

# Perfect hashing for strings: Formalization and Algorithms

Martin Farach\*  
Rutgers University

S. Muthukrishnan†  
Univ. of Warwick

March 10, 1996

## Abstract

Numbers and strings are two objects manipulated by most programs. Hashing has been well-studied for numbers and it has been effective in practice. In contrast, basic hashing issues for strings remain largely unexplored. In this paper, we identify and formulate the core hashing problem for strings that we call *substring hashing*. Our main technical results are highly efficient sequential/parallel (CRCW PRAM) Las Vegas type algorithms that determine a perfect hash function for substring hashing. For example, given a binary string of length  $n$ , one of our algorithms finds a perfect hash function in  $O(\log n)$  time,  $O(n)$  work, and  $O(n)$  space; the hash value for any substring can then be computed in  $O(\log \log n)$  time using a single processor. Our approach relies on a novel use of the suffix tree of a string. In implementing our approach, we design optimal parallel algorithms for the problem of determining *weighted ancestors* on a edge-weighted tree that may be of independent interest.

## 1 Introduction

Two primitive objects manipulated by almost all programs are integers<sup>1</sup> and strings. For instance, any database application typically manipulates both. Whatever the object, a fundamental paradigm in computing is to develop a *fingerprint*, by which we mean a succinct representation, for each object in a set drawn from a large universe; the operations that we wish to support on the set of objects can be equivalently implemented on the fingerprints. The advantage is, of course, that concise fingerprints use less resources and offer more efficiency.

---

\*Dept. of Computer Sc., Rutgers Univ., Piscataway, NJ 08855, USA. ([farach@cs.rutgers.edu](mailto:farach@cs.rutgers.edu), <http://www.cs.rutgers.edu/~farach>) Supported by NSF Career Development Award CCR-9501942 and an Alfred P. Sloan Research Fellowship.

†[muthu@dcs.warwick.ac.uk](mailto:muthu@dcs.warwick.ac.uk); Partly supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), and partly supported by ALCOM IT.

<sup>1</sup>More generally, they manipulate fixed-precision numbers.

In this paper, we will define a *hash function* to be any function whose domain is the set of objects under consideration and whose range is the set of fingerprints; hash functions, therefore, implement fingerprint schemes. While hash functions have been well-studied in applications that manipulate integers, several aspects of hashing in equally prevalent string processing tasks remain unexplored. In this paper, we identify a core fingerprint problem in string processing, describe a novel hash function for these fingerprints and present two efficient algorithms for computing this hash function.

Consider fingerprints for integers, which are most commonly used in the following situation. Suppose we are given a set  $S$  of  $n$  integers from  $[1, N]$ ,  $N \gg n$ , and we wish support lookup queries on  $S$ . If we have a function  $f : S \rightarrow [1, R]$  such that  $f(x) = f(y) \Rightarrow x = y$ , then we can use the  $f$  values to address a table of size  $R$  and perform a lookup operation. The time taken is dominated by the time needed to compute  $f$ . In our present nomenclature, the integers  $[1, R]$  are the fingerprints for  $S$ , and  $f$  is a hash function. What makes this scheme interesting is that we can efficiently find hash functions which can be evaluated in constant time and where  $R = O(n)$ . Perhaps the most well-known scheme for this is due to Fredman, Komlós and Szemerédi [FKS84], but there are a number of other schemes (eg., [CHM92]). Such hash functions, which ensure fingerprint distinctness and which have  $R = O(n)$ , are called *perfect hash functions*. In practice, other hash functions are used which are trivial to find [K73]; however, they are *imperfect* hash functions since they may sometimes allow *collisions*, that is, pairs  $x \neq y$  such that  $f(x) = f(y)$ . The study of hash functions has recognized crucial tradeoffs, including the size of the fingerprints, resources needed to generate the fingerprints (time, number of random bits, etc.), and so on. There is a sophisticated theory of hash functions; it remains a significant intellectual achievement with practical impact.

Consider, now, fingerprints for strings. In contrast to the existing theory of hash functions for integers, hash functions for string processing remain relatively unexplored. Karp and Rabin [KR87] initiated the study of hash functions for strings by developing hash functions for the standard string matching problem<sup>2</sup>. Their hash functions, however, are not perfect hash functions. Since their seminal work, these hash functions have been applied to a variety of string matching problems [B93, R84, N91, AFM92]. Despite these developments, the principle issues involved in hash functions for strings have yet to be addressed. While the resource tradeoff issues are common to both integer and string hashing, in this paper we identify hashing issues which are *unique* to strings. It is these unique issues which we specifically address in this paper.

We see the contribution of this paper as two-fold. First, we identify the core hashing problem for string processing and second, we provide general (that is, parallel and when possible, deterministic) algorithmic solutions for the identified problem. We also use our results to obtain simple and highly efficient Las Vegas

---

<sup>2</sup>The standard string matching problem is that of finding all occurrences of a pattern string of length  $m$  in a text string of length  $n$ .

sequential and parallel algorithms for several pattern matching problems e.g., in all those where the imperfect hash function of [KR87] has been applied.

In Section 2 we discuss hashing relevant for strings and formulate the core problem. There, we also describe our technical results. In Section 3 we present our algorithms for the core hashing problem for strings. Finally, we list a number of interesting open problems in Section 4.

## 2 Formulating hash functions for strings

**Relevant fingerprints for string processing.** What are the fingerprints we can formulate that would have wide applicability for string processing? Drawing from the analogy of fingerprints for integers, consider the problem where we are given a set  $S$  of strings each of  $n$  characters<sup>3</sup>. Once again, we would like a perfect hash function, i.e. an  $f : S \rightarrow [1, O(|S|)]$  such that  $f(x) = f(y) \Rightarrow x = y$ . We will say that such an  $f$  gives us *string fingerprints*. While string fingerprints are used in practical systems, e.g. in looking up reserved words in programming languages, command words in operating systems etc., they are of limited theoretical interest since there is a trivial optimal deterministic algorithm to both find an  $f$  and to evaluate it<sup>4</sup>. Additionally, they fall short of solving all but the most trivial string processing problems. For example, using string fingerprints to solve even the standard string matching problem yields an algorithm that takes  $\Theta(nm)$  time (which is no better than the naïve complexity).

By carefully looking at various string processing tasks, we have been able to abstract the following primitive underlying all of them.

**Primitive P1.** We are given a string  $s[1 \cdots n]$  and a set  $S$  of explicitly specified tuples  $\langle l_i, r_i \rangle$ , for some  $1 \leq l_i \leq r_i \leq n$ , to preprocess. Then given a pair  $\langle l, r \rangle$ , determine an  $i$  (possibly some representative  $i$ 's) such that  $s[l, r] = s[l_i, r_i]$ . That is, we are given a set  $S$  representing substrings of string  $s$ , each given by a tuple of its leftmost and rightmost index within  $s$ , and we wish to support lookup queries on  $S$ .

In order to implement this primitive, we define *substring fingerprints*. Let  $\mathcal{T}(s)$  be the set  $\{\langle l_i, r_i \rangle \mid 1 \leq l_i < r_i \leq n\}$  for string  $s$ . We define a *substring hash function*  $f_s : \mathcal{T}(s) \rightarrow [1, O(|\mathcal{T}(s)|)]$  such that  $f_s(\langle i, j \rangle) = f_s(\langle k, l \rangle) \Leftrightarrow s[i, j] = s[k, l]$ . Then  $f_s(\langle k, l \rangle)$  is the fingerprint for the substring  $s[k, l]$ .

Consider also the following closely related primitive.

**Primitive P2.** This is same as P1 except that  $S$  comprises *all* possible tuples  $\langle l_i, r_i \rangle$ , for  $1 \leq l_i \leq r_i \leq n$ . Furthermore, they are not specified explicitly; therefore, the input is merely the string  $s$  and  $\mathcal{T}(s)$  consists of all substrings of  $s$ . Substring fingerprints are defined for this case the same way as above.

<sup>3</sup>The constraint that they be of the same length is merely for illustration.

<sup>4</sup>Indeed it suffices to construct a trie of the strings in  $S$  and uniquely number the leaves in a one-to-one fashion with the distinct strings; the numbers assigned to the leaves are the string fingerprints.

Primitive P2 is more general than P1 because if we determine a  $f_s$  for P2, then we can subsequently apply integer hashing (that is, determine a perfect hash function for the integer values  $f_s$ 's) to get a substring hash function for P1. Therefore, in this paper, we henceforth consider only substring hashing for P2.

**Applicability of substring hashing.** Substring hashing is *universal* in the following sense. Consider solving any off-line string matching problem (that is one in which the text(s) and pattern(s) are presented concurrently), as is usual, in the comparison model. Any comparison made by an algorithm for such a problem can be replaced by a constant number of substring hashing queries. Therefore substring hashing can be used to simulate any algorithm for off-line string matching that uses comparisons without loss of efficiency in terms of the number of substring hashing queries<sup>5</sup>.

More interestingly, we can derive very simple algorithms *directly* using substring hashing for several string processing problems: these include standard string matching, higher dimensional matching, multiple pattern matching, etc. Known algorithms (especially the parallel ones) for these problems that do not use substring hashing are substantially more involved. Also, substring fingerprints can be used as a subtask in practical string processing systems, eg., those that rely on heuristics to effectively solve the theoretically-hard problem of approximate dictionary matching [AGM<sup>+</sup>90]. Finally, substring fingerprints naturally allow parallelism since fingerprints for different substrings can be evaluated and used in parallel.

In this paper, we apply substring hashing to obtain Las Vegas type algorithms as stated below in more detail.

**How integer, string and substring hashing differ:** Integers can clearly be hashed using string hashing by viewing each integer as a binary string (say, of length  $b$ ), that is, integer and string hashing are equivalent “bit” and “word” model operations respectively. However, that yields integer hashing algorithms slower than direct ones by a  $\Theta(b)$  factor. A similar relationship exists between string and substring hashing since substrings are represented succinctly by tuples. At first glance, it may seem that substring hashing is the same as string hashing since in both cases we are addressing the question of membership in a set of strings. However, the crucial difference is the succinctness in referring to substrings. In general  $l$  substrings represent strings of total length  $\Theta(ln)$ . Therefore, using string hashing routine for substring hashing leads to a  $\Theta(n)$  factor blow up in the length of the representations of strings, and consequently, an efficient algorithm for string hashing can be grossly inefficient for substring hashing.

**Our Results.** Substring fingerprints of exactly optimal size, namely  $\lceil \log \binom{n}{2} \rceil$  bits, can be determined in  $\Theta(n^2)$  sequential time or parallel work<sup>6</sup>. Clearly we

---

<sup>5</sup>By introducing dynamism in the definition of substring hashing, any on-line string matching problem (that is one in which portions of the text or pattern are presented for preprocessing) can be solved by substring fingerprints as well. See Section 4.

<sup>6</sup>This is done by determining, for each suffix, its longest common prefix at each location

would like to find a perfect hash function that can be computed more efficiently than this naïve scheme, i.e., in  $o(n^2)$ , or preferably in  $O(n)$  sequential time or parallel work. (Computing a perfect hash function needs  $\Omega(n)$  time since that requires looking at each string position).

Our main results are efficient algorithms to compute a perfect hash function for substring hashing. Some bounds may be obtained by using previously available techniques. In what follows, we summarize them together with our bounds (Rand: Randomized; Det: Deterministic; Par: Parallel).

Ref	Preprocessing			Query Time
	Alg.	Space	Par. Work	
[AI <sup>+</sup> 88]	Det	$O(n^{1+\epsilon})$	$O(\frac{1}{\epsilon}n \log n)$	$O(\frac{1}{\epsilon} \log n)$
[SV94]	Det	$O(n^{1+\epsilon})$	$O(\frac{1}{\epsilon}n \log^* n)$	$\Omega(\frac{1}{\epsilon} \log n)$
Here	Det	$O(n^{1+\epsilon})$	$O(n^{1+\epsilon})$	$O(\frac{1}{\epsilon})$
[KR87]	Rand (imperfect)	$O(n)$	$O(n)$	$O(\log n)$
Here	Rand (perfect)	$O(n)$	$O(n)$	$O(\log \log n)$

The bounds above are for a string drawn from a fixed alphabet. Furthermore, we have only listed the complexity of parallel algorithms. The sequential bounds are omitted here although our bounds are significantly better than the existing ones there as well. (Those bounds will be tabled in the final version of this paper). Here  $0 < \epsilon \leq 1$  is not necessarily fixed. Also, our randomized algorithm is of the Las Vegas type. In addition, the fingerprints we generate use at most  $\log(n^2)$  bits which is at most 2 bits more than the information-theoretic lower bound of  $\log \binom{n}{2}$ .

If alternate models for alphabet size is assumed, the bounds differ. For strings over unbounded alphabet and if we consider only randomized algorithms, the work for preprocessing becomes  $O(n \log \sigma)$  where  $\sigma$  is the number of distinct symbols in the string; the time becomes  $O(\log n \log \sigma)$  and all other bounds remain unchanged. If the strings are drawn from an alphabet  $1 \dots n^k$  for some constant  $k$  and we consider only deterministic algorithms, then work bounds above get multiplied by  $O(\log \log n)$  without change in any other parameter. In what follows, we omit further discussion of these alternate models and consider only the fixed alphabet case.

Our results for substring hashing are obtained by developing parallel and sequential algorithms for the problem of *weighted ancestors* on trees we introduce here. The weighted ancestors problem generalizes the well-studied *level-ancestors* problem on trees [C87, D92, BV89]. Our algorithms for the weighted ancestors problem may be of independent interest. Also, our algorithm relies on constructing the suffix tree of  $s$ .

---

in the original string, a task that takes  $\Theta(n)$  sequential time or  $\Theta(n)$  parallel work for each suffix [HM94, GP94, KMP77].

Now we present a concrete application for substring hashing. We can directly apply our algorithms for perfect hash functions and derive Las Vegas algorithms for several string matching problems, eg., those in [KR87, R84, N91, AFM92]. The algorithms we derive are suboptimal in both time and work by a  $O(\log \log n)$  factor, but they are simple and of Las Vegas type. In contrast, the known algorithms are, in general, either more complicated or of Monte Carlo type. In what follows, we do not elaborate on this application; the bounds we obtain for the various problems will be listed in the final version.

### 3 Our Approach to Substring Hashing

One way to find a perfect hash function is to use a provably good hash function (not necessarily perfect) and to apply an efficient “checker” that detects collisions, that is, two unequal substrings that have the same fingerprint<sup>7</sup>. While the fingerprint functions of Karp and Rabin [KR87] can be directly used as a good hash function for substring fingerprints, unfortunately there are no known efficient checkers for these functions. An efficient checker is known [M93] for a specific application of [KR87], but efficient checkers in general remain elusive, and therefore, they remain the bottleneck<sup>8</sup>.

Our approach is structural and it uses the suffix tree of the string. We give a simple, yet novel, reduction of substring hashing to the problem of finding weighted ancestors of nodes in the suffix tree of the string, with appropriate weights on each edge. The reduction and the precise definition of this problem follow.

**The Weighted Ancestors Problem:**

**Preprocess:** An  $n$  node rooted tree  $T$  with integer weights on edges in the range 1 to  $n$ .

**Weighted Ancestor Query:** Given pair  $v, l$ , where  $v$  is a node in  $T$  and  $l$  is an integer, return the highest node above  $v$  with (weighted) depth at least  $l$ .

#### 3.1 Reduction to Weighted Ancestors on Suffix Trees

Consider a string  $s$  for which we wish to perform substring naming. Suppose we want a substring naming function on  $s$ . Consider taking any substring  $s[x, y]$  and tracing from the root of the suffix tree of  $s$  with the characters of  $s[x, y]$ . Then, when we read the last such character, that is,  $s[y]$ , we will be at some node or edge in  $T_s$ . Further, if  $s[x, y] = s[i, j]$ , then we will be at the same node or edge in  $T_s$  by tracing with either  $s[x, y]$  or  $s[i, j]$ . Thus, the location read by such a tracing procedure would seem a good fingerprint for substrings. To be

---

<sup>7</sup>Throughout this section, we only consider strings over a fixed-sized alphabet for exposition.

<sup>8</sup>In contrast, perfect hash functions for numbers successfully use this approach. There trivially in  $O(n)$  time we can check if two numbers have been assigned the same fingerprint.

more precise, assign a distinct integer to each node  $T_s$ . Then, we can specify a node position by  $(v, 0)$ , for node with number  $v$ . For an edge  $(v, u)$ , we must specify the number of  $u$ , as well as how far up the tree we are in the edge, so once again, a pair  $(u, l)$  suffices.

The question is, how do we efficiently compute such a fingerprint. The direct method of tracing from the root seems inefficient. Instead, take the depth of a node  $v$  in  $T_s$  to be the sum of the string lengths labeling the edges above  $v$ . Now, give a pair  $(x, y)$ , we note that  $s[x, y]$  is a prefix of  $s[x, |s|]$ . In fact, it is the prefix of length  $y - x + 1$ . We can quickly find the leaf in  $T_s$  corresponding to  $s[x, |s|]$ , and now, we need the highest ancestor of this leaf which has (weighted) depth at least  $y - x + 1$ . Either this node yields the desired fingerprint, or its parent edge does. We therefore have reduced the problem of computing substring fingerprints to the weighted ancestors problem.

## 3.2 Finding Weighted Ancestors

We design several algorithms for this problem; amongst them one is the first known work-optimal algorithm for the problem, as well as the first known sequential algorithm. Berkman and Vishkin [BV89], and Dietz [D92] considered the version in which each edge has weight 1. This special case is called the *level-ancestor* problem which has interesting applications of its own [C87]. Algorithms in [BV89, D92] rely crucially on the unit-weight assumption (in generating tables containing solutions to all small subproblems). For our more general scenario where the weights on edges are arbitrary integers from  $1 \cdots n$ , we develop alternate algorithms described below.

We will briefly sketch the sequential version of our two approaches to the weighted ancestor problem while noting that they can be parallelized optimally (details are deferred to the final version).

### 3.2.1 Algorithm I

First, take an Euler Tour of  $T$  and annotate each node with its weighted depth. The weighted ancestor of  $v$  at depth  $l$  is given by the nearest neighbor to the right of  $v$  in the Euler Tour which has weighted depth no more than  $l$ . A first trivial algorithm would tabulate for each position in the Euler Tour all  $n^2$  possible queries (recall that there are up to  $n$  edges on a root-leaf path and that no edge weight is greater than  $n$ ). A refinement would build one table of size  $n$  for the  $\log n$  high bits of each possible query and another for the low bits. That is, one table would store the answers to the queries where the integer is  $n, 2n, \dots, W(v)$ , for node  $v$ , where  $W(v)$  is the weighted depth of  $v$ . Another could store the answers to queries of the form  $W(v) - n, W(v) - n + 1, \dots, W(v)$ . Answering a query now consists of 2 jumps: one which goes to the nearest neighbor with depth no greater than  $n \lceil l/n \rceil$ , and the second which completes the jump to  $l$ . By extending this idea to multiple levels, we achieve a  $O(n^{1+\epsilon})$  preprocessing algorithm which answers queries in  $O(1)$  time for some fixed  $\epsilon$ ,  $0 < \epsilon \leq 1$ . We omit much of the details here.

**Theorem 3.1** *We can deterministically preprocess a tree in  $O(n^{1+\epsilon})$  time and space so each weighted ancestor query can be answered in  $O(1/\epsilon)$  time, where  $0 < \epsilon \leq 1$ . We can achieve optimal speedup for the preprocessing in  $O(\log n)$  time.*

### 3.2.2 Algorithm II

A second approach is as follows. Let the weight of each node be the number of its descendant leaves. As in [ST83], let a child be *heavy* if its weight is more than half that of its parent. We now partition the tree into heavy paths. Notice that no node has more than  $\log n$  light ancestors. Now, each node determines its light ancestors and puts them in a sorted list. If we start a query at a node, we can perform a binary search within the light list – in  $O(\log \log n)$  time to determine either which node or edge we should answer, or to localize the answer to a single heavy path. Now our query is reduced the following: given a sorted list of  $k$  integers in the range  $[1, kn]$  and given an integer  $l$ , find the integers in the list that bracket  $l$ . Of course, we can not afford a binary search since this would give us a  $O(\log n)$  query time. Instead, notice that it suffices to use a data structure that solves the static predecessor problem on this list of numbers. That gives a number of tradeoffs for our problem based on those for the static predecessor problem. For example, placing the elements of each list in a van Emde Boas tree [vKZ77] allows such queries to be answered in  $O(\log \log n)$  time.

The upshot is that we get a  $O(n \log n)$  preprocessing algorithm with  $O(\log \log n)$  time queries. To reduce the preprocessing, we can partition  $T$  into subtrees of size  $\Theta(\log n)$  which we collapse into nodes. Now the tree is of size  $\Theta(n/\log n)$  and our preprocessing from above on this tree becomes linear. If we land in a collapsed node, we must search within the subtree for the true answer. However, since these trees are of logarithmic size, we can apply a straightforward binary search within the heavy paths and still maintain a  $O(\log \log n)$  query time. Omitting details here, to sum,

**Theorem 3.2** *We can preprocess the tree in  $O(n)$  time and space with high probability and answer any weighted ancestor query in  $O(\log \log n)$  deterministic time. We can achieve optimal speedup for the preprocessing in  $O(\log n)$  time.*

### 3.2.3 Putting it together

We can combine the results of the two preceding subsections to get the following for binary strings.

**Theorem 3.3** *We can determine a perfect hash function for substring hashing*

1. *deterministically in  $O(n^{1+\epsilon})$  work and space and  $O(\frac{1}{\epsilon} \log n)$  time following which the hash function can be evaluated for any query substring in  $O(\frac{1}{\epsilon})$  time with a single processor.*
2. *in  $O(n)$  work,  $O(n)$  space, and  $O(\log n)$  time by a Las Vegas algorithm following which the hash function can be evaluated for any query substring deterministically in  $O(\log \log n)$  time with a single processor.*

**Proof.** We deterministically construct the suffix tree for the given string in  $O(\frac{2}{\epsilon} \log n)$  work,  $O(n^{1+\epsilon})$  space and  $O(\frac{1}{\epsilon} \log n)$  time using the algorithm in [AI<sup>+</sup>88]. Then we apply Theorem 3.1. That gives 1 above. For 2, we apply the suffix tree construction algorithm in [FM96] together with Theorem 3.2.  $\square$

## 4 Discussion

In this paper, we have identified the core hashing problem in strings, namely substring hashing, and we have presented the most efficient parallel (and as a corollary, sequential) algorithms to compute a perfect hash function for them. Applying those algorithms, we obtain simple Las Vegas type algorithms for several pattern matching algorithms. We believe we have only initiated a formal study of hashing in strings and a number of other problems remain unexplored. In what follows, we will list only two of them.

**Dynamic Substring Hashing.** If the given string changes between substring fingerprint queries, we have the *dynamic* variant of substring hashing. Typical modifications to the string are the *edit* operations, namely character *insertions*, *deletions* and *substitutions*. The following can be easily shown.

**Theorem 4.1** *Given a string of length  $n$ , a substring hash function can be computed in  $O(\log n)$  time and  $O(n)$  work. Following this, the hash function can be updated for any insertion, deletion or substitution in  $O(\log n)$  time using a single processor. The fingerprint for any substring can be computed in  $O(\log n)$  time as well using a single processor. Two distinct substrings have different fingerprints with high probability.*

We do not know of more efficiently computable hash functions or any perfect hash function for dynamic substring hashing, either sequentially or in parallel. The simple scenario of dynamizing hash functions of [KR87] when *only* substitutions are allowed, can be thought of as the dynamic prefix summation problem studied by Fredman and others (See e.g., [HF93]). Dynamic substring hashing is of importance in *on-line* string processing problems, that is, those in which portions of the text or the pattern are presented in batches, or on-line.

**Subsequence Hashing.** A very general hashing for strings is what we call *subsequence hashing*. Here we wish to design a hash function for every subsequence (not necessarily a substring) of a given string. There are nontrivial complexity issues in formalizing even how subsequences are specified. Hash functions for subsequences would be of importance in string processing tasks in Computational Biology. Finding efficiently computable subsequence hash functions remains open.  $\square$

Furthermore, several interesting technical problems arise as a result of our work. Besides the obvious problem of designing perfect hash functions which can be evaluated in  $O(1)$  time after  $O(n)$  preprocessing, here is one.

**Batched weighted ancestors query.** Can we perform significantly better if we batch queries. That is, consider  $f(n)$  *simultaneous* weighted ancestor queries for some  $f(n)$ . Can they be answered cumulatively in  $o(f(n) \log \log n)$  time while still preserving  $O(n)$  time preprocessing sequentially? As it turns out, in pattern matching applications, the batched versions (for suitable  $f(n)$ ) suffice.

## References

- [AFM92] A. Amir, M. Farach, and Y. Matias. Efficient randomized dictionary matching algorithms. *Proc. of 3rd Combinatorial Pattern Matching Conference*, pages 259–272, 1992. Tucson, Arizona.
- [AGM+90] S.F. Altschul, W. Gish, W. Miller, E.W Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.
- [AI+88] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Scieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.
- [B93] A. Broder. Applications of Karp-Rabin fingerprints. Manuscript, 1993.
- [BV89] Omer Berkman and Uzi Vishkin. Recursive \*-tree parallel data-structure. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 196–202, 1989.
- [C87] B. Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2:337–361, 1987.
- [CHM92] Z J. Czech, G. Havas, and B S. Majewski. An optimal algorithm for generating minimal perfect hash functions. Technical Report 24, DIMACS, 1992.
- [D92] P. Dietz. Finding level-ancestors in dynamic trees. Manuscript, 1992.
- [FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with  $O(1)$  worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.
- [FM95] M. Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression. *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1995.
- [FM96] M. Farach and S. Muthukrishnan. Optimal Logarithmic Time Randomized Suffix Tree Construction. To be presented at the *23rd Intl. Colloq. on Automata, Languages and Programming*, 1996.

- [GP94] L. Gasieniec and K. Park. Optimal parallel prefix matching. *Proceedings of E.S.A.*, 1994.
- [HF93] H. Hampapuram and M. Fredman. Optimal bi-weighted binary trees and the complexity of maintaining partial sums. *Proc. IEEE Symp. on Foundations on Computer Sc*, 1993, 480–485.
- [HM94] R. Hariharan and S. Muthukrishnan. Optimal parallel prefix matching. *Proc. of 21st International Colloquium on Automata Languages and Programming*, 1994.
- [K73] D. E. Knuth. *The Art of Computer Programming, V. 3: Sorting and Searching*. Addison-Wesley, Reading, 1973.
- [KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.
- [KR87] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.
- [M93] S. Muthukrishnan. Detecting false matches in string matching algorithms. In *Proc. of 4th Combinatorial Pattern Matching Conference*, 1993.
- [N91] M. Naor. String matching with preprocessing of text and pattern. *Proc. of 18th International Colloquium on Automata Languages and Programming*, pages 739–750, 1991.
- [R84] M. Rabin. An algorithm for finding all repetitions. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, pages 85–96. Springer-Verlag, Berlin, 1984.
- [ST83] D. Sleator and R. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 24, 1983.
- [SV94] S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. *Proc. of the 26th Ann. ACM Symp. on Theory of Computing*, 1994.
- [vKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.