1 Yet more suffix-trees, more hybrid dynamic programming

Although the suffix tree was initially designed and employed to handle complex problems of *exact* matching, it can be used to great advantage in various problems of *inexact matching*. This has already been demonstrated in Sections **??** and **??** where the k-mismatch and k- difference problems were discussed. The suffix tree in the latter application was used in combination with dynamic programming to produce a *hybrid dynamic programming* method that is faster than dynamic programming alone. One deficiency of that approach is that it does not generalize nicely to problems of *weighted* alignment. In this section, we introduce a different way to combine suffix trees with dynamic programming for problems of weighted alignment. These ideas have been claimed to be very effective in practice, particularly for large computational projects. However, the methods do not always lend themselves to greatly improved *provable worst case* time bounds. The ideas presented here loosely follow the published work of Ukkonen [7] and an unpublished note of Gonnet and Baeza-Yates [1]. A related idea for using suffix trees in regular expression pattern matching (with errors), and its large-scale application in managing genomic databases, appears in [5]. The method of Gonnet and Baeza-Yates has been implemented and extensively used for large-scale protein comparisons [6, 4].

Two problems

We assume the existence of a scoring matrix used to compute the value of any alignment, and hence "edit distance" here refers to *weighted* edit distance. We will discuss two problems in the text, and introduce two more related problems in the exercises.

1. **The P-against-all problem:** Given strings P and T, compute the edit distance between P and *every* substring T' of T.

2. **The threshold all-against-all problem:** Given strings P and T and a threshold d, find every pair of substrings P' of P and T' of T such that the edit distance between P' and T' is less than d.

The all-against-all problem is similar to problems mentioned in Section **??** concerning the construction of non-redundant sequence databases. However, the all-against-all problem is harder, because it asks for the alignment of all pairs of *substrings*, not just the alignment of all pairs of strings. This critical distinction has been the source of some confusion in the literature [2, 3].

1.1 The P-against-all problem

The problem again is: Given strings P and T, compute the edit distance between P and *every substring* T' of T. This is an example of a *large-scale alignment* problem

that asks for a great amount of related alignment information. If not done carefully, its solution will involve a large amount of redundant computation.

Assume that P has length n and T has length $m > n$. The most naive solution to the P-against-all problem is to enumerate all $\binom{m}{2}$ substrings of T, and then seperatly compute the edit distance between P and each substring of T. This takes $\Theta(nm^3)$ total time. A moment's thought leads to an improvement. Instead of choosing all substrings of T, we need only choose each *suffix* S of T and compute the dynamic programming edit distance table for strings P and S. If S begins at position i of T, then the last row of that table gives the edit distance between P and every substring of T that begins at position i. That is, the edit distance between P and $T[i..j]$ is found in cell $(n, j - i + 1)$ of the table. This approach takes $\Theta(nm^2)$ total time.

We are interested in the P-against-all problem when T is very long. In that case, the introduction of a suffix tree may greatly speed up the dynamic programming computation, depending on how much repetition is contained in string T [1]. (See also Section **??**.) To get the basic idea of the method, consider two substrings T' and T'' of T that are identical for their first n' characters. In the dynamic programming approach above, the edit distances between P and T' and between P and T'' would be computed separately. But if we compute edit distance *column-wise* (instead of in the usual row-wise manner), then we can combine the two edit distance computations for the first n' columns, since the first n' characters of T' and T'' are the same (see Figure 1). It would be redundant to compute the first n by n' subtable separately for the two edit distances. This idea of using the commonality of T' and T'' can be formalized and fully exploited through the use of a suffix tree for string T.

Consider a suffix tree $\mathcal{T}$ for string T and recall that any path from the root of $\mathcal{T}$ specifies some substring S of T. If we traverse a path from the root of $\mathcal{T}$, and we let S denote the growing substring corresponding to that path, then during the traversal we can build up (columnwise) the dynamic programming table for the edit distance between P and the growing substring S of T. The full idea then is to traverse $\mathcal{T}$ in a *depth- first* manner, computing the appropriate dynamic programming column (from the column to its left) for every substring S specified by the current path. When the traversal reaches a node v of $\mathcal{T}$, it stores there the last (most recently generated) column and last subrow of the current subtable (the last row will always be row n). That is, if S is the substring specified by the path to a node v, then what will be stored at v is the last row and column of the dynamic programming table for the edit distance between P and S. When the depth-first traversal visits a child v' of v, it adds columns (one for each character on the (v, v') edge) to this table to correspond to the extension of substring S. When the depth-first traversal reaches a leaf of $\mathcal{T}$ corresponding to the suffix starting at a position i (say) of T, it can then output the values in the last row of the current table. Those values specify the edit distances

[1] Recent estimates put the amount of repeated human DNA at 50 to 60%. That is, 50 to 60% of all human DNA is contained in *non- trivial length*, structured substrings that show up repeatedly throughout the genome.

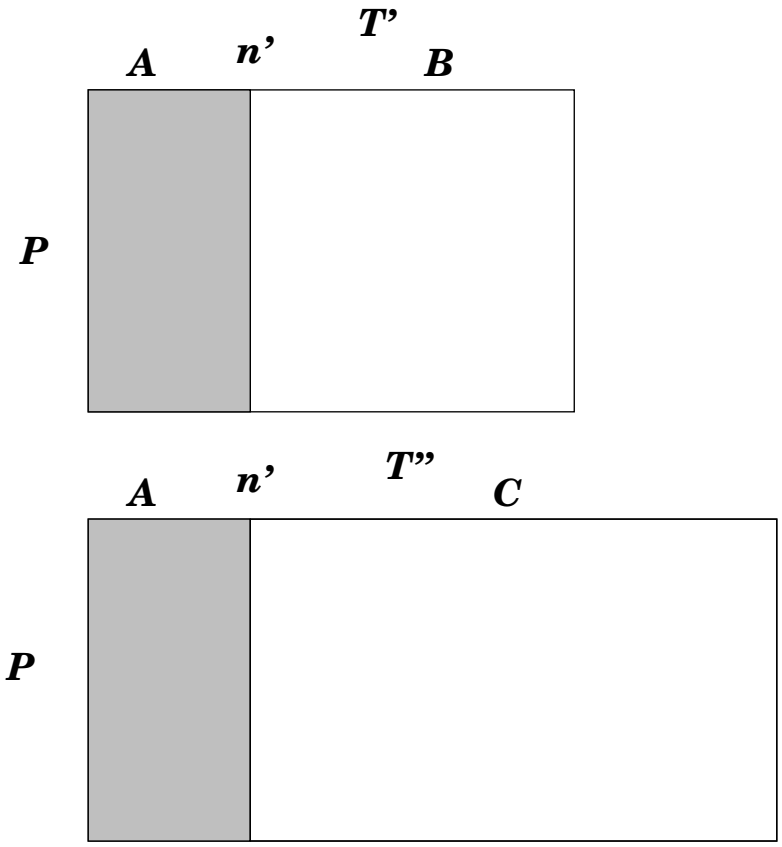

Figure 1: A cartoon of the dynamic programming tables for computing the edit distance between P and substring T', and between P and substring T''. The two tables share the subtable for P and substring A (shown as a shaded rectangle). This shaded subtable only needs to be computed once.

between P and every substring beginning at position i of T. When the depth-first traversal backs up to a node v and v has an unvisited child v', the row and column stored at v are retrieved and extended as the traversal follows a new (v, v') edge (see Figure 2).

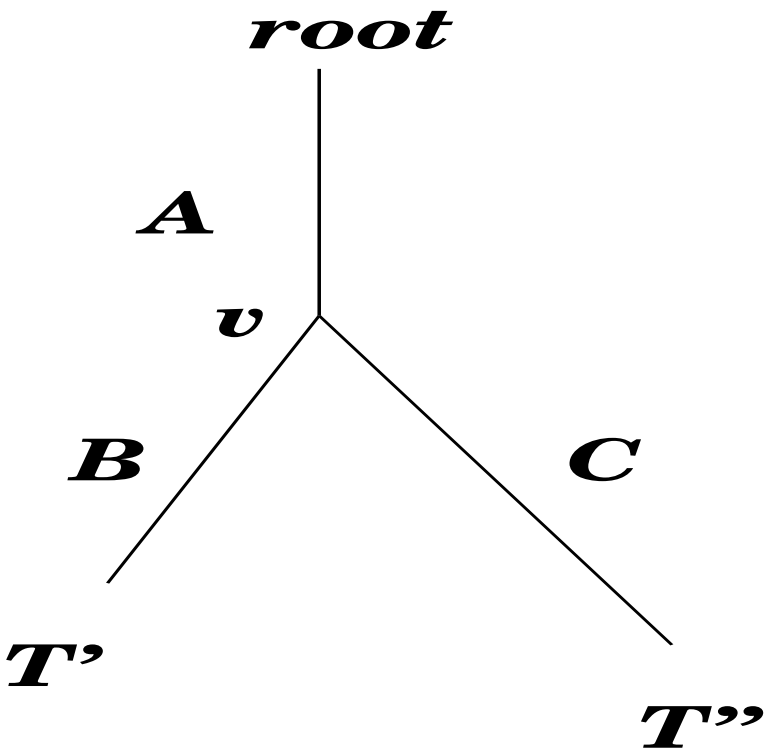

Figure 2: A piece of the suffix tree for T. The traversal from the root to node v is accompanied by the computation of subtable A (from the previous figure). At that point, the last row and column of subtable A are stored at node v. Computing the subtable B corresponds to the traversal from v to the leaf representing substring T'. After the traversal reaches the leaf for T', it backs up to node v, retrieves the row and column stored there, and uses them to compute the subtable C needed to compute the edit distance between P and T''.

It should be clear that this suffix-tree approach does correctly compute the edit distance between P and every substring of T, and it does exploit repeated substrings (small or large) that may occur in T. But how effective is it compared to the $\Theta(nm^2)$-time dynamic programming approach?

Definition The *string-length* of an edge label in a suffix tree is the length of the string labeling that edge (even though the label is compactly represented by a constant number of characters). The *length of a suffix tree* is the sum of the string-lengths for all of its edges.

The length for a suffix tree $\mathcal{T}$ for a string T of length m can be anywhere between $\Theta(m)$ and $\Theta(m^2)$, depending on how much repetition exists in T. In computational experiments using long substrings of mammalian DNA (length around one million), the string-length of the resulting suffix trees have been around $\frac{m^2}{10}$. Now the number of dynamic programming columns that are generated during the depth-first traversal of $\mathcal{T}$ is exactly the length of $\mathcal{T}$. Each column takes $\Theta(n)$ time to generate, so

Lemma 1.1 *The time used to generate the needed columns in the depth-first traversal is* $\Theta(n \times (\textit{length of } \mathcal{T}))$.

We must also account for the time and space used to write the rows and columns stored at each node of $\mathcal{T}$. In a suffix tree with m leaves there are $\Theta(m)$ internal nodes and a single row and column take at most $O(m+n)$ time and space to write. Therefore, $\Theta(m^2 + nm) = \Theta(m^2)$ time and space is needed for the row and column stores. Hence

Theorem 1.1 *The total time for the suffix-tree approach is* $\Theta(n \times (\textit{length of } \mathcal{T}) + m^2)$, *and the maximum space used is* $\Theta(m^2)$.

Reducing space

The size of the required output is $\Theta(m^2)$, since the problem calls for the edit distance between P and *each* of $\Theta(m^2)$ substrings of T, so the $\Theta(m^2)$ term in the time bound is acceptable. On the other hand, the space used seems excessive since the space needed by the dynamic programming solution without using a suffix tree is just $\Theta(nm)$ and can be reduced to $O(m)$. We now modify the suffix-tree approach to also use only $O(n+m)$ space, and the same time bounds as before.

First, there is no need to store the current column at each node v. When backing up from a child v' of v, we can use the current column at v' and the string labeling edge (v, v') to recompute the column for node v. This does however double the total time for computing the columns. There is also no need to keep the current row n at *each* node v. Instead, only $O(n)$ space is needed for row entries. The key idea is that the current table is expanded columnwise, so if the string-depth of v is j and the string- depth of v' $j+d$, then the row n stored at v and v' would be identical for the first j entries. We leave it as an exercise to work out the details. In summary,

Theorem 1.2 *The hybrid suffix-tree/dynamic programming approach to the P-against-all problem can be implemented to run in* $\Theta[n(\textit{length of } \mathcal{T}) + m^2]$ *time and* $O(n+m)$ *space.*

The above time and space bounds should be compared to the $\Theta(nm^2)$ time and $O(n+m)$ space bounds that result from a straightforward application of dynamic programming. The effectiveness in practice of this method depends on the length of

$\mathcal{T}$ for realistic strings. It is known that for *random* strings, the length of $\mathcal{T}$ is $\Theta(m^2)$, making the method unattractive. (For random strings, the suffix tree is bushy for string depths of $\log_\sigma m$ or less, where σ is the size of the alphabet. But beyond that depth, the suffix tree becomes very sparse, since the probability is very low that a substring of length greater than $\log_\sigma m$ occurs more than once in the string.) But strings with more structured repetitions (as occur in DNA) should give rise to suffix trees with length that is small enough to make this method useful. We examined this question empirically for DNA strings up to one million characters, and the lengths of the resulting suffix trees were around $m^2/10$.

1.2 The (threshold) all-against-all problem

Now we consider the most ambitious problems, to find the edit distance between *every pair* of *substrings*, one from P and one from T, or to find every pair of substrings where the edit distance is below a fixed threshold d. Computations of this type have been conducted when P and T are both equal to the combined set of protein strings in the database Swiss- Prot [6]. The importance of this kind of large-scale computation and the way in which its results are used, are discussed in and [4]. The way suffix trees are used to accelerate the computation is discussed in [1].

Since P and T have respective lengths of n and m, the full all- against-all problem calls for the computation of n^2m^2 pieces of output. Hence no method for this problem can run faster than $\Theta(n^2m^2)$ time. Moreover, that time bound is easily achieved: pick a pair of starting positions in P and T (in nm possible ways), and for each choice of starting positions i, j fill in the dynamic programming table for the edit distance of $P[i..n]$ and $T[j..m]$ (in $O(nm)$ time). For any choice of i and j, the entries in the corresponding table give the edit distance for every pair of substrings that begin at position i in P and at position j in T. So to achieve the $O(n^2m^2)$ bound for the full all-against-all problem, no suffix trees are needed.

But the full all-against-all problem calls for an amount of output that is often excessive, and the threshold version of the problem usually produces results that are more meaningful. And, one can further modify the threshold problem so that trivial alignments are also omitted. That is, one might only want to report a pair of substrings if they both exceed a certain length and yet have an edit distance below a given threshold. Or the criteria for reporting a substring pair might be a function of both length and edit distance. Whatever the specific reporting criteria, if it is no longer necessary to report the edit distance of every pair, it is no longer certain that $\Theta(n^2m^2)$ time is required. Here we develop a method whose worst case running time is expressed as $O(C + R)$, where C is a computation time that may be less than $\Theta(n^2m^2)$ and R is the output size, i.e., the number of reported pairs of substrings. In this setting, the use of suffix trees may be quite valuable depending on the size of the output and the amount of repetition in the two strings.

An $O(C + R)$-time method

The method uses a suffix tree $\mathcal{T}_P$ for string P and a suffix tree $\mathcal{T}_T$ for string T. The worst case time for the method will be shown to be $O(C+R)$, where C is the length of $\mathcal{T}_P$ times the length of $\mathcal{T}_T$ *independent of whatever the output criteria are*, and R is the size of the output. That is, the method will compute certain dynamic programming cell values, which will be the same no matter what the output criteria are, and then when a cell value satisfies the particular output criteria, the algorithm will collect the relevant substrings associated with that cell. Hence our description of the method holds for the full all-against-all problem, the threshold version of the problem, or any other version with different reporting criteria.

To start, recall that each node in $\mathcal{T}_P$ represents a substring of P and every substring of P is a prefix of a substring represented by a node of $\mathcal{T}_P$. In particular, each suffix of P is represented by a leaf of $\mathcal{T}_P$. The same is true of T and $\mathcal{T}_T$.

Definition The dynamic programming table for a pair of nodes (u, v), from $\mathcal{T}_P$ and $\mathcal{T}_T$ respectively, is defined as the dynamic programming table for the edit distance between the string represented by node u and the string represented by node v.

The threshold all-against-all problem could be solved (ignoring time) by computing the dynamic programming table for each pair of leaves, one from each tree, and then examining every entry in each of those tables. Hence it certainly would be solved by computing the dynamic programming table for each pair of nodes, and then examining each entry in those tables. This is essentially what we will do, but in a way that avoids redundant computation and examination. The following lemma gives the key observation.

Lemma 1.2 *Let u' be the parent of node u in $\mathcal{T}_P$ and let α be the string labeling the edge between them. Similarly let v' be the parent of v in $\mathcal{T}_T$ and let β be the string labeling the edge between them. Then, all but the bottom right $|\alpha||\beta|$ entries in the dynamic programming table for the pair (u, v) appear in one of the tables for (u', v') or (u', v) or (u, v'). Moreover, that bottom right part of the (u, v) table can be obtained from the other three tables in $O(|\alpha||\beta|)$ time. See Figure 3.*

The proof of this lemma is immediate from the definitions and the edit distance recurrences.

The computation for the new part of the (u, v) table produces an $|\alpha|$ by $|\beta|$ rectangular subtable which forms the lower right section of the (u, v) table. In the algorithm to be developed below, we will store and associate with each node pair (u, v) the last column and the last row of this $|\alpha|$ by $|\beta|$ subtable.

We can now fully describe the algorithm.

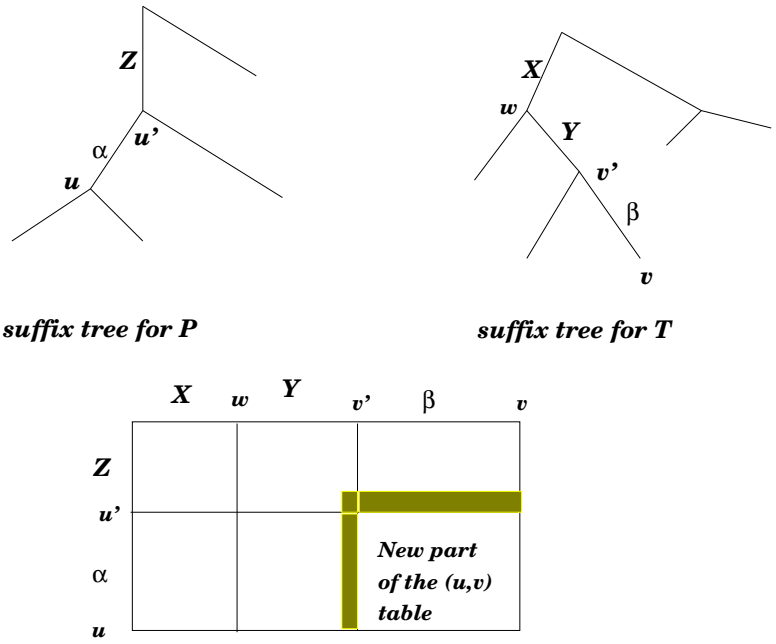

Figure 3: The dynamic programming table for (u, v) is shown below the suffix trees for P and T. The string on the path to node u is $Z\alpha$ and the string to node v is $XY\beta$. All cells in the (u, v) table, except those in the lower right rectangle, are also in the (u, v') or (u', v) or (u', v') tables. The new part of the (u, v) table can be computed from the shaded entries and substrings α and β. Exactly one entry from the (u', v') table is needed; $|\alpha|$ entries from the last column in the (u, v') table are needed; and $|\beta|$ entries from the last row in the (u', v) table are needed.

Details of the algorithm

First, number the non-root nodes of $\mathcal{T}_P$ according to string-depth, smaller string-depth first.[2] Separately, number the nodes of $\mathcal{T}_T$ according to string-depth. Then form a list L of all pairs of node numbers, one from each tree, in lexicographic order. Hence, pair (u, v) appears before pair (p, q) in the list if and only if u is less than p, or u is equal to p and v is less than q. See Figure 4. It follows that if u' is the parent of u in $\mathcal{T}_P$ and v' is the parent of v in $\mathcal{T}_T$, then (u', v') appears before (u, v).

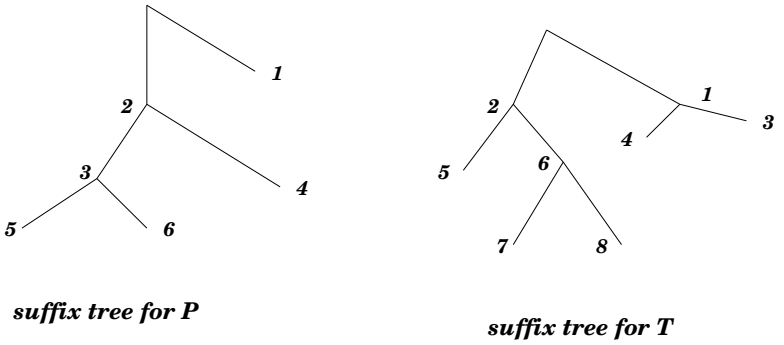

Figure 4: The suffix trees for P and T with nodes numbered by string-depth. Note that these numbers are not the standard suffix position numbers that label the leaves. The ordered list of node pairs begins $(1, 1), (1, 2), (1, 3)$... and ends with $(6, 8)$.

Next, process each pair of nodes (u, v) in the order that it appears in L. Assume again that u' is the parent of u, that v' is the parent of v, and that the labels on the respective edges are α and β. To process a node pair (u, v), retrieve the value in the single lower right cell from the stored part of the (u', v') table; retrieve the column stored with the pair (u, v'), and retrieve the row stored with the pair (u', v). These three pairs of nodes have already been processed, due to the lexicographic ordering of the list. From those retrieved values, and from the substrings α and β, compute the new $|\alpha|$ by $|\beta|$ subtable completing the (u, v) table. Store with pair (u, v) the last row and column of newly computed subtable.

Now suppose cell (i, j) is in the new $|\alpha|$ by $|\beta|$ subtable, and its value satisfies the output criteria. The algorithm must find and output all locations of the two substrings specified by (i, j). As usual, a depth- first traversal to the leaves below u and v will then find all the starting positions of those strings. The length of the strings is determined by i and j. Hence, when it is required to output pairs of substrings that satisfy the reporting criteria, the time to collect the pairs is just proportional to the number of them.

[2] Actually, any topological numbering will do, but string depth has some advantages when heuristic accelerations are added.

1.2.1 Correctness and time analysis

The correctness of the method follows from the fact that at the highest level of description, the method computes the edit distance for every pair of substrings, one from each string. It does this by generating and examining every cell in the dynamic programming table for every pair of substrings (although it avoids redundant examinations). The only subtle point is that the method generates and examines the cells in each table in an incremental manner to exploit the commonalities between substrings, and hence it avoids regenerating and re-examining any cell that is part of more than one table. Further, when the method finds a cell satisfying the reporting criteria (a function of value and length), it can find all the pairs of substrings specified by that cell using a traversal to a subset of leaves in the two suffix trees. A formal proof of correctness is left to the reader as an exercise.

For the time analysis, recall that the length of $\mathcal{T}_P$ is the sum of lengths of all the edge labels in $\mathcal{T}_P$. If P has length n, then the length of $\mathcal{T}_P$ ranges between n and $n^2/2$, depending on how repetitive P and T are. The length of $\mathcal{T}_T$ is similarly defined and ranges between m and $m^2/2$, where m is the length of T.

Lemma 1.3 *The time used by the algorithm for all the needed dynamic programming computations and cell examinations, is proportional to the product of the length of $\mathcal{T}_P$ and the length of $\mathcal{T}_T$. Hence that time, defined as C, ranges between nm and n^2m^2.*

Proof In the algorithm, each pair of nodes is processed exactly once. At the point a pair (u, v) is processed, the algorithm spends $O(|\alpha||\beta|)$ time to compute a subtable and examine it, where α and β are the labels on the edges into u and v respectively. Each edge label in $\mathcal{T}_P$ therefore forms exactly one dynamic programming table with each of the edge labels in $\mathcal{T}_T$. The time to build those tables is $|\alpha|$(length of $\mathcal{T}_T$). Summing over all edges in $\mathcal{T}_P$ gives the claimed time bound. □

The above lemma counts all the time used in the algorithm except the time used to collect and report pairs of substrings (by their starting position, length and edit distance). But the algorithm collects substrings when it sees a cell value that satisfies the reporting criteria. So the time devoted to output is just the time needed to traverse the tree to collect output pairs. We have already seen that this time is proportional to the number of pairs that are collected, and is R. Hence,

Theorem 1.3 *The complete time for the algorithm is $O(C + R)$.*

How effective is the suffix tree approach?

As in the P-against-all problem, the effectiveness of this method in practice depends on the lengths of $\mathcal{T}_P$ and $\mathcal{T}_T$. Clearly, the product of those lengths, C, falls as P and T increase in repetitiveness. We have built a suffix tree for DNA strings of total length around one million bases, and observed that the tree length is around one tenth of

the maximum possible. In that case, C is around $\frac{n^2m^2}{100}$, so all else being equal (which is unrealistic), standard dynamic programming for the all- against-all problem should run about one hundred times slower than the hybrid dynamic programming approach.

A vastly larger "all-against-all" computation on amino acid strings was reported in [6]. Although their description is very vague, they essentially used the suffix tree approach described here, computing similarity instead of edit distance. But, rather than a one-hundred fold speed up, they claim to have achieved nearly a million-fold speed up over standard dynamic programming[3]. That level of speedup is not supported either by theoretical considerations (recall that for a random string S of length m, a substring of length greater than $\log_\sigma m$ is very unlikely to occur in S more than once), or by the experiments we have done. The explanation may be the incorporation of an early stopping rule described in [6] only by the statement "Time is saved because the matching of patricia[4] subtrees is aborted when the score falls below a liberally chosen similarity limit". That rule is apparently very effective in reducing running time, but without a clearer description of it we cannot define precisely what specific all-against-all problem was solved.

References

[1] R. Baeza-Yates and G. Gonnet. All-against-all sequence matching. unpublised note.

[2] G. J. Barton. Computer speed and sequence comparison (letter to the editor). *Science*, 257:1609–1609, 1992.

[3] S. A. Benner, M. A. Cohen, and G. H. Gonnet. Response to barton's letter: Computer speed and sequence comparison. *Science*, 257:1609–1610, 1992.

[4] S. A. Benner, M. A. Cohen, and G. H. Gonnet. Empirical and structural models for insertions and deletions in the divergent evolution of proteins. *J. Mol. Biol.*, 229:1065–1082, 1993.

[5] P. Bieganski. *Genetic sequence data retrieval and manipulation based on generalized suffix trees*. PhD thesis, University of Minnesota, Department of Computer Science, 1995.

[6] G. H. Gonnet, M. A. Cohen, and S. A. Benner. Exhausive matching of the entire protein sequence database. *Science*, 256:1443–1445, 1992.

[7] E. Ukkonen. Approximate string-matching over suffix trees. *Proc. 4'th Symp. on Combinatorial Pattern Matching. Springer LNCS 684*, pages 228–242, 1993.

[3] They finish a computation in 405 cpu days that they claim would otherwise have taken more than a million cpu years without the use of suffix trees

[4] A patricia tree is a variant of a suffix tree.