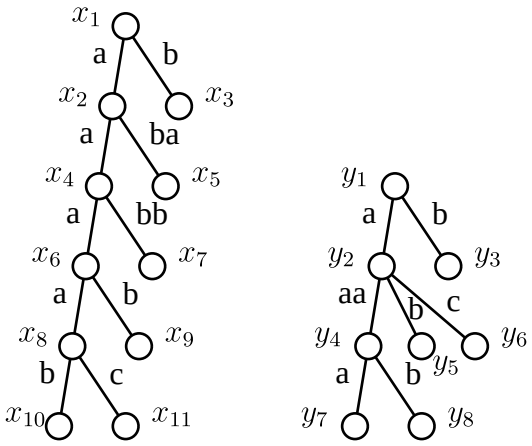
Let S be a set of integers. For an integer x, the *successor* of x in S is the minimal element $y \in S$ such that $y \geq x$. A *successor data-structure* for the set S supports answering queries of the form "Given an integer x, what is the successor of x in S?". A successor data-structure for a set $S \subseteq \{1, \ldots, U\}$ can be built in $O(|S|)$ time and space such that successor queries are answered in $O(\log \log U)$ time (such a construction is obtained, for example, by combining the van Emde Boas data-structure [30] with the static dictionary of Hagerup et al. [14]).

In order to build a dictionary query data-structure, we split every string in D into two parts. For each $s_i \in D$ define s'_i to be the longest prefix of s_i that contains at most $\lceil k/2 \rceil$ sets of size greater than 1. Also, define s''_i to be the prefix of s_i^R (i.e. the string s_i reversed) of length $m - |s'_i|$. For example, if $k = 2$ and $s_1 = ab\{c, d\}g\{a, b, c\}ad$ then $s'_1 = ab\{c, d\}g$ and $s''_1 = da\{a, b, c\}$.

Let $D_1 = \{s'_1, \ldots, s'_n\}$ and $D_2 = \{s''_1, \ldots, s''_n\}$. For $i = 1, 2$, let D'_i be the set of all strings that match to one of the strings in D_i. We wish to reduce the problem of matching a string p against the dictionary D to matching a prefix p' of p against D_1, and matching a prefix p'' of p^R against D_2, with $|p''| = m - |p'|$. However, there are two issues that need to be addressed: (1) It is possible that p' matches a string s'_i, while p'' matches to a string s''_j with $i \neq j$. This of course does not imply that p matches to a string in D. (2) We do not know the length of p', so we need to check all prefixes of p that match to a string in D_1.

Let T_1 be a compressed trie for D'_1 and T_2 be a compressed trie for D'_2. For each vertex of T_2 assign a *name* which is an integer from the set $\{1, \ldots, |T_2|\}$. The name assigned to a vertex v is denoted name(v). For now we assume that all the names are distinct.

The string that *corresponds* to a vertex v in a trie is the concatenation of the edge



(a) (b)

v	L_v	S_v (1)	S_v (2)
x_1	$\emptyset$	$\emptyset$	$\emptyset$
x_2	$\{1\}$	$\{7,8\}$	$\{1,2\}$
x_3	$\{1\}$	$\{7,8\}$	$\{1,2\}$
x_4	$\emptyset$	$\emptyset$	$\emptyset$
x_5	$\{2\}$	$\{5,6\}$	$\{3,4\}$
x_6	$\{2\}$	$\{5,6\}$	$\{3,4\}$
x_7	$\{3\}$	$\{2,3\}$	$\{1,5\}$
x_8	$\emptyset$	$\emptyset$	$\emptyset$
x_9	$\{3\}$	$\{2,3\}$	$\{1,5\}$
x_{10}	$\{4\}$	$\{1\}$	$\{1\}$
x_{11}	$\{5\}$	$\{1\}$	$\{1\}$

(c)

v	L_v
y_1	$\{4,5\}$
y_2	$\{3\}$
y_3	$\{3\}$
y_4	$\emptyset$
y_5	$\{2\}$
y_6	$\{2\}$
y_7	$\{1\}$
y_8	$\{1\}$

(d)

Figure 2: An example of the data-structure for the input strings $s_1 = \{a,b\}\{a,b\}aaa$, $s_2 = a\{a,b\}a\{b,c\}a$, $s_3 = aa\{a,b\}b\{a,b\}$, $s_4 = aaaab$, and $s_5 = aaaac$. The tries T_1 and T_2 are shown in Figures (a) and (b), respectively. The sets L_v for vertices of T_1 and T_2 are shown in Figures (c) and (d), respectively. Moreover, Figure (c) shows the sets S_v for $v \in T_1$, where the naming of the vertices is done according to two naming schemes: (1) $\text{name}(y_i) = i$ (2) naming according to the heavy path decomposition $Q_1 = [y_1, y_2, y_4, y_8]$, $Q_2 = [y_7]$, $Q_3 = [y_5]$, $Q_4 = [y_6]$, and $Q_5 = [y_3]$ of T_2.

labels in the path from the root to v. The *depth* of a vertex v in a trie is the length of the strings that corresponds to v. We say that the vertices $v \in T_1$ and $w \in T_2$ are *paired* if the sum of their depths is m. For a vertex v in T_1 (respectively T_2) whose corresponding string is s, let L_v be the set of all indices i such that s matches to s'_i (respectively s''_i). For a vertex $v \in T_1$, let $S_v = \{\text{name}(w) | w \in T_2 \text{ and } L_v \cap L_w \neq \emptyset\}$. See Figure 2 for an example.

The data-structure for the dictionary query problem consists of the tries T_1 and T_2, and each vertex $v \in T_1$ has a successor data-structure on the set S_v. Answering a query is done as follows.

1. Find the longest path P_1 in T_1 that corresponds to a prefix of the query pattern p, and the longest path P_2 in T_2 that corresponds to prefix of p^R.

2. Find all paired vertices $v \in P_1, w \in P_2$ by traversing P_1 from top to bottom, while concurrently traversing P_2 from bottom to top (note that a vertex $v \in P_1$ is paired with at most one vertex $w \in P_2$).

3. Check whether $\text{name}(w) \in S_v$ for some paired vertices $v \in P_1$ and $w \in P_2$ (by checking whether the successor of $\text{name}(w)$ in S_v is equal to $\text{name}(w)$).

Answering a dictionary query requires at most $|P_1| \leq m$ queries on the successor data-structures, where each such query takes $O(\log\log|D'|)$ time. Therefore, the time to answer a query is $O(m \log\log|D'|)$.

We now discuss the time complexity of building the tries. The tries T_1 and T_2 are built using the algorithms in Sections 2 and 3 in $O(nm + |\Sigma|^{k/2} n \log(\min(n,m)))$ time. In

order to build the sets S_v for all v, compute the intersections $L_v \cap L_w$ for all v and w. This is done as follows. For each i from 1 to n, go over all vertices $v \in T_1$ such that $i \in L_v$. For each such v, go over all $w \in T_2$ such that $i \in L_w$, and add the pair $(\text{name}(w), i)$ to a list I_v that is stored at v. Then, for each $v \in T_1$, lexicographically sort the list I_v and obtain all the intersections involving v. Therefore, computing all the intersections and building the successor data-structures takes $O(|\Sigma|^k n)$ time. The total preprocessing time is $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \log(\min\{n, m\}))$.

In order to speed up the query time, we use the technique of *fractional cascading* [3]. Fractional cascading is a method for efficiently searching for the same element in several successor data structures. Using a variant of this technique that is described in the next section, we can preprocess T_1 such that performing a successor query x on all the successor data structures of the vertices of some path P in T_1 is done in $O(|P| \log\log|\Sigma| + \log\log|D'|)$ time. Recall that in order to answer a query, we need to query for $\text{name}(w)$ in the successor data-structures of v for every paired vertices $v \in P_1$ and $w \in P_2$. In order to use the fractional cascading speedup, we need to decrease the number of names assigned to the vertices of P_2. Note that we can assign the same name to several vertices of T_2 if their corresponding strings have different lengths. Thus, we partition the vertices of T_2 into paths $Q_1, \ldots, Q_r$ using a *heavy path decomposition* [18].

A heavy path decomposition T_2 is as follows. For each node v define its size to be the size of the subtree rooted at v. For every internal node v we pick a child of maximum size and classify it as heavy. The remaining children are light. An edge to a light child is a *light edge*. Removing the light edges we obtain the decomposition of T_2 into paths. This decomposition has the important property that a path from some vertex of T_2 to the root passes through at most $\log|T_2|$ different paths in the decomposition.

We now assign names to the vertices of T_2 according to the heavy path decomposition: The name of a vertex w is the index i such that $w \in Q_i$.

Now, answering a query is done as follows.

1. Find the longest path P_1 in T_1 that corresponds to a prefix of the query pattern p, and the longest path P_2 in T_2 that corresponds to prefix of p^R.

2. For $i = 1, \ldots, r$, let v_i^{high} (respectively v_i^{low}) be the highest (respectively lowest) vertex in P_1 that is paired with a vertex $w \in P_2 \cap Q_i$, if there is such a vertex.

3. For every i such that v_i^{high} is defined, let $P_{1,i}$ be the path from v_i^{high} to v_i^{low},

4. For every path $P_{1,i}$, perform a successor query with the integer i on the successor data-structures of the vertices in $P_{1,i}$ using fractional cascading.

For example, consider the query $p = aaaaa$ on the structure in Figure 2. We have that $P_1 = [x_1, x_2, x_4, x_6, x_8]$ and $P_2 = [y_1, y_2, y_4, y_7]$. Moreover, $v_1^{\text{high}} = x_4$, $v_1^{\text{low}} = x_6$, and $v_2^{\text{high}} = v_2^{\text{low}} = x_2$, so $P_{1,1} = [x_4, x_6, x_8]$ and $P_{1,2} = [x_2]$.

We have that there are at most $\min\{m, \log|T_2|\} = O(\min\{m, \log|D'|\})$ different names assigned to the vertices of P_2. Therefore, the number of $P_{1,i}$ paths is $O(\min\{m, \log|D'|\})$. Since the $P_{1,i}$ paths are disjoint, it follows that the time to answer a dictionary query is $O(m \log\log|\Sigma| + \min\{m, \log|D'|\} \log\log|D'|)$.

4.1 Fractional cascading

Let T be a rooted tree of maximum degree d. Each vertex v of T has a set $C_v \subseteq \{1, \ldots, U\}$. The goal is to preprocess T in order to answer the following queries "given a connected subtree T' of T and an integer x, find the successor of x in C_v for every $v \in T'$". The fractional cascading technique of [3] gives search time of $O(|T'| \log d + \log\log U)$, with linear time preprocessing. We now present a variant of fractional cascading that gives $O(|T'| \log\log d + \log\log U)$ search time (our construction is similar to the one in [28]).

The preprocessing of T is as follows. Traverse the vertices of T in postorder, and for each vertex v of T construct a list A_v whose elements are kept in a non-decreasing order. For a leaf v, A_v contains exactly the elements of C_v. For an internal vertex v, A_v contains all the elements of C_v. Additionally, for every child w of v, A_v contains every second element of A_w. Each element of A_v stores a pointer to its successor in the set C_v. An element of A_v which came from a set A_w keeps a pointer to its copy in A_w. This pointer is called a *w-bridge*. For every vertex v, the elements of A_v are stored in a successor data-structures.

Handling a query (T', x) is done by finding the successor of x in each set A_v for $v \in T'$. Then, using the successor pointers, the successor of x in each set C_v is obtained. Finding the successor of x in each set A_v for $v \in T'$ is done by traversing the vertices of T' in postorder. For the root r of T', finding the successor of x in A_r is done by making a successor query on the successor structure of A_r. Suppose we have found the successor y of x in A_v and we now wish to find the successor of x in A_w, where w is a child of v. Let z be the first element that appears after y in A_v that has a w-bridge, and let z' be the elements in A_w pointed to by the w-bridge of z. Then, the successor of x in A_w is either z' or the element preceding z' in A_w.

In order to efficiently find the first w-bridge after some element of A_v, perform additional preprocessing as follows. Partition the elements of each list A_v into blocks $B_v^1, B_v^2, \ldots, B_v^{\lceil |A_v|/d \rceil}$ of d consecutive elements each (except perhaps the last block). Let $w_1, \ldots, w_{d'}$ be the children of v. For each block B_v^i build an array L_v^i, where $L_v^i[j]$ is the location of the first w_j-bridge that appears in the blocks $B_v^{i+1}, B_v^{i+2}, \ldots, B_v^{\lceil |A_v|/d \rceil}$. Moreover, for all j, build a successor data-structures $S_v^{i,j}$ that contains the indices of the elements of the block B_v^i that have a w_j-bridge.

Find the first w_j-bridge after some element of A_v takes $O(\log\log d)$ time. Therefore, the time complexity of answering a successor query is $O(|T'| \log\log d + \log\log U)$.

5 Conclusion and open problems

We have shown two solutions for the subset dictionary query problem: one based on building a trie for D' and one based on building two tries. We conjecture that the trie of D' can be built in $O(nm + |\Sigma|^k n)$ time.

References

[1] G. R. Abecasis, R. Martin, and S. Lewitzky. Estimation of haplotype frequencies from diploid data. *American Journal of Human Genetics*, 69(4 Suppl. 1):114, 2001.

[2] A. Amir, A. Apostolico, G. M. Landau, and G. Satta. Efficient text fingerprinting via parikh mapping. *J. of Discrete Algorithms*, 1(5-6):409–421, 2003.

[3] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[4] A. G. Clark. Inference of haplotypes from PCR-amplified samples of diploid population. *Molecular Biology and Evolution*, 7(2):111–122, 1990.

[5] R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th ACM Symposium on Theory Of Computing (STOC)*, pages 91–100, 2004.

[6] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. 34th ACM Symposium on Theory Of Computing (STOC)*, pages 592–601, 2002.

[7] R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. In *Proc. 33rd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 358–369, 2006.

[8] G. Didier, T. Schmidt, J. Stoye, and D. Tsur. Character sets of strings. *J. of Discrete Algorithms*, 5(2):330–340, 2007.

[9] L. Excoffier and M. Slatkin. Maximum-likelihood estimation of molecular haplotype frequencies in a diploid population. *Molecular Biology and Evolution*, 12(5):921–927, 1995.

[10] D. Fallin and N. J. Schork. Accuracy of haplotype frequency estimation for biallelic loci, via the expectation-maximization algorithm for unphased diploid genotype data. *American Journal of Human Genetics*, 67(4):947–959, 2000.

[11] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. of the ACM*, 47(6):987–1011, 2000.

[12] D. Gusfield. Haplotype inference by pure parsimony. In *Proc. 14th Symposium on Combinatorial Pattern Matching (CPM)*, pages 144–155, 2003.

[13] D. Gusfield and S. H. Orzack. Haplotype inference. In S. Aluru, editor, *CRC handbook on bioinformatics*, 2005.

[14] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *J. of Algorithms*, 41(1):69–85, 2001.

[15] M. T. Hajiaghayi, K. Jain, K. Konwar, L. C. Lau, I. I. Mandoiu, and V. V. Vazirani. Minimum multicolored subgraph problem in multiplex PCR primer set selection and population haplotyping. In *Proc. 6th International Conference on Computational Science (ICCS)*, pages 758–766, 2006.

[16] B. V. Halldórsson, V. Bafna, N. Edwards, R. Lippert, S. Yooseph, and S. Istrail. A survey of computational methods for determining haplotypes. In *Proc. DIMACS/RECOMB Satellite Workshop on Computational methods for SNPs and haplotype inference*, pages 26–47, 2002.

[17] E. Halperin and R. M. Karp. The minimum-entropy set cover problem. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 733–744, 2004.

[18] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing*, 13(2):338–355, 1984.

[19] M. E. Hawley and K. K. Kidd. Haplo: A program using the em algorithm to estimate the frequencies of multi-site haplotypes. *J. of Heredity*, 86:409–411, 1995.

[20] L. Helmuth. Genome research: Map of human genome 3.0. *Science*, 5530(293):583–585, 2001.

[21] P. Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *Proc. 39th Symposium on Foundations of Computer Science (FOCS)*, pages 166–173, 1998.

[22] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. of the ACM*, 53(6):918–936, 2006.

[23] R. Kolpakov and M. Raffinot. New algorithms for text fingerprinting. In *Proc. 17th Symposium on Combinatorial Pattern Matching (CPM)*, pages 342–353, 2006.

[24] J. C. Long, R. C. Williams, and M. Urbanek. An E-M algorithm and testing strategy for multiple-locus haplotypes. *American Journal of Human Genetics*, 56(2):799–810, 1995.

[25] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.

[26] P. Rastas, M. Koivisto, H. Mannila, and E. Ukkonen. A hidden markov technique for haplotype reconstruction. In *Proc. 5th Workshop on Algorithms in Bioinformatics (WABI)*, pages 140–151, 2005.

[27] P. Rastas and E. Ukkonen. Haplotype inference via hierarchical genotype parsing. In *Proc. 7th Workshop on Algorithms in Bioinformatics (WABI)*, pages 85–97, 2007.

[28] Q. Shi and J. JáJá. Novel transformation techniques using Q-heaps with applications to computational geometry. *SIAM J. on Computing*, 34(6):1471–1492, 2005.

[29] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):246–260, 1995.

[30] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[31] P. Weiner. Linear pattern matching algorithm. *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[32] P. Zhang, H. Sheng, A. Morabia, and T. C. Gilliam. Optimal step length EM algorithm (OSLEM) for the estimation of haplotype frequency and its application in lipoprotein lipase genotyping. *BMC Bioinformatics*, 4(3), 2003.