View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Theoretical Computer Science 131 (1994) 295–310
Elsevier

Dynamic dictionary matching with failure functions*

Ramana M. Idury** and Alejandro A. Schäffer***

Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251-1892, USA

Communicated by M. Nivat
Received March 1992
Revised October 1993

Abstract

R.M. Idury and A.A. Schäffer, Dynamic dictionary matching with failure functions, Theoretical Computer Science 131 (1994) 295–310.

Amir and Farach (1991) and Amir et al. (to appear) recently initiated the study of the dynamic dictionary pattern matching problem. The dictionary D contains a set of patterns that can change over time by insertion and deletion of individual patterns. The user may also present a text string and ask to search for all occurrences of any patterns in the text. For the static dictionary problem, Aho and Corasick (1975) gave a strategy based on a failure function automaton that takes $O(|D|\log|\Sigma|)$ time to build a dictionary of size $|D|$ and searches a text T in time $O(|T|\log|\Sigma|+tocc)$, where *tocc*, is the total number of pattern occurrences in the text.

Amir et al. (to appear) used an automaton based on suffix trees to solve the dynamic problem. Their method can insert or delete a pattern P in time $O(|P|\log|D|)$ and can search a text in time $O((|T|+tocc)\log|D|)$.

We show that the same bounds can be achieved using a framework based on failure functions. We then show that our approach also allows us to achieve faster search times at the expense of the update times; for constant k, we can achieve linear $O(|T|(k+\log|\Sigma|)+k\,tocc)$ search time with an update time of $O(k|P||D|^{1/k})$. This is advantageous if the search texts are much larger than the dictionary or searches are more frequent than updates.

Finally, we show how to build the initial dictionary in $O(|D|\log|\Sigma|)$ time, regardless of what combination of search and update times is used.

1. Introduction

Amir, Farach, Galil, Giancarlo, and Park [3, 5] (AFGGP for short) initiated the study of the *dynamic dictionary matching problem.* We are given a collection of

Correspondence to: A.A. Schäffer, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251-1892, USA.

* An extended abstract describing this research has been published in the proceedings of the Third Symposium on Combinatorial Pattern Matching held in Tucson, AZ in April 1992.

** Research partially supported by a grant from the W.M. Keck Foundation. Present address: Department of Mathematics, University of Southern California, Los Angeles, CA 90089-1113, USA.

*** Research partially supported by NSF grant CCR-9010534.

0304-3975/94/$07.00 © 1994—Elsevier Science B.V. All rights reserved
SSDI 0304-3975(93)E0193-8

patterns $D=\{P_1, P_2, \ldots, P_s\}$, called the *dictionary*, that can change over time. The basic matching operation is to *search* a text $T[1,t]$ and report all occurrences of dictionary patterns in the text. The dictionary can be changed by *inserting* or *deleting* individual patterns.

The *static dictionary matching problem*, in which inserts and deletes are not supported, was addressed in earlier papers by Aho and Corasick [2] and Commentz-Walter [9]. A recent survey article by Aho [1] provides a comprehensive overview of these papers. The *semi-dynamic dictionary matching problem*, in which only insertions are allowed, was introduced by Meyer [14]. Algorithms for the dictionary problems have applications to database searches and molecular biology [2, 5].

The two algorithms for static dictionary matching can be summarized as follows. We use $|D|$ to denote the total size of all the patterns in the dictionary; we use Σ to denote the alphabet and $|\Sigma|$ to denote its size. The Aho–Corasick algorithm (AC for short) builds the dictionary D in time $O(|D|\log|\Sigma|)$ and searches a text $T[1,t]$ in $O(t\log|\Sigma|+tocc)$ time, where *tocc* is the total number of occurrences reported. For small bounded alphabet the AC algorithm runs in linear time. The AC algorithm uses an automaton where states correspond to prefixes of dictionary patterns and transitions are determined by failure functions as in the Knuth–Morris–Pratt [11] (KMP for short) string searching algorithm. The Commentz-Walter (CW for short) algorithm instead uses ideas from the Boyer–Moore [7] string matching algorithm. Like the AC algorithm the CW algorithm also builds the dictionary in $O(|D|\log|\Sigma|)$ time; like the Boyer–Moore algorithm, the CW algorithm has good practical performance.

The AFGGP algorithm for dynamic dictionary matching is based on using a suffix tree [15, 12, 8] as an automaton. Each state corresponds to a substring of some pattern. The AFGGP algorithm is able to insert or delete a pattern $P[1,p]$ in time $O(p\log|D|)$, and it performs a search in time $O((t+tocc)\log|D|)$. If inserts and deletes are frequent enough, this algorithm is better than the simple alternative of using the AC algorithm for searches and rebuilding the dictionary in $O(|D|\log|\Sigma|)$ time at each update. It may be a little surprising that the AFGGP time bounds appear not to depend on the alphabet size; actually, the $\log|D|$ factor is short for $\log|D|+\log|\Sigma|$, but the $\log|\Sigma|$ factor is subsumed by the $\log|D|$ factor under certain assumptions.

In this paper, we present another algorithm for dynamic dictionary matching that addresses three questions raised by the AFGGP algorithm and time bounds.

First, is the idea of failure functions and the AC automaton of any use at all for dynamic dictionary matching? The AC automaton is a natural extension of the KMP automaton to multiple patterns. One would expect that the same approach may work for dynamic dictionary matching. We show that this is indeed the case. Our algorithm is based on the AC automaton. With a suitable choice of underlying data structures, our method achieves a search time of $O((t+tocc)\log|D|)$ and an update time of $O(p\log|D|)$ matching the AFGGP bounds.

Second, are other tradeoffs between search and update times possible? One would expect that in some applications, the updates are relatively infrequent or the text strings are much longer than the patterns. Under either condition we would prefer

a search time better than $O((t+tocc)\log|D|)$ while tolerating an update time worse than $O(p\log|D|)$. It is interesting to ask: how good can we make the update time if we insist that the search time match the AC bound of $O(t\log|\Sigma|+tocc)$ for the static problem?

We show that our algorithm can also use a different data structure such that for any constant $k\geqslant 2$, it can achieve search time $O(t(k+\log|\Sigma|)+k\,tocc)$ and update time $O(p(k|D|^{1/k}+\log|\Sigma|))$. We thereby match the static search time of [2] and have a sublinear update time if the patterns are not very long relative to $|D|$.

Third, can one match the $O(|D|\log|\Sigma|)$ preprocessing time of the AC and CW algorithms? The AFGGP algorithm builds the initial dictionary by repeated insertion of patterns in time $O(|D|\log|D|)$. We would like to avoid these insertions, especially if we are going to increase their cost. We show that regardless of the choice of the data structure, we can build our initial dictionary in $O(|D|\log|\Sigma|)$ time.

Our dynamic dictionary framework has another advantage. It can be used to solve a natural two-dimensional version of dynamic dictionary matching, while the AFGGP algorithm does not seem to generalize. One of the algorithmic reasons for this distinction is described at the end of Section 2.

In our two-dimension problem, the texts are rectangular and the patterns are square. The size of a text or pattern is its area. Amir and Farach [4] have shown how to solve the static two-dimensional dictionary problem with a preprocessing time of $O(|D|\log s)$ and a search time of $O(|T|\log s)$, where s is the number of patterns. In a separate paper [6], we describe how to combine their two-dimensional search technique, our dynamic dictionary framework, and some other ideas, to solve the dynamic two-dimensional dictionary problem, achieving the same time bounds for insertion, deletion, and search as in one dimension.

In sum, we show that it is possible to dynamize the AC approach to static dictionary matching to achieve the same preprocessing and search times that they do, while achieving an update time sublinear in $|D|$. Alternatively, we can match the search and update times of the AFGGP algorithm and improve the dictionary construction time to $O(|D|\log|\Sigma|)$. Our framework can be extended to two-dimensional dictionary matching.

The rest of this paper is organized as follows. In Section 2, we present our basic algorithm for dynamic dictionary matching. In Section 3, we show how to modify the underlying data structures, to improve the search time. In Section 4, we describe how to construct the initial dictionary in $O(|D|\log|\Sigma|)$ time.

2. Dictionary automaton and searching algorithm

Let $D=\{P_1,\ldots,P_s\}$ be a dictionary of patterns, where each P_i, $1\leqslant i\leqslant s$, is a string over a finite alphabet Σ. For convenience, we assume that the empty string ε is always a pattern in the dictionary. We append to each pattern a special symbol \$ that does not occur elsewhere in any pattern or text. We shall henceforth assume that $\$\in\Sigma$ and

that \$ is the largest symbol in the lexicographic order. Throughout this paper, we generally use w, x, y, z to denote a prefix of some pattern, and a, b to denote some character of Σ. We use the following dictionary as an example to explain various definitions and concepts of our automaton.

Example 2.1. Suppose $\Sigma = \{a, b, \$\}$. Let $\hat{D} = \{\$, b\$, aab\$\}$ be a sample dictionary where every pattern is appended with the special symbol \$.

We make the same assumptions on the character set Σ as in the papers on suffix trees [15, 12, 8, 3, 5]:

Assumption 2.2. Each character is represented by a constant number of bytes.

Assumption 2.3. The relative order of any two characters can be determined in constant time.

An additional assumption is implicit in [3, 5]:

Assumption 2.4. Even though $|\Sigma|$ may be very large, the actual number of distinct characters of Σ in any dictionary D is bounded by $|D|$. For stating time bounds, we assume that the $\log|\Sigma|$ factor is subsumed by the $\log|D|$ factor.

In [2], each state in the automaton corresponds to a prefix of some pattern in D. From now on, we use a prefix to denote its state. Intuitively, if we are in a state x after reading the first j characters of a text, x is the longest prefix of any pattern ending at the jth position of the text. Aho and Corasick [2] defined two important (partial) functions, *goto* and *fail*, that describe the transitions in their automaton. $goto(x, a) = xa$ if xa is a prefix of some pattern in the dictionary, else it is undefined, which we denote by $\perp$. Therefore, *goto* is a partial function in that each state may not have a transition on every symbol. For any prefix x, $fail(x) = w$ such that w is the longest prefix of some pattern such that w is a proper suffix of x. Figure 1 shows a sample automaton where *goto* transitions are shown by solid edges and *fail* transitions are shown by dotted edges. Whenever a transition cannot be made because *goto* is undefined, we repeatedly replace the state with *fail*(*state*) until a transition can be found. The basic search loop is:

```
while goto(state, symbol) = ⊥ do
  state ← fail(state)
  state ← goto(state, symbol)
  symbol ← nextsymbol
```

For any given choice of *state* and *symbol* we may have to take the *fail* transition repeatedly, but this shortens the length of the new *state*. The total time needed to scan a text T using this algorithm is $O(|T|(g+f))$, where $g+f$ is the time needed to make one evaluation of *goto* and *fail* [2].

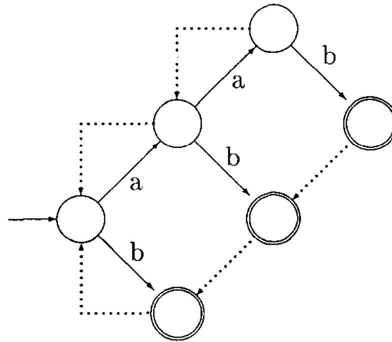

Fig. 1. Search automaton for the set of patterns $\{b, ab, aab\}$.

Like AC, we store the *goto* function as a directed rooted tree, where the nodes correspond to automaton states (pattern prefixes). The arcs of the tree are directed away from the root and every arc is labeled with a character from Σ. We organize the outgoing arcs of an internal node into a binary search tree with the arc labels as the keys for search, as suggested in [2]. The *degree*, or the maximum number of children of an internal node, of a *goto* tree is bounded by $|\Sigma|$. With this modification, computing each *goto* takes $O(\log|\Sigma|)$, or $O(\log|D|)$ time by Assumption 2.4. We also store with each node a pointer to its parent node. For each prefix x of a pattern, we keep a count of how many patterns have x as a prefix.

We call a string a *normal prefix* if it is a prefix of some pattern in D. For each proper prefix x we also define an *extended prefix*, $x\$$, by appending the character \$; the extended prefixes help in detecting patterns. The corresponding states are called normal or extended. In Example 2.1, the set of normal prefixes is $\{\varepsilon, a, b, \$, aa, b\$, aab, aab\$\}$ and the set of extended prefixes is $\{\$, a\$, b\$, aa\$, aab\$\}$. Note that some prefixes are both normal and extended. We extend the definition of *fail* to accommodate the extended prefixes as follows: Let w be a prefix (normal or extended). $fail(w) = x$ such that $|x| < |w|$ and x is the longest suffix of w such that x is a normal prefix. In Example 2.1, $fail(aab\$) = b\$$ but $fail(aa\$) \neq a\$$ since $a\$$ is not a normal prefix.

We recognize patterns as follows. When we reach a position of a text, we pretend that the next symbol is a \$. If we can make a transition to some normal prefix ending with a \$, then we know that a pattern has been matched at that position, since any normal prefix ending with a \$ must be a pattern in the dictionary. By applying *fail* repeatedly, we can report all the matching patterns in the order from the longest pattern to the shortest.

Suppose we are searching the text *abaabba* for the occurrences of the patterns in the dictionary D of Example 2.1. After reading the prefix *abaab*, we will be in the normal state *aab*. When we pretend to read \$ as the next symbol, we will *temporarily* enter a state *aab*\$ and since this is a normal state in the dictionary we report that a pattern

aab is recognized at the current location of the text. If we take $fail(aab\$)=b\$$ we see that we have matched another (smaller) pattern *b*. Again, if we take $fail(b\$)=\$$ we realize that no more patterns can be matched as this corresponds to the empty pattern ε. Since we keep track of the state *ab*, we can continue our search by reading the next symbol *b* from the text.

In [2], *fail* is stored as a directed, rooted tree with the arcs pointing towards the root. If $fail(s)=t$ then there will be an arc $s \to t$. The (in)degree of the *fail* tree is unbounded. We use a new method to store *fail* that enables us to insert new patterns, updating the *fail* function. Before describing our representation of *fail*, we explain some auxiliary prefixes and data structures that we use.

Let $* \notin \Sigma$ be a new symbol such that $* > a$ or any $a \in \Sigma$ in lexicographic comparison. For every prefix $w \in \Sigma^*$ we define $*w$ as the *complement* of w. We call w a *regular prefix*, and $*w$ a *complementary prefix*. For the purpose of representation, we extend the definition of *fail* as follows: for a regular prefix x, if $fail(x)=z$ then $fail(*x)=z$. We define a total ordering on the set of prefixes and their complements and call it the *inverted order* denoted by $<_{\text{inv}}$. For two distinct strings w and x, $w <_{\text{inv}} x$ if w^{R} comes before x^{R} in the lexicographic ordering, where x^{R} is the reverse of the string x.

Example 2.5. Consider the list of prefixes of $\hat{D}$ in Example 2.1 in the inverted order. It can be represented as: $\hat{S}=\varepsilon, a, aa, *aa, *a, b, aab, *aab, *b, \$, a\$, aa\$, *aa\$, *a\$, b\$, aab\$, *aab\$, *b\$, *\$, *$.

The number of extended prefixes is at most equal to the number of normal prefixes, so the number of regular prefixes is $O(|D|)$. The number of complementary prefixes is exactly equal to the number of regular prefixes. Therefore, the total number of prefixes in our dictionary structure is only $O(|D|)$, independent of Σ.

Let S denote the set of all normal and extended, regular and complementary prefixes of patterns in D. An important property of S is that for any string x, if S contains x then S contains every prefix of x. All string comparisons are made w.r.t. $<_{\text{inv}}$ ordering unless stated otherwise. For a nonempty string $x \in S$, $pred(x)$ is the largest string in S smaller than x. We need to compute $pred(x)$ when inserting a new string to know where the new string lies in the $<_{\text{inv}}$ order. In Example 2.5, $pred(aab)=b$ and $pred(b)=*a$. The following lemmas state some important properties of the prefixes in S that follow directly from the definition of $<_{\text{inv}}$ and the character $(*)$.

Lemma 2.6. *Let $w, x \in S$ be arbitrary regular prefixes. Let $y \in S$ be any prefix.*

1. $w <_{\text{inv}} *w$; *w is smaller than its complement.*

2. $w <_{\text{inv}} x <_{\text{inv}} *w$ *if and only if* $w <_{\text{inv}} *x <_{\text{inv}} *w$; *if we replace a regular prefix with a '(' and its complement with a ')' then the prefixes of S in the $<_{\text{inv}}$ order yield a list of well balanced parentheses.*

3. *If* $w <_{\text{inv}} y <_{\text{inv}} *w$ *then* $y=y'w$ *for some nonempty* y'; *w and $*w$ are, respectively, the smallest and largest prefixes in S with the suffix w.*

Lemma 2.7. *Let xa be the prefix we are inserting into S. Suppose $yb \neq xa$ is a prefix already in S. Then the following relations hold:*

1. $\varepsilon <_{\text{inv}} xa$.
2. *If $b <_{\text{inv}} a$ then $yb <_{\text{inv}} xa$. Similarly if $a <_{\text{inv}} b$ then $xa <_{\text{inv}} yb$.*
3. *If $b = a$ then $yb <_{\text{inv}} xa$ if and only if $y <_{\text{inv}} x$. Since x and y must already be in S we can determine whether $y <_{\text{inv}} x$ from S alone.*

Lemma 2.7 suggests a way to obtain the relative order of two prefixes without making a complete lexicographic comparison. We utilize Lemma 2.7 to compute the *pred* function by building an auxiliary search tree, called *ST*, on the top of all the prefixes of *S*. The elements of *ST* are basically pointers to the states of the *goto* tree sorted by the inverted order of the corresponding prefixes. We actually store *ST* as an a–b tree [13] with all the prefixes as leaves. For any state corresponding to a prefix *yb* we can determine the character *b* and the state with the prefix *y* in constant time as *y* is the parent of *yb* in the *goto* tree and hence this information is available with the parent pointer of *yb*. To utilize Lemma 2.7 we need to know the relative order of all the existing prefixes in *S*. Since we implement *ST* as an a–b tree, we can determine the relative order of two prefixes in $O(\log|D|)$ time.

We now describe one way to compute *pred*(*xa*) for a nonempty prefix *xa* not yet in *S*. Our aim is to find the largest prefix in *ST* smaller than *xa*. We start the search at the root of the *ST* and proceed towards the leaves. Suppose *yb* is the key associated with an internal node of the *ST*. At the next level we take the right child of the node if $yb <_{\text{inv}} xa$ and the left child otherwise. With this scheme we need $O(\log|D|)$ tree comparisons to find *pred*(*xa*). From now on we assume that we have a routine FINDPRED that computes the *pred* of a new prefix. We can conclude the following lemma.

Lemma 2.8. *Let xa be a prefix to be inserted into S. We can compute pred(xa) using* FINDPRED(*xa*) *in $O(\log^2|D|)$ time.*

Proof. We can determine *pred*(*xa*) with $O(\log|D|)$ tree comparisons. Since *ST* is organized as an a–b tree, the relative order of two existing prefixes can be determined in $O(\log|D|)$ time. Therefore, by Lemma 2.7 each tree comparison takes $O(\log|D|)$ time. From this it follows that we can determine *pred*(*xa*) in $O(\log^2|D|)$ time. □

To reduce the time to compute *pred*, we use a data structure invented by Dietz and Sleator [10] for solving the *order maintenance problem* efficiently. In the order maintenance problem, we define the following operations on a linear list *L*, initially containing one element. We attach an *L* to the name of every operation to distinguish them from the dictionary operations.

L_Insert(x, y): insert a new element y after the element x in *L*.
L_Delete(x): delete the element x from *L*.
L_Order(x, y): return true if x is before y in *L*, false otherwise.

Theorem 2.9 (Dietz and Sleator [10]). *The above three operations can be implemented in worst-case* O(1) *time.*

We store the prefixes of S in an auxiliary list data structure that we call the Dietz–Sleator list, or *DSL*. Using *DSL* we can determine the relative order of two prefixes in constant time instead of $O(\log|D|)$ time as achieved by using only *ST*.

Lemma 2.10. *Let x and y be two prefixes in S. By building DSL for all the prefixes of S, we can determine the relative order of x and y in constant time by Theorem 2.9. Hence, we can compute pred in* $O(\log|D|)$ *time.*

When we insert a new prefix xa, we need to find the value of $fail(xa)$. Below we show how to use $\text{pred}(xa)$ to compute $fail(xa)$ quickly.

Lemma 2.11. *Suppose w is a normal regular prefix, and* $w <_{\text{inv}} x <_{\text{inv}} *w$. *Then* $fail(x)=w$ *if and only if there is no normal regular y such that* $w <_{\text{inv}} y <_{\text{inv}} x <_{\text{inv}} *y <_{\text{inv}} *w$.

Proof. By Lemma 2.6, w is a suffix of x. If there is a normal regular prefix $y \in S$ such that $w <_{\text{inv}} y <_{\text{inv}} x <_{\text{inv}} *y <_{\text{inv}} *w$ then y must be a suffix of x longer than w. By definition, $fail(x) \neq w$ because of the presence of y. On the other hand, if there is no such y then $fail(x)=w$. □

In Example 2.5, $fail(aab)=b$ since there is no normal y such that $b <_{\text{inv}} y <_{\text{inv}} aab <_{\text{inv}} *y <_{\text{inv}} *b$, but $fail(aab) \neq \varepsilon$ since b can play the role of y in that case.

Lemma 2.12. *Let x be a regular prefix. Suppose* $pred(x)=w$. *If w is normal and regular then* $fail(x)=w$ *else* $fail(x)=fail(w)$.

Proof. If w is normal and regular then $w <_{\text{inv}} x <_{\text{inv}} *w$. From the definition of *pred*, there is no normal regular y such that $w <_{\text{inv}} y <_{\text{inv}} x <_{\text{inv}} *w$. Hence by Lemma 2.11, $fail(x)=w$. Suppose that w is not both normal and regular, and that $fail(w)=z$. Then from Lemma 2.11, $z <_{\text{inv}} w <_{\text{inv}} *z$ and there is no normal regular y such that $z <_{\text{inv}} y <_{\text{inv}} w <_{\text{inv}} *y <_{\text{inv}} *z$. Since x is right after w in the $<_{\text{inv}}$ order the same conditions hold for x. Hence $fail(x)=z=fail(w)$. □

In Example 2.5, $fail(aab)=b$ since $b=pred(aab)$ is normal and regular. Similarly $fail(b)=\varepsilon=fail(*a)$ and one may observe that $pred(b)=*a$.

We show later how SEARCH, INSERT and DELETE can be implemented using the *fail* and *pred* functions on the set S. We first describe how *fail* is represented. We store *fail* as a forest of *a–b* trees. Amir and Farach [3] use 2–3 trees in their suffix-tree automaton for a similar purpose. Each tree is called a *fail tree*, and the forest is called a *fail forest*. There is a one-to-one correspondence between the set of a–b trees and the

set of normal regular prefixes of S. If w is a normal regular prefix, then T_w denotes the a–b tree associated with w and contains as leaves all strings x such that $fail(x)=w$. In Example 2.5, T_a contains $\{aa, *aa\}$ as leaves, whereas T_{aab} is an empty tree. The root of the tree T_x contains a pointer to x. Each prefix $x \in S$ is a leaf in exactly one a–b tree, T_y, where $y=fail(x)$. The leaves of any *a–b* tree are sorted in $<_{\text{inv}}$ order. For any $x, fail(x)$ can be computed by starting at the leaf x finding the root of the tree T_y and taking the pointer to y.

When we insert a new pattern P into D, we inset its prefixes and extended prefixes into S, in increasing order of length. Thus, when we insert a string x into S, we have already inserted all the prefixes of x. Using Lemmas 2.8 and 2.10 we can find $w=pred(x)$. From Lemma 2.12 we can obtain $y=fail(x)$. If $w=y$ insert x into T_y as the leftmost leaf. Otherwise, insert x into T_y right after w. We can similarly find $z=pred(*x)$ (z could be x) and insert $*x$ into T_y right after z. We keep a separate bidirectional link between x and $*x$. Similarly when we delete a pattern P from D, we delete its prefixes in the order of decreasing lengths.

After inserting x and $*x$ into S we must create T_x. For this we must first identify those prefixes whose *fail* value changes to x. By Lemma 2.11, these are the prefixes with a suffix x and whose current *fail* value is y. But these are exactly the leaves of T_y properly enclosed between x and $*x$. We change *fail* for these nodes by a special *split* of T_y into T_y and T_x. We can similarly handle the case when a normal prefix x with $fail(x)=y$ becomes extended as a result of deletion of some pattern. In this case we fuse T_y and T_x into T_y with a special *concatenate*. These special operations were used similarly in [3].

As an example, consider inserting a new pattern $ab\$$ into the dictionary $\hat{D}$ of Example 2.1. For this we need to insert the prefixes $a, a\$, ab$, and $ab\$$ into the set $\hat{S}$ of Example 2.5. Since a and $a\$$ are already present in $\hat{S}$ we simply increment the reference count for them. We insert ab after $pred(ab)=b$, and $*ab$ after $pred(*ab)=*aab$. After this we have $\hat{S}=\{\ldots, b, ab, aab, *aab, *ab, *b, \ldots\}$. T_b contained $\{aab, *aab\}$ as leaves before the insertion of ab and $*ab$. We create T_{ab} by splitting T_b as described above. After this step, T_{ab} contains $\{aab, *aab\}$ as leaves, and T_b contains $\{ab, *ab\}$ as leaves. We similarly insert $ab\$$ and $*ab\$$.

We summarize the algorithms by giving pseudocode to describe our procedures for searching a text, and inserting and deleting a pattern from a dictionary. We use prefixes instead of states for clarity.

```
SEARCH(T = t_1 ... t_n)
  state ← ε
  for i ← 1 to n do
    while goto(state, t_i) = ⊥
      state ← fail(state)
    state ← goto(state, t_i)
    temp ← goto(state, $) /* Pretend a $ is read to check if any patterns match */
    if temp is not normal then temp ← fail(temp)
```

```
    while temp≠$ do /* Report all nonempty patterns */
        Print the pattern associated with temp /* Since temp ends in $ we have matched
        a pattern */
        temp←fail(temp) /* See if any smaller patterns match */

INSERT(P=p1 ... pm) /* pm=$ */
  suppose p1 ... pj is the longest prefix of P shared by some other pattern.
  increment the reference count for the prefixes of p1 ... pj.
  for i←j+1 to m do
    1. let x=p1 ... pi-1. Let a=pi. /* xa is being inserted. x is already in S*/
    2. compute y=pred(xa) using FINDPRED
    3. compute w=fail(xa) using Lemma 2.12
    4. if w≠y then
         insert xa into Tw right after y.
       else insert xa into Tw as the leftmost leaf.
    5. insert xa into ST after y.
    6. L_Insert xa into DSL after y.
    7. compute z=pred(*xa) using FINDPRED /* w=fail(*xa). */
    8. insert *xa into Tw right after z.
    9. insert *xa into ST after z.
   10. L_Insert *xa into DSL after z.
   11. repeat steps 1–10 to insert xa$ and *xa$. /* extended prefixes */
   12. from Txa by a split of Tw into Tw and Txa.
DELETE(P=p1 ... pm) /* pm=$ */
    suppose p1 ... pj is the longest prefix of P shared by some other pattern.
    decrement the reference count for the prefixes of p1 ... pj.
    for i←m downto j+1 do
      let x=p1 ... pi /* x is a normal prefix */
      if x$ is still in S then
        delete x$ and *x$ from their fail tree.
        delete x$ and *x$ from ST.
        L_Delete x$ and *x$ from DSL.
      let y=fail(x). Fuse Tx and Ty into Ty by a concatenate.
      delete x and *x from Ty.
      delete x and *x from STy.
      L_Delete x and *x from DSL.
```

The correctness and running time bounds for the above procedures follow from the previous lemmas.

Theorem 2.13. *Let D be a dictionary of patterns over an alphabet Σ. We can search a text T for occurrences of patterns of D in time* $O(|T|(\log|\Sigma|+\log|D|)+tocc\log|D|)$, *where tocc is the total number of patterns reported. We can insert or delete a pattern P in*

time $O(|P|(\log|\Sigma|+\log|D|))$. *Moreover, we require only* $O(|D|)$ *space to store the automaton.*

The Dietz–Sleator [10] list plays an interesting role in the generalization to two dimensions [6]. A key component in the semi-dynamic two-dimensional dictionary algorithm [4] is a subroutine that answers queries of the form: "Is x a prefix of y?", where x and y are suffixes of dictionary patterns in constant time. In [4] the prefix query is converted into a least common ancestor query on a suffix tree; least common ancestor queries on fixed trees can be answered quickly with enough preprocessing. One of the obstacles to making the algorithm of [4] dynamic is that it is not known how to efficiently answer the least common ancestor queries in a dynamic setting.

Therefore, we find a different way to handle the prefix queries in a dynamic setting. First, we convert the prefix queries into suffix queries on the reversed strings. To answer a suffix query, we can use the following lemma (closely related to Lemma 2.6).

Lemma 2.14 (Amir et al. [6]). *x is a suffix of y if and only if* $x <_{\text{inv}} y <_{\text{inv}} {*y} <_{\text{inv}} {*x}$.

The middle $<_{\text{inv}}$ ordering always holds and the outer two can be tested in constant time using the *L_Order* operation. This works even when the lists are dynamically changing due to insertions and deletions.

3. Linear time searching

We show how to improve the search time in this section. From Theorem 2.13, a text T can be searched in $O(\log|T|(|\Sigma|+\log|D|)+tocc\log|D|)$ time. The $\log|\Sigma|$ factor comes from the computation of each *goto* and the $\log|D|$ factor from the computation of each *fail*. We show that it is possible to speed up the computation of *fail* and achieve a faster searching algorithm. When $|\Sigma|$ is small and finite, or when $|\Sigma| \ll |D|$, this is better than the searching algorithm of Section 2. It does, however, slow down the update times of the dictionary.

In Section 2, we used *a–b* trees to store every fail tree. In an *a–b* tree, the number of children or the *degree* of a nonroot internal node v, denoted by $\delta(v)$, must be in the range $[a, b]$, and must be in the range $[2, b]$ if v is a root. In this section, we use a variant of a–b trees, which we call *hybrid a–b trees*, to store the fail trees. In a hybrid a–b tree different nodes may have different ranges for the number of children permitted. The ranges depend on the number of leaves in the fail forest.

Our hybrid a–b trees depend on a parameter, which is a fixed integer independent of $|D|$. For any $k \geq 2$, the hybrid trees will allow us to perform a *findroot* in $O(k)$ time, and any update operation in $O(k\ n^{1/k})$ time, where n is the total number of leaves in the forest of fail trees. Recall that n is twice the number of states in the automaton.

Let $\alpha \geq 16$ be the smallest power of 2 such that $\alpha^k \leq n \leq (2\alpha)^k$. Each internal node is designated as *small* or *big*, but may change its designation during the algorithm.

A small nonroot v has $\delta(v) \in [\alpha, 2\alpha]$, a small root v has $\delta(v) \in [2, 2\alpha]$, a big nonroot v has $\delta(v) \in [2\alpha, 4\alpha]$, and a big root v has $\delta(v) \in [2, 4\alpha]$. We maintain lists of small and big nodes using a separate link. Let $\#small$ and $\#big$ denote the number of such nodes. Our ranges imply that any nonroot has $\Theta(n^{1/k})$ children and there will be $O(k)$ levels in any hybrid tree.

The operations *insert, delete, concatenate, and split* can be implemented in a similar fashion to that used for regular a–b trees. Each operation visits or modifies at most b nodes at each level of the tree and may cause at most a constant number of nodes per level to violate the constraints on the number of children allowed.

In the rest of the section we show how to handle overflows and underflows in the number of children of a node of a hybrid tree. Define $excess = n - \alpha^k$, and $m = (\alpha^k + (2\alpha)^k)/2$. We maintain two invariants:

$$\#small + excess \leqslant (2\alpha)^k - \alpha^k \qquad (1)$$

$$\#big \leqslant excess \qquad (2)$$

Note that all nodes will be small when $n = \alpha^k$, and big when $n = (2\alpha)^k$. If n goes above $(2\alpha)^k$, we redefine all the big nodes as small nodes and start operating in the interval $(2\alpha)^k \leqslant n \leqslant (4\alpha)^k$ with small nonroot nodes having the range $[2\alpha, 4\alpha]$ and big nonroot nodes having the range $[4\alpha, 8\alpha]$. We redefine *excess* and *m* accordingly. We can do a similar thing when n falls below α^k. The following two lemmas simplify invariant maintenance.

Lemma 3.1. *If there are n leaves in the fail forest, then the number of internal nodes of the fail trees is at most* $3n/5$.

Proof. The number of fail trees is equal to the number of normal regular prefixes. Since there is a complementary prefix for every regular prefix, the total number of prefixes n is at least twice the number of normal regular prefixes. This implies that the number of fail trees is at most $n/2$. Thus the number of root nodes is at most $n/2$.

Since $\alpha \geqslant 16$, the degree of any nonroot internal node in a fail tree must be at least 16. This means that the number of nonroot internal nodes can be at most $n/15$. Hence the total number of internal nodes of all fail trees is at most $n/2 + n/15$ which is less than $3n/5$. □

Lemma 3.2. *If* $n \leqslant m$ *then invariant* 1 *is satisfied. Similarly if* $n \geqslant m$ *then invariant* 2 *is satisfied.*

Proof. If $n \leqslant m$ then $excess \leqslant ((2\alpha)^k - \alpha^k)/2$. From Lemma 3.1 it follows that $\#small \leqslant 3n/5 \leqslant 3/5((\alpha^k + (2\alpha)^k/2)$. Hence $\#small + excess \leqslant \frac{4}{5}(2\alpha)^k - \alpha^k/5 \leqslant (2\alpha)^k - \alpha^k$; the last inequality holds provided $k \geqslant 2$. One can similarly verify that $\#big \leqslant excess$ whenever $n \geqslant 5\alpha^k/2$, which holds if $n \geqslant m$ and $k \geqslant 2$. □

Lemma 3.2 implies that when $n \leqslant m$ we only have to control $\#big$, as invariant 1 is always satisfied in this case, and the value of $\#small$ has no effect on invariant 2. Similarly when $n \geqslant m$, we only have to control $\#small$.

If $n \leqslant m$, we do any update operation in such a way that $\#big$ never increases. Any update operation can increment or decrement the value n by at most one. If invariant 2 is satisfied before a *delete* but violated afterwards, we have to decrement $\#big$ by only one to maintain invariant 2. Similarly, when $n \geqslant m$, we update in such a way that $\#small$ never increases. We may have to decrement $\#small$ by at most one after an *insert*. The operations *concatenate and split* do not change the value of n.

Two primitive operations needed for implementing hybrid trees are those that handle overflow and underflow of an internal node. Those are the cases when the degree of an internal node goes one above or below its declared range. We show how overflow and underflow can be eliminated without violating the invariants. We annotate each line of pseudocode by an ordered pair (i, j) implying that $\#small$ changes by i and $\#big$ by j after executing the line.

overflow(v):
  Suppose we are controlling $\#small$.
    If v is small redesignate it as a big node. $\{(-1, 1)\}$
    If v is big break it into two big nodes. $\{(0, 1)\}$
  Suppose we are controlling $\#big$.
  (*) If v is small break it into two small nodes. $\{(1, 0)\}$
  (*) If v is big break it into one big and one small node. $\{(1, 0)\}$
underflow(v):
  suppose v is a root node.
    if $degree(v) = 1$ remove v. Name its only child as the new root. $\{(-1, 0)$ or $(0, -1)\}$
  suppose v is a nonroot node with an immediate sibling w.
    let $[a, b]$ be the range of w.
    if $\delta(w) \geqslant a+1$ transfer one child of w to v. $\{(0, 0)\}$
    otherwise
      if v and w are both big(small) then
        fuse them into one big(small) node. $\{(0, -1)$ or $(-1, 0)\}$
      if one of them is big and the other small fuse them into one big node. $\{(-1, 0)\}$

One can check that each change in $\#small$ or $\#big$ does not violate the invariants. Also *underflow*(v) never increases $\#small$ or $\#big$. If an invariant is violated by 1 as a result of an *insert* or *delete*, applying *overflow* or *underflow* does not exacerbate the violation, and we restore the violated invariant as explained below.

A single overflow or underflow can be handled in $O(\alpha)$ time. Correcting the overflow(underflow) of a node may cause an overflow(underflow) of its parent. This is the way any update operation is implemented; we make changes to an internal node and correct any overflows or underflows that may propagate all the way to the root. Since there are $O(k)$ levels in any tree, we can implement any update operation in $O(k\,\alpha)$ time.

Finally we show how to decrement *#small* or *#big* by at least one in $O(k\alpha)$ time. These operations are necessary to restore the invariants that may be violated after an *insert* or *delete* operation. There is no problem with the invariants if the following operations cause *#small* or *#big* to go down by more than one.

decreasesmall( ): Pick some small node v. If v is a root, redesignate it as a big node. If v is not a root, it must have an immediate sibling w. If w is small, then fuse v and w into a big node. If w is big, fuse v and w into one or two big nodes depending on $\delta(v)+\delta(w)$. This may propagate underflows to the ancestors of v which can be handled without increasing *#small* as we noted earlier that *#small* does not increase during an *underflow* operation.

decreasebig( ): Pick some big node v. Split v into one or two small nodes depending on $\delta(v)$. Any propagated overflows can be handled without increasing *#big* as can be observed from the lines marked (*) of *overflow*.

Theorem 3.3. *For a fixed integer $k \geqslant 2$, we can search a text T in $O(|T|(k+\log|\Sigma|)+k\,tocc)$ time. Furthermore we can insert or delete a pattern P into a dictionary D in $O(|P|(k|D|^{1/k}+\log|\Sigma|))$ time.*

Proof. For any state v, *fail*(v) can be obtained with a *findroot* operation. Since this takes only $O(k)$ time on a hybrid a–b tree we can search T in $O(|T|(k+\log|\Sigma|)+k\,tocc)$ time. Any update operation will be accompanied by at most one *decreasesmall or decreasebig* operation. Each of these operations take $O(k\alpha)$ time. Since α is $O(|D|^{1/k})$, it follows that P can be inserted or deleted in the specified time. □

4. Building *D* in linear time

In this section, we show how to build the initial dictionary in $O(|D|\log|\Sigma|)$ time. This is better than building the dictionary by repeated insertion of patterns which requires $O(|D|\log|D|)$ (as in [3,5]) or $O(|D|\ |D|^{1/k})$ time depending on the data structure used. We can build the dictionary in linear time using either regular a–b trees or hybrid a–b trees to store the *fail* function.

Let $D=\{P_1,\dots,P_s\}$ be the initial set of patterns given. Our first major goal is to sort all the (normal and extended, regular and complementary) prefixes of S in $<_{\text{inv}}$ order. For this purpose we partition S into two disjoint sets S_1 and S_2. S_1 contains those prefixes of S ending in a \$ and S_2 contains the rest of S.

We start by building a sorted list of the prefixes of S_2. These prefixes are exactly the prefixes of the patterns $P_1,\dots,P_s$ and $*P_1,\dots,*P_s$ with the \$ stripped from their right ends. We build a suffix tree for the *reverses* of these $2s$ strings using the suffix tree construction of [12] as modified by [5]. There will be a one-to-one correspondence between the leaves of the suffix tree and the prefixes of S_2 as proved in [5, Section 2, Lemma 2]. If we sort the children of every internal node of the suffix tree by the labels

of the edges connecting the children and the parent, then the left-to-right order of the leaves of the suffix tree is the $<_{\text{inv}}$ order of the prefixes of S_2. We can build the suffix tree in $O(|D|\log|\Sigma|)$ time. Rearranging the children and scanning the leaves in the left-to-right order takes only $O(|D|\log|\Sigma|)$ time. So we can order all the prefixes of S_2 in $O(|D|\log|\Sigma|)$ time.

The prefixes of S_1 (and S_2) occur consecutively in S (this follows from the assumptions made on \$ in Section 2). Furthermore, the prefixes of S_1 are in one-to-one correspondence with the prefixes of S_2, and occur in exactly the same $<_{\text{inv}}$ order as the corresponding prefixes of S_2. Thus we can build a sorted list of the prefixes of S_1 by scanning the list of the prefixes of S_2, and for each prefix x of S_2 inserting (in the same relative order) the prefix $x\$$ into S_1. Finally we concatenate both S_2 and S_1 to obtain S in the $<_{\text{inv}}$ order. This takes $O(|D|)$ time.

In our dictionary structure, we used both a binary search tree and a Dietz–Sleator list on the sorted list of prefixes. Both of these can be built in $O(|D|)$ time (see the discussion on building a–b trees below).

Our second major goal is to compute the *fail* function. Specifically, to each prefix w, we want to associate those strings x, such that $fail(x)=w$. We temporarily keep these strings associated with w in a sorted list in $<_{\text{inv}}$ order, which we call *w*'s *fail list*. The strings on the fail list will become the leaves of the fail tree T_w. By Lemma 2.11, these are precisely the strings x such that $w <_{\text{inv}} x <_{\text{inv}} *w$, and there is no y such that $w <_{\text{inv}} y <_{\text{inv}} x <_{\text{inv}} *y <_{\text{inv}} *w$. Intuitively, if we think of each regular complementary pair of a normal prefix as a pair of matching parentheses, we want to find the deepest pair of parentheses containing x; if w is the left parenthesis in that pair, then $fail(x)=w$. The natural way to do this is to scan the list S keeping track of the unmatched normal prefixes on a stack STK. We use the following scanning rules in the order below:

1. If $x \neq \varepsilon$ is regular, then $fail(x)=top(STK)$, so append x to the fail list for $top(STK)$.
2. If x is normal and regular, then $push(x)$ onto STK.
3. If x is normal and complementary, $pop(STK)$.
4. If $x \neq \$$ is complementary, then $fail(x)=top(STK)$, so append x to the fail list for $top(STK)$.

The scanning rules ensure that $fail(x)$ is never x for a regular prefix, and $fail(*x)=fail(x)$ for every complementary prefix, as desired. Scanning the list takes constant time per prefix, so this step takes $O(|D|)$ time.

Finally we organize each fail list into a fail tree. For each w, we have computed the leaves of T_w in sorted order as *w's* fail list. To build T_w we build a search tree with the elements of the fail list as leaves. How we proceed depends slightly on whether we are using standard a–b trees or hybrid a–b trees. We can build any standard a–b tree given the sorted list of leaves in linear time [13]. To build hybrid a–b trees, let n be the total number of leaves in all trees. Given $k \geqslant 2$, choose α as described in Section 3 and define the value m. Build the trees with all small nodes if $n \leqslant m$, and with all big nodes if $n > m$. The correctness of this construction follows from Lemma 3.2. This also takes only $O(|D|)$ time.

We can summarize our linear time construction algorithm as follows:
BUILD($D = \{P_1, \ldots, P_s\}$)
1. Build suffix tree on the *reverses* of $P_1, \ldots, P_s$, $*P_s * P_1, \ldots$, to order S_2.
2. Scan S_2 and build S_1.
3. Concatenate S_2 and S_1 to obtain S.
4. Build the search tree, ST, and the Dietz–Sleator list DSL for the sorted list of prefixes.
5. Scan the sorted list and compute the fail list of each prefix.
6. Convert each fail list into a (regular or hybrid) a–b tree.

The entire process takes linear time as every step has been shown to take $O(|D|\log|\Sigma|)$ time. We can summarize:

Theorem 4.1. *The initial dictionary D can be constructed in* $O(|D|\log|\Sigma|)$ *time.*

Acknowledgments

We thank Amihood Amir for answering our questions about [3, 5] and for pointing out the importance of the large/unbounded alphabet case, thereby motivating us to find a simpler, more general solution to the basic problem.

References

[1] A.V. Aho, Algorithms for Finding Patterns in Strings, in: J. van Leeuwen ed., *Handbook of Theoretical Computer Science* (Elsevier, Amsterdam, 1990) 255–300.
[2] A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search, *Comm. ACM* **18** (1975) 333–340.
[3] A. Amir and M. Farach, Adaptive dictionary matching, *Proc. 32nd IEEE Symp. Found. Comput. Sci.* (1991) 760–766.
[4] A. Amir and M. Farach, Two-dimensional dictionary matching, *Inform. Process. Lett.* **44** (1992) 233–239.
[5] A. Amir, M. Farach, Z. Galil, R. Giancarlo and K. Park, Dynamic dictionary matching, *J. Comput. System Sci.*, to appear.
[6] A. Amir, M. Farach, R.M. Idury, J.A. La Poutré and A.A. Schäffer, Improved dynamic dictionary matching, *Proc. 4th Ann. ACM-SIAM Symp. on Discrete Algorithms* (1993) 392–401.
[7] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Comm. ACM* **20** (1977) 762–772.
[8] M.T. Chen and J. Seiferas, Efficient and elegant subword-tree construction, in: A. Apostolico and Z. Galil eds., *Combinatorial Algorithms on Words*, NATO ASI Series, Vol. F12 (Springer, Heidelberg, 1985) 97–107.
[9] B. Commentz-Walter, A string matching algorithm fast on the average, *Proc. ICALP'79*, Lecture Notes in Computer Science, Vol. 71 (Springer, Berlin, 1979) 118–132.
[10] P. Dietz and D.D. Sleator, Two algorithms for maintaining order in a list, *Proc. 19th ACM Symp. Theoret. Comput. Sci.* (1987) 365–372.
[11] D.E. Knuth, J.H. Morris and V.B. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977) 323–350.
[12] E.M. McCreight, A space economical suffix tree construction algorithm, *J. ACM* **23** (1976) 262–272.
[13] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching* (Springer, Berlin, 1984).
[14] B. Meyer, Incremental string matching, *Inform. Process. Lett.* **21** (1985) 219–227.
[15] P. Weiner, Linear pattern matching algorithms, *Proc. 14th IEEE Symp. on Switching and Automata Theory* (1973) 1–11.