Dictionary Automaton in Optimal Space

Stefan Edelkamp

Institut für Informatik, Albert-Ludwigs-Universität,
Am Flughafen 17, D-79110 Freiburg;
eMail: edelkamp@informatik.uni-freiburg.de

Abstract. In this paper we describe the data structure of a time and space efficient string dictionary automaton, providing insertion and deletion of strings and finite state machine based substring searching.
If the input alphabet is bounded and the pattern are mutually substring free an optimal worst case bound of $O(|m|)$ for both insertion and deletion of a pattern m is achieved.
The underlying structure is a multi suffix tree. Let M be the set of strings stored in the dictionary resulting from an arbitrary sequence of *Insert* and *Delete* operations. One main result in this paper is that the space complexity of the augmented multi suffix tree is $O(d)$ with $d = \sum_{m \in M} |m|$. This bound is optimal if we assume that each pattern in the dictionary uses at least linear space.
Additionally, we present a new on-line substring search algorithm to find one substring of x in time $O(|x|)$. The novelity of the approach is to realize a multi suffix tree based finite state automaton for substring searching that performs a state transition in amortized constant time.

1 Introduction

String matching algorithms for multiple pattern are applied to a variety of practical relevant tasks such as text editing, molecular biology and information retrieval. The input is a possibly large set of strings and a query string. The problem is to determine one (or all) occurrences of the stored strings as substrings within the query. Usually, the algorithm of choice is Aho and Corasick's algorithm [1] or one of its refinements [5,7,9]. The approach is time and space optimal and the output can easily be transferred into a finite state machine, processing one character of the query at a time.

The drawback of this algorithm emerges, if dynamic changes such as insertion and deletions to the set of strings have to be performed. Therefore, a time and space efficient representation of a dictionary for strings is required, leading to the so-called *dynamic dictionary matching problem*, *DDMP* for short [3].

Amir et al. and Idury and Schäffer independently provide (sequential) algorithms for the *DDMP* with worst case bounds $O(|m| \log d)$ for insertion and deletion and $O((|x| + tocc) \log d)$ for searching all substrings of x, with $tocc$ being the number of substring pattern in the output [3,15]. Amir et al. improve this to $O(|m| \log d / \log \log d)$ for insertion and deletion and to $O((|x| + tocc) \log d / \log \log d)$ for searching substrings of x in the dictionary [4].

We achieve the optimal time bounds for insertion and deletion assuming that no dictionary string is a substring of another. This is a very important special case if one is just interested in finding but one substring of a query.

Several published papers highlight different aspects of dictionary matching, such as randomization [2], two-dimensionality [16] and parallelism [12]. We concentrate on two aspects: Achieving optimal space bounds and incremental search.

Our work is inspired by the application of an efficient dictionary automaton for duplicate elimination in huge regular problem spaces that do not fit into main memory [22]. The problem spaces are implicitly described by a bounded number of transitions such that transition sequences can be regarded as strings.

The strings in the dictionary are interpreted as a set of forbidden words determined in a learning phase together with abbreviations that lead to the same situation. When in the search phase a duplicate is found as a substring of the generated path, the exploration tree can be pruned, since the regular structure of the search space guarantees that a shorter path has been or will be examined. The finite state machine pruning technique has successfully been applied to the (n^2-1)-Puzzle [17].

Instead of the above off-line scenario, an incremental learning scheme is desirable, in which duplicate gathering and exploration are carried out concurrently [8]. In this case, we exhibit the following requirements to the string dictionary: It has to provide efficient insertion and deletion algorithms, such that new duplicate information can be implemented without leading to a bottleneck in the search. Secondly, since state space search algorithm s usually consider very different horizon nodes, the dictionary has to serve as an finite state automaton with a characterwise scanning procedure of the input query [14]. Last but not least, the space consumption of the dictionary should be small, since the exploration itself will be very space consuming.

As the main result of this paper in contrast to the deletion of a string in the dictionary provided by Amir et al. we release all space used by the deleted string since we will not refer to the string afterwards. Their algorithm does not tackle this problem at all so we have to conclude that the space complexity is bounded by $O(d^*)$, with d^* being the total length of all strings ever be considered. In this case the allocation of free space is accumulated in several transactions and memory resources are soon exhausted.

The paper is organized as follows. Firstly, the algorithm of Aho and Corasick and Meyer's approach for an incremental construction of the failure function are recalled. Afterwards we present the data structure multi suffix trie as used in Amir et al. Then we turn to our new deletion algorithm that achieves the optimal space bound. Finally, we present a new automaton based incremental searching scheme accepting if a dictionary string is contained in the query.

2 Dictionary Matching

Aho and Corasick maintain a trie to represent the strings in the dictionary. Each node in the tree has outgoing edges labeled by pairwise different characters of the alphabet. In order to simultaneously search x the authors devise a failure function f defined as follows: $f(t) = t'$ if and only if $v(t')$ is the maximum suffix of $v(t)$ which is also prefix of at least one string in the dictionary, with $v(t)$ being the corresponding prefix of t. The time for calculating f is linear to the sum of strings in the dictionary but Aho and Coarsic only describe how to determine f from scratch. The authors further propose an additional breadth-first search algorithm to infer a finite state automaton accepting stored substrings of a given queries. A query is processed one character after another with a constant time consumption for each step.

The difficulties of the incremental construction of T (strings are only inserted) and its failure function are first addressed by Meyer [19]. Let t' be a new node in the trie and z be incoming character leading from an existing node t to t'. The failure link $f(t')$ is the the first existing successor of $f(t)$, $f(f(t))$, $f(f(f(t)))$, and so on, for which the state transition with z is defined. The problem, however, is to find all nodes, whose failure function has to change, since these nodes might be located in different branches of the trie.

Meyer maintains the inverse f^{-1} of f at each node in T and argued that f^{-1} can be efficiently represented as the so-called *inverse failure tree* with nodes in T. The main procedure $Compute f^{-1}$ recursively determines all potential predecessors of nodes with a failure link to be changed and checks whether or not the transition with z is defined. Let $m_{\max}$ be the maximum length of all dictionary strings. Meyer postulates an overall time for his node-by-node insertion scheme of $O(m_{\max} d)$. Since $m_{\max}$ is not constant, the algorithm is not time optimal.

3 Multi Suffix Trees

A *patricia tree* (*practical algorithm to retrieve information coded in alphanumeric*, cf. [20]) is a compact representation of a trie where all nodes with one successor are merged with their parents.

A *suffix tree* is a patricia tree corresponding to the suffixes of a given string and were proposed as an space-efficient alternative to Weiner's position trees [18,24]. The edge labels correspond to substrings of a dictionary string m and are addressed by $m(f, l)$, with f and l describing the index of the first and the last character of the interval, respectively. The substring information is stored at the outgoing node, which gives the extended locus of the stored substring. The operations for accessing the successor nodes are *getChild*, *setChild*,*removeChild*, and *anyChild* with obvious semantics. Besides the substring and the successor information a suffix node v consists of a predecessor link (*pred*), the number of children (c), the length (*len*) of the string referring to the path from the root to v and the so-called *suffix link* (*sl*) allowing shortcuts in the traversal of the tree.

McCreight presents a linear time algorithm to build a suffix tree of a given string $m\$$ with \$ not being represented in the alphabet [18]. Linear time construction algorithms were also given for a more general case of integer alphabets with characters belonging to the range $[0, |m|^c]$ where c is a constant [10]. This and other approaches in constructing suffix trees build partial suffix trees for some mutually disjoint subsets of positions, and merge the forest into one final suffix tree [11,13,21].

Some notations adopted from McCreight: A *locus* of a string is the node representation of the string in the suffix tree and *extension* of string u is any string having u as a prefix. Thus, an *extended locus* of string u is the locus of the shortest extension to u that is represented in the suffix tree. Similar the *contracted locus* of string u is the locus of the longest prefix of u that is represented in the suffix tree. Recall, that in the construction of a suffix tree according to McCreight the suffixes of a string $m\$$ are processed from the longest to the shortest in contrast to the construction proposed by Ukkonen [23].

McCreight's approach can be naturally extended to more than one string for example by building the suffix tree of the string $m_1\$_1, \ldots, m_n\$_n$. The remaining step to a full dictionary is based on the observation of Amir et al. that the suffix tree for $m_1\$_1, \ldots, m_n\$_n$ is isomorphic to the compacted trie for all suffixes of $m_1\$_1$ up to to all suffixes of $m_n\$_n$. Furthermore, the trees are identical except for the labels of edges incident to leaves. This observation allows to insert a string $m_{n+1}\$_{n+1}$ into an existing suffix tree. To avoid an unbounded extension of the alphabet, $\$$ represents all additional $\$_i$ characters. This causes the comparison of $\$$ with itself to fail. The construction of a multi suffix tree is composed of a sequence of *InsertBetween* and *InsertAfter* operations. The former one leads to a new node on a given edge, while the latter one just adds a branch to an already existing node. The insertion scheme and its correctness result can be found in Amir et al. which in fact refer to McCreight. The time complexity for inserting a string m into the dictionary is $O(|m|)$.

4 Deletion in Optimal Space

Deleting a pattern $m\$$ in a multi suffix tree S as proposed in Amir et al. does not release all the space consumed by $m\$$, since a contracted node in the suffix tree might need a removed pattern to extract the characters of the described substring. The proposed strategy will be first to find a valid string representation for the leaves and then to propagate it to the inner nodes. Let a trie T refer to the set of reversed pattern $(m\$)^R$. Since the suffixes of $m\$$ are the prefixes of $(m\$)^R$ the following fact is obvious and allows to set links from the nodes in T to the leaves of suffix tree S.

Theorem 1. *(Bijection trie - multi suffix tree) There is a bijection from every node (except of the root) in T to every leaf node in S.*

A direct consequence is that the number of nodes in the multi suffix tree is bounded by $O(d)$, since in a patricia tree there are less inner nodes than leaves.

In the deletion process of Amir et al. all leaves of a given string $m\$$ are found by searching the multi suffix tree structure [3]. According to Theorem 1, however, this set can directly be determined by the associated trie T. This leads to the following concise deletion routine.

1. Delete all nodes in T according to $(m\$)^R$ and fill a stack with links to the leaves in S. (These links can be created during the insertion process.)
2. Delete the leaves in S and, if necessary, its parent left with a single child.

The stack sequentializes the deletion in S and T. Deletion can be executed from the shortest suffix to the longest one, but the chosen direction is more convenient, since in all iterations the suffix links remain correct.

The unconsidered problem of Amir et al. is that the deleted string may still be required at inner nodes in the stored substring information. According to the space limitation of $O(d)$ we have only $O(1)$ space at each node. Therefore, the string information has to be resolved by referring intervals of previously inserted strings. However, these links are sensible with respect to the order of the insertion process. Therefore, all nodes that refer to the deleted string have to be found and updated.

The basic idea of the proposed improvement is to draw a connection between the string representations found at leaves and at the corresponding inner node. Unfortunately, the ordinary predecessor relationship changes over time due to insertion of new suffixes such that we have to compensate this loss of information.

Therefore, we introduce *twins* referring to the history of leaf generations. If *InsertAfter* is called both a new successor and a new twin successor is generated. In case of *InsertBetween* only the twin of a new inner node is created pointing to the new leaf. The twin of the preceding node remains unchanged. In Fig. 1 the combined structure of successor and twin successor information is visualized.

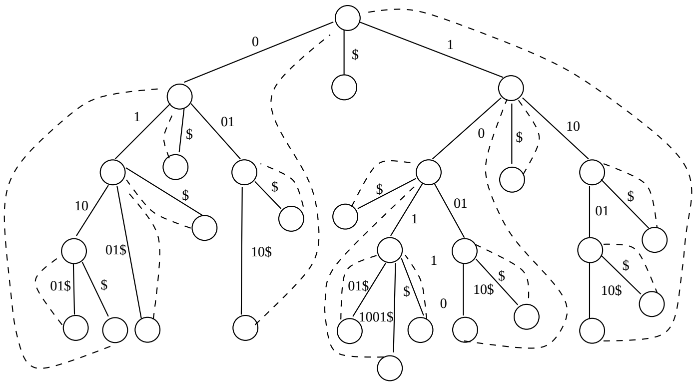

Fig. 1. The successor and twin successor structure in a multi suffix tree of *1100110$*, *1011001$*, and *10101$*.

Now we can turn our attention to the deletion procedure *RemoveLeaf* as shown in Table 1 and illustrated in Fig. 2. Let q be the leaf to be deleted and s be its predecessor. Furthermore, let p be the unique twin predecessor of q, and r and r' two (further) twin successors of s (if existing).

The ordinary predecessor s is equal to the twin predecessor p.
If s has no further successor, delete s according to the contraction condition of patricia trees. Otherwise take an arbitrary twin successor r and transfer the string representation from this leaf to s.

The ordinary predecessor s is different to the twin predecessor p.
Take the twin successor r of s and assign r to be the twin successor of p.

```
s ← q.pred; p ← q.twin.pred                      {assign (twin) predecessor of q}
j ← findLabel(p.twin, q)                              {label of the outgoing edge}
removeChild(s, q); removeChild(p.twin, q)                    {delete leaf node q}
r ← anyChild(s.twin)                                   {find twin successor of s}
if (s = p)                                         {direct predecessors are equal}
  if (s.c > 1)                                   {s has more than one successor}
    ChangeLeaf(r)                                         {adjust representation}
  else                                                {s has only one successor}
    RemoveInner(s)                                          {delete inner node s}
else                                           {direct predecessors are not equal}
  removeChild(s.twin, r); setChild(p.twin, r, j)                  {transfer twin}
  ChangeLeaf(r)                                     {adjust representation for p}
  if (s.c > 1)                                   {s has more than one successor}
    r' ← anyChild(s.twin)                           {find another twin successor}
    ChangeLeaf(r')                                        {adjust representation}
  else                                                {s has only one successor}
    RemoveInner(s)                                          {delete inner node s}
```

Table 1. Pseudo code for *RemoveLeaf* deleting a single leaf q in the multi suffix tree. The sub-procedure *RemoveInner* deletes a node in the multi suffix tree structure due to the enforced contraction and *ChangeLeaf* propagates the string representation to the parent node.

If s does not have another twin successor r' it can obviously be deleted. Otherwise, take r' and transfer its string representation to s.

Lemma 2. *Let* I *and* L *be the set of respective internal an leave node in the multi suffix tree S. Further, let $\Gamma(p)$ be the set of ordinary successors of p and let $\Gamma'(p)$ be the set of twin successors of p. Then we obtain the following invariance*

(Ia) *For all inner nodes p in* I *there exist a twin successor q of p in* L *with the same string representation. ($p.m = q.m$).*

(Ib) *For all inner nodes p in* I *we have $|\Gamma'(p)| = |\Gamma(p)| - 1$, i.e., for each inner node the number of ordinary successors exceeds the number of twin successors by one.*

Proof. The three only operations that change the structure of a multi suffix tree are *InsertAfter*, *InsertBetween* and *RemoveLeaf*.

InsertAfter Let cl be the contracted locus according to the insertion of a suffix of string m. The newly generated leaf enlarges both sets $\Gamma(cl)$ and $\Gamma'(cl)$ by one element. Therefore, we have *(Ib)*. The string representation at cl is set to m such that *(Ia)* is also fulfilled.

InsertBetween Since the string representation of both newly generated nodes is set to m condition *(Ia)* remains valid. The inner node gets two successors: The extended locus and the new leaf. Therefore, we have $2 = |\Gamma(p)| = |\Gamma'(p)| + 1$.

RemoveLeaf Let q be the node to be deleted, s its ordinary predecessor and p its twin predecessor. The algorithm considers two cases (cf. Fig. 2):

$s = p$ Since q is in $\Gamma(s) \cap \Gamma'(s)$ the condition *(Ib)* is trivially fulfilled. If $|\Gamma(s)| > 1$, then there is a leaf r in $\Gamma'(s)$. Through the *ChangeLeaf* operation this leaf determines the string representation of s such that s does not access $q.m$ any more. Therefore, we have *(Ia)* at node s. If, however, $|\Gamma(s)| = 1$, then s is deleted and nothing remains to be shown.

$s \neq p$ This case is the most difficult one. If $|\Gamma(s)| = 1$ then s will be deleted. Furthermore, $\Gamma'(p)$ is set to $\Gamma'(p) - \{q\} \cup \{r'\}$, such that $|\Gamma'(p)|$ remains unchanged. Otherwise $|\Gamma(s)| = k > 1$. Therefore, by *(Ib)* at the time when q was deleted s had $k+1$ ordinary successors and k twin successors. Hence, beside r' there exists another twin successor r of s, which can be utilized to adjust the string representation of p. More precisely we set $\Gamma'(p)$ to $\Gamma'(p) - \{q\} \cup \{r\}$ fulfilling *(Ia)* and *(Ib)* as before.

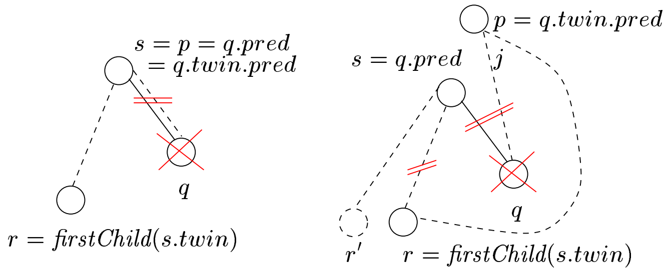

Fig. 2. The proof of the degree invariance of twin successorship.

Theorem 3. *(Space Optimality) Let M be the set of strings represented in a dictionary after an arbitrarily given sequence of string insertion and deletion and let d be the sum of the lengths of all strings in M. The space consumption of the dictionary represented by a multi suffix tree data structure is bounded by $O(d)$.*

Proof. The invariance *(Ia)* in Lemma 2 proves that after any sequence of insert and deletion operations at all inner nodes only those string representation are found that are also referred by the leaves.

By Theorem 1 we know that all leaves in the multi suffix tree correspond to nodes in T. Therefore, all space consumed by m can be set free after *Delete(m)* has been finished, leading to a space complexity of $O(d)$.

5 Searching One Substring

There are two different approaches to find a substring of a given string x in the suffix tree. Amir et al. determine the longest pattern prefix h of the string

```
at ← (at + 1) mod |el.l − el.f + 1|             {init h_j (set i_j ← i_{j−1})}
if (at = 0) return SearchChild(q, x_j)          {a new node has been reached}
while (el.m[at + f] ≠ x_j)        {as long as the character does not fit}
  if (cl.IsRoot) at ← at − 1                    {special case cl is root}
  w ← el.m(f, f + at − 1)                 {w is substring of edge {cl, el}}
  cl ← cl.sl                                {omit character (i_j ← i_j + 1)}
  if (Rescan(w, at, cl, el))                                   {rescan w}
    return SearchChild(q, x_j)                    {w exhaust edge label}
h ← el.m(f + at − cl.len, f + at)                {maximal suffix detected}
```

Table 2. δ: State transition of suffix state $q = (at, cl, el, h)$ and input character x_j resulting in the state representation of the successor state.

stored in the suffix tree which matches x *starting* at position i. In contrast to this algorithm we fix the longest substring h of the strings which matches x *ending* at position i, $i \in \{1, \ldots, |m|\}$. If the strings m to be stored are *mutually substring free*, i.e. no string is a substring of another, then in both cases the only thing to do is to check if longest prefix h coincides with one string in the dictionary. The representation of h as a substring is determined by the string stored at the extended locus of h. Therefore, we only have to test if h is maximal, i.e., if the length of h equals to the length of $h.m$.

The requirement of substring freedom is not a severe restriction if one is interested only in one substring contained in the query, since a substing of a given string is trivially a substring of any superstring which in this case is the query itself. Therefore, we restrict the dictionary to minimal substrings.

Our algorithm will be called *strongly incremental* because it does not refer to characters x_j with $j < i$. The main idea is to split the *Search* operation into several suffix state transitions δ. A *suffix state* is a momentary picture of the search algorithm in the multi suffix tree. Through the compressed description of the suffix tree the reference to only one node as in the case of the algorithm of Aho and Corasick is not sufficient.

Therefore, a suffix state is defined as a triple of the contracted locus cl the extended locus el and the distance on the edge drawn from cl to el. For a better description of the algorithm we add a fourth component to the state, given by the substring information of the already scanned substring. Obviously, a suffix state consumes only constant space.

We now describe the transition δ from one suffix state q to another one given the input character x_j (cf. Table 2). By traversing the suffix links we continuously omit the first character, unless we find a match of the next given character x_j.

1. First of all try to scan x_j at the position given by suffix state q. Therefore, increase at by one.
2. In case a new node is encountered, search for the extended locus of the substring h_j to be build. If no such node exists, traverse the suffix link until eventually a match is established. Thereby successively shorten the substring h_{j-1} by its leading character.

$cl \leftarrow el$; $el \leftarrow cl.getChild(x_j)$	`{try to find a successor}`
while (**not** el **and** $el \neq cl$)	`{unless such a successor exists}`
$\quad cl \leftarrow cl.sl$	`{omit character` $(i_j \leftarrow i_j + 1)$`}`
$\quad el \leftarrow cl.getChild(x_j)$	`{try to find successor}`
if $(el = cl)$	`{special case,` el `is root}`
$\quad at \leftarrow -1$; $h \leftarrow \epsilon$	`{initialize suffix state}`
else $h \leftarrow el.m(f - cl.len, f)$	`{maximal suffix found}`

Table 3. *SearchChild*: Search a child node with the character x_j on the incident edge. Note that rescanning is not necessary, since in this case at equals zero.

3. Check if the character at position $at + f$ of the string represented at the extended locus coincides with x_j. In case of a failure omit a character in h_{j-1} by traversing the suffix link of the contracted locus and *rescan* the edge label. Fortunately, based on the index structure of a suffix tree rescanning has to succeed.
4. If the word w to be rescanned is exhausted continue with Step 2, otherwise iterate on Step 3. If the matching phase finally succeeds, h_j can be fixed as $el.m(f + at - cl.m, f + at)$, corresponding to the path from the root to the position at which x_j has been found. This position gives the new suffix state.

The finer granularity of Step 2 leads to the following sequence (cf. Table 3).

2a. Try to find an extended locus el, beginning with label x_j.
2b. If a successor is found, determine h_j to be $el.m(f - cl.len, f)$, i.e. the path of the root to the first character of the incoming edge of el. Otherwise traverse the suffix link and continue with Step 2a.

Theorem 4. *(Correctness of δ) Let the string x be scanned up to x_{j-1}. The returned value h_j of $\delta(x_j)$ is the longest suffix of $x(1, j)$, which coincides with a substring of one dictionary string m in the multi suffix tree.*

Proof. The proof is based on induction, i.e. we assume that h_{j-1} is the longest suffix of $x(1, j-1)$, which is also a substring of one dictionary string m. Let i_j be defined as the index of the start of h_j in x, than h_{j-1} is the longest substring in the dictionary with $h_{j-1} = x(i_{j-1}, j-1)$.

The algorithm δ utilized a sequence of suffix link travesals and rescanning operations to eventually fix h_j as $el.m(f + at - cl.len, f + at)$. Therefore, h_j is substring of one dictionary string.

We now assume that the substring $x(i_j, j)$ of $x(1, j)$ is not maximal. The condition for δ to terminate is $x_j = el.m[f + at]$. By traversing the suffix links and, therefore, by dropping leading characters, all substrings of $x(i_{j-1}, j)$ to $x(i_j, j)$ were checked to be a prefix of a dictionary string.

Therefore, the start index k of the substring is smaller as i_{j-1}. This contradicts to the maximality of $x(i_{j-1}, j-1)$, since $x(k, j-1)$ is a suffix of $x(1, j-1)$, being longer as $x(i_{j-1}, j-1)$ *and* representing a string in the dictionary.

Theorem 5. *(Running time of δ) Let the string x be scanned up to x_{j-1}. The amortized worst case time for $\delta(x_j)$ is constant.*

Proof. In the amortized analysis we show that the total processing time for query x is bounded by $O(|x|)$. The running time of δ proportional to the sum of the calls of Step 2a and Step 3 (corresponding to the loops in δ and *SearchChild*) and the total rescanning time.

Accumulating the calls of the Steps 2a and 3. By traversing the suffix links in each iteration one leading character of h_{j-1} is dropped. Let i_j denote the start of the substring intervall h_j in x. Then the calls of the Steps 2a and 3 are bounded by $i_j - i_{j-1}$ with i_0 initially being set to 1. Summing this value for $j \in \{1, \ldots, |x|\}$ most terms cancel leading to a remaining value of $i_{|x|} - i_0$. Since $i_{|x|}$ is an index of x it is trivially bounded by the length of x.

Determining the total rescanning time. The total number of operations in *Rescan* is bounded by the number of traversed nodes. Therefore, in each step the length of the substrings shrink with the lengths of the substrings labeling the corresponding edges. These length are maintained in at. Let at_j be the contents of at after determination of h_j. Since at is only incremented at the beginning of the procedure, the rescanning time is bounded by $1 + at_{j-1} - at_j$ with at_0 being fixed to -1. The accumulated sum of $1 + at_{j-1} - at_j$ for $j \in \{1, \ldots, |x|\}$ leads to $at_0 - at_{|x|} + |x|$. Since at is bounded by $|x|$ the total rescanning time is $O(|x|)$ as proposed.

Chang and Lawler describe a different *on-line* search strategy entitled *matching statistics*, which searches the longest prefixes h_i of position i in x [6]. They observed that the end marker $k = i + |h_i|$ increases monotonically. Before the next character x_{k+1} is considered the longest substring up to x_k is found.

Nevertheless, the authors use an additional pointer j in between i and k, which leads to retrievals of x_j with $j < k$. As in the application of dynamic duplicate matching for our proposes this access is not available.

Finally, we draw a conection between multi suffix trees and the failure function. The key observation is that inverse failure tree as proposed by Meyer is similar to the multi suffix tree, having just a few nodes less. (The proof of the following theorem can be found in the appendix.)

Theorem 6. *(Failure function in multi suffix tree) Let M be a set of strings and T its trie representation. Furthermore, let S be the multi suffix tree according to the set $(\$M)^R = \{(\$m)^R \mid m \in M\}$ and $l(v(t')^R)$ the location of a node t' in T. If $l(v(t')^R)$ is the deepest ancestor of $l(v(t)^R\$)$ then $f(t) = t'$.*

Theorem 6 allows to use the suffix tree as an efficient tool to calculate the failure function. Although there are exceptions like $M = \{(011\ldots1)\}$ we empirically observed that in most cases the calculation of the failure function via the multi suffix tree takes only constant time. Therefore, incremental substring matching with the multi suffix tree based failure function might be an alternative to the approach given above, since in the search only the information of a single node has to be maintained.

6 Experiments

We have implemented the multi suffix tree acceptor in C++ and chose a spell-checker dictionary with 25143 entries as a benchmark set of strings. Starting from the end of the file we inserted all words that had neither a substring nor a superstring in the dictionary. Substrings were found according to the strongly incremental search algorithm described in Section 5.

Exacly 14344 of the 25143 entries remained. The multi suffix tree for this mutually substring free set of strings had 69207 nodes in total and 48854 leaf nodes leading to 29 percent inner nodes. We deleted 8000 randomly chosen strings according to the algorithm given in Section 4, and observed that this ratio remained almost constant as Fig. 3 depicts.

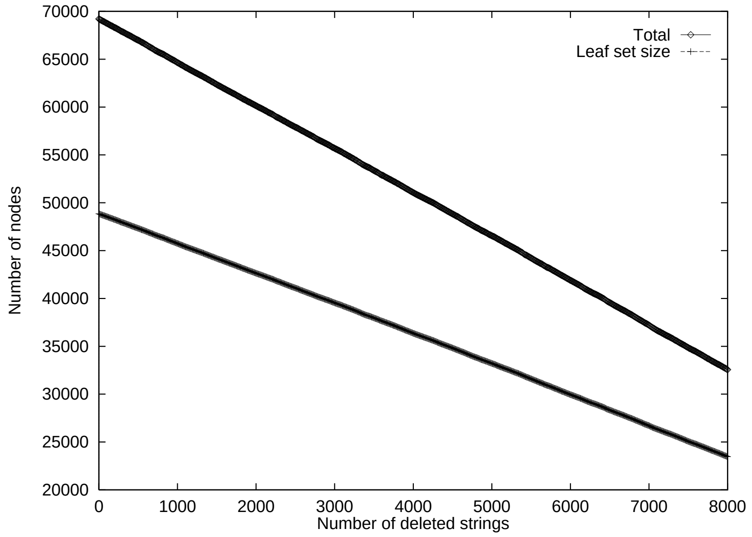

Fig. 3. The leaf set size and the total number of nodes in the multi suffix tree acceptor according to the deletion of strings.

7 Conclusions

The dictionary multi suffix tree has been extended with some neat new aspects. Strongly incremental search is achieved in time $O(|x|)$ and time bounds of $O(|m|)$ were found for inserting and deleting a string m in the dictionary.

The multi suffix tree structure was augmented by a trie with nodes linking to the leafs of the suffix tree and by *twin* nodes. This has led to optimal space complexity for the dictionary. An open question is whether the constraint of substring freedom can be dropped by maintaining the optimal time bounds.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
2. Amir, Farach, and Matias. Efficient randomized dictionary matching algorithms. In *CPM: 3rd Symposium on Combinatorial Pattern Matching*, 1992.
3. A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic dictionary matching. *J. Comput. Syst. Sci.*, 49(2):208–222, 1994.
4. A. Amir, M. Farach, R. M. Idury, J. A. L. Poutré, and A. Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.
5. J. I. Aoe. *Computer Algorithms: String Pattern Matching Strategies.* IEEE Computer Society Press, 1994.
6. W. I. Chang and E. L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.
7. B. Commentz-Walter. A string matching algorithm that is fast on the average. *Int. Colloq. Automata, Lang. and Progr.*, pages 118–132, 1979.
8. S. Edelkamp. Suffix tree automata in state space search. In *KI-97*, volume 1303 of *LNCS*, pages 381–385, 1997.
9. J.-J. Fan and K.-Y. Su. An efficient algorithm for matching multiple patterns. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):339–351, 1993.
10. Farach. Optimal suffix tree constructioin with large alphabets. *IEEE Symp. Found. Computer Science*, pages 137–143, 1997.
11. M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. *Int. Colloq. Automata Lang. and Progr.*, pages 550–561, 1996.
12. Ferragina and Luccio. On the parallel dynamic dictionary matching problem: New results with application. In *Annual European Symposium on Algorithms*, 1996.
13. Hariharan. Optimal parallel suffix tree construction. *IEEE Symp. Found. Computer Science*, 1994.
14. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Trans. on SSC*, 4:100, 1968.
15. R. M. Idury and A. A. Schaeffer. Dynamic dictionary matching with failure functions. *Theoretical Computer Science*, 131(2):295–310, 1994.
16. D. K. Kim and K. Park. Linear-time construction of two-dimensional suffix trees. In *ICALP'99*, volume 1644 of *LNCS*, pages 463–472, 1999.
17. R. E. Korf and L. A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *AAAI*, pages 1202–1207, 1996.
18. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
19. B. Meyer. Incremental string matching. *Inf. Process. Lett.*, 21:219–227, 1985.
20. D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
21. S. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. *J. Assoc. Copmput. Mach.*, pages 300–309, 1994.
22. L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *AAAI*, pages 756–761, 1993.
23. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
24. P. Weiner. Linear pattern matching algorithms. In *Conference Record, IEEE* 14^{th} *Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Appendix

Theorem 6

We have to proof the followin result on the failure function in a multi suffix tree.

Let M be a set of strings and T its trie representation. Furthermore, let S be the multi suffix tree according to the set $(\$M)^R = \{(\$m)^R \mid m \in M\}$ and $l(v(t')^R)$ the location of a node t' in T. If $l(v(t')^R)$ is the deepest ancestor of $l(v(t)^R\$)$ then $f(t) = t'$.

Proof. Since $l(v(t')^R)$ is a predecessor of $l(v(t)^R\$)$, we have that $v(t')^R$ is prefix of $v(t)^R$. Therefore, $v(t')$ is suffix of $v(t)$.

We recall the definition of the failure function $f(t)$ is defined as the node t' in T, with its string representation $v(t')$ being the maximal prefix of one string in T *and* a suffix of $v(t)$.

Therefore, it remains to show, that $v(t')$ is maximal. Suppose not. Then there is an extension $v(t'')$ of $v(t')$, being a prefix of a string in T and a suffix of $v(t)$ Hence, $v(t'')^R$ is prefix of $v(t)^R$. The bijektion of the set of trie nodes and suffix tree leaves stated in Theorem 1 leads to the existance of the location of $v(t'')^R\$$ in S. Furthermore, $v(t'')$ is an extension of $v(t')$, i.e., $|v(t'')^R| > |v(t')^R|$. Since both $v(t')^R$ and $v(t'')^R$ are prefixes of $v(t)^R$, the locus of $l(v(t'')^R)$ is deeper than the locus of $l(v(t')^R)$. This contradicts the assumption that $l(v(t')^R)$ is the deepest ancestor of $l(v(t)^R\$)$ in S. Therefore, we have $f(t) = t'$ as required.