Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation

Paul Bieganski, John Riedl and John V. Carlis
Computer Science Department, University of Minnesota
Ernest F. Retzel
Medical School, University of Minnesota

This paper addresses applications of suffix trees and generalized suffix trees (GSTs) to biological sequence data analysis. We define a basic set of suffix tree and GST operations needed to support sequence data analysis. While those definitions are straightforward, the construction and manipulation of disk-based GST structures for large volumes of sequence data requires intricate design. GST processing is fast because the structure is content addressable, supporting efficient searches for all sequences that contain particular subsequences. Instead of laboriously searching sequences stored as arrays, we search by walking down the tree. We present a new GST-based sequence alignment algorithm, called GESTALT. GESTALT finds all exact matches in parallel, and uses best-first search to extend them to produce alignments. Our implementation experiences with applications using GST structures for sequence analysis lead us to conclude that GSTs are valuable tools for analyzing biological sequence data.

1 Introduction

Genetic codings transmit information from a parent to its progeny. The primary representation of information is sequences of symbols from a limited alphabet. The most commonly used alphabets represent the four nucleic acids forming strands of DNA and the twenty amino acids constituting polypeptides. The information encoding power of these limited sets of molecules lies in their ability to form long chains.

Sequence information is most commonly stored in computer memory in contiguous locations, in order of the molecules in the biological sequence. This storage method is not efficient for a large group of sequence information processing applications. The key problem lies in the fact that data stored sequentially must be processed sequentially. The information within the sequence is often encoded through the presence of a certain subsequence of molecules; for example a sequence of DNA coding for a certain protein. In order to detect the presence of any given subsequence the entire sequence must be accessed, and in order to detect a subsequence in a set of sequences, each sequence must be accessed sequentially. As the volume of sequence data increases, the data access time itself becomes the limiting factor of sequence information retrieval regardless of advances in sequence comparison speeds. What is needed is a system capable of *content-addressing* of sequence. Such a system allows a sequence to be accessed in terms of *what* it contains without having to specify *where* it is contained.

In this, paper we describe the data structures and operations providing content-addressable access to sequence data, outline their possible applications and report on our work on their implementation. In section 2, we provide a short review of suffix trees and generalized suffix trees. In section 3, we review the operations that provide the basis for construction of applications using suffix tree structures. In section 4 we outline a generalized suffix tree-based sequence homology search algorithm. In section 5, we present an overview of sequence information processing applications that can be constructed using the basic operations described in section 3. In section 6 we report on our experiences with implementations of some of the applications. In section 7, we summarize and outline future work.

2 Suffix trees

A *trie* is an indexing structure used for indexing sets of key values of varying sizes[13]. A trie is a tree in which the branching at any level is determined by a partial key value (see figure 1). A suffix tree is a PATRICIA trie [4] built over the set of all suffixes of a given sequence S. Figure 2 illustrates a sample suffix tree of a short DNA sequence. Each path from the root of the suffix tree represents a suffix of the original string. Any individual suffix of the original sequence can be recreated by walking along a path from the root and catenating the labels of the edges traversed along the way. Nodes with a single outgoing edge can be collapsed, resulting in edges with multi-

1060-3425/94 $3.00 © 1994 IEEE

Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, 1994

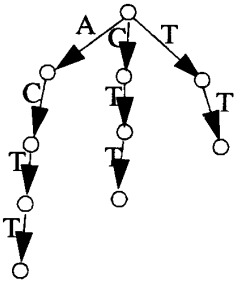

FIGURE 1. A suffix tree for sequence ACTT

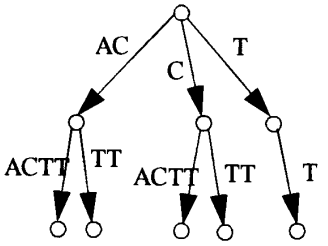

FIGURE 2. Suffix tree for sequence ACACTT. Some edges have multi-symbol labels resulting from collapsing nodes with single children.

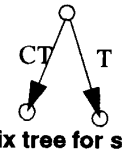

FIGURE 3. Suffix tree for sequence CT

symbol labels, such as the edge labeled AC coming out of the root of the tree in figure 2.

A suffix tree for sequence S of length n can be constructed in O(n) time [8] and the number of nodes in the tree is in general a linear function of n [30]. Heuristic arguments [30] and our experiments with trees containing the rodent subsection of GenBank indicate that the number of nodes is equal to less than twice the number of bases in the sequence for long DNA sequences.

A generalized suffix tree (GST) is an augmented version of the suffix tree allowing for multiple sequences to be stored in the same tree. A GST can be viewed as a suffix tree with additional sequence-identifier leaves added to the leaves of the original suffix tree. A generalized suffix tree containing multiple sequences contains all suffixes of each of the original sequences (see figures 2, 2 and 4). For every suffix, its sequence of origin is identified. A GST can be augmented with information about the number of different sequences that contain suffixes expressed by descendants of each node (see figure 4). This number is also known as the Color Set Count (CSC) [7]. A GST can

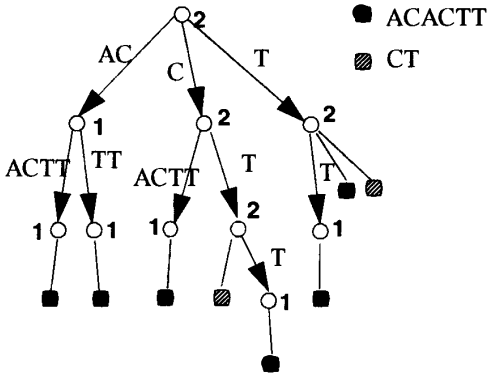

FIGURE 4. A generalized suffix tree for two sequences, ACACTT and CT. Notice the sequence identifier leaves (squares) denoting the origin of every suffix of every sequence. The number of identifier leaves for every sequence is equal to the number of suffixes of that sequence (equal to the number of characters in the sequence).

be constructed in O(n) time [30], where n is the sum of lengths of all sequences stored in the tree. Parallel algorithms for GST construction will be described later.

3 Basic suffix tree operations

In this section we describe a set of basic operations on generalized suffix trees. Molecular biology applications are implemented by combining one or more of these operations.

In the remainder of the paper we use seq to denote a single sequence and SEQS to denote a set of sequences. In suffix trees with multi-letter edge labels, it is not sufficient to give the node in order to describe a subsequence because the subsequence may terminate within an edge. For the sake of clarity we assume subsequences terminate at nodes in the discussion.

3.1 Suffix tree construction [tree(seq)]

Suffix tree construction algorithms have been described in the literature [8, 17, 23]. They operate by constructing an initial tree with a single branch corresponding to the entire sequence and incrementally modifying the tree to include all of its suffixes. An important variable in the construction process is the choice of data structures used to represent the tree. Fixed node-size trees have a node access time advantage over child list-based ones since a descendant can be accessed directly, without having to traverse a list. They are particularly useful when the number of descendant pointers is small. For instance, each node of a DNA sequence GST can have at most four children, labelled A, C, G and T. Child list-based nodes are

more space efficient in applications when the number of descendant nodes may be large, for example when representing amino acid sequences.

3.2 Generalized suffix tree construction [gst(SEQS)]

The GST construction process is usually approached [7, 23] by adding a special *sequence separator* symbol to the alphabet. The sequences to be included in the tree are catenated, separated from each other by the separator symbol. The GST is created using the ordinary suffix tree construction algorithm on the catenated sequence. The GST created using this process has to be kept in main memory during construction, hence this approach is not feasible when sets of thousands or more sequences are involved. We have developed an incremental disk-based GST construction method using binary merging of GSTs. Two GSTs representing two disjoint sets of sequences are merged to produce a single GST representing the union of the two sets. A GST for a large set of sequences can be constructed by performing a series of binary merges of GSTs of increasing size, starting with n/2 merges of single sequence trees and ending with a single merge of two trees of n/2 sequences. All merges at each level of this binary merge tree can be performed in parallel. Our merge procedure operates in limited main memory on GSTs stored in disk files, thus making it well-suited for execution on clusters of workstations.

3.3 Suffix tree walking [match(tree(seq_1), seq_2)]

Matching of a single sequence against a suffix tree is the simplest variant of a family of tree-based matching operations. A path expressing a given sequence seq_2 is traversed from the root of the suffix tree constructed for sequence seq_1. Traversal is terminated when the end of seq_2 is reached or a node in $tree(seq_1)$ is reached beyond which further traversal is not possible. The point in the tree at which the traversal is terminated determines the longest prefix of seq_2 contained within the sequence seq_1.

3.4 GST walking [match(gst(SEQS), seq)]

A path expressing a given sequence seq is traversed from the root of a GST constructed for a set of sequences SEQS. Traversal is terminated when the end of seq is reached or a node in gst(SEQS) is reached beyond which further traversal is not possible. The point in the tree at which the traversal is terminated determines the longest prefix of seq contained within the set of sequences SEQS. Examination of nodes descendant to the traversal termination point can be used to determine where and in what sequences of SEQS the fragments matching a prefix of seq are located. The number of sequences of SEQS matching the prefix of seq can be determined by examining the sequence count information in the nearest descendant of the traversal termination point (see the description of GST in section 2).

3.5 Suffix tree matching [match(tree(s), tree(p))]

Matching of suffix trees against suffix trees is performed similarly to matching of sequences against suffix trees. Instead of traversing a single path, however, all paths corresponding to an exhaustive traversal of one tree are traversed in the other tree. If the trees were truncated every time the traversal process reaches a dead end in either tree, the resulting tree would contain all common subsequences of the two sequences. This tree can be examined to determine the lengths and locations for common subsequences. To avoid truncating the trees, summary information about the match can be collected during the traversal process. For instance, the longest common subsequence can be determined by keeping track of the longest match encountered during the traversal.

3.6 GST and suffix tree matching [match(gst(SEQS), tree(seq))]

Matching of a suffix tree against a GST is performed identically to the matching of two suffix trees. The matches found between the two trees will indicate subsequences common between the sequence seq and *any* of the subsequences contained in SEQS. Matches can be analyzed in a manner described in section 3.4 to determine their exact location and the members of SEQS involved.

3.7 GST addition [add(gst($SEQS_1$), gst($SEQS_2$))]

GST addition allows GSTs for unions of disjoint sets of sequences to be constructed by merging GSTs of the individual sets. Addition of GSTs is performed by pre-order traversal of both GSTs and merging of branches corresponding to common subsequences. The GST merge operation is in its practical implementation less expensive computationally than the construction of a new GST, even though both operations are of the same order of complexity. GST addition also has the advantage of supporting disk-based representations of GSTs in limited main memory. We use GST addition in GST construction for very large sequence sets.

3.8 GST subtraction [subtract(gst($SEQS_1$), gst($SEQS_2$))]

GST subtraction is the reverse of GST addition. A GST corresponding to the difference of two sets of sequences $SEQS_1$ and $SEQS_2$ ($SEQS_2$ contained in $SEQS_1$) is constructed by traversing both trees and inserting in the

result tree only nodes corresponding to sequences not present in $SEQS_2$.

3.9 Suffix tree and GST operation summary

Table 1 summarizes all operations described in this section and their computational complexities. It also includes brief comments on their potential for parallelization and lists relevant references.

4 Generalized suffix tree alignment algorithm

In this section we outline a sequence similarity search algorithm based on the content-addressability of sequence data provided by generalized suffix trees.

Sequence *alignment* is a widely used method of addressing sequence similarity [1, 9, 21, 32]. An alignment of two sequences can be measured by defining an alignment cost function [1, 3, 18]. The cost function determines the cost or penalty for mismatched, deleted and inserted sequence elements in the alignment. An alignment with the minimal cost with respect to a given cost function is the *optimal* alignment. Determination of optimal and near-optimal alignments of sequences is an important research tool, since sequences with low alignment costs have been shown to be frequently functionally related.

An alignment of two sequences can be uniquely represented by a path in a two-dimensional lattice (see figure 5). The problem of optimal alignment determination becomes the problem of finding the lowest-cost path in the lattice.

A number of dynamic programing-based optimal alignment algorithms have been developed and described in the literature [1, 9, 10, 11, 15, 16, 18, 21, 25, 32]. Most of them function by enumerating alignments of prefixes of the two sequences of increasing length. This corresponds

Operation	Time complexity[a]	Space complexity[a]	Parallel implementation	References
tree(seq)	O(n)	O(n)		8, 23, 17
gst(SEQS)	O(N)	O(N)	Parallel binary divide-and-conquer[b]	8, 23, 17, 7
match(tree(seq_1), seq)	O(n)	O(1)		23, 7
match(gst(SEQS), seq)	O(n)	O(1)		23, 7
match(tree(seq_1), tree(seq_2)	O(m)[c]	O(m)	Parallel multiple branch traversal	
match(gst(SEQS), tree(seq_2)	O(m)[d]	O(m)	Parallel multiple branch traversal	
add(gst($SEQS_1$), gst($SEQS_2$))	O(N_1+N_2)	O(N+N_2)[e]	Parallel multiple branch merging	
subtract(gst($SEQS_1$), gst($SEQS_2$))	O(N_1)	O(N_1)[h]	Parallel multiple branch processing	

TABLE 1. Basic suffix tree and generalized suffix tree operations; n = length(seq), N = length(SEQS), N1 = length(SEQS1), N2 = length(SEQS2).

a. Complexities of nested operations include the cost of the outermost operation only.

b. The last step of the binary merge limits the performance to O(N)

c. m is proportional to the number and length of common subsequences of seq_1 and seq_2 and m <= min(-length(seq_1), length(seq_2))

d. Additional processing of match information may increase cost

e. Can be performed in-place by modifying an existing GST

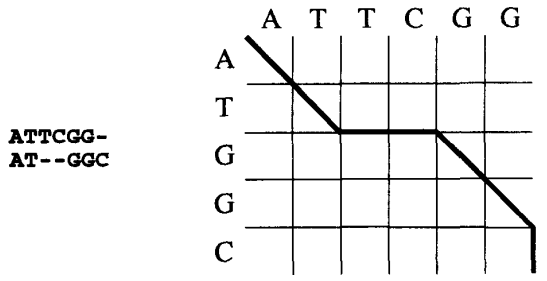

FIGURE 5. Graphical representation of a sequence alignment as a path in a two-dimensional graph. Diagonal edges correspond to a match (or mismatch) of two characters whereas horizontal and vertical segments correspond to insertions (gaps) in the vertical and horizontal sequences, respectively.

to enumerating all paths originating in the upper-left corner of Figure 5. The number of possible paths grows exponentially with their length. To bound the number of paths that must be considered, the dynamic programming algorithms use cost functions with properties ensuring that, whenever paths cross, only the lowest cost path must be remembered. By repeatedly considering paths of increasing lengths the algorithms eventually arrive at the lower-right corner of the lattice in figure 5 in O(nm) operations, which is the number of possible path crossings in the lattice. Algorithms vary in details such as the form of the cost functions supported and the way the information about the path chosen at every crossing point is stored.

Improvements to the basic algorithm outlined above have been suggested. One approach is to pre-compute path segments corresponding to exact matches of subsequences in order to restrict the areas of the lattice — and the number of potential path crossing points — that have to be considered [6, 14].

Both the basic and improved versions of the alignment algorithm process two sequences at a time. Two ways of improving the search process for alignments of multiple sequences are possible: processing multiple alignments and multiple paths at the same time, and reducing the number of paths considered by considering lowest cost paths first. Both of these improvements could be achieved if an algorithm were able to search all subsequences of all sequences simultaneously. Such an algorithm would operate in the content domain ("*what*"), instead of the position domain ("*where*"). A generalized suffix tree provides exactly this ability and forms the basis of the Generalized Suffix Tree Alignment (GESTALT) algorithm outlined below.

The GESTALT algorithm determines alignments of a given sequence **seq** against a set of sequences SEQS = $\{S_1, ..., S_n\}$ by aligning a suffix tree and a generalized suffix tree. By performing the match operation (match(gst(SEQS), tree(seq))) on the two trees the algorithm determines all common subsequences of **seq** and any of the members of SEQS. This is equivalent to diagonal path segment pre-computation for *all* pairs of sequences simultaneously. Figure 6 shows a graphical representation of the problem of determining the best alignment of a single sequence **seq** against a set of sequences SEQS. Path segments M_1 through M_5 (figure 6) corresponding to all exact matches against fragment f of sequence **seq** are all discovered by a single pair of matching tree branches[1]. Path segments such as these are likely to be parts of paths corresponding to low cost alignments (figure 6).

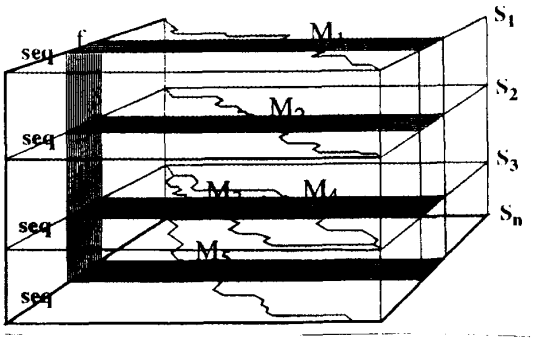

FIGURE 6. Graphical representation of the search space of a GST and suffix tree alignment algorithm. The algorithm determines the optimal alignments of sequence seq against a set of sequences S_1 through S_n. For clarity all sequences $S_1 ... S_n$ are shown as being of the same length.

In order to find the actual alignments the algorithm extends the branches corresponding to exact matches one sequence element at a time while allowing mismatches. Every time a mismatch between the two branches in two trees is considered a number of new potential paths (i.e. pairs of branches) is created. The cost for every branch-pair is re-computed according to some cost function. The algorithm searches branch-pairs in a best-first [27] manner, considering the lowest cost branch-pair first.

The algorithm as outlined above exhibits an undesirable bias: paths having long exact matches as prefixes would be traversed before a potentially lower cost path with a long exact match suffix (compare alignment (B) versus alignment (A) in figure 7). This occurs because paths are extended in one direction only. To search more fairly in both directions the GESTALT algorithm operates on two pairs of trees: a 'forward' pair and a 'reverse' pair. The

1. To avoid overloading the term *path* we will use *branch* to refer to a path from the root to some point in the tree.

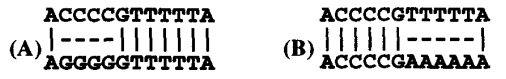

FIGURE 7. Alignments with low-cost suffixes (A) and low-cost prefixes (B).

reverse pair is constructed by reversing each sequence in SEQS and seq. The forward and reverse trees are augmented with pointers to nodes corresponding to the reverses of sequences expressed by every node in the other tree (see figure 8). By traversing the forward and reverse tree the algorithm considers extensions of the best path in both directions. The forward and reverse tree pair can be implemented as a single forward tree augmented with additional pointers to nodes expressing the current node's sequence with single-character prefixes (see figure 8).

Following is a simplified pseudo-code outline of the GESTALT algorithm:

```
// find exact matches first
AlignmentSet <- match(gst(SEQS), tree(seq))
do // process until satisfied
   // consider best alignment
   alignment <- lowestCostAlignment(AlignmentSet)
   // extend it
   newAlignments <- extendAlignment(alignment)
   // add to the match set
   AlignmentSet <- AlignmentSet - alignment + newAlign-
ments
// stop when satisfied
until terminationCriteria(AlignmentSet) = TRUE
```

The extendAlignment procedure is responsible for generating new alignment paths by extending the path corresponding to a given alignment. Path extension is performed by traversing appropriate pointers in the forward/reverse suffix tree pair (figure 8) or the augmented suffix tree (figure 8). The terminationCriteria function is application dependent. It may terminate the search process when a specified number of complete alignments have been completed, when a sub-alignment of a specified length has been found, or when the cost of every alignment under consideration has exceeded a specified maximum.

If one were to dynamically visualize the space of possible alignment paths depicted in figure 6 and highlight paths as they are considered by the GESTALT algorithm the image would resemble a dynamically changing 'sponge', initially filling the entire space (many short exact matches), with gradually emerging diagonal segments (fewer, but longer exact matches), the ends of long diagonal segments becoming 'fuzzy' (multiple mismatching paths being considered) and eventually connecting to form complete alignment paths.

GESTALT simultaneously searches all of the target sequences (SEQS) for subsequences of the pattern sequence (seq). Because the search process takes place in the content domain it has the ability to effectively use a best-first search strategy to reduce the search space.

The GESTALT algorithm outlined above introduces a number of interesting analytical and implementation issues, some of which we will refer to in section 7.

5 Applications of suffix tree and generalized suffix tree operations

By providing a content-addressable way of encoding sequence information suffix trees form the basis of a family of sequence analysis applications. In the following section we outline a simple taxonomy of these applications.

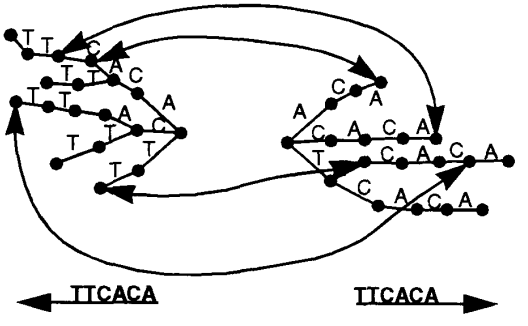

FIGURE 8. A forward/reverse suffix tree pair for the sequence ACACTT. Links corresponding to four selected node pairs have been marked. Note that in order to support the links between the trees the trees do not have to be expanded to one character per edge - compacted (see figure 2) trees with additional edge offset information can be used.

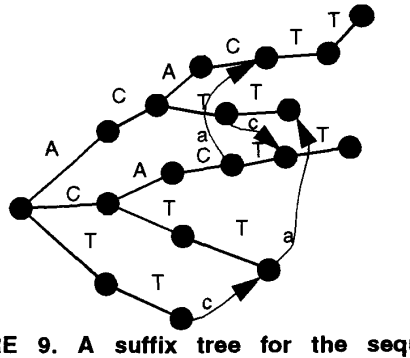

FIGURE 9. A suffix tree for the sequence ACACTT augmented with prefix pointers. Prefix pointers point to nodes expressing the sequence of the source node prefixed with the label of the pointer: a pointer labeled 'a' originating at node corresponding to sequence CAC points to a node corresponding to sequence ACAC. Four selected pointers are shown.

Suffix tree-based sequence analysis applications can be divided into the following major categories:

1.Search applications,

2.Single sequence analysis applications, and

3.Multiple sequence analysis applications.

5.1 Search applications

Suffix trees provide a good tool for performing sequence searches. They can be applied to a number of search applications with varying degrees of speed and accuracy.

Exact match searches: exact searches provide the basic dictionary and spelling-checker functions for a sequence database. They support very fast ('immediate') detection of the presence of a sequence in the database. These capabilities are important during interactive work such as computer-guided cloning experiment design or primer design, where they can allow a system to display a list of all matching sequences in real time, as the clone or primer is being modified by the user. Exact match searches are implemented using the match operation on a generalized suffix tree of the sequences in the database of interest (SEQS) and the sequence in question (seq): ANSWER = match(gst(SEQS), seq)).

Subsequence composition searches: subsequence composition searches are a variant of exact match searches that detect exact matches of all subsequences of a given sequence against any subsequences in the database. Their main application is quick screening of sequences, either for identification purposes or for filtering of data produced by sequencing machines. The filtering process can increase the quality of sequence data by detecting sample contamination, eliminating vector sequences and bringing 'unusual' matches to the attention of researchers. Subsequence composition searches are implemented using the tree matching operation: ANSWER = match(gst(SEQS), tree(seq)), where seq is the sequence in question and SEQS the set of sequences in the database.

Homology searches: homology searches are performed when sequences similar but not necessarily identical to a given sequence are to be found. The GESTALT tree matching algorithm described in section 5 can be used for homology searches. The cost function used by the algorithm can be adjusted to detect only sequences within a specified distance from the pattern sequence. We believe that homology search applications based on this approach may be able to outperform currently used tools such as FASTP and FASTA [22]. We are currently developing a GESTALT-based sequence search tool.

5.2 Single sequence analysis applications

Analysis of a suffix tree constructed for any given sequence (tree(seq)) can reveal a wealth of interesting information about the sequence, such as internal repeats, shortest unique subsequence and longest common subsequence. Internal repeats in the sequence are represented by internal (non-leaf) nodes in the tree. The shortest unique subsequence is determined by finding the shortest tree branch with a single descendant node. The longest common subsequence is determined by finding the longest tree branch with more than one descendant node. tree(seq) can also provide a measure of the information content of the sequence [12].

A suffix tree of a single sequence allows all occurrences of any number of short subsequences to be easily detected. This method can be used for enzyme cut site determination: CutSites = match(gst(SEQS), tree(seq)), where SEQS is the set of sequences corresponding to enzyme cut sites.

5.3 Multiple sequence analysis applications

Constructing a single generalized suffix tree for a set of sequences allows all of the sequences to be analyzed simultaneously. A GST can be analyzed in a manner similar to a single-sequence suffix tree. The answers obtained, however, relate to the entire set of sequences stored in a given GST. Specifically, the presence of a given sequence fragment in *any* of the sequences stored in a GST can be determined using the match operation. Efficient detection of common subsequences within a set of sequences forms the basis for contig reassembly applications, among others. Using basic tree operations, the process of contig reassembly can be reduced to the following algorithm:

```
// construct the initial gst
T = gst(GELS)
// process until a single contig is produced
while memberCount(T) > 1 do
   // pick best candidates for merging
   c1, c2 = bestMergeCandidates(T)
   // remove them from the tree
   T = T - tree(c1) - tree(c2)
   // add a merged sequence to the tree
   T = T + gst(merge(c1, c2))
endwhile
```

The bestMergeCandidates function is an automatic (using the match operation or the GESTALT algorithm) or human-guided function responsible for selecting the best candidates for merging. The merge function merges two selected sequences into one. The rest of the loop body maintains a generalized suffix tree of sequence contigs of increasing lengths. Upon completion, the algorithm produces a generalized suffix tree containing a single contig. This process provides natural support for merging of mul-

tiple sequence projects. Also, the high efficiency of the process allows experimentation with various **bestMergeCandidates** and **merge** functions. We have implemented a simple sequence reassembly engine similar to the one outlined above using our basic tree operator package.

Another application for which generalized suffix trees provide a useful framework is multiple sequence alignment. By analyzing a generalized suffix tree and choosing the subsequences common to the largest number of sequences[1] as the initial anchor points, the search space of the problem can be greatly reduced. We are currently developing a multiple sequence alignment system using the basic tree operator package.

6 Implementation issues

In this section, we outline three major issues in implementing suffix tree based applications: implementation of basic tree operations; construction, storage and management of large persistent generalized suffix tree structures; and actual applications. We also comment on our experiences with such implementations.

6.1 Basic tree operation implementations

In the course of our experiments with suffix trees and generalized suffix trees, we have developed several implementations of the basic set of tree operations described in section 3. These applications range in stage of completeness from alpha-test to production tools. We found that the data structures and algorithms involved are complicated. Thus, well-documented, defensive coding techniques are essential for producing correct, robust implementations.

The initial tree operation package we have developed supports suffix tree and generalized suffix tree creation and disk-based storage of DNA sequences. It also includes modules supporting compression of disk-based GSTs and transparent access to both compressed and uncompressed GST files. The package is written in C and runs on Sun SPARC workstations.

In order to facilitate experiments with other suffix tree operations and symbol alphabets, we have implemented a C++ GST object package. It supports the complete set of tree operations described in section 3, including in-place versions of the GST addition and subtraction operators. The package supports symbol alphabets of arbitrary size.

6.2 Persistent generalized suffix tree maintenance for large volumes of sequence data

Our early analysis and experiments with GSTs convinced us that only space-efficient, disk-based storage of GST structures will make their application feasible. We have implemented a system for construction and disk-based storage of GSTs for large (up to GenBank size) sets of DNA sequences in structures we call H-trees (H stands for *huge*). H-trees are stored in a fashion allowing binary merging of disk-based H-trees in limited main memory, as described in section 3.2. H-trees of append-only sequence databases can be updated by merging the H-trees of new sequences with the H-tree of the extant database.

In order to support large databases, we had to make the system more space efficient. We have developed a compression scheme satisfying two basic criteria: transparency to existing tools, and support of fast, on-the-fly decompression. The second requirement implies a local or node-level compression scheme. We have implemented a system satisfying these criteria and reduced the space requirement of compressed H-trees from about 100 bytes per element to 10-20 bytes per element (nucleotide or amino acid). These compression levels make construction and maintenance of GSTs for today's large databases feasible

We have implemented the binary merge procedure on a workstation cluster and used it to construct a single GST for the rodent section of the GenBank database. The rodent section of the GenBank version used contained 17,776,128 bases of DNA in 15,930 sequences. The resulting tree contained 26,854,572 nodes and occupied approximately 400M bytes of disk space.

Support for persistent GST structures is being added to the C++ GST object package. This package will also incorporate the H-tree compression method.

6.3 Tree operation-based applications

We have implemented two tools utilizing H-tree structures. **hscan** is a DNA sequence dictionary tool. It provides instant identification of a given DNA sequence based on exact matching of sequences using the **match** operation. **hftscan** is a more sophisticated search tool matching a suffix tree of a given DNA pattern sequence against an H-tree of the database and analyzing the exact matches of subsequences detected by the **match** operation in order to determine the set of sequences closest to the pattern. The weighing of individual matches between the two trees based on their length and relative 'uniqueness' can be adjusted.

The new C++ implementation of a GST object package has provided us with an excellent platform for prototyping

1. The number of sequences a given subsequence belongs to is also equivalent to the Color Set Count (CSC) problem. In [7] an efficient (O(n)) solution to the CSC problem using suffix trees is presented. When the merge-based GST construction technique is used, however, the CSC information is derived 'free' as a side effect of the construction process.

and experimentation with new applications. We currently have a simple prototype of a contig reassembly engine operating on top of the C++ package. GST-based representation and manipulation of sequences during the contig reassembly process has greatly simplified the process and its implementation.

7 Future work

We believe that feasible, efficient implementations of GST structures for large volumes of sequence data will provide the basis for a variety of new research tools and produce exciting results. Three major areas of future research effort can be identified.

First, the implementation techniques of GST-based data structures have to be perfected. This includes further advances in compression and manipulation of secondary storage-based structures. Issues of integrating these structures with existing and future sequence databases will have to be considered, possibly leading to a development of a GST server or layer operating on top of sequence databases. Suffix trees cannot be maintained on a per-tool basis because of the volume of the raw data and resulting computational costs of creating, manipulating and maintaining these data structures. They have to be treated as separate data repositories, similar to other databases, or as specialized server layers linked to databases and providing the services needed by the various tools.

Second, GST-based structures and algorithms such as GESTALT provide the basis for a family of new, high performance sequence search tools. The unstructured nature of searches performed by such tools appears to make them good candidates for studying dynamic load balancing issues in their parallel implementations.

Finally, the high levels of performance of GST-based tools may allow some machine reasoning techniques to be effectively applied to sequence analysis. In addition, new applications of position-independent sequence data processing may arise. An interesting example of such an application is 3-D protein structure analysis. In [26], it has been shown that 3-D protein structures can be represented as symbols of a limited size alphabet. By translating protein structures into this representation, it is possible to use suffix tree-based analysis techniques. Specifically, it may be possible to perform 3-D protein structure searches and alignments[1] using the GESTALT algorithm.

1. Alignment of actual 3-D protein structures has been in the past performed using dynamic programming techniques [34].

8 Bibliography

[1] Altschul, Stephen F. and Erickson, Bruce W., *Optimal Sequence Alignment Using Affine Gap Costs*, Bulletin of mathematical biology, Vol. 48, No. 5/6, pp. 603-616, 1986.

[2] Altschul, Stephen F. and Erickson, Bruce W., *A Nonlinear Measure Of Subalignment Similarity And Its Significance Levels*, Bulletin of mathematical biology, Vol. 48, No. 5/6, pp. 617-632, 1986.

[3] Altschul, Stephen F. and Erickson, Bruce W., *Locally Optimal Subalignments Using Nonlinear Similarity Functions*, Bulletin of mathematical biology, Vol. 48, No. 5/6, pp. 633-660, 1986.

[4] Bays, J. C., The Complete PATRICIA, Ph.D. dissertation, University of Oklahoma, 1974.

[5] Chan, S.C., Wong, A. K. C. and Chiu, D. K. Y., *A Survey of Multiple Sequence Comparison Methods*, Bulletin of Mathematical Biology, Vol. 54, No. 4, pp. 563-598, 1992.

[6] Chao, Kun-Mao, Hardison, Ross C. and Miller, Webb, *Constrained Sequence Alignment*, Bulletin of Mathematical Biology, Vol. 55, No. 3, pp. 503-524, 1993.

[7] Chi Kwong Lui, Lucas, *Color Set Size Problem with Applications to String Matching*, Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching, Tucson, Arizona, USA, April/May 1992.

[8] McCreight, E. M., *A Space Economical Suffix Tree Construction Algorithm*, JACM, 23, 262-272, 1976.

[9] Davison, Dan, *Sequence Similarity ('Homology') Searching For Molecular Biologists*, Bulletin of Mathematical Biology, Vol. 47, No. 4, pp. 437-474, 1985.

[10] Gotoh, Osamu, *Consistency of Optimal Sequence Alignments*, Bulletin for Mathematical Biology, Vol. 52, No. 3, pp. 509-525, 1990.

[11] Gotoh, Osamu, *Optimal Sequence Alignment Allowing for Long Gaps*, Bulletin of Mathematical Biology, Vol. 52, No. 3, pp. 359-373, 1990.

[12] Grassberger, Peter, *Estimating the Information Content of Symbol Sequences and Efficient Codes*, IEEE Transactions on Information Theory, Vol. 35, No. 3, May 1989

[13] Horowitz, Ellis and Sahni, Sartaj, *Fundamentals of Data Structures*, Computer Science Press, 1987.

[14] Landau, Gad M. and Vishkin, Uzi, *Fast Parallel and Serial Approximate String Matching*, Journal of Algorithms, 10, pp.157-169, 1989.

[15] Landau, Gad M., Vishkin, Uzi and Nussinov, Ruth, Locating alignments with k differences for nucleotide and amino acid sequences, CABIOS, Vol. 4, No. 1, 1988, pp. 19-24.

[16] Lawrence, Charles B., Goldman, Daniel A. and Hood, Robert T., *Optimized Homology Searches of the Gene and Protein Sequence Data Banks*, Bulletin of Mathematical Biology, Vol. 48, No. 5/6, pp. 569-583, 1986.

[17] Martinez, H. M., *An Efficient Method for Finding Repeats in Molecular Sequences*, Nucleic Acids Research, 11, pp. 4629-4634, 1983.

[18] Miller, Webb and Myers, Eugene, *Sequence Comparisons With Concave Weighing Functions*, Bulletin of Mathematical Biology, Vol. 50, No. 2, pp. 97-120, 1988.

[19] Mott, Richard, *Maximum-Likelihood Estimation of the Statistical Distribution of Smith-Waterman Local Sequence Similarity Scores*, Bulletin of Mathematical Biology, Vol. 54, No. 1, pp. 59-75, 1992.

[20] National Center for Biotechnology Information, *Genetic Sequence Data Bank, Release 76.0*, 15 April 1993.

[21] Needleman, Saul B. and Wunsch, Christian D., *A General method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins*, Journal of Molecular Biology, 48, pp. 443-453, 1970.

[22] Pearson, W.R. *Rapid and Sensitive Sequence Comparison with FASTP and FASTA*, Methods in Enzymology, vol 183, pp. 63-98, 1990.

[23] Powell, Patrick A., *Using and Constructing the Suffix Tree Index Structure*, U. of Minnesota Computer Science Department, Technical Report TR 89-90

[24] Powell, P.A., P. Bieganski, E. Shoop [1989]. *X11 - Based Tools for Network Access to and Comparison of DNA Sequence Data*. Presentation at MacroMolecules, Genes and Computers, Chapter Two, Waterville Valley, New Hampshire.

[25] P.A. Powell [1990]. *FASTSIM: A New Algorithm for Rapid Sequence Similarity Determination*. Presentation at American Medical Informatics Association First Annual Research Conference: Computers, Molecular Biology and Medicine. Snowbird, Utah.

[26] Prestrelski, S.J., Williams, A.L. Jr. and Liebman, M.N., *Generation of a substructure library for the description and classification of protein secondary structure - Overview of the methods and results*, Proteins, Dec;14(4):430-9, 1992

[27] Rich, Elaine, *Artificial Intelligence*, McGraw-Hill Book Company, 1983.

[28] P. Rice, Elliston, K. and Gribskov, M., in Sequence Analysis Primer, M. Gribskov and J. Devereux, eds., Stockton Press, New York, NY, 1991

[29] Shasha, Dennis and Zhang, Kaizhong, *Fast Algorithms for the Unit Cost Editing Distance between Trees*, Journal of Algorithms, 11, pp. 581-621, 1990.

[30] Szpankowski, Wojciech. *Probabilistic Analysis of Generalized Suffix Trees*, Combinatorial Pattern Matching, Third Annual Symposium, Proceedings, Springer Verlag, 1992, 1-14.

[31] Ukkonen, Esko, Finding *Approximate Patterns in Strings, Journal of Algorithms*, 6, pp. 132-137, 1985.

[32] Waterman, Michael S., *Efficient Sequence Alignment Algorithms*, Journal of Theoretical Biology, 108, pp. 333-337, 1984.

[33] Waterman, M. S., Smith, T. F., and Beyer, W. A., *Some Biological Sequence Metrics*, Advances in Mathematics, 20, pp. 367-387, 1976.

[34] Zuker, M. and Somorjai, R. L., *The Alignment of Protein Structures In Three Dimensions*, Bulletin for Mathematical Biology, Vol. 51, No. 1, pp. 55-78, 1989.