

Manuscript Title: Faster Protein Classification Using Suffix Trees

Running Head: Protein Classification Using Suffix Trees

Authors:

Bogdan Dorohonceanu*
CAIP Center, 96 Frelinghuysen Rd., CORE 703, Piscataway, NJ 08854-8088.
dbogdan@cs.rutgers.edu

Craig G. Nevill-Manning
Computer Science Department, HILL Center, Rutgers University, Piscataway, NJ 08854.
neville@cs.rutgers.edu

Keywords: suffix tree, motif identification, sequence classification, scoring matrix, lookahead scoring.

* To whom correspondence should be addressed

Faster Protein Classification Using Suffix Trees

Bogdan Dorohonceanu and Craig G. Nevill-Manning

Computer Science Department, Rutgers - The State University of New Jersey, Piscataway, NJ 08854

Abstract

Motivation: 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.

Methods: 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 compactly in memory and on the disk without significant loss of speed.

Results: Our implementation requires as little as 12 bytes of disk storage per input symbol. Searches are accelerated by a factor of up to eleven under typical conditions.

Availability: The package source code is available at: <http://sequence.rutgers.edu/sat>.

Contact: dbogdan@cs.rutgers.edu

Abbreviations: EOS, end of sequence symbol; K, kilo; M, mega; MB, megabyte; PSSM position-specific scoring matrix.

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 (Hofmann *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 (Delcher, 1999). We reduce this to between 12 and 14 bytes per input symbol in secondary memory by storing the tree nodes in compressed chunks on the disk and bringing into primary memory only the necessary chunks of nodes. The following section describes scoring matrices and existing techniques for their acceleration. Next, we introduce suffix trees, and detail our implementation and application of them to accelerating the evaluation of scoring matrices or sequence classification using 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, \dots, r_L , and a segment of width W beginning at position k ($1 \leq k \leq L - W + 1$) the segmental score is:

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

	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.

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 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 a 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, in order 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 8th 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 1(a) shows a trie of suffixes for the string ANANAS. This trie is compacted in the following way. Wherever two edges beginning at the same node

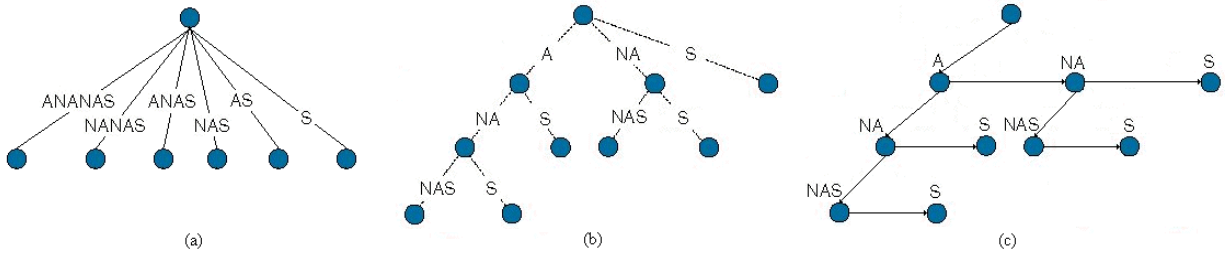


Figure 1. (a) The suffix trie for string ANANAS. (b) The trie compacted in a suffix tree. (c) The suffix tree with the children nodes represented using linked lists.

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 trie is the suffix tree shown in Figure 1(b). 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.

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 note of the ends of the sequences, and adding extra leaves to the tree to specify where sequences end (Dorohonceanu and Nevill-Manning, 2000a and 2000b). The second approach concatenates all the sequences, placing a special end of sequence symbol (EOS) between them. This tree includes useless suffixes that cross sequences, 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. The second approach requires less storage space for the suffix trees and produces faster execution times when classifying protein sequences or evaluating scoring matrices.

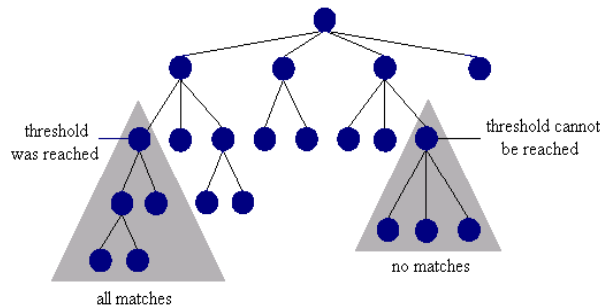


Figure 2. Two accelerations that suffix trees provide: a subtree where every leaf points to a matching segment, and a subtree where every segment can be ignored.

Algorithm and Implementation

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 an example, 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 $30,000,000 \text{ nodes} \times 22 \text{ pointers/node} \times 4 \text{ bytes/pointer} = 2.6 \text{ GB}$. Because not all nodes have outdegree 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 1(c) 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 $30,000,000 \text{ nodes} \times 4 \text{ pointers/node} \times 4 \text{ bytes/pointer} = 480 \text{ MB}$. However, this computation 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 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 the index $index(node)$ and ends right before the index $index(child(node))$. The same technique ensures that during tree construction there are only node

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

Figure 3. The reverse-sorted suffix array for the string ANANAS.

creations (insertions), and no node deletions. Therefore, the node fields can be compactly represented in arrays. The suffix insertion procedure is given in Figure 4. Figure 5 gives a step-by-step diagram of the creation of a suffix tree from the reverse suffix array of Figure 3.

Memory Management

The suffix tree requires a significant amount of memory for storage, e.g., a suffix tree for about 60 K protein sequences (about 20 MB of input symbols) needs a little less than 400 MB of storage. This is a large amount of primary memory to devote to searching, and we will eventually want to deal with much larger databases, so it is important to be economical. One solution would be to compress the suffix tree, but that approach makes the tree traversal slower and the classification using the tree several times slower than using a multi-sequence.

Taking into account that, during motif searches, the suffix tree is traversed in-order with jumps (cuts) in traversal, a better approach is to store the suffix tree on the disk in depth-first order and to fetch into the memory only the needed sequential chunks of nodes.

Building such a (generalized) suffix tree from a multi-sequence requires three steps:

1. Read the multi-sequence and create the suffix array consisting in all the suffixes sorted in reverse order.
2. Take each suffix from the suffix array and build the suffix tree by inserting the suffixes sorted in reverse order as described in Figure 4. Instead of storing all

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

Figure 4. The pseudo-code for inserting a suffix from a reverse-sorted suffix array into a suffix-tree.

the nodes in memory, we save them on secondary storage as soon as they no longer belong to the current insertion path, due to insertion of new nodes (Figure 5). When we save a subpath on the disk we save the nodes so that the node saved first is the rightmost node in the subpath and the last saved node is the nearest to the root. This technique generates on the disk the *post-order* sequence of nodes.

- Reverse the node sequence on the disk so that we obtain the sequence of the depth-first traversal of the suffix tree. When we saved the nodes on the disk in step 2 we did not know the final number of nodes in the tree (because we were constructing the tree). When we reverse the node sequence to obtain the sequence of the depth-first traversal, we have to replace the values for *sibling* and *child* fields of the nodes with the differences between the number of nodes (now being known) and their current values (the last step in Figure 5), i.e. for each node: $child(node) = n - child(node)$ and $sibling(node) = n - sibling(node)$, where $n = \#nodes - 1$.

This algorithm only needs storage in primary memory for the multi-sequence, the suffix-order vector from the

suffix array (temporarily), and for the nodes from the current insertion path (which will be never longer than the tree depth, occupying a few thousand bytes). The rest of the tree nodes are on the disk.

When the suffix tree construction is finished, its nodes are saved in depth-first order on the disk. We can easily implement or re-write the traversal-based algorithms (like printing the tree, finding the depth of the tree, classifying sequences, or evaluating scoring matrices) such that the node structures are accessed in depth-first order only. Consequently, the chunks of nodes are accessed in sequential order, as in the following pseudo-code for first-depth traversal:

```

depth-first(int node) {
1.   int child = child(node);
2.   while (child != NULL) {
3.     int sibling = sibling(child);
4.     depth-first(child);
5.     child = sibling;
   }
}

```

For a given *node*, for each call of *child(node)* or *sibling(node)* the chunk containing the *node* must be fetched into the primary memory. In line (1.) we visit the *node*, fetching into primary memory the chunk containing that *node*. In line (3.) the chunk containing

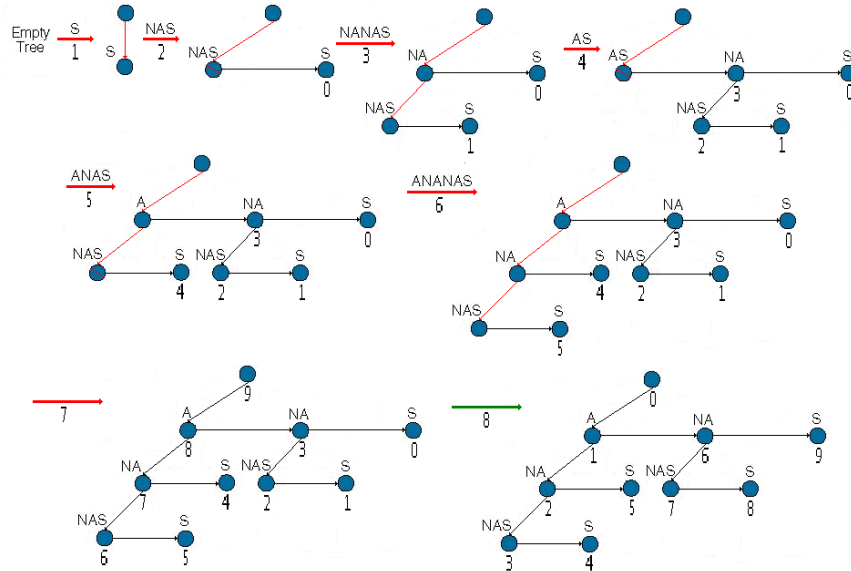


Figure 5. The order the nodes are written in on the disk when the suffix tree for string ANANAS is built and the reversal of the tree nodes on the disk (the last step). Step 1: When suffix NAS is inserted, node S becomes a sibling of node NAS and it is saved on position 0 in the node file on the disk... Step 3: When suffix AS is inserted, a part of the current insertion path becomes a subtree rooted in node NA, which is a sibling of node AS. The subpath is saved on the disk in the order NAS, NA, such that the farthest node from the root is saved first... Step 7: Finally, after all suffixes have been inserted, the main insertion path remains is saved such that the farthest nodes from the root are saved first. Step 8: During tree creation, the nodes were saved in post-order: 0 (S), 1 (S), 2 (NAS), ... 8 (A), 9 (ROOT). The tree has 10 nodes. After node reversal and updates of node fields (i.e., for each node: $child(node) = 9 - child(node)$ and $sibling(node) = 9 - sibling(node)$) the tree nodes are in depth-first traversal order: 0 (ROOT), 1 (A), ... 8 (S), 9 (S).

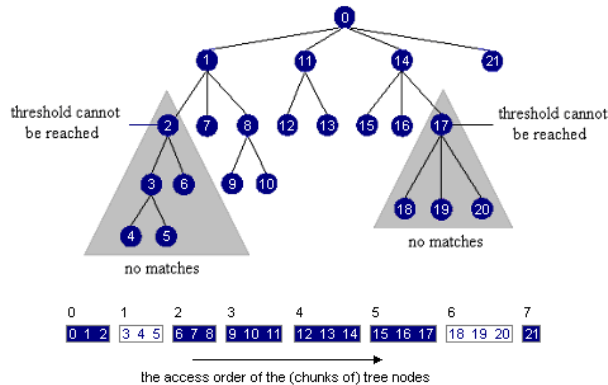


Figure 6. An example of access order of the (chunks of) tree nodes: The suffix tree has 22 nodes (0...21) stored in the secondary memory in 8 chunks (0...7). During an in-depth tree traversal for the evaluation of a scoring matrix it was determined that the threshold cannot be reached at nodes 2 and 17 (two traversal cuts). Therefore, nodes 3, 4, 5, 6, 18, 19, and 20 were skipped and chunks 1 and 6 were not fetched into the primary memory.

the *child* of the *node* must be fetched into the primary memory and we save the *sibling* of the child in a temporary variable, before calling the function recursively in line (4.). Line (5.) does not fetch back into the primary memory the chunk for the *child*, because the *sibling* value can be retrieved from the temporary variable.

This way, the algorithms that are based on in-order tree traversal and are implemented using our suffix-tree data structure have a minimal number of disk accesses because they read (fetch from secondary memory into primary memory) the (chunks of) nodes in sequential order only. An example is given in Figure 6.

This works well when we search for motifs in a suffix-tree using one scoring matrix. For multiple scoring matrices, for each scoring matrix we would have to read from the beginning the file storing the tree nodes again, unless we use threads of execution. For each scoring matrix we launch a thread. All threads run in parallel and read from the same chunk of nodes. When the threads finish with a chunk and need the next chunk, they will synchronize first waiting for all of them to finish with the nodes in the current chunk. Then the next necessary node chunk is fetched into the memory and all the threads continue their work in parallel. This way the file containing the nodes is still read only once and in sequential order, regardless of the number of scoring matrices involved in classification. Due to traversal cuts, some threads may skip more node chunks than the others may. In those cases, they will wait until the chunks they need are fetched into memory.

During motif searches, because the threads associated with scoring matrices run in the same process the operating system overhead in execution time due to thread context switching is negligible (milliseconds). The overhead that counts is the one associated with disk accesses and is proportional to the number of nodes. Indeed, we need $m = \#nodes / node_chunk_size$ accesses for each full traversal and each access takes about $t = k * node_chunk_size$ time, where k is constant. Therefore, the overhead due to disk accesses is $m * t = k * \#nodes = O(\#nodes)$.

Cuts in searches in the tree may skip (forward) over entire chunks of nodes, but the accessed chunks remain ordered sequentially by their starting offsets in the node file, i.e. the file pointer is always only increased from the beginning (zero) towards the end of the file (thus never decreased). If we agree to pay a small penalty in the classification time, we can decrease storage space on the disk for the tree nodes from 18 to 12 bytes per input symbol, using the ZIP algorithm. In order to access separately each chunk of nodes, each chunk is stored as a separate entry in the ZIP archive.

Writing the nodes on the disk can be accelerated using a buffered data output stream with a buffer of size between 10 and 20 MB stored in primary memory. The performance for reading the nodes from the disk during node reversal phase and, later, during tree traversal for motif searches in the case of uncompressed nodes, does not improve if we use a buffered input data stream, because we need to read chunks of nodes of a fixed size. Operating system buffers appear to be sufficient for that.

This implementation has better performance than the multi-sequence and worse or almost the same performance as the suffix tree stored entirely into the primary memory. The time performance of tree traversal varies with the overhead introduced by the number of disk accesses (less or equal to the number of node chunks, because some chunks may be skipped due to traversal cuts as we have shown).

Experimental Results

We measured the speedup from the suffix tree (with nodes split in chunks of 20 MB) over lookahead scoring as the threshold p increases, for 10 MB, 20 MB, 30 MB, 40 MB, 50 MB, 60 MB, and 170 MB (542,868 peptide sequences from GENPEPT database) of input symbols.

The compact suffix tree provided a speedup of between 1.9 and 4.2 over the *lookahead scoring* scheme and between 2.2 and 7.0 over the *simple scoring* scheme that always evaluates the whole segment. Because the system was implemented in Java, the times cannot be

Scheme	Threshold (p)		
	10^{-10}	10^{-20}	10^{-30}
Simple scoring	173	173	173
Lookahead scoring	199	257	291
Compact suffix tree	386	661	1,216
- improvement over simple	2.2	3.8	7.0
- improvement over lookahead	1.9	2.6	4.2

Table 2. Residues per second processed for 4,034 scoring matrices and 170 M input symbols.

compared directly with Wu *et al.* (2000), where the system was implemented in C, but the relative speedup should be similar.

Table 2 shows the speedup gained from the suffix tree over lookahead scoring as the threshold increases for 170 MB input symbols. The total times to apply 4,034 scoring matrices to 170 MB of input symbols for $p=10^{-10}$ were: 272.85 hours for the simple scheme, 237.22 hours for lookahead scheme, and 124.9 hours for compact suffix tree (of which 2.63 hours (2.1%) were used to build the tree). For smaller inputs, such as the 10 to 60 MB of input symbols in the first 185,821 peptide sequences of GENPEPT, building the compact suffix tree required between 0.9 and 1.5% of the total time. It is true, then that the compact tree construction time is negligible, and we should trade off time for space where possible in constructing the tree.

Table 3 summarizes the statistics for storage requirements for various excerpts from GENPEPT, ranging in size from 10 MB (33,215 sequences) to 170 MB (542,868 sequences, almost the entire current GENPEPT). The total time to build the compact suffix trees ranged from 5.7 minutes to 2.63 hours, of which an important part (from 3.61 minutes to 0.8 hours respectively) was consumed by the array sorting. The average length of the common prefixes between two adjacent entries of the array was between 135 and 189 symbols. The time to build the compact tree from the array depends on the fixed size of the node chunks.

The largest tree had 276,246,780 nodes, and occupied 2,292.15 MB on disk, at a fixed rate of 8.7 bytes/node.

input symbols	sequences	building time [sec]			average prefix	nodes	compacted nodes [MB]	bytes per input symbol	bytes per node	uncompressed nodes [MB]
		array	tree	total						
10,000,093	33,215	129	217	346	204.47	15,286,964	122.18	12.81	8.38	174.95
20,000,078	67,360	226	398	624	135.22	30,570,995	249.09	13.06	8.54	349.86
30,000,485	99,543	363	644	1,007	151.98	45,887,763	378.22	13.22	8.64	525.14
40,000,025	133,880	518	884	1,402	160.79	62,321,494	514.42	13.49	8.66	713.21
50,001,533	161,472	636	1,170	1,806	145.41	77,241,470	640.63	13.43	8.70	883.96
60,000,199	185,821	805	1,881	2,686	147.93	92,727,820	772.67	13.50	8.74	1,061.19
170,000,195	542,868	2,885	6,581	9,466	188.71	276,246,780	2,292.15	14.14	8.70	3,161.39

Table 3. Compact suffix tree statistics for excerpts from the GENPEPT peptide database.

The space corresponds to about 14 bytes per input symbol. The peak primary memory usage when creating that tree was 810.62 MB (of which 648.5 MB were needed to store the *position* array for the 170 M suffixes during the sorting of the suffix array), but this need only be done once. The peak primary memory usage when using that tree was under 183 MB. Conceivably, the tree could be created on a large memory machine but used on a more modest one. With a small decrease in performance, the memory requirement for the suffix array can be decreased by swapping it on disk and fetching into memory only chunks of it during the sorting process.

Figure 7 shows the space requirement for the data structures presented in this paper. The *memory* requirements are also the ones for traversals during sequence classification.

We also measured the speedup with the increase of threshold p , gained by the uncompressed suffix tree (stored entirely into memory at a fixed rate of 12 bytes per node or 12 bytes per input symbol) and the compact suffix tree. We tested with 4,034 scoring matrices and 10 MB, 20 MB, 30 MB, and 40 MB (133,880 peptide sequences from GENPEPT database) of input symbols. (Table 4 shows the results for 40 MB input symbols). While the compact suffix tree provided a speedup of between 1.6 and 3.2 over the *lookahead scoring* and between 1.9 and 5.5 over the *simple scoring*, the uncompressed suffix tree performed even better. It provided a speedup of between 2.5 and 6.5 over the *lookahead scoring* and between 2.9 and 11.2 over the *simple scoring*. The results suggested that it is advantageous to use a suffix tree until the primary memory storage limit is reached and a compact suffix tree after that.

The code was implemented in Java (JDK 2) and the experiments were run on a DELL Pentium machine with 700 MHz processor, 256 KB cache memory, 1 GB RAM, 100 MHz bus, 20 GB hard-disk capacity, and Windows NT 4.0 as operating system.

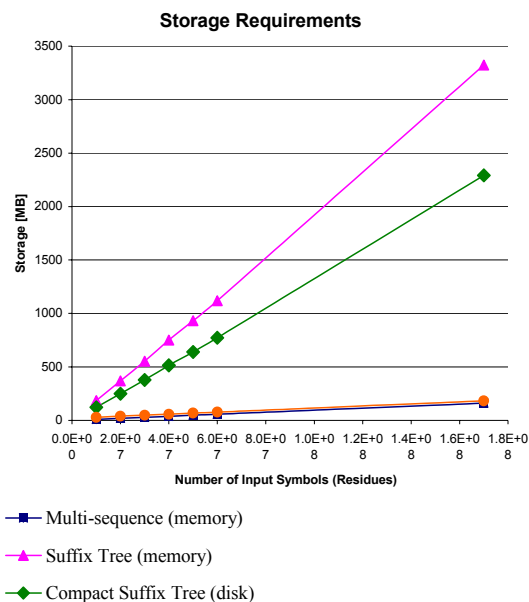


Figure 7. Storage requirements for multi-sequence, suffix tree stored entirely in primary memory, and compact suffix tree stored in secondary memory with a node buffer (chunk) size of 20 MB of primary memory.

Conclusions

Suffix trees can be used to accelerate scoring matrix calculations by a factor of between 2 and 11 over a straightforward method and between 2 and 6 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 extend the technique to other probabilistic applications.

Acknowledgements

We would like to thank Professor Martin Farach-Colton (Rutgers University) for many enlightening discussions on suffix-tree creation and use, Professor Alistair Moffat (Melbourne University, Australia) for suggestions on storing suffix trees in secondary memory, and Professor Ivan Marsic (CAIP Center, Rutgers University) for hardware resources for experiments.

Scheme	Threshold (p)		
	10^{-10}	10^{-20}	10^{-30}
Simple scoring	182	182	182
Lookahead scoring	211	276	316
Compact suffix tree	339	575	1,010
- improvement over simple	1.9	3.2	5.5
- improvement over lookahead	1.6	2.1	3.2
Suffix tree	527	961	2,043
- improvement over simple	2.9	5.2	11.2
- improvement over lookahead	2.5	3.5	6.5

Table 4. Residues per second processed for 4,034 scoring matrices and 40 M input symbols, using a compact suffix tree and an uncompressed suffix tree (stored entirely in the primary memory).

References

- 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.
- Dorohonceanu, B.; and Nevill-Manning, C.G. (2000a). Accelerating Protein Classification Using Suffix Trees, *Proceedings of the International Conference on Intelligent Systems for Molecular Biology (ISMB'00)*, 128-133, La Jolla, CA.
- Dorohonceanu, B.; and Nevill-Manning, C.G. (2000b). A Practical Suffix Tree Implementation for String Search, *Dr. Dobb's Journal, Algorithm Alley*, July.
- Durbin, R.; Sean, E.; Krogh, A.; and Mitchison, G. (1998) Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. *Cambridge University Press*, Cambridge UK.
- Gribnikov, 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*, New York.
- 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 Research*, **27**:215-219.
- McCreight, E.M. (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.; and Brutlag, D.L. (1998). Highly Specific Protein Sequence Motifs for Genome Analysis, *Proceedings of the National Academy of Sciences USA*, **95**(11):5865-5871.
- Ukkonen, E. (1995). On-line Construction of Suffix Trees. *Algorithmica*, **14**(3):249-260.
- 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