TUM

INSTITUT FÜR INFORMATIK

Efficient approximate dictionary look-up for long words over small alphabets

Abdullah N. Arslan   Johannes Nowak

TUM-I0708
Januar 07

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-01-I0708-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Efficient approximate dictionary look-up for long words over small alphabets

Abdullah N. Arslan
University of Vermont
Department of Computer Science
Burlington, VT 05405, USA
aarslan@cs.uvm.edu

Johannes Nowak
Technische Universität München
Fakultät für Informatik
D-85748 Garching, Germany
nowakj@in.tum.de

Abstract

Given a dictionary $\mathcal{W}$ consisting of n strings of length m each, the *approximate dictionary look-up* problem asks if there is a member in $\mathcal{W}$ within a given distance d from a given query string q. We present new hybrid tree/dynamic programming algorithms for the approximate dictionary look-up problem. The approach works for both Hamming and edit distances. We use a new tree data structure, *prefix-partitioned look-up tree*, which generalizes Patricia trees. The complexities of the algorithms depend on the height h and branching factor b of the tree used. Our approach is space-efficient, and answers a query in $O(m(b-1)^d h^{d+1})$ time when Hamming distance is used (the time complexity increases by a factor of $O(db^d h^d)$ for the edit-distance case) where h and b denote the tree height and branching factor, respectively.

Keywords: d-query, approximate dictionary look-up, Patricia tree, Hamming distance, edit distance, hybrid tree/dynamic programming technique, space efficient algorithm, space-time tradeoff, preprocessing.

1 Introduction

1.1 Approximate Dictionary Look-up

Let $\mathcal{W}$ be a dictionary consisting of n binary strings of length m each. A *d-query* asks if there exists a string in $\mathcal{W}$ within Hamming distance d of a given binary query string q. Hamming distance between two strings is the number of positions at which the strings differ. The problem was originally posed by Minsky and Papert in 1969 [24] in which they asked if there is a data structure that supports fast d-queries. Algorithms for answering d-queries and its variations have been a topic of interest in the literature [1, 3, 4, 9, 10, 23, 34]. *Approximate dictionary look-up* is a problem of dictionary look-up within distance d to a given query string q. It is essentially a d-query problem over a larger but finite alphabet, and it allows for various notions of proximity.

A variation of the approximate dictionary look-up problem is the *approximate query* problem: given a query string q in some alphabet Σ, list all words in the dictionary that are close to q [9, 10]. Dolev et al. [10] analyze time and space complexity tradeoff of a class of algorithms answering approximate query using various notions of proximity. They assume that $\mathcal{W}$ is stored in buckets, and preprocessing of $\mathcal{W}$ is allowed. Dolev et al. [9] study hashing functions to buckets so that no bucket is too large, and uses the results to derive bounds for the approximate query problem.

For the approximate dictionary look-up problem, the cases of small d and large d seem to require different techniques for their solutions. In this paper, we consider dictionary look-up within distance d for small d, and alphabets whose sizes can be larger than two. This variant of the problem has applications in password security [23]. Other applications include spell-checking, speech-recognition, study of bird-singing, and searching biological sequence databases for an approximate match for a given query pattern (or motif) (see [28] for many possible applications).

A naive method for answering the approximate dictionary look-up problem is to generate all possible strings differing from q in at most d positions, and perform $O((|\Sigma| - 1)^d m^{d+1})$ exact queries using $O(m)$ additional space. If we use $O((|\Sigma| - 1)^d m^d n)$ additional space to store all possible words within difference d of words in $\mathcal{W}$ we can answer an approximate dictionary look-up in $O(m)$ time by performing one exact query. Therefore, there is a tradeoff between time and space required to answer a d query. We are interested in finding a solution that does not require unreasonable space or time. The algorithm of Arslan and Eğecioğlu [1] answers an approximate dictionary look-up in time $O((|\Sigma| - 1)^d m^{d+1})$ using additional space $O(m)$.

Recently several results for text indexing with errors have been published [5, 22]. These results improve the complexity of answering approximate dictionary look-up. Results shown by Maaß [21] imply that when Hamming distance is used, and the dictionary is stored in a trie, the average time of trie-search to answer an approximate dictionary look-up is $O(\log^{d+1}(nm))$. The method presented by Cole et al. [5] can be used to answer an approximate dictionary look-up (where d can be the edit distance) in time $O(m + \log^d(nm) \log\log(nm))$, and it requires additional space $O(nm \log^d(nm))$ for indexing. Maaß and Nowak [22] have shown two results for text indexing with errors. Their results imply that when edit distance is used approximate dictionary look-up can be answered: 1) in $O(m)$ time if we use on average $O(n \log^d n)$ additional space for indexing; 2) in $O(m)$ average time if we use $O(n \log^d n)$ additional space for indexing. Although these methods are time-efficient, they are not practical for answering approximate dictionary look-up in very large databases because of their space requirements.

1.2 Our Results

Patricia trees are widely used to store, process, and query dictionaries. We devise a hybrid tree/dynamic programming approach for fast approximate dictionary queries in Patricia trees for both Hamming and edit distance models. Let h be the height and b be the branching factor of the Patricia tree storing for the considered dictionary. The time complexity of the queries is bounded by $O(m(b-1)^d h^{d+1})$ in the Hamming model. When edit distance is considered, the complexity is increased by a factor of $O(db^d h^d)$.

These algorithms are efficient compared to existing solutions when the problem involves long words defined over a small alphabet. Main property of our method is that whenever we have a tree with small height and branching factor the worst-case performance of our algorithm is within the expressions we state unlike other space-efficient methods which guarantee average-case performances (for example [21, 22]). It has been shown that *almost surely*, the height of a Patricia tree constructed from n independent random strings satisfying certain probabilistic constraints is $\leq c \log n$ where c is a small constant. This implies that our method answers an approximate dictionary look-up in $O(m(|\Sigma| - 1)^d \log^{d+1} n)$ (Hamming model) in this case. We generalize the notion of Patricia trees to *prefix-partitioned look-up trees*. We show that the hybrid look-up technique can be applied to dictionaries stored by prefix-partitioned look-up trees. Moreover, we show that prefix-partitioned look-up trees exhibit nice scaling properties. In particular, it is possible to trade height in for an increased branching factor. This property makes prefix-partitioned look-up trees especially useful for secondary memory data structure design, i.e., it is possible to optimize the branching factor according to the given disk block size. The overall complexity of the look-up algorithms is proven to be independent of this choice. Moreover, in the case that the height of the Patricia tree associated with $\mathcal{W}$ is large, the running time of the look-up algorithms is slightly improved by the appropriate trade-off.

The outline of this paper is as follows: in Section 2 we introduce notational concepts and give preliminary definitions including the generalization of Patricia trees. In Section 3 we describe and analyze the look-up algorithms. Finally, in Section 4, we show that prefix-partitioned look-up trees are not sufficient for provably efficient dictionary look-ups. On the other hand, we examine the trade-off between height and branching factor in prefix-partitioned look-up trees.

2 Preliminaries

2.1 Strings and String Distances

Throughout this work, let Σ be any finite alphabet and let $|\Sigma|$ denote its size. We consider $|\Sigma|$ to be constant. Let $w \in \Sigma^m$ be a string of length m over Σ. A dictionary $\mathcal{W}$ is a collection of n strings of over Σ. For ease of exposition, we assume that all strings in $\mathcal{W}$ are of equal length m. Note that all of our results apply also to dictionaries without this restriction.

There are various measures of *proximity* between strings. In this work, we focus on the well known *Hamming distance* [14] and the *edit distance* (Levenshtein distance [20]). The edit distance of two strings u and v, *eduv*, is defined as the minimum number of *edit operations* (*deletions*, *insertions*, and *substitutions*) that transform u into v. For the Hamming distance, only substitutions are allowed. We restrict our attention to the unweighted model, i.e., every operation is assigned a cost of one.

2.2 Tries and Patricia Trees

For fast look-up, strings can be stored in a *trie*. A trie $\mathcal{T}$ for a set of strings $S \subset \Sigma^*$ is a rooted tree with edges labelled by characters from Σ. All outgoing edges of a node are labelled by different characters (*unique branching criterion*). Each path from the root to a leaf can be read as a string from S.

A Patricia tree is a compact representation of a trie where any node which is an only child is merged with its parent. Figure 1 shows a Patricia tree storing $\mathcal{W} = \{001, 010, 011, 101\}$. Patricia trees were introduced by Morrison [25], and studied in depth first by Knuth [18]. Two parameters of Patricia trees (and prefix-partitioned look-up trees, see below) are of particular importance in this paper. The *height* h of a tree T is the length if a longest path from the root to a leaf in T. The *branching factor* b is the largest number of children a single node in T has. Observe that in the case of Patricia trees we have $b = O(|\Sigma|)$. The following result is well known and due to Pittel [27].

Lemma 1 ([27]). *Let $\mathcal{W}$ be a collection of n infinite strings created independently by a strongly mixing, stationary and ergodic random source [31]. Then the height of the Patricia tree built over $\mathcal{W}$ is bounded by $O(\log n)$, almost surely.*

Observe that important random sources as a Bernoulli source (memoryless source) and a Markovian source fall under this model. For more details on the probabilistic behavior we refer to [31, 8] and the references therein.

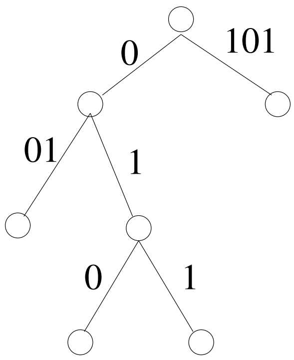

Figure 1: A Patricia tree for $\mathcal{W} = \{001, 010, 011, 101\}$.

2.3 Prefix-Partitioned Look-Up Tree

A prefix-partitioned look-up tree (PPT) S is —just as a Patricia tree— an edge-labelled, rooted tree. We denote by $p_{v,w}$ the label of the arc from a given parent node v to its child (or successor) w in S. For any given node v in S, p_v denotes the concatenation of arcs from the root to node v. In prefix-partitioned look-up trees, we relax the unique branching criterion. Consider the labels of the arcs from a given node v to its children. In contrast to Patricia trees, we do not require that these labels start with different characters. Instead, we only require that none of these label is a prefix of another.

A prefix-partitioned look-up tree S has the following properties:

1) S has n leaves and there is a one-one correspondence between the leaves and the members in $\mathcal{W}$. That is, the leaf nodes can be numbered from 1 to n such that for the ith leaf l_i we have $p_{l_i} = w_i$ where $\mathcal{W} = \{w_1, w_2, \ldots, w_n\}$.

2) All strings that are stored in a subtree rooted at node v in S have a common prefix p_v.

3) Let v be an arbitrary node with children $\{v_1, \ldots v_k\}$. Then for $i \neq j$ it holds that p_{v,v_i} is not a prefix of p_{v,v_j}.

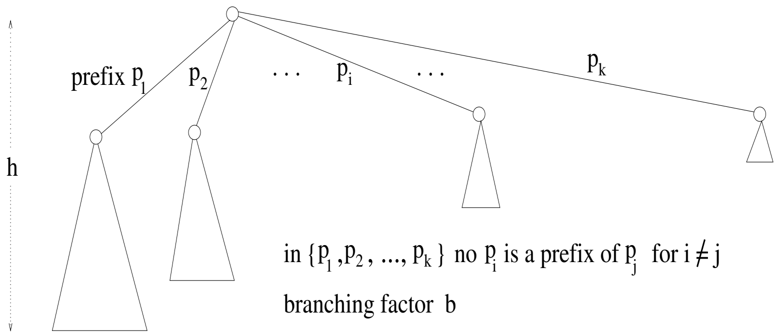

Figure 2: A prefix-partitioned look-up tree $\mathcal{S}$.

As noted above, a Patricia tree is a prefix partitioned look-up tree. On the other extreme, the tree of height 1 with n leaves in which each leaf corresponds to exactly one member in $\mathcal{W}$ is also a prefix-partitioned look-up tree. Ideally, we want prefix-partitioned look-up trees with small height, and small branching factor.

3 Look-up Algorithms

We use a hybrid tree/dynamic programming technique to develop an algorithm for approximate dictionary look-up. Our algorithms run on any prefix-partitioned look-up tree T that stores $\mathcal{W}$. Algorithms using this technique on tries for the same problem were presented by Arslan and Egecioglu [1]. Our algorithms specialize for height-compressed trees that store the dictionary $\mathcal{W}$. This replaces in the time complexity, the factor m^{d+1} that appears in the results of the algorithms of Arslan and Egecioglu [1] by mh^{d+1} in our results. Therefore our results are better when $h^{d+1} = o(m^d)$, i.e. when the words in the dictionary, and the query are long compared to the height of the tree that stores the dictionary $\mathcal{W}$. Our algorithms are superior to existing solutions that are space-efficient (such as those proposed by Arslan and Egecioglu [1]) when m is large, and in the tree that stores the dictionary $\mathcal{W}$, the branching factor is small (the branching factor is bounded from above by $|\Sigma|$ when we use a Patricia tree).

We call a function F an *ordinary node-weight* function for a given tree T if F assigns non-negative integral weights to nodes such that for any node p at most one of its children has the same weight as p, and the weights of the other children are strictly larger than that of p.

The following is a generalization of a lemma in [1].

Lemma 2. *Let T be a tree whose height is h, and whose branching factor is b. Let the nodes of T be assigned weights by an ordinary node-weight function. Then the number of nodes N in T with weight $\leq d$ is $O(\ (b-1)^d\ h^{d+1})$.*

Proof. To find an upper bound for N we consider the complete tree T with branching factor b, and height h. Since the weights are assigned by an ordinary node-weight function, N is maximized when each parent node whose weight is w has exactly one child with weight w, and each of its other children has weight $w+1$. Let $L(l,w)$ denote the number of nodes with weight w at level l in T. Then $L(l,w)$ satisfies the recursion $L(l+1,w) = L(l,w) + (b-1)L(l,w-1)$ with $l \geq w$ and $L(l,0) = 1$. The solution of this recursion is $L(l,w) = \binom{l}{w}(b-1)^w$. Therefore the total number of nodes with weight w in T is $(b-1)^w \sum_{l=w}^{h} \binom{l}{w} = (b-1)^w \binom{h+1}{w+1}$, and therefore $N = \sum_{w=0}^{d}(b-1)^w \binom{h+1}{w+1} = O((b-1)^d h^{d+1})$.

Let $s[i..j]$ represent the substring $s_i s_{i+1} \dots s_j$ of any given string $s = s_1 s_2 \dots s_k$ with length k. With respect to a given query string q let function f assign a weight to each node v in the tree rooted at node r,

$$f(v) = H(p_{r,v}, q[1..|p_{r,v}|]) \tag{1}$$

where H denotes the Hamming distance. We note that f is an ordinary node-weight function for $\mathcal{S}$. Consider any parent node u, and its child w. The weight $f(w) = H(p_{r,w}, q[1..|p_{r,w}|]) = f(u) + H(p_{u,w}, q[(|p_{r,u}|+1)..(|p_{r,u}|+|p_{u,w}|)])$. Clearly $f(w) \geq f(u)$. If $f(w) = f(u)$ then for any other child of u the weight is larger than $f(u)$ because over all the arc labels from u only $p_{u,w}$ exactly matches $q[(|p_{r,u}|+1)..(|p_{r,u}|+|p_{u,w}|)]$, and among these arc labels no label is a prefix of another. We reach the following corollary from Lemma 2:

Corollary 3. *Let N be the number of nodes in $\mathcal{S}$ with weight $\leq d$ as defined in (1). Then $N = O((b-1)^d h^{d+1})$.*

We present an algorithm shown in Figure 3 for the approximate dictionary look-up within Hamming distance d. Algorithm *DFT-LOOK-UP*$_H(r,q,d)$ checks if the tree rooted at r has a leaf whose Hamming distance from q is $\leq d$. The algorithm searches for a member in a depth-first manner. If $d < 0$ then it returns false since there is no such member. If r is a leaf and q is an empty string then the algorithm returns true since a member is found. Otherwise for each child the algorithm calculates a weight $d' = H(p_{r,c}, q[1..|p_{r,c}|])$, and recursively checks if the subtree rooted at each child contains a member within an updated distance $d - d'$. If any of these searches returns true then the algorithm returns true, otherwise, it returns false.

Since f in (1) is an ordinary node-weight function for $\mathcal{S}$, by Corollary 3 the algorithm in $\mathcal{S}$ visits $O((b-1)^d h^{d+1})$ nodes, and at each leaf spends time $O(m)$. The time complexity of the algorithm is, therefore, $O(m(b-1)^d h^{d+1})$. We can modify the algorithm such that the words in $\mathcal{W}$ as well as the query string can be of different lengths.

Theorem 4. *Algorithm DFT-LOOK-UP$_H$ solves the approximate dictionary look-up problem with respect to the Hamming distance on a dictionary $\mathcal{W}$ in time $O(m(b-1)^d h^{d+1})$, where h is the height and b is the branching factor of any prefix-partitioned look-up tree representing $\mathcal{W}$.*

```
Algorithm DFT-LOOK-UP_H(r, q, d)
   If d < 0 return FALSE
   If r is a leaf, and q is an empty string then return TRUE
   for each child c of r in S do {
      d' := H(p_{r,c}, q[1..|p_{r,c}|])
      if DFT-LOOK-UP_H(c, q[(|p_{r,c}| + 1)..|q|], d - d') then return TRUE
   }
   return FALSE
```

Figure 3: Algorithm *DFT-LOOK-UP*$_H$ for approximate dictionary look-up within Hamming distance d.

Lemma 1 yields:

Corollary 5. *Let $\mathcal{W}$ be a dictionary such that its members are created by a strongly mixing, stationary and ergodic random source. Then Algorithm DFT-LOOK-UP$_H$ solves the approximate dictionary look-up problem respect to the Hamming distance on a dictionary $\mathcal{W}$ in time $O(m(|\Sigma|-1)^d \log^{d+1} n)$ almost surely.*

We present another similar algorithm for the problem when simple edit distances are used. Given two strings $X = x_1 \dots x_m$ and $Y = y_1 \dots y_m$, the simple edit distance $ed(X, Y)$ is the minimum number of edit operations which transform X into Y using three types of operations: insert, delete, and substitute. A common framework for computing an edit distance is the *edit graph* (see [13]) for definition), and it has a simple dynamic programming formulation [13]:

$$D_{i,j} = \min\{\ D_{i-1,j} + 1,\ D_{i-1,j-1} + H(x_i, y_j),\ D_{i,j-1} + 1\} \tag{2}$$

for all i, j, $0 \le i, j \le m$ with boundary conditions $D_{i,0} = i$, $D_{0,j} = j$.

With respect to a given query string q, let e be a function that assigns a weight to a given node v in $\mathcal{S}$ rooted at r as described in the following:

$$e(v) = \min\{ed(p_{r,v}, t) \mid t \text{ is a prefix of } q\} \tag{3}$$

where ed denotes the simple edit distance. Note that in this definition $p_{r,v}$ and the prefix t of q can be of different lengths. We can see that e is not an ordinary node-weight function because it is possible that more than one children of a node can have the same weight as the parent node due to insert, and delete operations that can be performed on the labels to ancestor nodes.

Lemma 6. *Let N be the number of nodes in $\mathcal{S}$ with weight $\le d$ as defined in (3). Then $N = O((b-1)^d b^d h^{2d+1})$.*

Proof. We imagine that we traverse the tree in breath-first manner starting at root at level 0, and consider the minimum possible weight for each node. Clearly for any node u, the weight $e(u)$ in (3) is less than or equal to $f(u)$ in (1). Suppose that initially for every node u in $\mathcal{S}$, $e(u) = f(u)$. If v is a parent node of u then $e(u) \ge e(v)$, i.e. e is non-decreasing. Due to possible delete, and insert operations on the label of the arc $p_{v,u}$ from v to u there may be nodes w in the subtree rooted at u such that w has more than one children sharing the same weight as w. When we studied an upper bound in Lemma 2 we considered that every node v has exactly one child with the same weight as v. This time, being overly pessimistic, we assume that all of v's children have the same weight as v if it is given that there are insertions, or deletions on $p_{v,u}$. We consider possibility of insertions, and deletions on all labels on arcs each connecting a node at level $i-1$ to a node at level i for a given i. This increases the number of nodes at level i with the same weight as their parents (and all $\leq d$) by a factor of $\leq b$, where b is the branching factor in $\mathcal{S}$. Since there are at most d insertions, or deletions, for each permutation of the levels they can occur, the number of nodes with the same weight as their parents (and all $\leq d$) is increased by a factor of $\leq b^d$. The number of levels where an insertion, or a deletion can occur is the same as the height of the tree, h. Since there can be at most d such operations, we need to consider $\binom{h}{d}$ possibilities. Putting all together, the product of $\binom{h}{d}b^d$ and the upper bound in Lemma 2 gives the upper bound in this lemma.

We present Algorithm *DFT-LOOK-UP*$_{ed}$ for the approximate dictionary look-up problem when edit distance is used. The steps of the algorithm are shown in Figure 4. The algorithm is based on depth-first traversal (DFT) of $\mathcal{S}$ during which the entries of the dynamic programming matrix are partially computed. To determine if two strings are within edit distance d it is sufficient to consider a diagonal band of the edit graph [33]. Algorithm *DFT-LOOK-UP*$_{ed}$ uses this observation (see Figure 6).

For a given node v in $\mathcal{S}$ rooted at r, we define $D_{v,j}$ where $\max\{0, i - \lfloor d/2 \rfloor\} \leq j \leq \min\{m, i + \lfloor d/2 \rfloor\}$, and $i = |p_{r,v}|$ (see Figure 6) as follows:

$$D_{v,j} = ed(p_{r,v}, q[1..j])$$

That is, $D_{v,j}$ is the minimum simple edit distance between $p_{r,v}$ and $q[1..j]$, and the weight of node v defined in (3) is

$$e(v) = \min\{\ D_{v,j} \mid \max\{0, i - \lfloor d/2 \rfloor\} \leq j \leq \min\{m, i + \lfloor d/2 \rfloor\} \text{ where } i = |p_{r,v}|\ \}.$$

If we process $\mathcal{S}$ in depth-first manner, we can compute $D_{v,j}$ for all nodes using a single matrix $D_{i,j}$ where $0 \leq i \leq m$.

Algorithm *DFT-LOOK-UP*$_{ed}$ starts with the initialization of scores for the first row, and it invokes Function *DFT-COMPUTE-D*$_{ed}$ for each child v of the root r. If any of these invocations returns a value $\leq d$ then the algorithm returns YES; otherwise it returns NO.

Given a parent node v, a child node u, Function *DFT-COMPUTE-D*$_{ed}(v, u)$ computes the shaded region of the edit graph shown in Figure 6 using $p_{v,u}$, and starting with the values in the row of parent node v. The minimum of the values in the row of u is set as the weight $e(u)$ of u . If this value is equal to d then the function examines every position j in the row of u where d is achieved. These are the only starting positions for a suffix of the query string q with which weight d is preserved in a subtree rooted at u . That is, these are the only positions which potentially lead to a leaf with weight d . Therefore the algorithm checks if starting from u at each such position j if there is a path to a leaf on

```
Algorithm DFT-LOOK-UP_ed(d)
   D_{0,j} := j for all j,   1 ≤ j ≤ ⌊d/2⌋
   D_{i,0} := 0 for all i,   1 ≤ i ≤ m
   D_{0,0} := 0
   for each child v of the root r in S do
     if DFT-COMPUTE-D_ed(r, v) ≤ d then return YES
   return NO
```

Figure 4: Algorithm *DFT-LOOK-UP*$_{ed}$ for dictionary look-up within edit distance d.

$q[(j+1)..m]$. If the answer is yes then the algorithm returns d, otherwise it returns $d+1$ which is a number sufficiently large to yield a no answer when we only consider the subtree rooted at u . If the weight of u is smaller than d then the function traverses recursively the subtree rooted at u in depth-first manner, computes and returns the minimum edit distance (leaf-weight) achievable in this subtree. Note that if the final value returned is $\leq d$ then it must be the weight of a leaf.

```
Function DFT-COMPUTE-D_ed(v, u)
   i_start := |p_{r,v}|; i_end := |p_{r,v}| + |p_{v,u}|
   for i := i_start to i_end do
     for j := max{0, i − ⌊d/2⌋} to min{m, i + ⌊d/2⌋} do
       D_{i,j} := min{D_{i−1,j} + 1,  D_{i−1,j−1} + H(p_{v,u}[i − i_start], q_j),  D_{i,j−1} + 1}
   weight := min{D_{i_end,j} | max{0, i − ⌊d/2⌋} ≤ j ≤ min{m, i + ⌊d/2⌋}}
   if u is a leaf or weight > d then return weight
   if weight= d then {
    for j := max{0, i_end − ⌊d/2⌋} to min{m, i_end + ⌊d/2⌋} do {
     if D_{i_end,j} = d and there is a path from u to a leaf in S
         on q[(j + 1)..q_m] then return d }
    return d + 1
   }
   return min{DFT-COMPUTE-D_ed(u, w) | w is a child of u}
```

Figure 5: Function *DFT-COMPUTE-D*$_{ed}$ for computing the minimum edit distance achieved in subtree rooted at u whose parent is v.

Since processing at each node takes time $O(dm)$, the algorithm's time complexity is $O(dm(b-1)^d b^d h^{2d+1})$ by Lemma 6, and it requires additional space $O(dm)$.

With high probability, a Patricia tree storing $\mathcal{W}$ has height $\leq c \log n$ for some small constant c [6, 7, 30]. Whenever such Patricia tree exists, our algorithms take $O(m(|\Sigma|-1)^d c^{d+1} \log_2^{d+1} n)$ time when Hamming distance is used, and $O(dm(|\Sigma|-1)^d |\Sigma|^d \log_2^{2d+1} n)$ when edit distance is used.

Theorem 7. *Algorithm DFT-LOOK-UP*$_{ed}$ *solves the approximate dictionary look-up problem with respect to the edit distance on a dictionary* $\mathcal{W}$ *in time* $O(dm(b-1)^d h^{2d+1})$*, where* h *is the height and* b *is the branching factor of any prefix-partitioned look-up tree representing* $\mathcal{W}$.

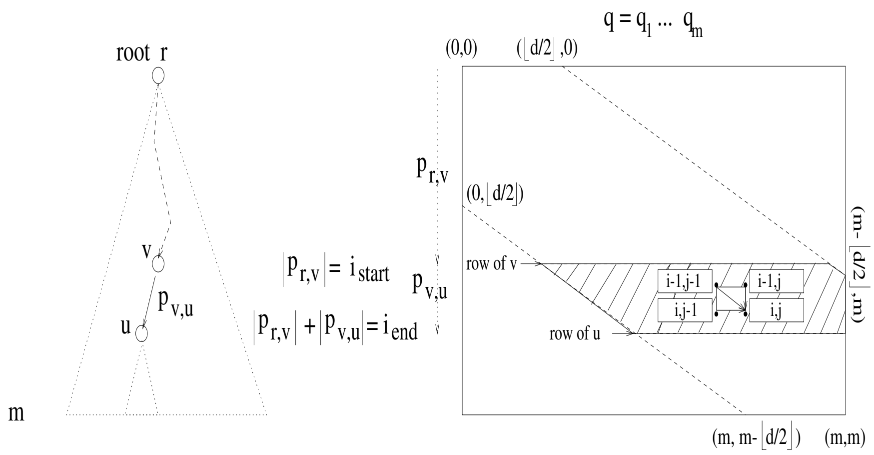

Figure 6: The hybrid tree/dynamic programming approach used by Function *DFT-COMPUTE-D*$_{ed}$.

Again, by Lemma 1

Corollary 8. *Let $\mathcal{W}$ be a dictionary such that its members are created by a strongly mixing, stationary and ergodic random source. Then Algorithm DFT-LOOK-UP$_{ed}$ solves the approximate dictionary look-up problem respect to the edit distance on a dictionary $\mathcal{W}$ in time $O(m|\Sigma|^{2d}\log^{2d+1} n)$ almost surely.*

When a Patricia tree storing $\mathcal{W}$ has large height, we propose a heuristic algorithm that aims to construct from this tree a prefix-partitioned look-up tree with smaller height at the expense of increasing the branching factor. We also propose an algorithm which guarantees an incremental improvement over a Patricia tree.

4 Trade-Offs

In this section, we demonstrate the versatility of the generalized look-up tree definitions.

We start by showing that in general it is not possible to bound the relevant tree parameters, i.e. height and branching factor, by functions logarithmical in n as incorrectly claimed in [2]. However, we show that in prefix-partitioned trees —as opposed to Patricia trees— height can be traded in for branching factor and vice versa. This trade-off exhibits the following useful properties:

- The trade-off can be carried out algorithmically on any prefix-partitioned tree.
- The size of the data structure is not affected by the trade-off.
- The performance of the hybrid look-up algorithms is independent from the particular choices of the parameters.
- The appropriate trade-off improves the worst-case running time of the look-up algorithms in the case of large Patricia tree heights.

The trade-off can nicely be applied to optimize the storage locality of the considered data structure in case of operating on secondary memory.

To see why the efficiency of prefix-partitioned look-up trees is limited, we consider a dictionary $\mathcal{W}$ over $\Sigma = \{0, 1\}$. Let $\mathcal{W} = \{w_1, \ldots, w_n\}$ such that $w_i = 0^i 10^{m-i-1}$ for all $1 \leq i \leq m - 1$.

The following lemma is obvious for this $\mathcal{W}$:

Lemma 9. *Let $W' \subset W$ and $|W'| \geq 2$. Let α be a common prefix of all $w' \in W'$ of length k. Then $\alpha = 0^k$.*

Let now T be a prefix-partitioned look-up tree for $\mathcal{W}$ with height h and branching factor b such that $b = O(\log n)$. We show that the height h of S cannot be bounded from above by $\Theta(\log n)$. Lemma 9 and $b = O(\log n)$ imply the following: let r be the root node, and let $T_1, \ldots, T_k$ be the subtrees rooted at the children $\{v_1, \ldots, v_k\}$ of r. We denote by $|T_w|$ the size, i.e. the number of nodes, in subtree rooted at T_w for any w in T. Then there is at most one subtree T_l such that $|T_l| \geq 2$. This follows from the fact that no edge label can be a prefix of another label. Hence, at most one edge label can be of the form 0^k and all other subtrees are leaves. Moreover, this reasoning can be continued recursively. The strings in T_l are now of the form $0^i 10^{m-k-1}$. Again, among all the subtrees rooted at children of v_l, only one has size ≥ 2. This leads to

Lemma 10. *T has height $h = \Omega(n/\log n)$*

Proof. Let $c \log n$ be the maximal branching factor of T for some real positive c. Then at every level, there are at most $c \log n$ leaf nodes. Hence there are at least $n/c \log n = \Omega(n/\log n)$ levels in T.

Let now T be the Patricia tree for an arbitrary dictionary $\mathcal{W}$. Let h be the height of T. For a given trade-off factor k we show how to construct recursively a prefix-partitioned look-up tree T' from T with the following properties: the height h' of T' is bounded by $O(h/k + \log n)$ and the branching factor b' of T' is bounded by $O(|\Sigma| \cdot k)$.

Definition 11. *A centroid path of T to be a root-leaf path $\{r = v_0, v_1, \ldots, v_l\}$ such that for all $1 \leq i \leq l$ it is true that $v_i = \underset{w \text{ is a child of } v_{i-1}}{\arg\max} \{|T_w|\}$ (ties broken arbitrarily).*

The intuition behind this construction is the following: in each recursion, a subtree T_v of T is converted. The height of T_v is reduced by centralizing subtrees branching off the first k nodes of the centroid path of T_v. Obviously, since the first k nodes of the centroid paths are deleted, the lengths of all paths in the subtree rooted at v_k (the kth node on the centroid path) are decreased by k. However, the lengths of the paths of the off-path subtrees (that are centralized at v) are only diminished by i, where $1 \leq i \leq k$. On the other hand, we know that in each off-path subtree the number of nodes is small ($\leq (1 - 1/|\Sigma|)|T_v|$) since it is not on the centroid path.

The algorithm is given in Figure 7. First, the centroid path P starting at the current root node r is determined. Then the off-path subtrees (rooted at nodes c) are centered at r, i.e., the new parent $p(c)$ of c is set to be r. The nodes v_i on the centroid path are deleted, since these are now unary. Finally, the children of r are converted recursively.

```
Algorithm CONVERT(T, k)
  Let r be the root of T
  If r is a leaf then return TRUE
  let P = {r = v_0, v_1, ..., v_l} be a centroid path in T_r
  for i = 1 to min{|P|, k} do {
    for each off-path child c of v_i in S do {
      p(c) = r
    }
    delete v_i
  }
  for each child c of r do {
    Let T_c be the subtree rooted at c.
    Convert(T_c, k)
  }
  return TRUE
```

Figure 7: Algorithm *CONVERT*(T, k, r) for reducing the height of a PAT in expense of an increased branching factor.

Lemma 12. *The running time of CONVERT*(T, k) *is bounded by* $O(|T|k)$.

Proof. Suppose that for each node $v \in T$, $|T_v|$, the size of the subtree rooted at v, is stored with v. A Patricia tree T can clearly be augmented with this information in a linear time traversal, e.g. a depth-first search. In every recursive step, we only need the first k nodes on the centroid path starting at the root. With the information about the subtree sizes stored with nodes, these k nodes can be clearly chosen in $O(k)$ time. Observe that the branching factor of the parts of the tree relevant for the centroid path is $|\Sigma|$ by the time the centroid paths are considered. The centralizing of the nodes can be carried out in constant time per node. Finally, for each node v, T_v is considered exactly once.

Theorem 13. *For every dictionary* $\mathcal{W}$ *and parameter* k *it is possible to construct a prefix-partitioned lookup-tree* T' *with height* $h' = O(h/k + \log n)$ *and branching factor* $b' = O(k)$ *(for constant alphabet size), where* h *is the height of the Patricia tree* T *for* $\mathcal{W}$.

Proof. Let $P = \{u_0, \ldots, u_h\}$ be a longest path in T'. We distinguish *on-path* and *off-path* nodes on P. We say u is an on-path node if the parent of u in T' was also a predecessor of u on a considered centroid path in T, i.e. u has not been relinked during the conversion. On the other hand, we say u is an off-path node if u was hanging off a centroid path in T, i.e. u has been relinked during the conversion. The following is true for P.

- There are at most $O(\log n)$ off-path nodes on this path. In fact, let u be an off-path node and let v be its parent. Let u' be the child of v such that u' is on the centroid path of T_v. By the pigeonhole principle, we conclude that $|T_{u'}| \geq (1/|\Sigma|)|T_v|$. It readily follows that $|T_u| \leq (1 - 1/|\Sigma|)|T_v|$. Moreover, this implies that there can be only $\log_{|\Sigma|-1/|\Sigma|} n$ off-path nodes on P.

- There are at most $O(h/k)$ on-path nodes on P. Again, this follows from the fact that each on-path node decreases the length of the concerning path by k.

We still have to show that T' is a prefix partitioned look-up tree. T' is a prefix-partitioned look-up tree. This follows directly from the construction since the subtrees which are centralized at the root all branch off the same path.
The maximal branching factor b' of T' is obviously bounded by $(|\Sigma| - 1) \cdot k$ which again follows directly from the construction.

Consider now a dictionary $\mathcal{W}$ where the associated Patricia tree T has size $\omega(\log n)$. Algorithm *DFT-LOOK-UP*$_H$ takes $O(m|\Sigma - 1|^d h^{d+1}) = O(m(|\Sigma - 1| \cdot h)^d h)$time. Let now $k = h/\log n$ be the trade-off factor and T' the resulting prefix-partitioned look-up tree. The running time of *DFT-LOOK-UP*$_H$ on T' can be bounded by $O((|\Sigma| \cdot h)^d \log n)$. Hence, the running time of *DFT-LOOK-UP*$_H$ on T' is $o(m(|\Sigma - 1| \cdot h)^d h)$ since $\log = o(h)$. Similarly, the running time improves in the case of edit distance by a factor of $h/\log n$.

5 Conclusion

We introduce a new approach for the approximate dictionary look-up problem. Our approach improves theoretically upon previous solutions using linear size indices. Moreover, the hybrid algorithms are practically appealing. The algorithms are easy to implement and can be used with Patricia trees. The trade-off properties of prefix-partitioned look-up trees can be used to tune the data structure for use in secondary memory. In fact, the algorithms are currently being implemented to run within PAST [32], a protein structure searching server based in generalized suffix trees. First experiments are promising.

As a future line of research, we intend to explore the values of prefix-partitioned look-up trees in secondary memory settings. What bounds can be guaranteed for the number of I/O operations necessary for dictionary look-ups. Can an appropriate prefix-partitioned look-up tree be constructed directly in secondary memory? How does the tree compare with recently developed secondary memory data structures [12, 16]?

References

[1] A. N. Arslan and Ö. Eğecioğlu. Dictionary look-up within small edit distance, *Inter. J. of Found. of Comp. Sci.*, Vol. 15, No 1, pp. 57-71, February 2004.

[2] A. N. Arslan. Efficient approximate dictionary look-up for words over small alphabets, *Latin American Theoretical Informatics LATIN'06, Lecture Notes in Computer Science 3887*, Valdivia, Chile, March 20-24, 2006.

[3] G. S. Brodal and L. Gasieniec. Approximate dictionary queries, *In Proc. 7th Combinatorial Pattern Matching, LNCS, Vol. 1075, Springer, Berlin*, 65–74, 1996.

[4] G. S. Brodal and S. Velkatesh. Improved bounds for dictionary look-up with one error. *IPL*, 75, 57–59, 2000.

[5] R. Cole, L.-A. Gottlieb, and N. Lewenstein. Dictionary matching and indexing with errors and don't cares. *Proc. The 36th ACM STOC*, pp. 91-100, 2004.

[6] L. Devroye. A note on the probabilistic analysis of Patricia trees. *Random structures and algorithms*, vol. 3, pp. 203-214, 1992.

[7] L. Devroye. A study of trie-like structures under the density model. *Annals of applied probability*, vol. 2, pp. 402-434, 1992.

[8] L. Devroye. Universal asymptotics for random tries and Patricia tress. *Algorithmica*, 42: pp. 11-29, 2005.

[9] D. Dolev, Y. Harari, N. Linial, N. Nisan and M. Parnas. Neighborhood preserving hashing and approximate queries. *Proc. The Fifth ACM SODA*, 1994.

[10] D. Dolev, Y. Harari and M. Parnas. Finding the neighborhood of a query in a dictionary. *Proc. The Second Israel Symp. on Theory of Comp. and Sys.*, 1993.

[11] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21:246–260, 1974.

[12] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 26:236–280, 1999.

[13] D. Gusfield. Algorithms on strings, trees, and sequences: computer science and computational biology. *Cambridge University Press*, 1997.

[14] R. W. Hamming. Error Detecting and Error Correcting Codes. Bell Systems Technical Journal, 29, pp. 147–160, 1950.

[15] M. Hirao, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find the best subsequence patterns. DS 2000, LNAI 1967, pp. 141-153, 2000.

[16] P. Ko and S. Aluru. Obtaining provably good performance from suffix trees in secondary storage. *Proceedings of CPM 2006*, LNCS 4009, pp. 72–83, 2006.

[17] X. Ji, J. Bailey, and G. Dong. Mining minimal distinguishing subsequence with gap constraints. *Proceedings of 5th IEEE International Conference on Data Mining (ICDM)*, pp. 194-201, Houston, Texas, USA, 27 - 30 November 2005.

[18] D. E. Knuth. Distinguishing string selection problems. *The art of computer programming: sorting and searching*, Addison-Wesley, Reading, MA, 1973.

[19] J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. *Information and Computation*, 2003.

[20] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. Doklady Akademii Nauk SSSR, 163, 4, pp. 845–848, 1965.

[21] M. G. Maaß. Average-case analysis of approximate trie search. *CPM 2004, LNCS 3109*, pp. 472-484, 2004.

[22] M. G. Maaß and J. Nowak. Text indexing with errors. *CPM 2005, LNCS 3537*, pp. 21-32, 2005.

[23] U. Manber and S. Wu. An algorithm for approximate membership checking with applications to password security. *IPL*, 50, 191–197, 1994.

[24] M. Minsky and S. Papert. Perceptrons. *MIT Press, Cambridge, MA*, 1969.

[25] D. R. Morrison. PATRICIA - Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, Vol. 15, pp. 514-534, 1968.

[26] G. Navarro, R. Baeza-Yates, E. Sutinen and J. Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 24(4):19-27, 2001.

[27] B. Pittel. Aymptotical growth of a class of random trees. *Annals of probability*, vol. 13, pp. 414-427, 1985.

[28] D. Sankoff and J. B. Kruskal. Time wraps, strings, edits and macromolecules: the theory and practice of sequence comparison. *Addison-Wesley Publishing Company, Inc.*, 1983.

[29] H. Shang and T. H. Merrett. Tries for approximate string matching. *IEEE Trans. Knowl. Data Eng.*, 8(4):540-547, 1996.

[30] W. Szpankowski and C. Knessl. Heights in generalized tries and Patricia tries. *in: LATIN'2000, Lecture Notes in Computer Science*, vol. 1776, pp. 298-307, Springer-Verlag, Berlin, 2000.

[31] Wojciech Szpankowski. Average Case Analysis of Algorithms on Sequences. Wiley-Interscience, 2000.

[32] Hanjo Täubig, Arno Buchner and Jan H.G. Griebsch. PAST: fast structure-base searching in the PDB. *Nucleic Acids Research*, vol. 34, W20–W23, 2006.

[33] E. Ukkonen. Algorithms for Approximate String Matching. *Information and Control*, 64, 100-118, 1985.

[34] A. C. Yao and F. F. Yao. Dictionary look-up with one error. *J. of Algorithms*, 25(1), 194–202, 1997.