Indexing a Dictionary for Subset Matching Queries

Gad M. Landau[1,2], Dekel Tsur[3], and Oren Weimann[4]

[1] Department of Computer Science, University of Haifa, Haifa - Israel.
[2] Department of Computer and Information Science, Polytechnic University, New York - USA.
`landau@cs.haifa.ac.il`
[3] Department of Computer Science, Ben-Gurion University, Beer-Sheva - Israel.
`dekelts@cs.bgu.ac.il`
[4] Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge MA - USA.
`oweimann@mit.edu`

Abstract. We consider a subset matching variant of the *Dictionary Query* paradigm. Consider a dictionary D of n strings, where each string location contains a set of characters drawn from some alphabet Σ. 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 *appears* in D. p is said to appear in D if there exists some $s \in D$ where $|p| = |s|$ and $p[i] \in s[i]$ for every $1 \leq i \leq |p|$. Furthermore, for every pattern p that appears in D we would like to know the number of times p appears in D.

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 $O(nm + |\Sigma|^k n \lg(\min\{n, m\}))$, 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 $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \lg(\min\{n, m\}))$ preprocessing time and $O(|p| \lg \lg |\Sigma| + \min\{|p|, \lg(|\Sigma|^k n)\} \lg \lg(|\Sigma|^k n))$ query time by cutting the dictionary strings and constructing two compressed tries.

Our problem is motivated by haplotype inference from a library of genotypes [14, 17]. 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. In particular, algorithms based on the "pure parsimony criteria" [13, 16], greedy heuristics such as "Clarks rule" [6, 18], EM based algorithms [1, 11, 12, 20, 26, 30], and algorithms for inferring haplotypes from a set of Trios [4, 27].

1 Introduction

In the *Dictionary Query* paradigm, one is given a dictionary D of strings $s_1, \ldots, s_n$ and subsequent queries ask whether a given query pattern p appears in the dictionary. In [7], 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 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 $O(n \lg^2 n)$ deterministic time [8] and $(n \lg n)$ randomized time [22], 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?" and "to how many strings in D does p match?".

Let m denote the length of the longest string in D and let D' be the set of all patterns that appear in D. For example, if D contains a single string $ab\{c,d\}g\{a,b,c\}ad$ then $D' = \{abcgaad, abcgbad, abcgcad, abdgaad, abdgbad, abdgcad\}$. Notice that a compressed trie of D' can support a query time of $O(|p|)$ for a pattern p. Such a trie 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. [7] can be used to solve the problem with $O(nm \lg(nm) + n(c_1 \lg n)^{k+1}/k!)$ preprocessing time, and $O(m + (c_2 \lg n)^k \lg \lg n)$ query time (c_1 and c_2 are some constants). For small $|\Sigma|$, this approach is less efficient than the 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 $O(nm + |\Sigma|^k n \lg n)$ preprocessing time. The second construction uses ideas introduced in [2] for text fingerprinting and requires $O(nm + |\Sigma|^k n \lg m)$ preprocessing time. The space complexity is $O(|\Sigma||D'|)$, and it can be reduced to $O(|D'|)$ by using suffix tray [9] ideas. This comes at the cost of $O(|p| + \lg \lg |\Sigma|)$ query time. In Sections 4 we show that by cutting the dictionary strings and constructing two tries we can obtain $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \lg(\min\{n, m\}))$ preprocessing time at the cost of $O(|p| \lg \lg |\Sigma| + \min\{|p|, \lg |D'|\} \lg \lg |D'|) = O(|p| \lg \lg |\Sigma| + \min\{|p|, \lg(|\Sigma|^k n)\} \lg \lg(|\Sigma|^k n))$ 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 [13].

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 [21]. 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.

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" [13, 16], greedy heuristics such as "Clarks rule" [6, 18], EM based algorithms [1, 11, 12, 20, 26, 30], and algorithms for inferring haplotypes from a set of Trios [4, 27].

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

In this section we present an $O(nm + |\Sigma|^k n \lg n)$ time construction for the compressed trie of D'. To simplify the presentation, for the rest of the paper we assume w.l.o.g. that all strings in D have the same length m.

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 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 $\{T_2^a\}_{a \in \Sigma}$.

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 all remaining 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. Next, we 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 $\hat{T}$ containing all suffixes of strings in 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 $\lceil \lg n \rceil$ 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 $j + 1 \le l \le j + |b| - 1$. Using the suffix tree $\hat{T}$, computing the longest prefix 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 $O(|\Sigma|^k n)$. Thus, the overall time complexity of the algorithm is $O(nm + |\Sigma|^k n \lg n)$.

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

In this section we present an $O(nm + |\Sigma|^k n \lg m)$ time construction for the compressed trie of D'. Consider a 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 order. We next describe how to obtain the required ordering and LCP information in $O(nm + |\Sigma|^k n \lg m)$ time.

We assign a unique name to every string in D' using fingerprinting techniques [2, 10, 25]. A *naming table* of some $p \in D'$ is a labeled complete binary tree whose leaves are the characters of p (without loss of generality $|p|$ is a power of two[1]). The naming table has

---
[1] Otherwise, we can extend p until $|p|$ is a power of two by concatenating to p a string of a repeated new character.

therefore $1 + \lg |p|$ rows and the cells in the last row are named by the characters of p. For example, if "$ab\{a,c\}b\{b,c\}cab$" $\in D$ then $p=$"$abcbbcab$" $\in D'$ and the naming table of p might look like this:

25							
9				17			
1		2		3		1	
a	b	c	b	b	c	a	b

We assign integer names to the rest of the table using the *naming technique* [3, 24], which is a modified version of the Karp, Miller and Rosenberg algorithm [23]. The names are assigned bottom-up, such that when we assign a name to a cell we look at the pair of cells below it. If this pair appeared before then we give the same name that was given before, otherwise we give a new name. Finally, the unique name assigned to p is in the root of the naming table.

The following property is what makes the naming technique appealing in our settings. Consider two strings (over the alphabet Σ) p and q that both match the same string in D. If p and q differ in one location then the naming table of p differs from the naming table of q only in $1 + \lg |p|$ cells (these cells are in bold in the following example where $p=$"$ababbcab$" and $q=$"$abcbbcab$").

37							
13				17			
1		**1**		3		1	
a	b	**a**	b	b	c	a	b

Consider all the strings that match a specific string $s \in D$. It is possible to enumerate these strings in an order $s^{(1)}, s^{(2)}, \ldots$ in which two consecutive strings differ in exactly one location. This means that we can compute names for these strings in $O(m + |\Sigma|^k \lg m)$ time as follows. We first build the naming table of $s^{(1)}$ from bottom to top. During the building of the naming table, we use an two-dimensional table B that stores the names given so far. More precisely, $B[a,b]$ is the name given for the pair of names (a,b), if the pair (a,b) was named. Since checking whether a pair of names has appeared before takes constant time, the time for building the naming table is linear in the number of cells of 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 $O(nm + |\Sigma|^k n \lg m)$ time. The space complexity is $O((nm + |\Sigma|^k n \lg m)^2)$ due to the table B. The space complexity can be reduced to $O(nm + |\Sigma|^k n \lg m)$ as shown in [10]. The algorithm of [10] 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.

In addition to the reduction in space, the algorithm of [10] has the following property. For every two strings $s, s' \in D'$ with names a and a' respectively, $a < a'$ if and only if s is lexicographically smaller than s'.

After naming all strings in D', we sort these strings using the names as keys. As noted above, this gives the lexicographical ordering of D'. Furthermore, the LCP of any two strings in D' can be computed in $O(\lg m)$ time by comparing their naming tables top-down as noticed in [25]. Therefore, we can compute the length of the LCP of every two consecutive strings in the lexicographic ordering of D' in $O(|\Sigma|^k n \lg m)$ time, and then construct the trie in $O(|D'|) = O(|\Sigma|^k n)$ time by adding the strings in lexicographical order.

4 An $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \lg(\min\{n, m\}))$ 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 given a penalty in the query time.

Let S be a set of elements with keys from some set U. For every $x \in U$, the *successor* of x in S is the element $y \in S$ such that $\mathrm{key}(y) \geq x$ and $\mathrm{key}(y)$ is minimal. A *successor data-structure* stores a set S of elements with keys, and supports answering queries of the form "Given a value $x \in U$, what is the successor of x in S?". For the universe $\{1, \ldots, U\}$, a successor data-structure for a set S can be built in $O(|S|)$ time and space such that successor queries are answered in $O(\lg \lg U)$ time (such a construction is obtained, for example, by combining the van Emde Boas data-structure [29] with the static dictionary of Hagerup et al. [15]).

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 $|s_i| - |s_i'| = m - |s_i'|$. 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 T_1 be a compressed trie for $\{s_1', \ldots, s_n'\}$ and let T_2 be a compressed trie T_2 for $\{s_1'', \ldots, s_n''\}$. For each vertex of T_2 assign a distinct integer from the set $\{1, \ldots, |T_2|\}$. The integer assigned to a vertex v is denoted $\mathrm{id}(v)$. The string that *corresponds* to a vertex in a trie is the concatenation of the edge labels in the path from the root to the vertex. 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 s_i' (respectively s_i''). For a vertex $v \in T_1$, let S_v be the set containing every vertex $w \in T_2$ that is paired with v and for which $L_v \cap L_w \neq \emptyset$.

Each vertex $v \in T_1$ has a successor data-structure, that stores the set S_v. The key of an element w in the set is $\text{id}(w)$.

Answering a query is done as follows. First 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. For a vertex $v \in P_1$, if a vertex $w \in$ is paired with v then by definition the depth of w is equal to m minus the depth of v. Since there is at most one vertex on P_2 with a given depth, we conclude that there is at most one vertex $w \in P_2$ that is paired with v. Moreover, it is possible to find all paired vertices $v \in P_1, w \in P_2$ in $O(|P_1| + |P_2|) = O(m)$ time by traversing P_1 from top to bottom, while concurrently traversing P_2 from bottom to top. To answer the query p, we just need to check whether $w \in S_v$ for some paired vertices $v \in P_1$ and $w \in P_2$. Checking whether $w \in S_v$ for some fixed v and w is done by performing a successor query on the successor data-structure of v. Answering a query requires at most $|P_1| \leq m$ searches in the successor structures, where each search takes $O(\lg \lg |D'|)$ time. Therefore, the time to answer a query is $O(m \lg \lg |D'|)$.

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 $O(nm + |\Sigma|^{k/2} n \lg(\min(n, m)))$ time. Computing the intersections $L_v \cap L_w$ for all v and w 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 $(\text{id}(w), i)$ 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 $O(nm + |\Sigma|^k n + |\Sigma|^{k/2} n \lg(\min\{n, m\}))$.

In order to speed up the query time, we use the technique of fractional cascading [5]. Using a variant of this technique that is described in the next section, we can preprocess T_1 such that searching for a key x in all the successor data-structures of the vertices of some path P in T_1 is done in $O(|P| \lg \lg |\Sigma| + \lg \lg |D'|)$ time. Recall that in order to answer a query, we need to locate $\text{id}(w)$ 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 different keys in the vertices of P_2. Note that we can give the same key 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 heavy path decomposition [19]. This decomposition has the property that a path from some vertex of T_2 to the root passes through at most $\lg |T_2|$ different paths in the decomposition. We now define $\text{id}(w)$ to be index i of the path Q_i that contains w.

Now, locating $\text{id}(w)$ in the successor data-structures of v for every paired vertices $v \in P_1, w \in P_2$ is done by grouping all vertices $w \in P_2$ with the same key i, and performing a search for i in all the successor data-structures of the vertices of some sub-path of P_1 (the subpath that contains all the vertices in P_1 between the highest and lowest vertices of P_1 that are paired to vertices in P_2 with key i). We have that there are at most $\min\{m, \lg |T_2|\} = O(\min\{m, \lg |D'|\})$ different keys in the vertices of P_2, and the different

subpaths of P_1 that correspond to the different keys in P_2 are disjoint. Therefore, the time to answer a query is $O(m \lg \lg |\Sigma| + \min\{m, \lg |D'|\} \lg \lg |D'|)$

4.1 Fractional cascading

Let T be a rooted tree of maximum degree d. Each vertex v of T has a set $C_v \subseteq \{1, \dots, U\}$. 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 [5] gives search time of $O(|T'| \lg d + \lg \lg U)$, with linear time preprocessing. We now present a variant of fractional cascading that gives better search time (our construction is similar to the one in [28]).

The preprocessing of T is as follows. For each vertex v of T construct a list A_v whose elements are kept in a non-decreasing order (the order of constructing the A_v lists is from the leaves up). 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*.

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. Suppose we have found the successor y of x in A_v and we now wish to find the successor y' of x in A_w, where w is a child of v. If we know the first element that appears after y in A_v and has a w-bridge, then we can follow the bridge to A_w and y' is either the element at the end of the bridge or the element preceding it in A_w.

In order to efficiently find the first w-bridge after some element of A_v, perform additional preprocessing: Partition the elements of each list A_v into blocks $B_v^1, B_v^2, \dots, B_v^{\lceil |A_v|/d \rceil}$ of d consecutive elements each (except perhaps the last block). Let $w_1, \dots, w_{d'}$ 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 appear in the blocks $B_v^{i+1}, B_v^{i+2}, \dots, B_v^{\lceil |A_v|/d \rceil}$. Moreover, for all j, build a successor data-structures $S_v^{i,j}$ that contains all the elements of the block B_v^i that have a w_j-bridge. The key of an element in $S_v^{i,j}$ is its rank in the block B_v^i.

Given an element y of A_v, finding the first element of A_v starting from y that has a w_j-bridge is done in $O(\lg \lg d)$ time. Therefore, the overall search time is $O(|T'| \lg \lg d + \lg \lg U)$

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. *Journal of Discrete Algorithms*, 1(5-6):409–421, 2003.
3. A. Apostolico, C.S. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.
4. Dumitru Brinza, Jingwu He, Weidong Mao, and Alexander Zelikovsky. Phasing and missing data recovery in family trios. In *Proceedings of the 5th International Conference on Computational Science (ICCS)*, pages 1011–1019, 2005.
5. B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.
6. A.G. Clark. Inference of haplotypes from pcr-amplified samples of diploid population. *Molecular Biology and Evolution*, 7(2):111–122, 1990.
7. R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the 36th annual ACM Symposium on Theory Of Computing (STOC)*, pages 91–100, 2004.
8. R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proceedings of the 34th annual ACM Symposium on Theory Of Computing (STOC)*, pages 592–601, 2002.
9. R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 358–369, 2006.
10. G. Didier, T. Schmidt, J. Stoye, and D. Tsur. Character sets of strings. *Journal of Discrete Algorithms*, to appear.
11. 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.
12. 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.
13. D. Gusfield. Haplotype inference by pure parsimony. In *CPM*, pages 144–155, 2003.
14. D. Gusfield and S.H. Orzack. Haplotype inference. In *CRC handbook on bioinformatics (S. Aluru Editor)*, 2005.
15. T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *J. of Algorithms*, 41(1):69–85, 2001.
16. 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 *Proceedings of the 6th International Conference on Computational Science (ICCS 2006), Part II*, pages 758–766, 2006.
17. B.V. Halldórsson, V. Bafna, N. Edwards, R. Lippert, S. Yooseph, and S. Istrail. A survey of computational methods for determining haplotypes. In *Proceedings of the DIMACS/RECOMB Satellite Workshop on Computational methods for SNPs and haplotype inference*, pages 26–47, 2002.
18. E. Halperin and R.M. Karp. The minimum-entropy set cover problem. In *Proceedings of the 31st annual International Colloquium on Automata, Languages and Programming (ICALP)*, pages 733–744, 2004.
19. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.
20. M.E. Hawley and K.K. Kidd. Haplo: A program using the em algorithm to estimate the frequencies of multi-site haplotypes. *Journal of Heredity*, 86:409–411, 1995.
21. L. Helmuth. Genome research: Map of human genome 3.0. *Science*, 5530(293):583–585, 2001.

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

23. R.M. Karp, R.E. Miller, and A.L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th annual ACM Symposium on Theory Of Computing (STOC)*, pages 125–136, 1972.

24. Z.M. Kedem, G.M. Landau, and K.V. Palem. Parallel suffix-prefix-matching algorithm and applications. *SIAM Journal of Computing*, 25(5):998–1023, 1996.

25. R. Kolpakov and M. Raffinot. New algorithms for text fingerprinting. In *Proceedings of the 17th annual symposium on Combinatorial Pattern Matching (CPM)*, pages 342–353, 2006.

26. 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.

27. J. Marchini, D. Cutler, N. Patterson, M. Stephens, E. Eskin, E. Halperin, S. Lin, Z.S. Qin, H.M. Munro, G. Abecasis, and P. Donnelly for the International HapMap Consortium. A comparison of phasing algorithms for trios and unrelated individuals. *American Journal of Human Genetics*, 78:437450, 2006.

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

29. 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.

30. 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.