Fast Filtering for HMMs in Domain Discovery

Ethan Kim
School of Computer Science
McGill University, Montreal, Québec, Canada
ethan@cs.mcgill.ca

Harley Cooper
Department of Biology
McGill University, Montreal, Québec, Canada
harley.cooper@mcgill.ca

April 19, 2006

Abstract

Hidden Markov Models(HMMs) have been very popular tools for domain detection problem in computational biology. However, when the size of the input data is large, simple linear searches through the entire database against the HMM are not sufficiently fast for realistic applications. In this paper, we propose different methods to accelerate this process by using techniques from combinatorial data structures and graph theoretic algorithms.

1 Introduction

Hidden Markov Models (HMMs) are useful tools for modeling a sequence of unknown stochastic processes, and have been widely used in speech recognition problems [13]. Beginning with the famous Viterbi algorithm [14], many efforts have been made to use Markov models for pattern recognition applications [3], and such approaches are also of interest in the domain of computational biology, such as for multiple sequence alignment and protein modeling [11].

We consider the problem of detecting a particular domain in a database of sequences. In particular, we assume that the HMM for the domain is given, and we wish to find a set of sequences from the database which score high in the HMM. An example of such a problem is to find CpG islands in the human genome [12]. Detecting domains with HMMs is a well studied topic in many other applications, but as the database of the sequences becomes large, even linear search methods are not sufficiently fast for realistic applications. Here, we look

at different techniques from random data structures and combinatorial optimizations in an attempt to accelerate the process of domain detection.

This paper is organized as follows: In Section 2, we give a short introduction on trie data structures, and compare the sizes of computation to be done via simple linear searches and via a trie structure. In Section 3, we describe our main results using the implicit trie structure from the preprocessing of the database. In Section 4, we propose another method for domain detection, which runs in parameterized running time and gives the user the freedom to select the quality of output, or the time available for the computation. Finally, in the Appendix we give a short proof of the upper bound on the size of trie structures, for the purposes of the theoretical analysis of our approaches.

2 Preliminaries

2.1 Tries

Tries, also known as prefix trees or digital search trees, are d-ary position trees which can efficiently store and access a collection of n strings, $S = \{s_1, s_2, \ldots, s_n\}$. Assume that each element is a string of length w over an alphabet of size d together with a terminal character, \$, i.e: $s_i \in \Sigma^w \cup \$$. For the remainder of the paper, we let $\mathcal{S}$ denote the sum of the length of all strings in S. A trie T can be either (1) null, (2) a string, or (3) a node with at most d child nodes which are also tries. Where strings share the same prefix, they will be indexed by the same path in the trie, branching off where they differ, until the point of uniqueness, where the remainder of the string is stored in a node [9]. For an example of a trie, see Figure 1.

Tries are extremely useful for highly repetitive data, such as dictionaries and gene sequences; as more strings are added, the space savings increase rapidly. Intuitively, we can see that this is especially true for large numbers of short strings, since a string that is very long and does not share a prefix with any others will contribute its full length to the size of the trie. Insertion of a string can result in a space increase ranging from zero if the string is fully contained within those already in the trie—much more likely in the case of a shorter string—to some number less than or equal to the total size of the string.

To the best of our knowledge, there appears to have been no formal examination in the literature about the worst-case and average-case analysis on the size of a trie. Since we are examining tries as a method of speeding up HMM searches, a size bound is crucial in the projection of the degree to which tries can accelerate the process. Hence, we give a formal proof on the exact bound on trie structures in the Appendix.

2.2 Comparison of Trie-Based and Tabular Approaches

We took both genomic data (random sequence data from *H. sapiens* chromosome 1) and randomly-generated nucleotides ($\Pr(\texttt{A}) = \Pr(\texttt{G}) = \Pr(\texttt{C}) = \Pr(\texttt{T}) = 0.25$) both totalling $\mathcal{S} =$ 3,223,233 characters and split everything into words of size w for $w = 5, 10, 15, \ldots, 50$. Then for each word size we plotted:

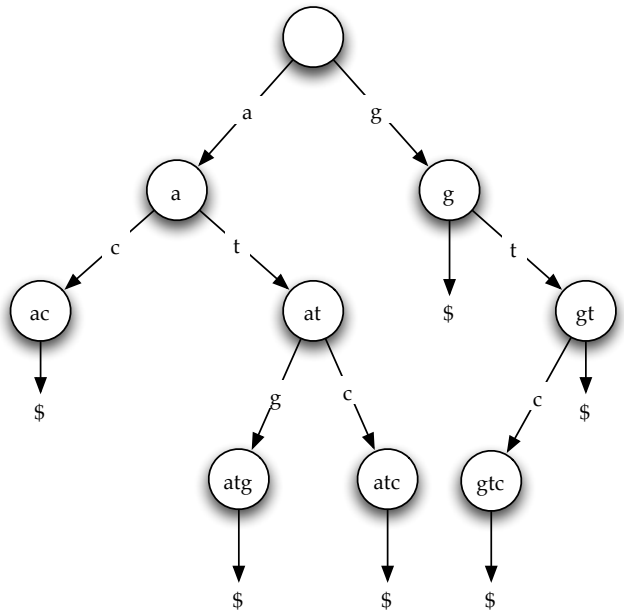

Figure 1: An example of a trie indexing the sequences `ac`, `atg`, `atc`, `g`, `gt` and `gtc`

- The size of the table needed to store all of the data ($w(\mathcal{S} - w + 1)$)
- The size of the trie generated from the data
- The theoretical bound on the space required by the trie, given as

$$\frac{d^{\lfloor \log_d n \rfloor + 1} - 1}{d - 1} + n\left(w - \lfloor \log_d n \rfloor\right)$$

  where d is the size of the alphabet and n is the number of strings in S. (For the proof of this bound, see the Appendix)

In figure 2, we can see that the random data almost exactly followed the theoretical upper bound, as would be expected, since multiple randomly-generated sequences are much more likely to all be quite different from one another, impairing trie performance. The genomic data resulted in tries that were generally around 55% smaller than the theoretical bound, showing that tries offer a significant reduction in space requirements for biological data.

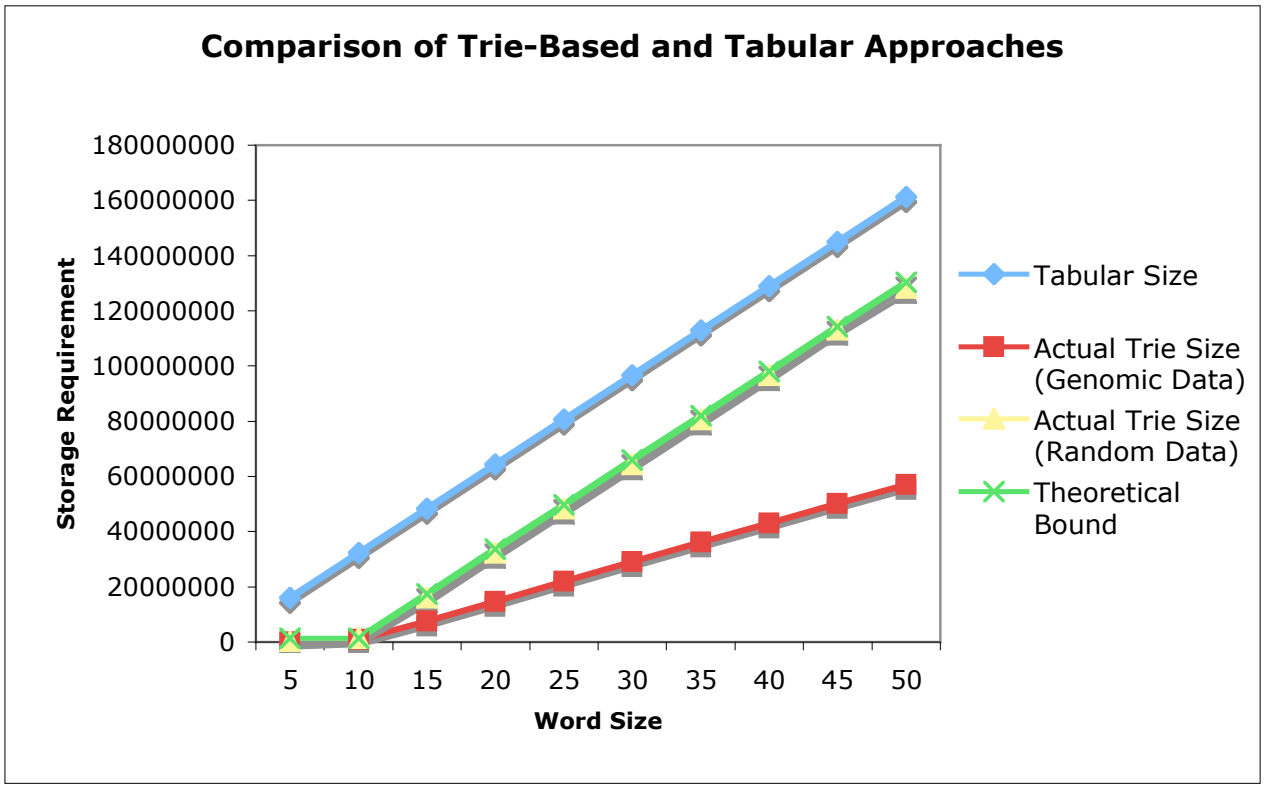

Figure 2: Plot of the space required to store 3,223,233 characters split up into words of size w in a table (blue), in a trie, given a theoretical worst case scenario (green), in a trie, given the genomic data (red), and in a trie, given randomly-generated genomic sequence data (yellow).

3 Trie-based Approach with DAWGs

The conventional method of domain detection works by taking a substring from the database, and simply run the Viterbi algorithm on the substring. The average size of a single instance of a domain is implicitly given by the HMM, and for the remainder of the paper, we assume that the substrings to be found are of length w. A single run of the Viterbi algorithm takes $O(|Q|^2 w)$ time, where $|Q|$ denotes the number of states in the given HMM. Since there are exactly $\mathcal{S} - w + 1$ substrings, the overall running time is $O(|Q|^2 w(\mathcal{S} - w + 1))$. As the typical value for $\mathcal{S}$ is large, even a small change by a constant factor for $\mathcal{S}$ would hurt (or save) the running time significantly.

The method proposed in this section uses two data structures to accelerate the process of domain detection. First, we introduce the method of using tries to reduce the computation time required for the Viterbi algorithm. We then make further improvements by preprocessing the database, and as a result, each query with a different HMM can be carried out in much less time.

3.1 Trie-based approach

We have shown both theoretically (see the Appendix) and empirically (see Section 2) that the sizes of tries are significantly smaller than the tabular representation of strings. The method we propose is based on the compactness of the trie data structure. The first step of the approach is to scan through the database, looking for substrings of size w. As new strings are discovered, we insert the strings into the trie structure. Once the construction of the trie T is complete, we can then run the Viterbi algorithm on T, in a *Depth-First-Search*(DFS) fashion.

To see the correctness of this approach, notice that the Viterbi algorithm uses a forward recurrence for the dynamic programming such that the solution for the n^{th} character only depends on the solution for the first $n-1$ characters. In particular, the Viterbi algorithm typically models first-order markov chains such as HMMs, and thus we can safely run the Viterbi algorithm on the trie using characters on the stack during the DFS.

The running time for this approach would consist of: (1) constructing the trie T with a sliding window of size w on the database and (2) running the Viterbi algorithm on T. The time taken for Step 2 is $O(|Q|^2|T|)$, where $|T|$ denotes the size of the trie (the exact bound on $|T|$ is given in Corollary 1.) However, the construction of T in Step 1 would be $O(w(\mathcal{S}-w+1))$ if we have to look at all possible substrings by a sliding window of length w. In the next section, we improve this method by preprocessing the database.

3.2 Preprocessing using a Directed Acyclic Word Graph (DAWG)

The problem of storing a collection of strings has been studied extensively. One of the most surprising results on space-efficient data structures for strings [4] is that the size of the minimal automaton accepting the suffixes of a string is linear. The first graph structure to represent subwords of a word was the *Directed Acyclic Word Graph* (DAWG), and many other variations were developed to store suffixes of a string in linear space. See Figure 3 for an example of a DAWG. Here, we define DAWGs formally for completeness.

Definition 1. *The Suffix Automaton DAWG of a string s is the minimal deterministic automaton that accepts the set of suffixes of s.*

An alternative data structure to represent suffixes of a string is a *suffix tree*, which is constructed by inserting all of the suffixes into a trie-like structure. Of course, a suffix tree would contain a greater number of vertices, as each suffix would be represented by a unique root-to-leaf path in the tree. For an extensive survey on different data structures to represent strings, see [10].

Recently, further improvements were made to reduce the size of DAWGs. A Compact Directed Acyclic Word Graphs (CDAWG) (Figure 4) is the compact version of a DAWG, where each arc (u, v) is contracted if (u, v) is the only incoming arc for v, thus allowing multiple characters to be represented on a single arc. After applying the compaction, the size of the DAWG becomes at most the size of the string. The following lemma is a result from [7].

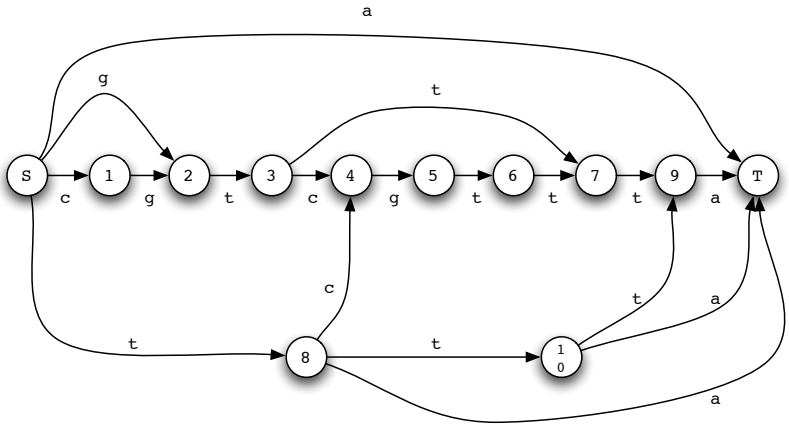

Figure 3: DAWG for `cgtcgttta`

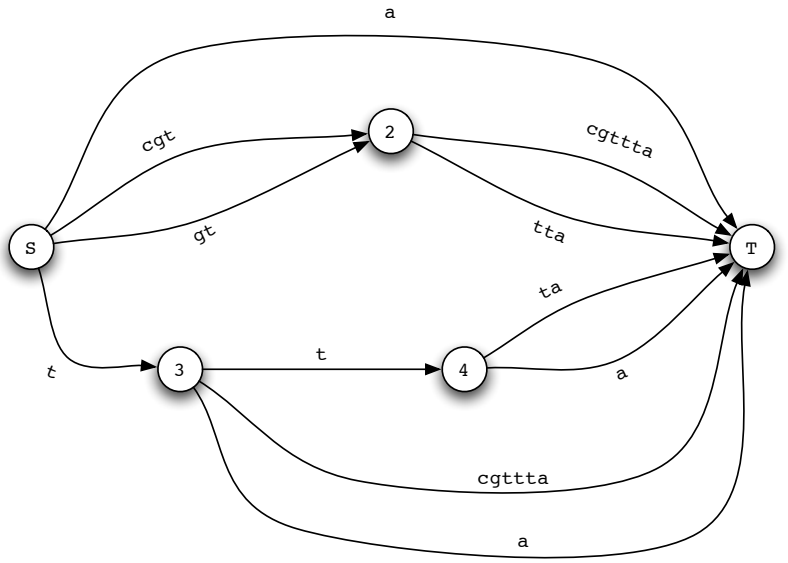

Figure 4: CDAWG for `cgtcgttta`

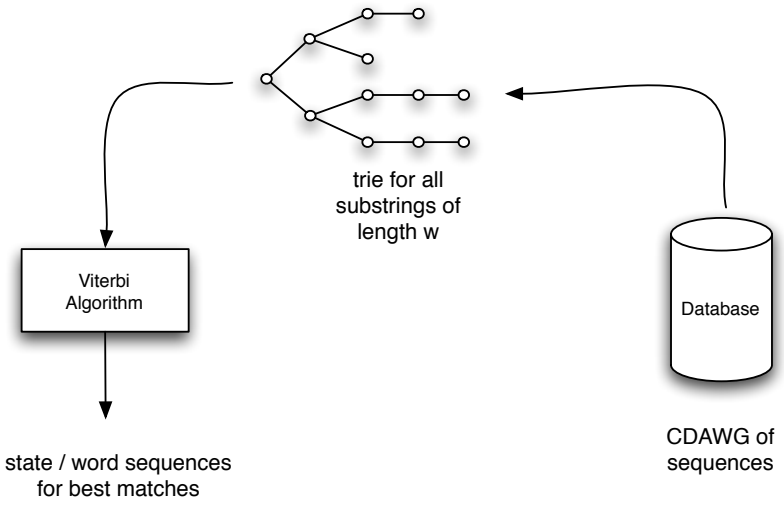

Figure 5: System layout for the trie-based method

Lemma 1 (Crochemore and Vérin, 1997). *Given $x \in \Sigma^*$, if $|x| \geq 2$, then $2 \leq \text{States}(x) \leq |x| + 1$. Further, the algorithm that builds the CDAWG of a word x runs in time $O(|x|)$.*

We propose the construction of a CDAWG of the database as a preprocessing step. Due to Lemma 1, the size of the database would stay the same as tabular representation, and any updates on the database can be carried out in linear time. See Figure 5 for an overview of the system.

3.3 Algorithms and Complexity Issues

There are three phases in the method we propose: (1) Construction of a CDAWG of the database, (2) construction of a trie T of length w from the CDAWG, and (3) running the Viterbi algorithm on T. The first step can be done in $O(\mathcal{S})$ time and space. The second step can be done via looking for all paths of length w, starting from the root node. This can be done in $O(|T|)$ time via DFS-like search. Finally, running the Viterbi algorithm on the trie can be done in $O(|Q|^2|T|)$. Notice, however, that the first step can be done as a preprocessing step, since it does not depend on the HMMs given at each query. Therefore, the overall running time per query is simply $O(|Q|^2|T|)$.

This time complexity shows a significant improvement over the conventional sliding window approach. The improvement is due to the indexing done in the CDAWG, which allows us to construct the trie without carrying out a repetitive linear search on the database. However, there is a speed vs. space trade-off. The size of the trie helps us quickly evaluate all of the substrings, but the trie structure needs to be stored in memory throughout the computation. For a large number of distinct substrings, Step 2 may not be completed simply due to space requirements. This problem can be resolved, however, by merging the Step 2

and 3. If the Viterbi algorithm is run as we visit the nodes in the CDAWG in a DFS manner, the computation required at each node v only depends on the unique root-to-v path(ie. the vertices stored on the stack). Thus, we can run the Viterbi algorithm on the trie that is implicitly embedded in CDAWG, and the overall space complexity for each query is then $O(w)$. A simplified pseudocode for the algorithm is given below.

DAWG-VITERBI(node v, depth d)

```
1  Evaluate v based on characters on stack.
2  Push v onto stack
3  if d = w
4      then Output the state sequence of the characters on stack.
5      else
6          for each neighbor u of v
7              do DAWG-VITERBI(u, d+1)
```

4 The k-Best Algorithm on profile HMMs

In this section, we propose another method for the domain detection problem. In particular, we look at hidden Markov models with dummy states for the begin and end states, as in the form of profile HMMs. The main bottleneck of the Viterbi algorithm-based approaches is that one needs to scan through the entire database of DNA or protein sequences, and then feed all the sequences into the Viterbi algorithm, regardless of how high the substrings would score. As the size of the database grows, the running time tends to slow down rapidly, and even linear-time algorithms are not sufficiently fast for realistic applications.

Recalling the method we proposed in the previous section, the running time of the algorithm is dominated by the size of the trie. Although such trie-based approaches guarantee that one never misses high-scoring substrings, having to consider all substrings in the database may result in exponential time, regardless of how high the string should score to be considered a "good match". The approach we propose in this section is sensitive to the *probability score* computed via the HMM model, in the sense that the running time is inversely proportional to the lowest score of all detected domains, and thus the user has the freedom to choose how good the output should be, or stated differently, how long the computation should be carried out.

First, let ϕ denote the probability threshold such that we consider a word sequence s_i a "good hit" against the given HMM if the probability of state sequence found on s_i has a probability above ϕ. Suppose we somehow have the list of all the permuted sequences whose score is above ϕ. Then, we simply need to find where in the database such sequences occur. Using the DAWG-based representation of the database, the search can be done in $O(kw)$ time, where k is the number of such sequences. The method presented in this section deals with how to find such high-scoring sequences from an HMM.

4.1 Transformation of profile HMMs

Consider an HMM in its profile HMM form, and denote it by H. The model H can then be defined by

- $Q = \{q_{begin}, q_{end}\} \cup M \cup I \cup D$, the set of states, where
  - q_{begin} and q_{end}, the begin and end states
  - $M = \{m_1, m_2, \ldots\}$, the matching states
  - $I = \{i_1, i_2, \ldots\}$, the insertion states
  - $D = \{d_1, d_2, \ldots\}$, the deletion states
- $t : Q \times Q \to R^+$, the transition probability function between states
- $e : M \cup I \to R^+$, the emission probability function from each matching and insertion states.

Intuitively, the model H already exhibits a graph-like structure: each state can be considered as a vertex, and transition probabilities can be used as weights on each arc. However, some states emit different characters, which complicates the matter. Unlike the Viterbi algorithm, where the output is the state sequence of a given word sequence, what we wish to compute here is the word sequence, given the HMM H. Thus, simply finding the state sequence with the highest total transition probability alone on H may not give us the optimal word sequence. To modify the model H to suit our needs, we perform the following transformation on H to obtain a directed graph G.

For every state q_i such that q_i emits at least one character (e.g. matching states and insertion states,) we split the vertex into two new vertices, q_i^{in} and q_i^{out}. Define the sets

$$\delta^-(q_i) = \{v \text{ a state} | t(v, q_i) > 0\}$$
$$\delta^+(q_i) = \{v \text{ a state} | t(q_i, v) > 0\}.$$

Then, join every state in $\delta^-(q_i)$ to q_i^{in} by an arc, and add the transition probability on its weight. Similarly, join q_i^{out} to every state in $\delta^+(q_i)$ by an arc, and add the transition probability on its weight. To connect the two split vertices, we join them via multiple edges from q_i^{in} to q_i^{out} and label each arc by the character that q_i emits. The weight on these arcs between q_i^{in} and q_i^{out} will be the emission probability of q_i emitting that character. Finally, if q_i contains a self-loop, add an arc (q_i^{out}, q_i^{in}) to allow the state to emit multiple characters. To see an example, see Figure 6.

Notice that the only change in the structure from H to G is the splitting of emitting states. Moreover, each emitting state now has one duplicate, with constant number of multiple edges between q_i^{in} and q_i^{out}. Thus, the size of the graph G is still linear in the size of H. Now, consider a path p from q_{begin} to q_{end}; p is an alternating path between transition arcs and emission arcs, possibly repeating on some transition arc, due to self-loops on deletion vertices. Let s be the sequence of characters labeled on the emission arcs on p. Then, the

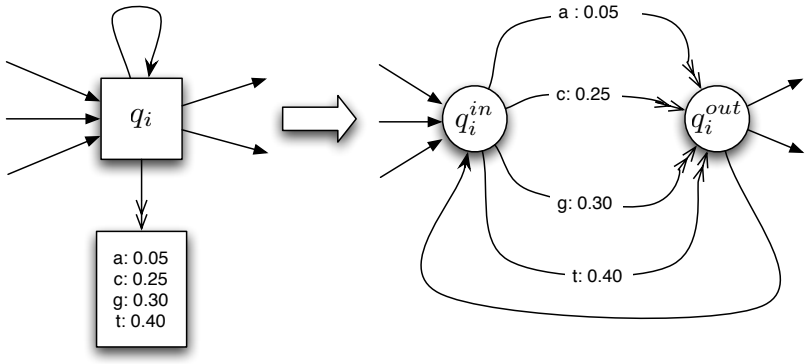

Figure 6: Transformation from an HMM to a digraph

product of all the weights on p is the probability score of s on H. Moreover, the path p also contains the state sequence that corresponds to the output of the Viterbi algorithm for s on H.

The directed graph G computed has the following properties: (1) There are no cycles except self-loops on delete vertices, and (2) all edge weights are positive if we convert the probability into negative log-space. Furthermore, the weights in log-space imply that the distance of a $q_{begin} - q_{end}$ path can be computed additively. Thus, the shortest path from q_{begin} to q_{end} would give us the word sequence that will score highest in H, as well as its state sequence. Notice that the self-loops on deletion vertices are never considered in shortest paths, since they merely increase the total length of a path. Therefore, it is safe to delete such self-loops on deletion vertices.

4.2 k-shortest paths algorithm

The k-shortest paths problem is a long-studied problem in combinatorial optimization [6, 8, 2, 1, 5]. We draw our attention to the latest result in the literature; it is suggested by [8] that the k-shortest paths can be computed in time and space $O(m + n \log n + k)$, and the same algorithm can be used to list all paths shorter than some given threshold length. The algorithm can be easily implemented using basic data structures, such as *heap-ordered trees*, and the complexity results are shown to be optimal.

4.3 Complexity issues with the choice of k

As before, we assume that the database of biological data is stored in a DAWG via the preprocessing step. Each query is then composed of the HMM H, and a threshold value ϕ which we use to decide whether or not to accept a state sequence. The transformation from H to G takes linear time, as the size of the graph G still stays linear in the size of H. Looking for the k-shortest paths from q_{begin} to q_{end} takes $O(m + n \log n + k)$ time, and finally, looking for the word sequences from the database takes $O(kw)$ time.

Notice that the running time of the algorithm is largely dominated by the parameter k. As a comparison, recall that the running time of the trie-based algorithm is mostly due to the number γ of distinct substrings of length w, which causes the trie to become large. Thus, we must be careful not to choose too large a value for k with respect to γ. Let θ denote the number of all permuted strings of length w, such that each string would score higher than the threshold value ϕ against the HMM. If $\theta \leq \gamma$, then simply setting $k = \theta$ guarantees that the k-shortest paths approach would run faster than the trie-based approach.

The strength of the algorithm discussed in this section, however, is that its running time is sensitive to the threshold ϕ. For the trie-based algorithm (or for the classical method using the sliding window), the algorithm must scan through the database regardless of the threshold value. In the case where ϕ is set very high (e.g. the user wants the k best matches in the database), the method proposed here would run much faster than the conventional database scanning approach, simply because the size of our HMM graph G is significantly smaller than the size of database. The space complexity of the algorithm is not an issue, either; after the computation of k shortest paths algorithm, only those k paths are stored in the heap tree in the form of shortest paths tree, which may be more desirable than having to store the database in memory.

5 Concluding Remarks & Future Work

In this paper, we gave two different approaches to solve the domain detection problem. The first algorithm introduced in Section 3 shows a major improvement in running time over the conventional database searching method, while preserving the same space complexity. We believe that this method can be easily implemented for many applications, and even further improvements can be made to deal with different values of w for each domain.

The second method based on the k-shortest paths algorithm, however, may result in exponential time, if the threshold value ϕ is chosen too generously. The fault in the exponential blow-up is because our directed acyclic graph, on which the shortest paths are found, is constructed from only the HMM. Thus, the substrings corresponding to the shortest paths may not necessarily exist in the database. Furthermore, the k-shortest paths found may not necessarily induce k *distinct* word sequences. This is because a path in the graph constructed is an alternating path between transition arcs and emission arcs. Therefore, simply selecting the first k shortest paths may contain two (or more) paths with the same word sequence but with different state sequences. A natural question to be studied, then, would be how to make improvements in running time of this algorithm. One way to attack this problem is to modify the transformation discussed in Section 4.1, so that the graph constructed incorporates the input sequences from the database as well as the HMM.

References

[1] J. A. Azevedo, M. E. O. S. Costa, J. J. E. R. S. Madeira., and E. Q. V. Martins. An algorithm for the ranking of shortest paths. *Eur. J. Operational Research*, 69:97–106, 1993.

[2] A. Bako and P. Kas. Determining the k-th shortest path by matrix method. *Szigma*, 10:61–66, 1977.

[3] L. E. Baum, T. Peterie, G. Souled, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *Ann. Math. Stat.*, 41(1):164–171, 1970.

[4] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel. Linear size finite automata for the set of all subwords of a word: an outline of results. In *Bull. European Assoc. Theoret. Comput. Sci.*, volume 21, pages 12–20, 1983.

[5] A. W. Brander and M. C. Sinclair. A comparative study of k-shortest path algorithms. In *Proc. 11th UK Performance Engineering Worksh. for Computer and Telecommunications Systems*, 1995.

[6] E. I. Chong, S. R. Maddila, and S. T. Morley. On finding single-source single-destination k shortest paths. In *Proc. 7th Int. Conf. Computing and Information*, July 1995.

[7] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, number 1261, pages 192–211. Springer-Verlag, Berlin, 1997.

[8] D. Eppstein. Finding the k shortest paths. *SICOMP*, 28(2):652–673, 1998.

[9] E. Fredkin. Trie memory. *Comm. ACM*, 3(9):490–499, September 1960.

[10] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Sunderland, MA, 1997.

[11] A. Krogh, M. Brown, I. S. Mian, K. Sjolander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology*, 235:1501–1531, 1994.

[12] F. Larsen, G. Gundersen, R. Lopez, and H. Prydz. CpG islands as gene markers in the human genome. *Genomics*, 13:1095–1107, 1992.

[13] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE*, 77(2):257–286, 1989.

[14] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inform. Theory*, 13(2):260–267, 1967.

Appendix: Exact Bounds on Tries

In this section, we look at exact bounds of a trie, given a collection of words. Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of strings, where each string $s_i \in \{a, c, g, t\}^w$. (Here, we look at the case for tries for DNA sequences; the case for protein sequences can be stated analogously.) Let T denote the trie constructed from strings in S. Since the number of edges in a trie is always exactly one less than the number of vertices, we only consider the number of vertices as the *size* of T. Before we state the bound, let us introduce the notion of *cuttlefish* tries.

Definition 2. *Let T be a trie rooted at $r \in V(T)$. Let d denote the size of the alphabet that the T contains(e.g. $d = 4$ for DNA sequences.), and n denote the number of leaves in T. If every vertex v with degree strictly greater than 2 is at most $\lfloor \log_d n \rfloor$ distance away from r, then T is a* cuttlefish trie*. We say a cuttlefish trie is* complete*, if it contains a complete d-ary tree from r to $\lfloor \log_d n \rfloor^{th}$ level.*

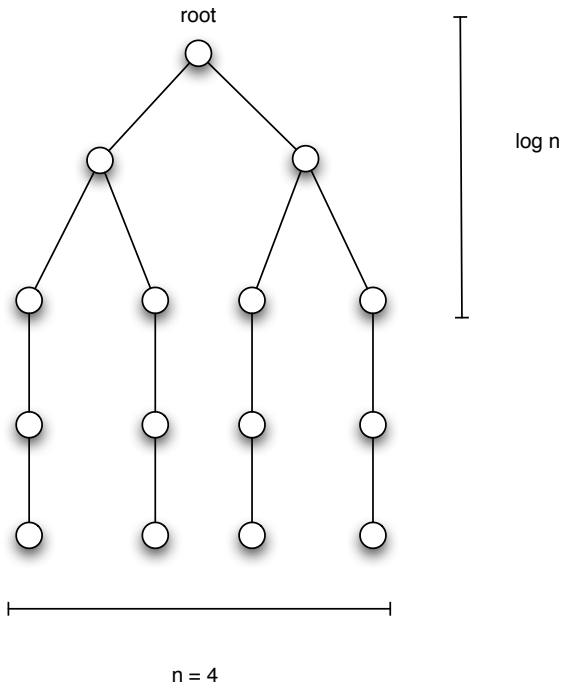

Figure 7: An example of a cuttlefish trie

In other words, if a trie contains no branching node below the $\lfloor \log_d n \rfloor^{th}$ level, we call it a cuttlefish trie. See Figure 7 for an example. The intuition is that, when the strings represented in a trie share very short prefixes, the size of the trie is large. Now, we are ready to state our claim.

Theorem 1. *Let T be a trie constructed from n distinct strings of length w over an alphabet of size d. Let C be the complete cuttlefish trie with same parameters as T. Then the number of vertices in T is bounded above by the number of vertices in C.*

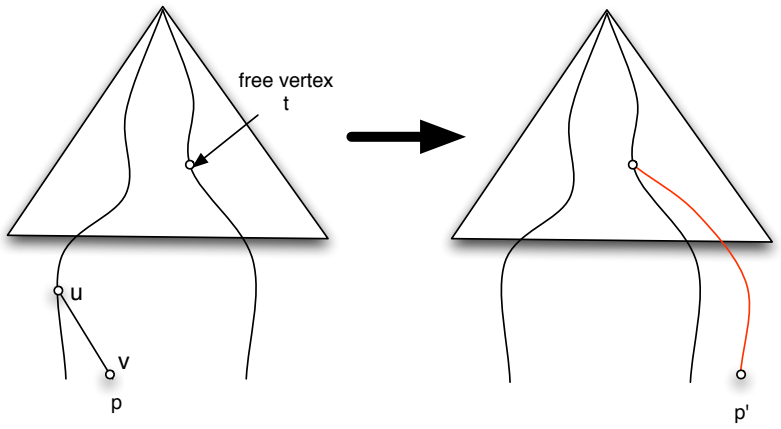

Figure 8: Proof step from T^i to T^{i+1}

Proof. To prove this claim, we consider different cases of the size of w. It is easy to see that w cannot be strictly less than $\lfloor \log_d n \rfloor$, since we assumed that all strings contained in T are distinct. Suppose $w = \lfloor \log_d n \rfloor$. Then, every leaf node in T is at most a distance of $\lfloor \log_d n \rfloor$ from the root node, and hence T is a complete cuttlefish trie.

Suppose $w > \lfloor \log_d n \rfloor$. Assume, for the sake of contradiction, that T is not a cuttlefish trie, and contains more vertices than C. Now, we show that T can be transformed into a cuttlefish graph by increasing the number of vertices in T. Let $T^0 = T$. We then create T^1 via the following transformation: First, take a root-to-leaf path p whose branching node is below the $\lfloor \log_d n \rfloor^{th}$ level from the root. Let v be the leaf node in p, and u be the lowest branching node above v. Delete the part of the path p from u to v, and instead, create a path p' branching off from a vertex t with fewer than d children, and located within distance $\lfloor \log_d n \rfloor$ away from the root. Note that the length of the newly added path p' is chosen appropriately so that the distance from root to leaf is w.

In other words, we are simply deleting the part of a path that violates the condition of a cuttlefish trie, and adding a new path at a free vertex in order to keep the number of leaves the same. To see the correctness of the argument, notice that the total number of leaf nodes is fixed. Since we assumed that T is not a cuttlefish, by definition, we can always find a violating path p. Furthermore, we are guaranteed to find a vertex with fewer than d children above the $\lfloor \log_d n \rfloor^{th}$ level, since otherwise we must have more than n leaves in T. Also, note that the newly added path is longer than the deleted $u - v$ path, since the branching node is now closer to the root.

Repeat the same process to go from T^1 to T^2, etc. Throughout this process, the number of vertices in T^i is increasing. Furthermore, the final residual tree T^R is a cuttlefish trie. Thus, we have the inequality

$$|V(T)| = |V(T^0)| < |V(T^1)| < |V(T^2)| < \cdots < |V(T^R)| = |V(C)|,$$

which is a contradiction.

□

Now, we can compute the exact upper bound on the size of tries.

Corollary 1. *Given a trie T that encodes n distinct strings of length w, where each string is over an alphabet of size d, the maximum number of vertices in T is exactly*

$$\frac{d^{\lfloor \log_d n \rfloor + 1} - 1}{d - 1} + n(w - \lfloor \log_d n \rfloor).$$

Proof. This follows immediately from Theorem 1. First, we look at the size of the complete trie of height $\lfloor \log_d n \rfloor$. This is a geometric series, evaluating to $\frac{d^{\lfloor \log_d n \rfloor + 1} - 1}{d-1}$. Then, if $w > \lfloor \log_d n \rfloor$, we must add the number of vertices in the simple paths below the complete trie, which evaluates to $n(w - \lfloor \log_d n \rfloor)$. □

Note that the second term in the equation evaluates to zero if $w = \lfloor \log_d n \rfloor$. Moreover, notice that the first term is in fact linear. If the strings are stored in a tabular form, the space required would be exactly nw. In the case of a trie, the space requirement is roughly at most $nw + n - n \log n$ in the worst case, exhibiting a considerable amount of improvement for large values of n.