Dictionary Look-Up Within Small Edit Distance

Abdullah N. Arslan and Ömer Eğecioğlu [*]

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106 USA
{arslan,omer}@cs.ucsb.edu

Abstract. Let $\mathcal{W}$ be a dictionary consisting of n binary strings of length m each, represented as a trie. The usual d-query asks if there exists a string in $\mathcal{W}$ within Hamming distance d of a given binary query string q. We present an algorithm to determine if there is a member in $\mathcal{W}$ within *edit distance* d of a given query string q of length m. The method takes time $O(dm^{d+1})$ in the RAM model, independent of n, and requires $O(dm)$ additional space.

1 Introduction

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. Algorithms for answering d-queries efficiently has been a topic of interest for some time, and have also been studied as the *approximate query* and the *approximate query retrieval* problems in the literature. The problem was originally posed by Minsky and Papert in 1969 [10] in which they asked if there is a data structure that supports fast d*-queries*.

The cases of small d and large d for this problem seem to require different techniques for their solutions. The case when d is small was studied by Yao and Yao [14] . Dolev et al. [5, 6] and Greene et al. [7] have made some progress when d is relatively large. There are efficient algorithms only when $d = 1$; proposed by Brodal and Venkadesh [3], Yao and Yao [14], and Brodal and Gasieniec [2]. The small d case has applications in password security [9]. Searching biological sequence databases may also use the methods of answering d-queries.

Previous studies for the d-query problem have focused on minimizing the number of memory accesses for a d-query, assuming other computations are free, and used cell or bit probe models to express complexity. We assume a RAM model with constant memory access time and take into account all computations in the complexity analysis. Dolev et al. [6] presented bounds for the space and time complexity of the d-query problem under certain assumptions using various notions of proximity. In the model, $\mathcal{W}$ is stored in buckets, and preprocessing of $\mathcal{W}$ is allowed.

[*] Supported in part by NSF Grant No. EIA–9818320.

In this paper we consider answering d-queries efficiently without limiting ourselves to the construction of a new data structure parametrized by d. The variant of the original d-query problem that we consider is when the string-to-string *edit distance* is used as the distance measure instead of the ordinary case of Hamming distance. We assume that $\mathcal{W}$ is stored as a trie $\mathcal{T}_m$, and propose two algorithms for the d-query problem in this case. Our algorithms use the hybrid tree/dynamic programming approach [4]. The first one (Algorithm *LOOK-UP*$_{ed}$, Figure 4) requires $O(dm^{d+2})$ time in the worst case, and $O(dm^{d+1})$ space (in addition to the space requirements of the trie $\mathcal{T}_m$). This complexity is of interest for small values of d under investigation. The second algorithm (Algorithm *DFT-LOOK-UP*$_{ed}$, Figure 7) has time complexity $O(dm^{d+1})$, and additional space complexity of only $O(dm)$.

There is reason to believe that the average performance of both algorithms is much better when $\mathcal{W}$ is sparse.

2 Motivation: Hamming Distance Based Methods

Hamming distance between two binary strings is the number of positions they differ. A d-query asks if there is a member in a dictionary $\mathcal{W}$ whose Hamming distance is at most d from a given binary query string q.

We assume a trie representation $\mathcal{T}_m$ for $\mathcal{W}$, and assume for simplicity that $\mathcal{W}$ consists of binary words of length m each. A *trie* is a tree whose arcs are labeled by the symbols of alphabet Σ, in this case $\Sigma = \{0, 1\}$. The leaf nodes of $\mathcal{T}_m$ correspond to the words in $\mathcal{W}$, and when concatenated, the labels of arcs on a path from the root to a given intermediate node gives a prefix of at least one word in $\mathcal{W}$. Clearly, in the RAM model assumed, accessing a word in $\mathcal{W}$ takes $O(m)$ time. Figure 1 part (a) shows an example trie $\mathcal{T}_5$ representing a dictionary $\mathcal{W} = \{00011, 01001, 11111\}$.

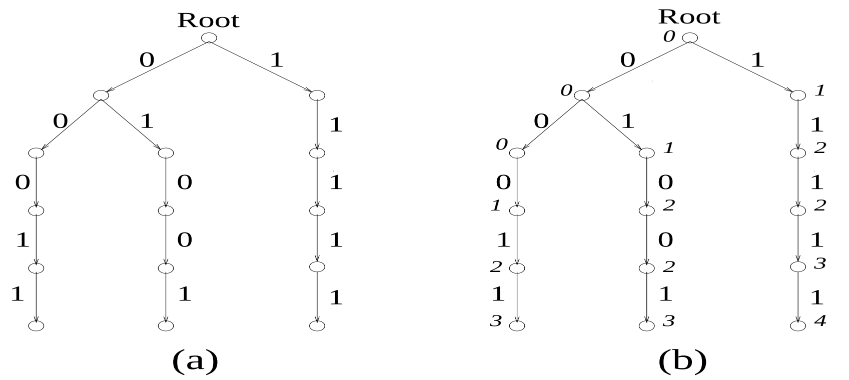

Fig. 1. a) An example trie with binary words 00011, 01001, and 11111 . b) The numbers in italic are the node weights computed with respect to query string 00100 .

A naive method for answering a d-query is to generate the whole set of $\sum_{k=0}^{d}\binom{m}{k}$ strings differing from q in at most d positions, and with every string generated, perform a dictionary look-up in $\mathcal{T}_m$ for an exact member in $\mathcal{W}$. This naive *generate and test* algorithm takes $O(m^{d+2})$ time and $O(m)$ additional space to store a generated string at a time. Another naive method is to add all strings within Hamming distance d from any member in $\mathcal{W}$ to obtain a bigger dictionary $\mathcal{W}'$. Then any d-query can be answered in $O(m)$ time using the corresponding trie $\mathcal{T}'_m$ for an exact member. This latter method significantly increases the size of $\mathcal{W}$ by a number roughly $O(nm^d)$ m-bit members. Cost of constructing and maintaining $\mathcal{T}'_m$ may be extremely high.

For Hamming distance, we can improve the first naive algorithm above as follows. Let $s(v)$ denote the prefix corresponding to trie node v. Given a query string q, suppose that we assign weight w_h to each trie node v in $\mathcal{T}_m$ as

$$w_h(v) = h(s(v), q_{1\cdots|s(v)|}), \tag{1}$$

where h denotes Hamming distance. As an example, in Figure 1 (b) the weights of the nodes have been computed with respect to query string $q = 00100$. The idea is that we can prune the trie in our search for q at the nodes in $\mathcal{T}_m$ with $w_h(v) > d$.

Lemma 1. *Let N be the number of nodes in $\mathcal{T}_m$ with weight $\leq d$ as defined in (1). Then $N = O(m^{d+1})$.*

Proof. It is easy to see that N is maximized over all tries $\mathcal{T}_m$ when $\mathcal{T}_m$ is a complete trie over Σ, i.e. $\mathcal{T}_m$ contains all binary strings of length m. Figure 2 shows node weights of a complete trie with respect to q up to level 4 starting with the root at level 0. The root has weight 0. For any other vertex v at level l, if the arc from its parent to v has label q_l (the lth symbol of query string q) then v and its parent have the same weight; otherwise, weight of v is 1 more than that of its parent . Let $L(l, w)$ denote the number of vertices with weight w at level l of the complete trie $\mathcal{T}_m$. At any level l, the largest weight is l . Using these observations we see that $L(l+1, w) = L(l, w) + L(l, w-1)$ with $l \geq w$ and $L(l, 0) = 1$. Therefore $L(l, w)$ is the binomial coefficient $\binom{l}{w}$. Furthermore since the smallest level at which weight w appears in $\mathcal{T}_m$ is $l = w$, the total number of vertices with weight w in $\mathcal{T}_m$ is $\binom{w}{w} + \binom{w+1}{w} + \cdots + \binom{m}{w} = \binom{m+1}{w+1}$. Hence

$$N = \sum_{w=0}^{d} \binom{m+1}{w+1} = O(m^{d+1})$$

Based on the above lemma, Figure 3 outlines Algorithm $LOOK\text{-}UP_h$ for dictionary look-up within Hamming distance d. The algorithm explores all nodes v in $\mathcal{T}_m$ with weight $w_h(v) \leq d$, i.e. $s(v)$ is a prefix of a word in $\mathcal{W}$ whose Hamming distance from q is potentially within d. S_k stores the set of node-weight pairs $(v, w_h(v))$ for all nodes v at levels $\leq k$ with weight $w_h(v) \leq d$. The algorithm iteratively computes S_k from S_{k-1} by collecting all pairs $(v[a], w(v) + h(a, q_k))$ in S_k where $(v, w(v)) \in S_{k-1}$, $w(v) + h(a, q_k) \leq d$, and $a \in \Sigma = \{0, 1\}$.

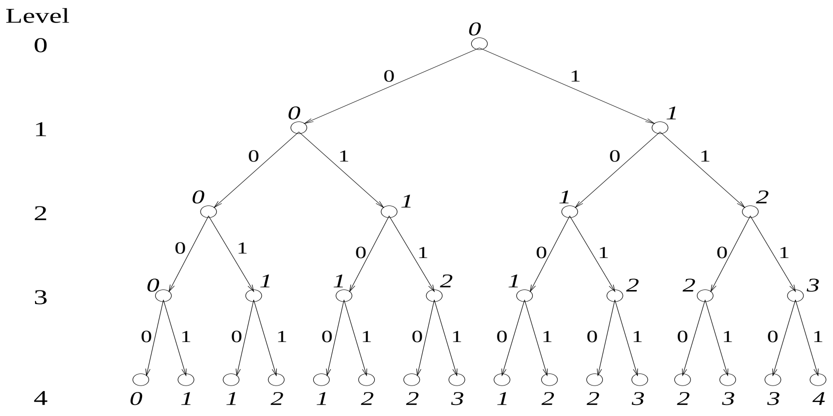

Fig. 2. In a complete binary trie of height 4, weights with respect to a given binary query string are shown in italic.

Clearly, if there is a member in $\mathcal{W}$ within Hamming distance d then it will be captured in S_m in which case the algorithm returns YES; otherwise it returns NO.

S_k contains $O(m^{d+1})$ node-weight pairs by Lemma 1. Therefore the time complexity in the assumed model is $O(m^{d+2})$. It also requires additional space to store $O(m^{d+1})$ trie nodes. The time complexity is no better in the worst case than that of the naive algorithm which generates and tries all possible strings within Hamming distance d from q. However for a sparse dictionary $\mathcal{W}$ Algorithm *LOOK-UP*$_h$ is bound to be much faster on the average.

```
Algorithm $LOOK\text{-}UP_h(d)$
$S_0 = \{(v_0, 0)\}$
for $k = 1$ to $m$ do
    $S_k = \{(v[a], w(v) + h(a, q_k)) \mid (v, w(v)) \in S_{k-1},\ w(v) + h(a, q_k) \le d,$ and $a \in \Sigma\}$
Return YES if the minimum weight in $S_m$ is $\le d$, NO otherwise
```

Fig. 3. Algorithm *LOOK-UP*$_h$ for dictionary look-up within Hamming distance d.

3 Edit Distance Based *d*-queries

Algorithm *LOOK-UP*$_h$ essentially computes the Hamming distance between q and a selected part of dictionary $\mathcal{W}$. Next we investigate the possibility of using a similar idea for d-queries defined with respect to the *edit distance*.

For the purposes of this paper, we use a simple type of edit distance. Given two strings $p = p_1 \cdots p_m$ and $q = q_1 \cdots q_m$, the edit distance $ed(p, q)$ is the minimum number of edit operations which transforms p into q. The edit operations are of three types: insert, delete, and substitute. Substituting a symbol by itself is called a *match*. A match operation is the only operation that does not contribute to the number of steps of the transformation. In terms of costs, all edit operations have cost 1 except for the match whose cost is 0. The usual framework for the analysis of edit distance is the *edit graph*. Edit Graph $G_{p,q}$ is a directed acyclic graph having $(m+1)^2$ lattice points (u,v) as vertices for $0 \le u, v \le m$. Horizontal and vertical arcs correspond to insert and delete operations respectively. The diagonal arcs correspond to substitutions. Each arc has a cost corresponding to the edit operation it represents. If we trace the arcs of a path from node $(0,0)$ to an intermediate node (i,j), and perform the indicated edit operations in the given order on $p_1 \cdots p_i$ then we obtain $q_1 \cdots q_j$. Edit distance between prefixes $p_1 \cdots p_i$ and $q_1 \cdots q_j$ is the cost of the minimum-cost path from $(0,0)$ to (i,j), and can be computed from the distances achieved at nodes $(i-1,j)$, $(i-1,j-1)$, and $(i,j-1)$. Hence it has a simple dynamic programming formulation [13]:

$$D_{i,j} = \min\{\ D_{i-1,j} + 1,\ D_{i-1,j-1} + h(p_i, q_j),\ D_{i,j-1} + 1\} \tag{2}$$

for $1 \le i, j \le m$, with $D_{i,j} = 0$ whenever $i = 0$ or $j = 0$.

3.1 Algorithm *LOOK-UP*$_{ed}$

With respect to a given binary query string q, we assign weight w_{ed} to any trie node v in $\mathcal{T}_m$ as

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

where *ed* denotes the edit distance.

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

Proof. Proof is similar to the proof of Lemma 1 for the Hamming distance case. Analysis of N over a complete binary trie gives the maximum N. We omit the details. We remark however that in general $w_{ed}(v) \ne w_h(v)$, and for non-complete binary tries the distribution of weights over the nodes can differ significantly for Hamming and edit distances.

Algorithm *LOOK-UP*$_{ed}$ shown in Figure 4 extends the dynamic programming formulation (2) of the edit distance computation by considering all prefixes of all members in $\mathcal{W}$. $S_{i,j}$ stores all the node-weight pairs $(v, w_{ed}(v))$ where $s(v) = p_1 \cdots p_i$ for some $p \in \mathcal{W}$, and $w_{ed}(v) = ed(p_1 \cdots p_i, q_1 \cdots q_j)$.

The computations in *LOOK-UP*$_{ed}$ involve sets, as opposed to just scores of the ordinary edit distance computations. Edit operation and the trie nodes involved determine an action on the sets. Consider the operations resulting in at node (i,j) of the edit graph as shown in Figure 5. Let the operation be the deletion of symbol $p_i \in \Sigma$. For $(v,t) \in S_{i-1,j}$ if there is an arc from v to $v[a]$ with label $a \in \Sigma$ then the delete operation causes weight $t+1$ in $v[a]$. This potential weight assignment is reflected in set $S'_{i-1,j}$, and realized in set $S_{i,j}$ only if no

```
Algorithm $LOOK\text{-}UP_{ed}(d)$
$S_{i,-1} = \emptyset$ for all $i$,   $1 \le i \le d$
$S_{-1,j} = \emptyset$ for all $j$,   $1 \le j \le d$
$S_{0,0} = \{(v_0, 0)\}$
for $i = 0$ to $m$ do
  for $j = \max\{0, i - \lceil d/2 \rceil\}$ to $\min\{m, i + \lfloor d/2 \rfloor\}$ do
   {
    if ($i = 0$ and $j = 0$) then continue with the next iteration
    else
     {
      $S'_{i-1,j} = \{(v[a], t+1) \mid t+1 \le d,\ (v,t) \in S_{i-1,j}$ and $a \in \Sigma\ \}$
      $S'_{i,j-1} = \{(v, t+1) \mid t+1 \le d,\ (v,t) \in S_{i-1,j}\ \}$
      $S'_{i-1,j-1} = \{(v[a], t+h(a,q_j)) \mid t+h(a,q_j) \le d,\ (v,t) \in S_{i-1,j-1},\ a \in \Sigma\ \}$
      $S_{i,j} = \{(v,t) \mid t$ is the minimum weight
                  paired with leaf $v$ in $S'_{i-1,j} \cup S'_{i,j-1} \cup S'_{i-1,j-1}\}$
     }
   }
Return YES if the minimum weight in $S_{m,m}$ is $\le d$; otherwise NO
```

Fig. 4. Algorithm $LOOK\text{-}UP_{ed}$ for dictionary look-up within edit distance d.

weight smaller than $t+1$ is achieved by other edit operations resulting in $v[a]$. Now consider the insertion of q_j. For each $(v,t) \in S_{i,j-1}$, the pair $(v, t+1)$ is inserted into $S'_{i,j-1}$. Similarly for each $(v,t) \in S_{i-1,j-1}$, $(v[a], t + h(a, q_j))$ is inserted into $S'_{i-1,j-1}$. Subsequently, all node-weight pairs in sets $S'_{i-1,j}$, $S'_{i,j-1}$, and $S'_{i-1,j-1}$ are collected into set $S_{i,j}$ by including for each node at most one pair, namely the one with the minimum weight.

The computations on the edit graph can be restricted to a narrow diagonal band of the edit graph as shown in Figure 5, since any edit path with total weight at most d completely lies in this band.

We can easily show by induction that for all i, j with $0 \le i \le m$, and $\max\{0, i - \lceil d/2 \rceil\} \le j \le \min\{i, i + \lfloor d/2 \rfloor\}$, $(v,t) \in S_{i,j}$ iff there exists a trie node v such that $|s(v)| = i$ and $ed(s(v), q_1 \cdots q_j) = t$. This implies that $S_{m,m}$ includes a node-weight pair with weight $\le d$ for a leaf node iff there exists a member in $\mathcal{W}$ within edit distance d. Therefore the algorithm is correct.

Figure 6 shows an example edit path with partial results which identify member 01001 as within edit distance 2 of $q = 00100$.

By Lemma 2, computing each $S_{i,j}$ from $S_{i-1,j}, S_{i,j-1}, S_{i-1,j-1}$ takes $O(m^{d+1})$ time since the size of each of these sets is bounded by $O(m^{d+1})$. Therefore the time complexity of Algorithm $LOOK\text{-}UP_{ed}$ is $O(dm^{d+2})$, and it requires $O(dm^{d+1})$ additional space since it is enough to store the sets of the previous and current rows as the processing is done row by row.

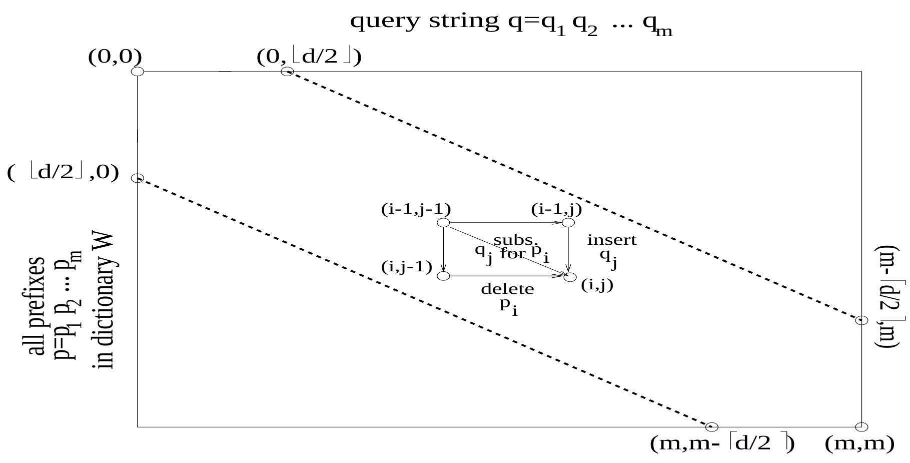

Fig. 5. Part of the edit graph explored during the computations.

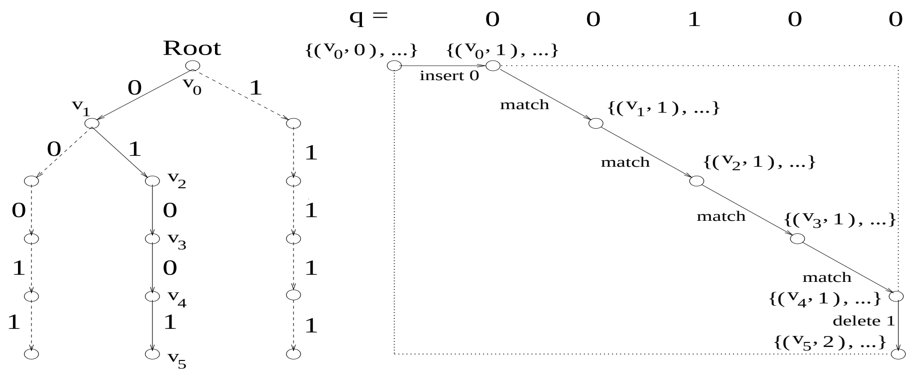

Fig. 6. Partial results on an edit path for a member within edit distance 2 of q.

3.2 Algorithm *DFT-LOOK-UP$_{ed}$*

Next we propose Algorithm *DFT-LOOK-UP$_{ed}$* with which we improve the time complexity of the d-query problem with respect to edit distance to $O(dm^{d+1})$, and space complexity to $O(dm)$. The steps of the algorithm are shown in Figure 7. The algorithm is based on depth-first traversal (DFT) of trie $\mathcal{T}_m$, during which the entries of the dynamic programming matrix are computed row by row. For trie node v, $i = level(v)$ (Figure 9), and $\max\{0, i - \lceil d/2 \rceil\} \leq j \leq \min\{m, i + \lfloor d/2 \rfloor\}$ we define $D_{v,i,j}$ as

$$D_{v,i,j} = ed(s(v), q_1 \cdots q_j)$$

Algorithm *DFT-LOOK-UP$_{ed}$* performs the initialization of scores for the first row, and invokes Procedure *DFT-COMPUTE-D$_{ed}$* for each arc from root v_0 to $v_0[a]$ with label a. If any of these invocations returns a value $\leq d$ then the algorithm returns YES; otherwise returns NO.

```
Algorithm $DFT\text{-}LOOK\text{-}UP_{ed}(d)$
    $D_{v_0,0,j} = 0$ for all $j$,   $0 \le j \le \lfloor d/2 \rfloor$
    for each arc from $v$ to $v[a]$ on any $a \in \Sigma$ do
        if $DFT\text{-}COMPUTE\text{-}D_{ed}(v_0, a, v_0[a]) \le d$ then return YES
    return NO
```

Fig. 7. Algorithm *DFT-LOOK-UP*$_{ed}$ for dictionary look-up within edit distance d.

Given a parent node v, children node u, and symbol $a \in \Sigma$ of the arc connecting these two, Procedure *DFT-COMPUTE-D*$_{ed}(v, a, u)$ first computes the values in the row of node u using the values in the row of parent node v in the edit graph (Figure 9). If all computed entries in this row are $> d$, then the procedure returns the minimum of these numbers. Otherwise it traverses the subtrie rooted at u in depth-first manner, computes and returns the minimum edit distance achievable in the leaf nodes of this subtrie.

```
Procedure $DFT\text{-}COMPUTE\text{-}D_{ed}(v, a, u)$
    $i = level(v)$
    for $j = \max\{0, i - \lceil d/2 \rceil\}$ to $\min\{m, i + \lfloor d/2 \rfloor\}$ do
        $D_{u,i,j} = \min\{\ D_{v,i-1,j} + 1,\ \ D_{v,i-1,j-1} + h(a, q_j),\ \ D_{v,i,j-1} + 1\}$
    weight $= \min\{D_{u,i,j}\ \mid\ \max\{0, i - \lceil d/2 \rceil\} \le j \le \min\{m, i + \lfloor d/2 \rfloor\}\}$
    if $u$ is a leaf node or weight $> d$ then return weight
    return $\min\{DFT\text{-}COMPUTE\text{-}D_{ed}(u, a, u[a])$ | there is an arc
                incident from $u$ to $u[a]$ on $a \in \Sigma\}$
```

Fig. 8. Procedure *DFT-COMPUTE-D*$_{ed}$ for computing the minimum edit distance achieved in subtrie rooted at v.

To show correctness, we claim that $D_{v,i,j}$ stores $ed(s(v), q_1 \cdots q_j)$ where $i = level(v)$. This can be shown by induction. Assume that before v is visited for the parent of v, and corresponding entries, the claim is true then following the computations for v we can easily see that the claim will be true for v after the processing is done for the entries of v. Another induction on the subtries of v reveals that the procedure call on v will return the minimum edit distance achieved in the leaves of the subtrie rooted at v. Therefore the algorithm returns YES iff there is a member in $\mathcal{W}$ within edit distance d of q.

Depth-first traversal visits $O(m^{d+1})$ trie nodes by Lemma 2. With every node visited $O(d)$ operations are performed. Therefore the time complexity of the algorithm is $O(dm^{d+1})$. Dept-first traversal requires that a branch of $O(m)$ trie nodes be stored. For each node $O(d)$ entries are maintained. Hence the space complexity of the algorithm is $O(dm)$.

We can adapt *DFT-LOOK-UP*$_{ed}$ (and *DFT-COMPUTE-D*$_{ed}$) to Hamming distance computations as well. In this case, we only need to consider the diagonal entries of the edit graph, and each entry is computed using only the value

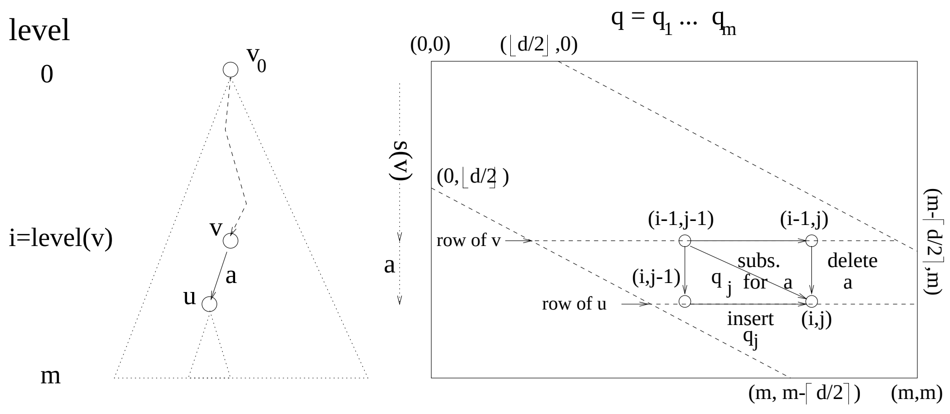

Fig. 9. Depth-first traversal on the trie, and the region of the dynamic programming matrix in which the computations are performed.

of the parent which is stored in the diagonal of the previous level. The resulting algorithm *DFT-LOOK-UP*$_h$ for Hamming distance based d-queries has time complexity $O(m^{d+1})$ and space $O(m)$.

4 Remarks

For clarity of presentation, we have assumed a dictionary of words of equal length. With some additional care our method can be generalized to the cases in which different word lengths, and larger alphabets are allowed for the dictionary.

Restricting the dynamic programming computation to a diagonal band of edit graph was used by Ukkonen [12]. We have essentially incorporated this idea in our method. For the purpose of developing new methods for d-queries, the idea of using sets to keep track of partial results may also be used in conjunction with suitable edit distance algorithms such as by Myers [11], and Kim et. al [8]. The algorithm in [8] is interesting in particular because it extends the definition of edit distance by allowing swaps.

As we remarked in section 2, the naive *generate and test* d-query algorithm for Hamming distance requires $O(m^{d+1})$ time and $O(m)$ space. If there exists an efficient algorithm to generate all binary strings within *edit distance* d of q then we can devise a similar *generate and test* d-query algorithm for the edit distance case. Similarly, the naive method for the Hamming distance in section 2 obtained by enlarging $\mathcal{W}$ can be adapted to the case of edit distance by efficient generation of words that are within edit distance d of the words in $\mathcal{W}$ if we agree to pay a very high cost for constructing, and maintaining the extended dictionary.

5 Conclusion

We have presented two algorithms *LOOK-UP* and *DFT-LOOK-UP* for answering d-queries in a dictionary of n binary words of length m. The algorithms incorporate the proximity search as part of the distance computation. This approach does not yield improved worst-case time complexity result in the case of Hamming distance compared to a naive generate and test approach. When edit distance is used we achieve worst-case $O(dm^{d+1})$ time and $O(dm)$ space complexities independent of n.

The average case analysis of the two algorithms presented for edit distance over larger alphabets and dictionaries consisting of arbitrary length words are additional topics of investigation.

Acknowledgement

The authors would like to thank the anonymous referee for pointing out the reference [1] in which suffix trees are used in regular expression pattern matching, and the hybrid tree/dynamic programming approach in [4], which is essentially the method presented here for a set of problems involving patterns in strings.

References

1. P. Bieganski. Genetic Sequence Data Retrieval and Manipulation based on Generalized Suffix Trees. *PhD thesis*, University of Minnesota, 1995.
2. G. S. Brodal and L. Gasieniec. Approximate dictionary queries, *in: Poc. 7th Combinatorial Pattern Matching, LNCS, Vol. 1075, Springer, Berlin*, 65–74, 1996.
3. G. S. Brodal and S. Velkatesh. Improved bounds for dictionary look-up with one error. *IPL*, 75, 57–59, 2000.
4. D. Gusfield. Algorithms on strings, trees, and sequences : computer science and computational biology. *Cambridge University Press*, 1997.
5. D. Dolev, Y. Harari, N. Linial, N. Nisan and M. Parnas. Neighborhood preserving hashing and approximate queries. *Proceedings of the Fifth ACM SODA*, 1994.
6. D. Dolev, Y. Harari and M. Parnas. Finding the neighborhood of a query in a dictionary. *Proceedings of the Second Israel Symposium on Theory of Computing and Systems*, 1993.
7. D. Greene, M. Parnas and F. Yao. Multi-index hashing for information retrieval. *Proceedings of 1994 IEEE FOCS*, pp. 722–731, November 1994.
8. D. K. Kim, J. S. Lee, K. Park and Y. Cho. Algorithms for approximate string matching with swaps. *J. of Complexity*, 15, 128–147, 1997.
9. U. Manber and S. Wu. An algorithm for approximate membership checking with applications to password security. *IPL*, 50, 191–197, 1994.
10. M. Minsky and S. Papert. Perceptrons. *MIT Press, Cambridge, MA*, 1969.
11. E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
12. E. Ukkonen. Algorithms for Approximate String Matching. *Information and Control*, 64, 100-118, 1985.
13. R. A. Wagner and M. J. Fisher. The string-to-string correction problem. *JACM*, 21(1):168–173, January 1974.
14. A. C. Yao and F. F. Yao. Dictionary look-up with one error. *J. of Algorithms*, 25(1), 194–202, 1997.