From: ISMB-00 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

Accelerating Protein Classification Using Suffix Trees

Bogdan Dorohonceanu and C.G. Nevill-Manning

Computer Science Department
Rutgers University
Piscataway, NJ 07310
{dbogdan, nevill}@cs.rutgers.edu

Abstract

Position-specific scoring matrices have been used extensively to recognize highly conserved protein regions. We present a method for accelerating these searches using a suffix tree data structure computed from the sequences to be searched. Building on earlier work that allows evaluation of a scoring matrix to be stopped early, the suffix tree-based method excludes many protein segments from consideration at once by pruning entire subtrees. Although suffix trees are usually expensive in space, the fact that scoring matrix evaluation requires an in-order traversal allows nodes to be stored more compactly without loss of speed, and our implementation requires only 17 bytes of primary memory per input symbol. Searches are accelerated by up to a factor of ten.

Introduction

Position-specific scoring matrices (Gribskov *et al.* 1987, Henikoff *et al.* 1999) capture the characteristic distribution of amino acids in ungapped sequence motifs. They are more sensitive than regular-expression based methods such as PROSITE (Hoffman *et al.* 1999) and EMOTIF (Nevill-Manning *et al.*, 1998) but require less data for estimation than hidden Markov models (Durbin *et al.* 1998). Their disadvantage in comparison with PROSITE and EMOTIF is their speed: whereas a single mismatch in a regular expression allows the search to skip forward in the sequence, a scoring matrix always matches with some probability, and so skipping is more problematic. Wu *et al.* (2000) describe a technique for stopping early that relies on setting a score threshold ahead of time, and results in a significant speedup over a simple algorithm. We extend this technique by scanning a suffix tree constructed from the sequence rather than the sequence itself. The suffix tree groups all identical substrings into a single path, and if it is clear that the path cannot yield a score that meets the threshold, then all sequences in the subtree can be discarded.

The disadvantage of using a suffix tree is that it is expensive to store—a straightforward implementation requires about 37 bytes per input symbol (Salzberg, 1999). We reduce this to 17 bytes, as sketched in Manber and Myers (1993) by storing children in a linked lisy.

The next section describes scoring matrices and existing techniques for their acceleration. Next, we introduce suffix trees, and detail their application to accelerating the evaluation of scoring matrices. Finally we address efficiency concerns and present results.

Scoring matrices

A position-specific scoring matrix (PSSM) S represents a gapless local alignment of a sequence family. The alignment consists of several contiguous positions, each position represented by a column in the scoring matrix. Each column j consists of a vector $S_j(r)$, one score for each possible residue r. Table 1 gives an example of a scoring matrix for part of a zinc finger motif.

A scoring matrix can be used in sequence analysis by sliding the matrix along the sequence and computing segmental score. Each segmental score is the sum of the appropriate matrix entries, with each residue corresponding to a score in a column of the matrix. For a sequence of length L consisting of the residues $r_1, \ldots, r_L$, and a segment of width W beginning at position k ($1 \le k \le L - W + 1$) the segmental score is:

$$T = \sum_{j=1}^{W} S_j(r_{k+j-1})$$

Intuitively, a higher segmental score indicates a greater likelihood that the sequence matches the given scoring matrix. To give a probabilistic interpretation to the segmental scores of for given sequence and a set of scoring matrices we compute the relationship between segmental scores and probability, or p, values. These p values represent the probability of obtaining a given score in a random segment.

The intuition behind scoring matrices is as follows. Amino acids that are abundant in a position in the alignment get

	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y	max	thresholds
1	-19	**92**	-45	-49	-30	-36	-38	-12	-41	-21	-22	-40	-46	-44	-44	-30	-25	16	-35	-34	92	2
2	5	-17	17	**22**	-28	-15	-7	-23	-8	-27	-21	26	18	-7	-13	-9	9	-19	-33	-25	22	24
3	7	-8	-29	-28	2	-25	-10	25	-23	-4	-5	-25	-32	-26	-25	-18	13	22	-11	**36**	36	60
4	-29	**99**	-55	-61	-42	-45	-47	-31	-52	-34	-36	-49	-56	-55	-55	-38	-35	-29	-44	-46	99	159
5	-14	-22	14	**22**	-28	9	-8	-26	15	-27	-20	-7	-26	-3	31	-13	5	-23	-30	-24	22	181
6	-25	-34	-25	-16	-37	-30	-15	-36	45	-34	-26	-18	-35	-9	**49**	-25	-26	-33	-39	-31	49	230
7	7	-8	-25	-24	-19	-23	-22	4	-15	-10	-8	-19	-29	-21	11	-13	**31**	**31**	-31	-22	31	261
8	-34	-27	-44	-43	60	-41	-8	-16	-38	-14	-17	-39	-51	-40	-36	-39	-35	-21	-1	**56**	56	317
9	7	**40**	-16	-14	-9	-14	-6	-17	14	-20	-15	-10	-24	-11	12	15	9	-13	-16	20	40	357
10	-7	**43**	16	-7	-27	-15	-9	-24	-5	-26	-18	-6	-25	25	13	25	-8	-21	-30	-24	43	400

Table 1 A position-specific scoring matrix and the intermediate thresholds used for early stopping when the global threshold is 400

high scores, and those that are rare get low scores. Scores are log-odds scores: $\log(f_{a,o}/f_{a,e})$, where $f_{a,o}$ is the proportion that amino acid a is observed in the position, and $f_{a,e}$ is the proportion of amino acid a in nature (its expected proportion). Multiplying odds scores for amino acids in a particular segment of the protein of length m yields the likelihood that this segment belongs to the same distribution as the multiple alignment. Adding log-odds scores is equivalent to multiplying odds scores, but it is faster and is not susceptible to underflow or overflow. For a more comprehensive treatment of PSSMs, see matrices Gribskov *et al.* (1987) and Henikoff *et al.* (1999.)

Many implementations of PSSMs for protein function prediction, such as BLIMPS apply the scoring matrix to each segment of each protein in the input, sort the segments by their scores, and present the top few hits. Wu *et al.* propose setting a threshold for the score *a priori*, and storing and presenting only those hits that score above the threshold. This saves storage space for lower-scoring segments, and eliminates sorting. They present a method for computing an appropriate score for a particular specificity, i.e. the score that a segment of random amino acids would be likely to exceed with a probability p. The probability p is chosen according to the size of the database in order to minimize false positives but maintain sensitivity.

The next optimization that Wu *et al.* introduce is stopping a score computation early using a technique known as *lookahead scoring*. Consider the matrix in Table 1 with a threshold of 400. The last column shows that a sequence must score at least 400 after the entry from the last row is added to be considered a match. The maximum score that can be achieved in the last position is 43, so the score by the end of the 9th position must be at least 357 to be able to make 400 by the 10th position. Similarly, the highest score in the second-last position is 40, so the score must be at least 317 in the eight position. These intermediate thresholds can be calculated for all other positions in a similar way. They are calculated ahead of time, and during the evaluation of a particular protein segment, the computation can be stopped early if the score fails to reach the appropriate intermediate threshold. This is the scoring matrix analog to skipping forward with a regular expression: when it is clear that the overall threshold cannot be reached for this segment, the rest of the segment can be skipped. In the rather extreme case of Table 1, the segment can be skipped if it does not begin with V or C: they are the only residues that score at least 2 in the first position. Note that stopping is more likely to occur as the global threshold is increased, or equivalently, as the overall probability of match is decreased. Wu *et al.* report that this optimization doubles the speed of scoring for $p = 10^{-20}$ and increases speed by a factor of five for $p = 10^{-40}$.

Suffix trees

A suffix tree is a compacted trie of suffixes in a string, i.e. for every suffix of a string, there is a path in the corresponding suffix tree from root to leaf labeled with that string. Figure 1a shows a tree of suffixes for the string *abab$*. This tree is compacted in the following way. Wherever two edges beginning at the same node share a prefix, a new edge and node are created, and the edge is labeled with the shared prefix. The two original edges no longer share a prefix. This operation is performed until no two edges from the same node share a prefix. This compacted tree is a suffix tree, and is shown in Figure 1b. Suffix trees allow operations such as finding whether a substring occurs in a string to be performed efficiently. This query can be performed in time proportional to the length of the substring by following the path from the root of the suffix tree. If the path exists, the substring exists in the string.

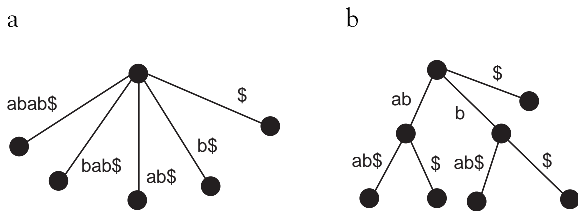

Figure 1 (a) uncompacted and (b) compacted suffix trees

The cost of creating the suffix tree must be added to any operations. A suffix tree can be inferred from a string of length n in $O(n)$ time (McCreight 1976, Ukkonen 1995). In our experiments, the cost of creating the suffix tree is negligible compared to the cost of evaluating the scoring matrices. For a tutorial on suffix tree construction, see Nelson (1996) or Gusfield (1997)

To find a high-scoring segment in a set of protein sequences, we first form a suffix array from the sequences, then do a depth-first traversal of the tree, calculating the scores for the edge labels. Whenever the score at some node in the tree reaches the threshold, all the substrings represented by the leaves below that node must also reach the threshold, and can be reported. More significantly, whenever the score at some node falls below the intermediate threshold, no substrings corresponding to leaves in the subtree can reach the threshold, and the entire substring can be ignored. This is the key to the acceleration: many substrings can be discounted based on looking at a single path. Figure 2 shows the two cases: a subtree of matches and a subtree of non-matches.

We have to deal specially with the ends of the sequences, and have investigated two approaches. The first approach forms a suffix tree from the set of sequences, taking special

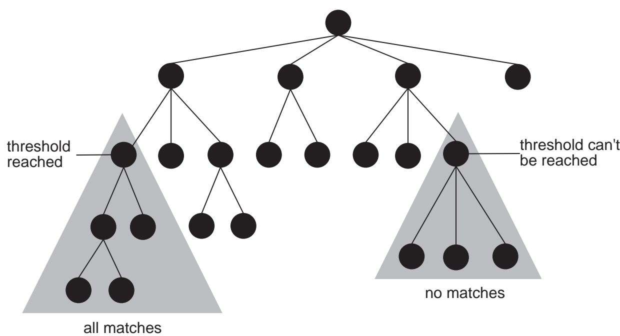

Figure 2 Two accelerations that suffix trees provide: a subtree where every leaf points to a matching segment, and a subtree where every segment can be ignored

note of the ends of the sequences, and adding extra leaves to the tree to specify where sequences end. The second approach concatenates all the sequences, placing a special end of sequence symbol (EOS) between them. This tree includes suffixes that cross sequences, which are suffixes are useless, but can be easily ignored. In order not to introduce new special cases in the tree traversal algorithm, we add EOS to the scoring matrix and give it a large negative score in all positions. If the threshold is reached in the suffix tree before encountering the EOS symbol, then care must be taken in calculating the final score for the resulting matches: when EOS is encountered, the summation should finish. If, on the other hand EOS is encountered before the threshold is reached, the large negative score ensures that the sequence boundary is not crossed, and that the subtree is pruned.

Compact suffix trees

A significant problem with suffix trees is their size in primary memory. For an alphabet of size $|\Sigma|$, each node in a suffix tree has at most $|\Sigma|$ nodes. For concreteness, we will assume that the alphabet consists of the natural amino acids, so that $|\Sigma| = 20$. A straightforward implementation would use a 20 element array of pointers for the children, a pointer to the suffix corresponding to this node, and a suffix pointer, which is used in construction. In our experiments, a suffix tree for the entire SWISSPROT database (20 million amino acids) has about 30 million nodes, giving 30,000,000 nodes × 22 pointers/node × 4 bytes/pointer = 2.6 Gb. Because not all nodes have outdgree 20, we can save memory by using a linked list to store the children. In this case, each node points to a linked list of its children, so it has a child pointer and a sibling pointer, as well as the two other pointers, for 4 integers per node. This restricts us to sequential access to children, rather than the random access that the fixed array of children provides, but because we are interested in a depth-first traversal of the entire tree, there is no performance penalty when evaluating a scoring matrix. Figure 4 shows the transformation between a regular suffix tree and one that uses a linked list to access children. Random access is important for suffix-tree construction: we will deal with this problem next. This gives a total size of 30,000,000 nodes × 4 pointers/node × 4 bytes/pointer = 480 Mb. However, this ignores the overhead of allocating memory in 16 byte chunks, which can be considerable. For this reason, we do our own memory allocation from an array.

At this point, we consider construction of the suffix tree. We begin by constructing a suffix array, which is a sorted array of pointers to all suffixes of a string. It is very simple to construct: the elements of the array are initialized to

1	→S
2	→NAS
3	→NANAS
4	→AS
5	→ANAS
6	→ANANAS

Figure 3 The suffix array for the string ANANAS

point to every suffix, then the sorting scheme compares the suffixes referenced by the pointers to sort the pointers. The suffix tree for the string *ANANAS* is shown in Figure 3.

Because the tree is constructed by inserting suffixes sorted in reverse lexicographic order, the node insertion will always take place either as a child of the tree root, or as a node with leaf or leaf on the last visited path (during the previous insertion). On the current insertion path, all the nodes (from the root to the insertion point) will have their *index* field updated so that they point in the same suffix (i.e. the one that is inserted).

This technique allows each node to represent a substring from a suffix in the multi-sequence, which starts at index index(node) and ends right before the index index(child(node)). The same technique ensures that during tree construction there are only node creations (insertions), and no node deletions. Therefore the node fields can be compactly represented in arrays. The suffix insertion procedure is given in Table 2. Figure 5 gives a step-by-step diagram of the creation of a suffix tree from

let $l \leftarrow$ length of the longest common prefix between the current suffix and previous suffix in suffix array.

if $l = 0$ **then**

  add a new child to the root

else

  let $s \leftarrow$ position in the string of the current suffix

  for each node n on the path from the root to the leftmost leaf,

    let index(n) $\leftarrow s$, so that it points in the current inserted suffix

    let $s \leftarrow s$ – length(n).

    let $l \leftarrow l$ – length(n).

    if $l = 0$ **then** add a leaf to n

    else if $l <$ length(n) **then** split n

      After splitting, n will have two children: child(n) is a leaf or a node with a leaf and continues the new inserted path, sibling(child(n)), which has all of n's children

Table 2 The pseudo-code for inserting a suffix in the suffix-tree

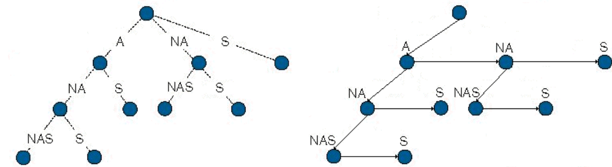

Figure 4 Representing children in a suffix tree using a linked list

the reverse suffix array of Figure 3.

The next space-saving modification, which we have not yet implemented, removes the sibling pointers by using an array instead of a linked list. This requires all of the siblings to be contiguous, which is not possible when the tree is created in a depth-first manner. So we make an extra pass to traverse the finished tree breadth first, so that siblings are contiguous, and do not require sibling pointers. When traversing the tree, it is necessary to know how many siblings a node has, i.e. the size of the sibling array. In a depth-first traversal, we can easily follow the child pointer for a node's sibling. Because the sibling's children are stored immediately after the current node's children, the first child of the sibling forms the boundary for the child array of this node. When the node is at the very right side of a larger subtree, there are no siblings to the right. In this case, the boundary is given by the right sibling of the most recent ancestor that has a right sibling.

Kurtz (1999) describes other space-saving techniques for suffix trees that allow linear-time construction. However, we chose not to use his compressed representation because of concerns about speed.

Results

Table 3 shows the speedup gained from the suffix tree over lookahead scoring as the threshold increases. The suffix tree provides a speedup factor of between 2.0 and 5.5 over lookahead scoring, and between 2.7 and 10.6 over the simple scheme that always evaluates the whole segment. Because the system was implemented in Java, the times

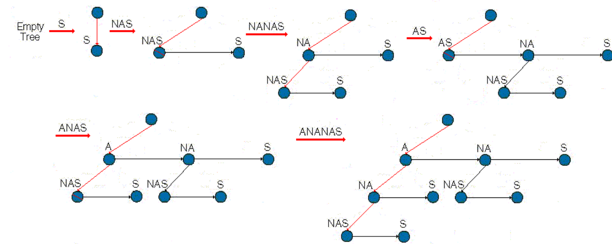

Figure 5 Creating a suffix tree from a reverse-sorted suffix array

Scheme	Threshold (p) 10^{-10}	10^{-20}	10^{-30}
Simple	114	114	114
Lookahead scoring	158	189	222
Suffix tree	308	568	1209
Improvement over lookahead	2.0	3.0	5.5
Improvement over simple	2.7	5.0	10.6

Table 3 Residues/second processed for 4034 scoring matrices

cannot be compared directly with Wu *et al.* (2000), where the system was implemented in C. However, the relative speedup should be similar. The total time to apply 4034 scoring matrices to 20 Mb of sequences was 20 hours, of which 4 minutes (0.3%) was required to build the suffix tree. For smaller inputs, such as the 4 Mb of sequences in the first 1000 proteins of SWISSPROT, building the tree only requires 0.2% of the total time. It is true, then that tree construction time is negligible, and we should trade off time for space where possible in constructing the tree.

The drawback of this approach, as we have mentioned, is the space required by the suffix tree. Table 4 summarizes statistics of suffix trees for various exerpts from SWISSPROT, ranging in size from 4MB (100 sequences) to the full 21MB (59021 sequences). The time to build the tree ranged from 3 seconds to 5 minutes, of which most was consumed by the array sorting. The average length of the common prefixes between adjacent entries of the array was about 20 residues. The largest tree had 30 million nodes, and occupied 345 MB, at a rate of 12 bytes/node. The space corresponds to just over 17 bytes per input symbol, and decreases slightly as the sequence grows.

Conclusions

Suffix trees can be used to accelerate scoring matrix calculations by a factor of 4.7 over a straightforward method, and 2.8 times faster than previous work on lookahead scoring. Although suffix trees are generally expensive in space, we have found ways to minimize this space to the point where these techniques would be practical on a small server machine. This efficient annotation technique is important both in the context of high-throughput sequencing centers and as a part of a system that serves multiple users .

This technique also demonstrates the versatility of the suffix tree data structure, even in situations where a probabilistic approach to sequence matching is involved. In the future, we plan to improve absolute speed by rewriting the system in C++, reduce the space requirements of the suffix array, and extend the technique to other probabilistic applications.

Acknowledgements

We would like to thank Professor Martin Farach-Colton for many enlightening discussions on suffix-tree creation and use.

References

Bieganski, P.; Riedl, J.; and Carlis, J. 1994. Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation. University of Minnesota. http://www.cbc.umn.edu/VirtLibrary/Bieganski/htree/htree-paper.ss.html.

Delcher, A. L.; Kasif, S.; Fleischmann, R. D.; Peterson, J.; White, O.; and Salzberg S. L. 1999. Alignment of Whole Genomes. Nucleic Acids Research 27(11):2369–2376.

Durbin, R. Eddy, S. Krogh, A. and Mitchison, G. (1998) *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.*

Gribskov, M., McLachlan, A. D., and Eisenberg, D. (1987) Profile analysis: Detection of distantly related proteins. Proceedings of the National Academy of Sciences USA, 84:4355–4358.

Gusfield, D. 1997. Algorithms on Strings, Trees, and Sequences-Computer Science and Computational Biology. Cambridge University Press.

sequences	residues	building time (sec) array	tree	total	average prefix	nodes	space (MB)	bytes per residue
1000	402468	3	1	4	24	582183	7	17.4
10000	3778757	38	10	48	23	5494537	63	17.4
30000	11044460	118	30	148	19	15910398	182	17.3
59021	21210388	257	64	321	19	30143093	345	17.1

Table 4 Suffix tree statistics for excerpts from the SWISSPROT database

Henikoff, J. G.; Henikoff, S.; and Pietrokovski, S. 1999. New Features of the Blocks Database Servers. Nucleic Acids Research 27(1):226–228.

Hofmann K., Bucher P., Falquet L., Bairoch A. (1999) The PROSITE database, its status in 1999 *Nucleic Acids Res*. 27:215–219.

Kurtz, S (1999) “Reducing the Space Requirement of Suffix Trees,” *Software–Practice and Experience*, 29(13), 1149–1171.

Manber., U; and Myers, G. 1993. Suffix Arrays: A New Method for On-line String Searches. SIAM Journal of Computing 22(5):935–948.

McCreight, E. 1976. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM* 23:262–272.

Nelson, M. R. 1996. Fast String Searching with Suffix Trees. *Dr. Dobb's Journal*, *Algorithm Alley*, August.

Nevill-Manning, C.G. Wu, T.D. & Brutlag, D.L. (1998) Highly Specific Protein Sequence Motifs for Genome Analysis, *Proc. Natl. Acad. Sci*. USA, 95(11), 5865-5871.

Ukkonen, E. 1995. On-line Construction of Suffix Trees. *Algorithmica*.

Wu, T. D.; Nevill-Manning, C. G.; and Brutlag, D. L. 2000. Fast and Accurate Sequence Analysis Using Scoring Matrices. Bioinformatics, 16(1) 1–12.