OASIS: An Online and Accurate Technique for Local-alignment Searches on Biological Sequences

Colin Meek Jignesh M. Patel Shruti Kasetty

University of Michigan
1301 Beal Avenue; Ann Arbor, MI 48109-2122; USA
{meek, jignesh, skasetty}@eecs.umich.edu

Abstract

A common query against large protein and gene sequence data sets is to locate targets that are similar to an input query sequence. The current set of popular search tools, such as BLAST, employ heuristics to improve the speed of such searches. However, such heuristics can sometimes miss targets, which in many cases is undesirable. The alternative to BLAST is to use an *accurate* algorithm, such as the Smith-Waterman (S-W) algorithm. However, these accurate algorithms are computationally very expensive, which limits their use in practice. This paper takes on the challenge of designing an *accurate and efficient* algorithm for evaluating local-alignment searches.

To meet this goal, we propose a novel search algorithm, called OASIS. This algorithm employs a dynamic programming A*-search driven by a suffix-tree index that is built on the input data set. We experimentally evaluate OASIS and demonstrate that for an important class of searches, in which the query sequence lengths are small, OASIS is more than an order of magnitude faster than S-W. In addition, the speed of OASIS is comparable to BLAST. Furthermore, OASIS returns results in decreasing order of the matching score, making it possible to use OASIS in an *online* setting. Consequently, we believe that it may now be practically feasible to query large biological sequence data sets using an accurate local-alignment search algorithm.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

1 Introduction

In the new and emerging area of functional genomics and proteomics, scientists often have to query biological sequences. Frequently, the scientist has a protein or nucleotide sequence that they want to match against a target database of known sequences. A common technique for determining a match is to compute a local alignment score between the query and a target sequence, with a high score implying that the two sequences are biologically related.

BLAST [1–3] is a popular tool for evaluating local sequence alignments. In BLAST, the input query is transformed into a set of fixed-length words that are matched against the database. These preliminary matches are then extended to the left and the right to determine whether they meet a user-specified score criteria, using some ceiling on the probability that such a match would occur by chance. While this heuristic filter has been well optimized to reduce the chances of missing any matching results, it is not 100% accurate. Consequently, some matching results that meet the specified score criteria may not be returned as part of the result. Given the high-stakes and cost involved in many life sciences experiments, such as drug discovery or gaining a better understanding of the causes of a disease, it is natural to wonder if it is possible to design a fast and accurate algorithm for local sequence alignment that is guaranteed to not miss any target matches given some similarity metric.

There are a few known algorithms for *accurately* identifying local alignments [34, 36], but these algorithms are computationally very expensive, limiting their use in practice. The most popular of these accurate search algorithms is the Smith-Waterman (S-W) algorithm [36]. S-W has been shown to be considerably more sensitive than BLAST [28, 35] in many cases, such as when a scientist is looking for a distant homology match. In such cases, often the scientist has to try a number of different BLAST queries with different parameters (such as the E-value). This process can be very cumbersome, and often requires a great deal of experience to get a feel for the "right" values of BLAST parameters. In such cases the use of S-W is desirable. However, the prohibitively high computational cost

of S-W has prevented its wide-spread use, especially for large data sets. Attempts have been made to address this performance issue by building special hardware to evaluate S-W [13, 35], but specialized hardware tends to be expensive as compared with commodity microprocessors.

In this paper, we take a novel approach to this problem. We design an algorithm called OASIS (an **O**nline and **A**ccurate **S**earch technique for **I**nferring local-alignments on **S**equences) for fast and accurate local alignment searches on sequences. OASIS can run on commodity microprocessors, and like S-W, it is an accurate algorithm.

OASIS employs a dynamic programming search technique which is driven by traversing a suffix tree index constructed on the sequence data set. By carefully managing the layout of the suffix tree in disk blocks, OASIS can be efficient even on large data sets. Furthermore, the OASIS search technique employs a *best-first* (A*) search strategy as it descends the suffix tree. As partial matches are computed, the search also computes an upper-bound on the cost of matching the remaining portion of the query. At any point in the search, OASIS picks the path capable of producing the highest scoring alignment. This design guarantees that OASIS will return alignments in the order of the alignment score. This *online* characteristic is ideal when the scientist wants to abort the query after seeing the top few matches. In contrast, both BLAST and S-W must compute the entire query before presenting any results.

In querying biological sequences, search tools often distinguish between long queries and short queries [6]. For example, BLAST classifies queries on nucleotide databases into short and long queries, with short queries being less than 20 symbols in length. Similarly, queries on protein databases are classified into short and long queries, with short queries being less than 15 symbols long.

Short query sequences are often used in practice; for example, queries using peptides, which are short protein sequences, are often used to find matching proteins that have a similar peptide [6]. OASIS is especially suitable in such cases. In this paper, we compare OASIS with S-W using an actual protein data set, and show that for short query sequences, OASIS is at least an order of magnitude faster than S-W.

We also compare the performance of OASIS with the popular heuristic search tool BLAST, and show that the performance of OASIS is often comparable to BLAST.

Finally, we experimentally evaluate the online characteristics of OASIS, and show that OASIS produces the first set of results very quickly, which makes it ideal for use in online environments.

The key contribution of this paper is the design and evaluation of a novel *accurate* search algorithm for evaluating local-alignments. As this paper demonstrates, with OASIS it may now be practically feasible to accurately evaluate local-alignments for short query sequences against actual biological data sets.

The remainder of this paper is organized as follows: We cover some background information in Section 2, and describe the OASIS algorithm in Section 3. Experimental evaluations are presented in Section 4, and related work is discussed in Section 5. Finally Section 6 contains our conclusions and directions for future work.

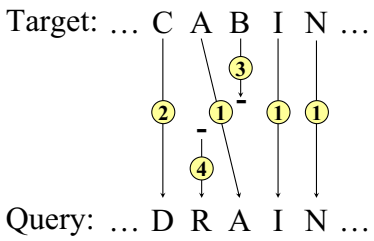

Figure 1: Example of Local Alignment

2 Background

In this section we present some background material crucial to understanding the OASIS algorithm.

2.1 Local Alignment

Given two sequences of symbols $Q = q_1 q_2 ... q_m$ and $T = t_1 t_2 ... t_n$, a local alignment is some way of "lining up" any two sub-sequences of Q and T. Figure 1 shows an example of one such local alignment. This figure illustrates the three types of local alignment operation:

- *Replacement* with either the same symbol or a different symbol (see labels 1 and 2 in Figure 1).
- *Deletion* allows us to *skip* a symbol in the *target* (see label 3 in Figure 1).
- *Insertion* allows us to *skip* a symbol in the *query* (see label 4 in Figure 1).

Alignments are given scores based on the sum of the scores of each operation involved in the alignment. Every operation is generalized to a replacement of "$\alpha \rightarrow \beta$", where insertions are represented as "$- \rightarrow \beta$" and deletions as "$\alpha \rightarrow -$". The score of an operation is denoted $S(\alpha \rightarrow \beta)$. A substitution matrix S stores the scores, so the cost of an operation $\alpha \rightarrow \beta$ is simply $S_{\alpha,\beta}$. Table 1 shows a specific example of a substitution matrix, which is called a *unit* edit distance matrix (since it has scores of 1 for exact matches, and -1 otherwise).

	A	**C**	**G**	**T**	**–**
A	1	-1	-1	-1	-1
C	-1	1	-1	-1	-1
G	-1	-1	1	-1	-1
T	-1	-1	-1	1	-1
–	-1	-1	-1	-1	-

Table 1: "Unit" Edit Distance Matrix

2.1.1 Notations

Throughout this presentation, we use the notations introduced above. Q will be used to represent an input query of length m, and q_i will represent the i^{th} symbol in the query sequence. Similarly, T will be used to represent the target sequence of length n, and t_j will represent the j^{th} symbol.

2.2 Smith-Waterman (S-W) Algorithm

The Smith-Waterman algorithm [36] finds the local alignment between two sequences with the maximum possible score using a dynamic programming approach that runs in $O(mn)$ time. This algorithm generates an m by n matrix G, where each entry $G_{i,j}$ stores the score of the maximum alignment between a query (Q) and target (T) ending at q_i and t_j. Each entry $G_{i,j}$ is computed as follows:

$$G_{i,j} = \max \left\{ \begin{array}{ll} 0 & \text{“start over”} \\ G_{i-1,j-1} + S(q_i \rightarrow t_j) & \text{Replacement} \\ G_{i-1,j} + S(q_i \rightarrow -) & \text{Insertion} \\ G_{i,j-1} + S(- \rightarrow t_j) & \text{Deletion} \end{array} \right\} \quad (1)$$

For example, consider a query $Q =$ TACG against a target $T =$ AGTACGCCTAG. Table 2 outlines the execution of S-W, using the substitution matrix shown in Table 1. In this example, $\nwarrow$ indicates that a replacement produced the maximum score, $\leftarrow$ indicates deletion, and $\uparrow$ indicates insertion. The bold score entry indicates the maximum score alignment. By backtracking using the bold arrows, we uncover the highest scoring alignment, TACG $\rightarrow$ TACG, which has a score of 4.

	A	G	T	A	C	G	C	C	T	A	G
T	0	0	$\nwarrow$1	0	0	0	0	0	$\nwarrow$1	0	0
A	$\nwarrow$1	0	0	$\nwarrow$2	$\leftarrow$1	0	0	0	0	$\nwarrow$2	$\leftarrow$1
C	0	0	0	$\uparrow$1	$\nwarrow$3	$\leftarrow$2	$\nwarrow$1	$\nwarrow$1	0	$\uparrow$1	0
G	0	0	0	0	0	$\nwarrow$**4**	$\leftarrow$3	$\leftarrow$2	$\leftarrow$1	0	$\nwarrow$2

Table 2: G Matrix Built Using S-W

2.3 Generalized Suffix Tree Structure

In this section, we outline the basics of the suffix tree, the index structure that is used in OASIS. A suffix tree is a PATRICIA trie [27] that represents every suffix in the input sequence. Edges in the tree are labeled with strings from the alphabet, A, of sequence symbols. Each node can have only one outgoing edge labeled with a symbol from the alphabet. Consequently, the branching degree of the tree is at most $|A|$, the cardinality of the symbol alphabet. A *compact* suffix tree is one in which every node is either the root, a branching node or a leaf. In practice, this means that all nodes with only one successor are collapsed into their parents, creating multi-symbol arcs.

The tree in Figure 2 is an example of a compact suffix tree built on the sequence T = AGTACGCCTAG. \$ is the terminal symbol, which indicates the last position in the sequence. Also, in this figure each node contains a number, followed by a letter denoting if the node is a leaf (L) or a non-leaf (N). For the non-leaf nodes, the number in the node label is arbitrary. However for the leaf-nodes, the number in the node label points to the position in the input sequence that matches the start of the *path* to that node. A path is simply the concatenation of the strings labeling the edges traversed. Paths can be defined for both leaf and non-leaf nodes. For example, in Figure 2, $path(\texttt{8L}) =$ TAG\$, and $path(\texttt{5N}) =$ AG.

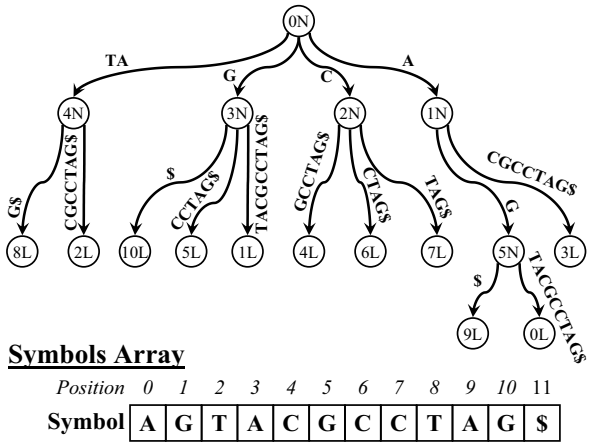

Figure 2: Suffix Tree on the Sequence AGTACGCCTAG

The suffix tree structure can also be used to index multiple sequences by appending the terminal symbol to each sequence, and building a single suffix tree over the concatenated sequences.

2.3.1 Finding Exact Matches in a Suffix Tree

Given a query $Q = q_1 q_2 ... q_m$, it is straightforward to confirm its presence in a sequence T represented in a suffix tree. It is simply a matter of tracing a path, defined by the query, from the root of the tree until either the query is consumed, or no match is found. If the query is consumed, this means that a match has been found; otherwise, the target does not contain the query as a subsequence. As an example, consider the query “TACG” posed against the structure in Figure 2. Starting from the root, we can match the first two symbols along the arc 4N, and the subsequent two symbols on the arc 2L. This indicates that this substring is present in the target sequence, beginning at position 2 in T. In general, once a match has been found, its location(s) in the target sequence can be identified by descending to all leaf descendants of the matching node.

3 OASIS

The basic idea of the OASIS algorithm is to perform a best-first search for local alignments, where the expansion of the nodes in the search space is driven by a suffix tree, and nodes are expanded in a manner closely related to S-W.

The procedure is outlined in Algorithm 1. OASIS, and its initialization function, take as input the following parameters:

- T: a suffix tree built on the sequence database
- Q: a query, $q_1 q_2 \ldots q_m$
- S: an arbitrary substitution matrix, and
- $minScore$: a minimum alignment score.

Algorithm 1 $OASIS(T, Q, S, minScore)$

$H, PQ = Initialize(T, Q, S, minScore)$
while $\neg Empty(PQ)$ **do**
  $searchNode \leftarrow PQ.Pop()$
  if $searchNode.tag = VIABLE$ **then**
    for $successor \in Successors(searchNode.sp)$ **do**
      $newNode \leftarrow Expand(searchNode, successor, H, Q, S, minScore)$
      if $newNode.tag = VIABLE \vee newNode.tag = ACCEPTED$ **then**
        $PQ.Add(newNode)$
      end if
    end for
  else if $searchNode = ACCEPTED$ **then**
    Return on-line all sequences containing $Path(searchNode.sp)$
  end if
end while

All alignments with scores greater than or equal to $minScore$ are returned, in reverse order of score.

Each search node in the OASIS algorithm corresponds to a node in the suffix tree. Search nodes represent partial alignments between the query and portions of the database sequence ending at the $path$ to the corresponding suffix tree node. Each search node stores scores for alignments ending at each position in the query, and we compute the maximum score that is possible for aligning the remaining portions of the query. The sum of these scores is used to organize the search nodes in a priority queue PQ. In each expansion step, OASIS picks the node at the head of the priority queue and explores all the children of the selected node. The close correspondence between the search expansion and the suffix tree implies that this step corresponds to exploring all the children of the corresponding suffix tree node. Since OASIS always expands the node at the head of the priority queue, it is a best-first search technique like A*.

Search nodes contain the following fields:

- sp: A pointer to a node in the suffix tree. This pointer is used to maintain the correspondence between the search node and its corresponding suffix tree node.
- Z: A vector $\begin{bmatrix} z_0 \\ z_1 \\ \cdots \\ z_m \end{bmatrix}$, where z_i is the score of the strongest alignment between the sequence $path(sp)$ and any subsequence of Q ending at q_i. z_i is set to $-\infty$ if the alignment has been pruned. This vector corresponds, roughly speaking, to a column of the S-W matrix shown in Table 2.
- $maxScore$: The maximum score alignment found along this path. This corresponds to the score of the strongest alignment between any prefix of the sequence $path(sp)$ and any subsequence of the query.

Algorithm 2 $Initialize(T, Q, S, minScore)$

{compute maximum alignment score beginning after each position in query}

$$H \leftarrow \begin{bmatrix} h_0 \leftarrow h_1 + \max_{\forall \beta} S(q_1 \rightarrow \beta) \\ \cdots \\ h_i \leftarrow h_{i+1} + \max_{\forall \beta} S(q_{i+1} \rightarrow \beta) \\ \cdots \\ h_m \leftarrow 0 \end{bmatrix}$$

{initialize first entry in priority queue}
$seed.sp \leftarrow Root[T]$

$$seed.Z \leftarrow \begin{bmatrix} \cdots \\ z_i \leftarrow \begin{cases} 0 & \text{if } h_i \geq minScore \\ -\infty & \text{if } h_i < minScore \end{cases} \\ \cdots \end{bmatrix}$$

$seed.maxScore \leftarrow Seed.g \leftarrow \max(seed.Z)$
$seed.f \leftarrow \max(H)$
if $seed.maxScore = 0$ **then**
  $seed.tag \leftarrow VIABLE$
  $PQ \leftarrow \{seed\}$
else
  $PQ \leftarrow \{\}$
end if
Return H, PQ

- f: The maximum possible score that can be achieved by further expanding this node.
- g: The maximum score in Z, or the best score ending at node sp in the suffix tree.
- tag: Indicates the status of the search node, and can take on one of three values: $ACCEPTED$, $VIABLE$, and $UNVIABLE$. A node is tagged as accepted when the strongest possible alignment of the query with this node or any of its descendants has been found, and it passes the $minScore$ threshold. When such nodes reach the top of the priority queue, we return the alignment on-line, since we can be certain, by the ordering of the queue, that no subsequent alignment will be stronger. A node is tagged as viable when a stronger alignment other than that already found along this path is possible, and the $minScore$ threshold can be reached. A node is tagged as unviable if no possible extension of this node can result in an alignment with the necessary strength. Unviable nodes are pruned from the search tree, while accepted and viable nodes are added to the priority queue.

The priority queue is ordered by the f-value, so that a node is only expanded when it can be guaranteed that no other node on the search frontier can produce a stronger alignment. After seeding the priority queue, OASIS expands the successors of the search node at the top of the heap using the algorithm outlined in Section 3.2. The expansion of a node may add new nodes to the priority queue, and the search terminates when the priority queue is empty (see Algorithm 1 for details).

In practice, several approaches can be adopted to the reporting of alignments. In the current implementation, we

duplicate the behavior of S-W, reporting only the single strongest alignment for each sequence in the database.

3.1 Initialization

OASIS begins by computing a "heuristic vector" H and initializing the priority queue with an entry representing the root of the suffix tree ($seed$). These steps are outlined in Algorithm 2. Each entry h_i in the vector H represents the maximum possible alignment score of $q_{i+1}q_{i+2}\ldots q_m$ with any arbitrary target (recall that the query $Q = q_1q_2\ldots q_m$). Calculating these values is trivial, assuming non-positive values for insertions and deletions. H_m is set to zero, since the leftover portion of the query is the empty string. We can then inductively calculate the remaining values: $H_{i-1} = H_i +$ the maximum score for the replacement of q_{i-1}.

The $seed$ search node corresponds to the root of the suffix tree ($seed.sp \leftarrow Root[T]$), which has a path length of 0. As a result, the alignment values stored in the Z vector are set to 0, or $-\infty$ in cases where the alignment can be pruned. An alignment can be pruned when the entry h_i is less than the value of $minScore$, which implies that this starting alignment point cannot yield a meaningful alignment.

3.2 Expanding arcs in the search tree

The $Expand$ function (see Algorithm 3) is the core of the OASIS algorithm. Essentially, this function fills in a portion of the S-W matrix for a particular node of the suffix tree. In addition to the scoring matrix S, the query sequence Q, and $minScore$ (described above), this function takes the following arguments:

- psn: the parent search node psn;
- stn: the suffix tree node being expanded, and;
- H: the heuristic vector computed by the initialization function.

$Expand$ returns an appropriately tagged search node corresponding to stn.

The relevant portion of the S-W expansion matrix (G) aligns the portion of the target labelling the incoming arc of the suffix tree node, denoted $C = c_1c_2\ldots c_n$. It is seeded with the last column of the matrix filled for the parent node ($psn.Z$). When the current search node ($node$) is returned, we only store the final column of G (in the Z field), as it is the only information relevant to the descendants of the search node.

Entries in G are filled inductively, as with S-W, with the exception that we do not permit a reset to 0 (outside of the seed entry). In S-W such resetting (see Equation 1), corresponds to starting a new alignment. Such resetting is not required in OASIS, because from the root of the suffix tree, *all* possible target alignments are considered. As a result resetting the score to 0 would duplicate the work done elsewhere. Since the algorithm explicitly considers all starting alignments for the query in the column expansion, and because of the fundamental property of the suffix

Algorithm 3 $Expand(psn, stn, H, Q, S, minScore)$

$node.sp \leftarrow stn$
$node.maxScore \leftarrow psn.maxScore$
$C = c_1c_2\ldots c_n \leftarrow IncomingPath(stn)$
$G \leftarrow$

$$\begin{bmatrix} g_{0,0} \leftarrow psn.z_0 & g_{0,1} \leftarrow -\infty & \ldots & g_{0,n} \leftarrow -\infty \\ g_{1,0} \leftarrow psn.z_1 & g_{1,1} & \ldots & g_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{m,0} \leftarrow psn.z_m & g_{m,1} & \ldots & g_{m,n} \end{bmatrix}$$

for $j \leftarrow 1\ldots n$ **do**
  for $i \leftarrow 1\ldots m$ **do**

$$g_{i,j} = \max\begin{pmatrix} g_{i-1,j-1} + S(q_i \rightarrow c_j), \\ g_{i-1,j} + S(q_i \rightarrow -), \\ g_{i,j-1} + S(- \rightarrow c_j) \end{pmatrix}$$

    if $g_{i,j} \leq 0 \vee g_{i,j} + h_i \leq node.maxScore \vee g_{i,j} + h_i < minScore$ **then**
      $g_{i,j} \leftarrow -\infty$
    end if

    if $g_{i,j} > node.maxScore$ **then**
      $node.maxScore \leftarrow g_{i,j}$
    end if

  end for

  $gValues \leftarrow$ the j^{th} column of G
  $fValues \leftarrow gValues + H$

  if $node.maxScore \geq \max(fValues) \wedge node.maxScore \geq minScore$ **then**
    $node.tag \leftarrow ACCEPTED$
    $node.f \leftarrow node.g \leftarrow node.maxScore$
    Return $node$
  else if $max(fValues) < minScore$ **then**
    $node.tag \leftarrow UNVIABLE$
    Return $node$
  end if

end for

$node.f \leftarrow \max(fValues)$
$node.g \leftarrow \max(gValues)$

if $IsALeaf(stn)$ **then**
  if $minScore \leq node.maxScore$ **then**
    $node.tag \leftarrow ACCEPTED$
    $node.f \leftarrow node.g \leftarrow node.maxScore$
  else
    $node.tag \leftarrow UNVIABLE$
  end if
else
  $node.Z \leftarrow gValues$
  $node.tag \leftarrow VIABLE$
end if

Return $node$

tree that guarantees every subsequence is the prefix of some path in the tree, all starting alignments are considered and the algorithm never misses any local alignments.

After computing a value $g_{i,j}$, OASIS performs *alignment pruning* (setting the score to $-\infty$) on alignments positions that are no longer viable. This process allows OASIS to discard specific alignment elements of a search node, which are either unviable or covered by other search paths. This pruning occurs in three cases:

1. Non-positive alignment scores ($g_{i,j} \leq 0$): To avoid duplicating work nodes with negative alignment scores are pruned. Consider a partial local alignment between a portion of a query $q_a q_{a+1} ... q_b$ and the suffix $t_c t_{c+1} ... t_d$. If the maximum alignment score beginning at $q_a \rightarrow t_c$ and ending at $q_b \rightarrow t_d$ is negative, then we can guarantee that the score of any alignment between $q_a ... q_b ...$ and $t_c ... t_d ...$ will have a lower score than the alignment between $q_{b+1} ...$ and $t_{d+1} ...$. This second alignment will be expanded along another tree path, so it is undesirable to maintain the (guaranteed) sub-optimal first alignment along the current search path.
2. Existing alignment is as good ($g_{i,j} + h_i \leq node.maxScore$): Based on an *optimistic* heuristic, we can not find an extension to this alignment with a better score than the strongest alignment already found along this search path. Some ancestor of the search node has an alignment score $maxScore$, so extensions to this alignment can not produce optimal scores for this portion of the database.
3. Threshold failure ($g_{i,j} + h_i < minScore$): No possible extension to this alignment can be equal to or greater than the $minScore$ threshold.

Under certain conditions, we can stop extending alignments along a path. Using the heuristic values H, we can determine an upper bound on the best possible alignment score resulting from further exploration, stored in the f field. If this value is less than or equal to a score already found along the path ($f \leq maxScore$), we can cease expansion, and return $ACCEPTED$ or $UNVIABLE$ depending on whether it meets the $minScore$ threshold. When improved scores are possible ($f > maxScore$), we must decide if further expansion is warranted: if f is below $minScore$, we can immediately return the node as "$UNVIABLE$", otherwise it is tagged as "$VIABLE$".

3.3 Example Execution of OASIS

Next we illustrate the execution of the OASIS algorithm using an example. For this example, we return to the target sequence and the suffix tree shown in Figure 2. The query used in this example is TACG. The minimum alignment score ($minScore$) is set to 1, and the substitution matrix shown in Table 1 is used in this example.

The priority queue is initialized with the following root $seed$ node:

	Z	H	$x =$0N
	0	4	$maxScore = 0$
T	0	3	$f = 4$
A	0	2	$g = 0$
C	0	1	
G	$-\infty$	0	

We now expand the children of this node. Using the suffix tree shown in Figure 2, the children search nodes to consider next correspond to the following suffix tree nodes: 1N, 2N, 3N, 4N.

Expanding Node 1N

The expansion of the node 1N starting from the root is shown below. Note that the first column in G is the Z column from the root node. Most of the G matrix is maintained in memory only while the node is expanded, but the final column (labelled (Z)) is maintained for subsequent expansion of the children.

	G	(Z)	H	F	$x =$ 1N
	-	A			←Target ($headPos = 3$)
	0	$-\infty$	4	$-\infty$	$maxScore = 1$
T	0	$\nwarrow(-1)$	3	$-\infty$	$f = 3$
A	0	$\nwarrow 1$	2	3	$g = 1$
C	0	$\uparrow(0)$	1	$-\infty$	$tag = VIABLE$
G	$-\infty$	$\nwarrow(-1)$	0	$-\infty$	

As mentioned in Section 3, the expansion corresponds to an extension of the G table using the symbols along the node's incident arc. For instance, in this example the second column of G corresponds to the initial alignment scores along the path beginning with the symbol A. The first two values in the G column refer to the replacement scores (indicated by $\nwarrow$) of $A \rightarrow T$ and $A \rightarrow A$. The third value is the result of an insertion ($\uparrow$). In cases where alignment pruning has occurred, we indicate the value initially computed in parentheses, but remember that such positions are actually assigned a value of $-\infty$. This node is tagged as $VIABLE$ because $f \geq minScore$.

Expanding Nodes 2N and 3N

The expansion of node 2N is similar to the expansion of node 1N, and results in a f value of 2 and g value of 1. The expansion of node 3N results in f and g values of 1, so this node is tagged as $ACCEPTED$. In the interest of space, we omit the details of these node expansions.

Expanding Node 4N

The expansion of the node 4N (shown below) is a case where multiple columns of G are expanded within a single node expansion.

	G		(Z)	H	F	$x =$ 4N
	-	T	A			←Target ($position = 8 - 9$)
	0	$-\infty$	$-\infty$	4	-	$maxScore = 2$
T	0	$\nwarrow 1$	$\leftarrow(0)$	3	$-\infty$	$f = 4$
A	0	$\uparrow(0)$	$\nwarrow 2$	2	4	$g = 2$
C	0	$\nwarrow(-1)$	$\uparrow(1)$	1	$-\infty$	$tag = VIABLE$
G	$-\infty$	$\nwarrow(-1)$	$-\infty$	0	$-\infty$	

Priority Queue
After expanding the root node's children, the priority queue contains the following entries, in $(nodePtr/f)$ pairs: $PQ = \{(\texttt{4N}/4), (\texttt{1N}/3), (\texttt{2N}/2), (\texttt{3N}/1)\}$. Note that although 3N has been accepted, it is last on the queue, and therefore we will not report its descendants as results unless all other paths have been shown to yield lower scores.

Picking the Next Node to Expand
In our example, the node 4N is at the front of the priority queue and it's children, (2L, 8L), are expanded next.

Expanding Node 2L
As soon as it is possible to do so, we cease the column-wise expansion of a node. In this case (node 2L as shown below), we reach an accept state ($f = g$) in the second column, and finish. We need not maintain an alignment column-vector (Z) for this node, both because it is a leaf node and an $ACCEPTED$ node.

	G			H	F	$x = \texttt{2L}$
	A	C	G ...			←Target ($position = 3 - 5$)
	$-\infty$	$-\infty$	$-\infty$	4	$-\infty$	$maxScore = 4$
T	$-\infty$	$-\infty$	$-\infty$	3	$-\infty$	$f = 4$
A	2	$\leftarrow 1$	$\leftarrow(0)$	2	$-\infty$	$g = 4$
C	$-\infty$	$\nwarrow 3$	$\leftarrow(2)$	1	$-\infty$	$tag = ACCEPTED$
G	$-\infty$	$\uparrow(2)$	$\nwarrow 4$	0	4	

Expanding Node 8L
The expansion of node 8L is similar to the expansion of node 2L, and results in a f value of 2 and g value of 2. In the interest of space, we omit the details of this expansion.

At the end of expanding nodes 2L and 8L, we reach a terminal symbol, \$. No further expansion is possible, so we simply set f and g to the maximum value seen along the path, which was uncovered in the earlier expansion of node 4N.

Search Termination
Now, the priority queue contains $PQ = \{(\texttt{2L}/4), (\texttt{1N}/3), (\texttt{8L}/2)(\texttt{2N}/2), (\texttt{3N}/1)\}$. The top element is tagged as $ACCEPTED$, therefore we have found the maximum local alignment with respect to the one sequence stored in the suffix tree. In a multi-sequence tree, we would continue the search in order to identify maximal alignments for all sequences, or until the queue is empty, indicating that all alignments with scores greater or equal to $minScore$ have been identified.

3.4 Suffix Tree Disk-based Representation

Since the OASIS search is driven by a suffix tree, the representation of the suffix tree on disk can have a large impact on the performance of OASIS. In this section, we present the disk-based representation that we use.

In general, accessing a suffix tree results in many random disk accesses, which could lead to poor performance. To minimize this effect, we organize the representation of the tree on disk such that siblings are stored contiguously whenever possible. This organization is beneficial, since OASIS must explore all children of a node when the node is expanded.

The suffix tree is conceptually represented using three arrays: a symbols array, an array for the internal nodes, and an array for the leaf nodes. The disk-based representations for the three arrays, with reference to the suffix tree in Figure 2, are as follows:

Symbols:
The symbol array (shown in Figure 2) is simply broken down into chunks that fit into a disk block, and the disk blocks are written out sequentially. The $position$ information that is shown in Figure 2, is simply the index of the symbols, and is shown only for illustration purposes; this information is not part of the actual disk representation.

Internal nodes:
Each internal node has the following four attributes:

- $depth$: This attribute stores the length of the node $path$.
- $position$: This attribute is a pointer to the symbol array, indicating the portion of the original sequence that labels the incoming arc. The length of the arc can be determined by subtracting the depth of the parent node from the depth of the incident node.
- $firstChild$: This attribute is a *node pointer* to the first successor or child of the node.
- $endBit$: Since siblings are adjacent, we use this attribute to indicate whether or not a node is the "last" sibling.

The internal nodes are traversed in a level-first order, and stored sequentially on disk in disk-block size chunks.

Leaves:
To save space, we organize the leaf nodes in such a way that the array index of a node indicates the relevant $position$ in the symbol array for the incident arc. Since there is a single leaf node for every suffix, the index tells us the starting position for the full node path in the symbol array. For instance, leaf node 0L is the terminus of the path beginning from $position$ 0 in the symbol array. Because of this, leaf nodes can not be clustered like the internal nodes, and we must maintain an explicit pointer to siblings ($rightSibling$). Again, the array is written to disk sequentially, and organized in disk blocks.

3.4.1 Suffix Tree Construction

For OASIS, the suffix tree is constructed only once. Traditional suffix tree construction algorithms [25, 38] realistically require the entire tree to be maintained in memory during the construction phase. With large data sets, this is not possible, and techniques for building suffix trees larger than the amount of memory are required. Recently such a technique was proposed [16]. This technique constructs sub-trees stemming from fixed-length prefixes of each suffix in memory, by making one pass through the sequence data for each subtree. We use this same general approach to construct the suffix tree, but select lexical ranges for each

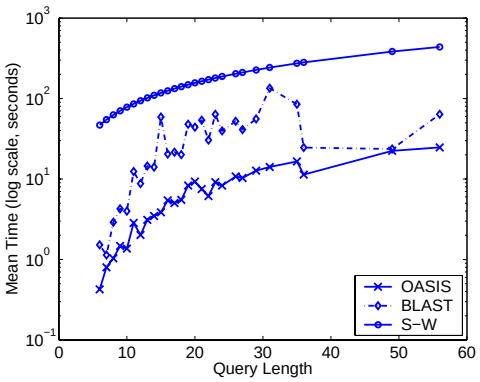

Figure 3: Performance Comparisons, SWISS-PROT, E=20,000.

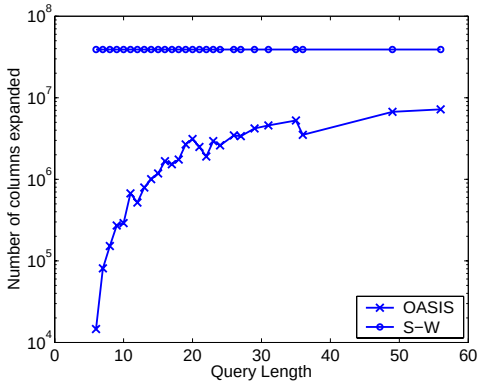

Figure 4: Filtering Efficiency, SWISS-PROT, E=20,000.

pass based on the contents of the underlying database sequences. Once the suffix tree has been constructed, we reorganize the disk-representation using the representation described above.

4 Experimental Evaluation

This section presents an experimental evaluation of OASIS.

4.1 Data Set and Queries

In the experiments described in this paper, we used the entire SWISS-PROT database [37]. This data set is an annotated collection of proteins with low redundancy, and is commonly used in practice. SWISS-PROT contains over 100K protein sequences, totaling roughly 40M amino acid symbols, ranging in length from 7 to 2048 symbols.

In addition to SWISS-PROT, we also tested OASIS on the entire *Drosophila* (fruit-fly) genomic nucleotide sequence. This nucleotide data set contains roughly 120M symbols in 1K sequences. The results for the nucleotide data sets are similar to those presented here, with OASIS outperforming S-W by orders of magnitude. In the interest of space, we omit these results in this paper.

For the query set, we used typical *short* query workloads, since OASIS is designed primarily for this task. The protein query workload consisted of a hundred queries that were randomly selected from the `ProClass` motif database [15]. `ProClass` is a non-redundant database which organizes the SWISS-PROT entries into family relationships, grouping similar proteins and placing them in the same family class. Such queries are frequently used to find matches to short peptide sequences [6]. The `ProClass` database contains roughly 70K motifs, ranging in length from 3 to 80 residues, with an average length of 17. The hundred queries in the selected workload range in length from 6 to 56 symbols and have an average length of 16 symbols.

4.2 Experimental Setup and Implementation Details

In these experiments, we used BLAST version 2.2, downloaded from the NCBI website [7]. We implemented both OASIS and S-W in C++. Both code sets were compiled with full optimization using gcc version 2.95.4. The OASIS algorithm reads disk pages from a buffer pool, which uses a simple clock replacement policy. The OASIS implementation used a block size of 2K. All experiments were run on a machine with a 1.70 GHz Intel Xeon processor, running Debian Linux 2.4.13, and configured with a 40GB Fujitsu MAN3367MP SCSI hard drive. Unless stated otherwise, in the following experiments, we set the buffer pool size to 256MB.

For OASIS the suffix tree was built using the technique described in [16]. The on-disk image was organized as described in Section 3.4. The space utilization of the suffix tree index is summarized below:

Data Set Size (# of symbols)	Index Size	Space Utilization (bytes per symbol)
40M	500MB	12.5

The space utilization of the suffix tree is comparable to that of the most compact suffix tree representations [22] (which uses 12.5 bytes per symbol).

In all the experiments, for the protein queries, we used the PAM30 substitution matrix, which is the popular choice for short queries [6]. (We also experimented with other substitution matrices, which produced similar results.)

All search tools were configured to use a fixed gap penalty model. With this model, a series of k insertions or deletions contributes $-ka$ to the alignment score, where a is a single gap penalty. Alternately, we can use an affine gap penalty, which charges a penalty b to open the gap, and a penalty c to extend the gap. With an affine gap model, a k-length gap contributes $-b-(k-1)*c$ to the alignment score. To manage affine gaps, OASIS and S-W must expand three dynamic programming matrices. The two additional matrices store the alignment scores associated with insertion gaps and deletion gaps respectively. Our current implementations of OASIS and S-W do not support such affine gap penalties.

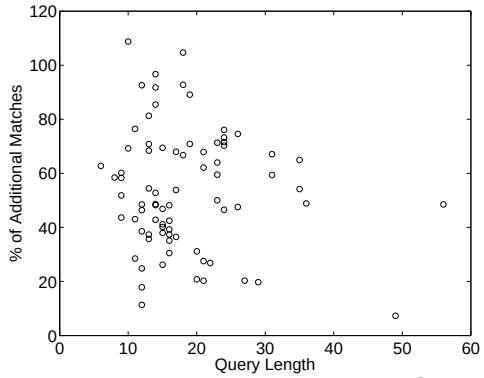

Figure 5: Accuracy Comparisons with BLAST, SWISS-PROT, E=20,000.

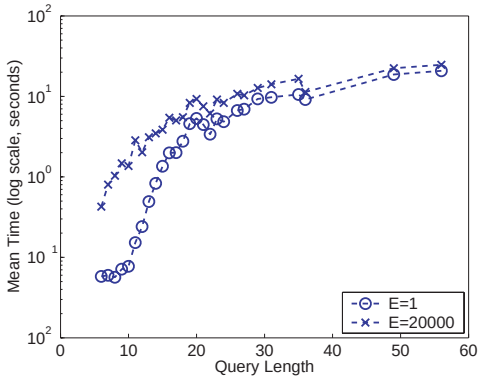

Figure 6: Effect of Selectivity, SWISS-PROT.

4.3 Comparison with BLAST and S-W

In this experiment, we compare the performance of OASIS with the Swiss-Waterman (S-W) algorithm, which is the traditional *accurate* local alignment algorithm. For this experiment, we also include a comparison with the popular search tool BLAST. Note that BLAST is a *heuristic* algorithm, so its functionality is not directly comparable to S-W and OASIS. The primary goal of the comparison with BLAST is simply to establish a baseline for performance comparison.

While OASIS controls query selectivity through the $minScore$ input parameter, BLAST uses an E-value parameter. Each alignment has an E value, which indicates the number of alignments we expect to find in a given database *by chance* having an equivalent or better alignment score. The following equation relates E-value and alignment score (S):

$$E = Kmne^{-\lambda S} \tag{2}$$

where m is the query length, n is the database size and K and λ are scaling constants computed by BLAST. In a BLAST search, all alignments with an E-value less than some input parameter are returned. The corresponding $minScore$ value for OASIS can be computed as follows:

$$minScore = \left\lceil \frac{\ln(Kmn) - \ln(E)}{\lambda} \right\rceil \tag{3}$$

BLAST performs additional statistical adjustments to the E value based both on the length of the query and on the lengths of individual sequences in the database [1]. For this reason, BLAST is used only for baseline comparisons with OASIS, which is designed as a replacement for S-W. OASIS can however perform the same adjustments: query length has a fixed effect on $minScore$, and it is straightforward to adjust reported statistics as OASIS uncovers alignments. To strictly maintain online properties, OASIS must also sort the queue based on an optimistic estimate of E-value, as it relates to alignment score. When a particular sequence is accepted, it must then be pushed back on the priority queue with a non-optimistic E value (adjusted for the actual sequence length).

Figure 3 shows the performance comparison between OASIS, S-W and BLAST, with the selectivity set to E=20,000, which is the BLAST recommended value for short protein queries. These results are organized according to query length, and the query execution times are plotted on a log scale. Where multiple queries of a particular length exist, we have plotted the mean time for all such queries. While OASIS and S-W optimally find the strongest alignments, BLAST offers no such guarantee. Despite this, *OASIS runs faster than BLAST, by an order of magnitude in some cases. In all cases, OASIS is an order of magnitude or more faster than S-W, which provides the same functionality as OASIS.*

In order to study the tradeoffs between OASIS and S-W in greater detail, we examine the number of column-wise expansions performed by each algorithm. This metric indicates what portion of the database is considered by OASIS, and thus reflects the *filtering* behavior of the algorithm. Figure 4 shows the filtering behavior of OASIS and S-W with respect to query length for the protein queries. In the worst case, OASIS expands 18.5% of the columns. On average, OASIS expands only 3.9% as many columns as S-W. OASIS generally does less work per column than S-W, due to alignment pruning, so this is a conservative metric of filtering efficiency.

We also examined the number of matches that were returned by OASIS and BLAST. Figure 5 plots the percentage of additional matches that were returned by OASIS. On average OASIS retrieved about 60% more matches than BLAST.

4.4 Effect of Selectivity

In this experiment we investigate the effect of query selectivity on the performance of OASIS. Query selectivity is changed by varying the E value in OASIS (we vary the E score rather than $minScore$ for consistency in the presentation of the experimental results; corresponding $minScore$ values can be computed using Equation 3).

For this experiment, the E value is varied from 1 to 20,000. Figure 6 plots these two extremes. Plots for the other E values are in between these two extremes, and are

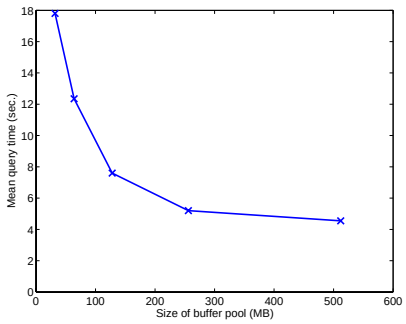

Figure 7: Effect of Buffer Pool Size, SWISS-PROT, E=20,000.

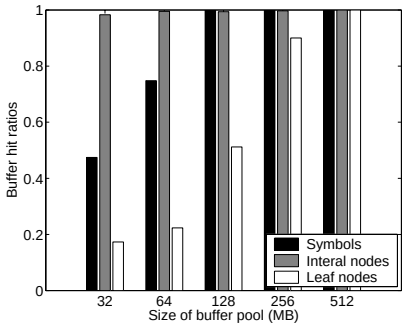

Figure 8: Buffer Pool Performance

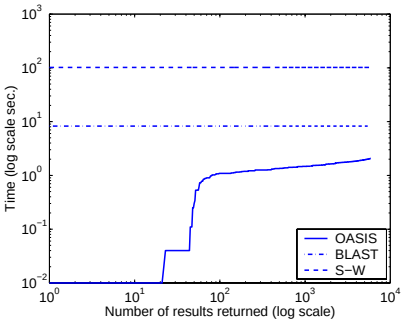

Figure 9: Sample Online Behavior, SWISS-PROT, Query: DKDGDGCITTKEL

not shown in the figure. A higher E value implies a more "relaxed" match, and more results are returned. For the experiment shown in Figure 6, with an E value of 20,000, OASIS returns roughly 1000 times more results than for E=1.

As can be seen from this figure, for very small query lengths, the more highly selective queries (E=1) take better advantage of our hierarchical search technique, and thus result in more efficient performance. However, as the query lengths increase, the effect of selectivity reduces sharply. For long queries, the difference in performance is relatively small, especially given that OASIS returns many more results for E=20,000. In practice, OASIS must search a relatively large space to guarantee that no alignments are missed. This implies that in uncovering strongly relevant matches, much of the groundwork has been laid for the discovery of weaker matches. Notice that the difference in performance is quite marked on the shortest queries, where high selectivity (or low E values) makes OASIS behave very much like exact suffix tree search.

4.5 Effect of Buffer Pool Size

This experiment evaluates the effect of the buffer pool on the performance of OASIS. Figure 7 plots these results as the mean times for the entire workload. Figure 7 shows that the performance of OASIS degrades for very small buffer pool sizes. As the buffer pool size increases the performance improves rapidly. With 512M allocated, the entire structure can fit in memory, and when one quarter of the tree (buffer pool size = 128M) can fit in memory, we see a 57.5% increase in query running time.

To understand the effect of the buffer pool size on the query performance, we measured the ratio of "buffer hits" over all protein queries. This ratio is defined as the ratio of disk pages requested that are already in the buffer pool. Figure 8 shows the buffer hit ratios for each of the three components of the suffix tree (see Section 3.4).

Notice the internal nodes are the only optimized elements in terms of disk layout, and as such, they are least susceptible to problems with smaller allocation. In the current suffix tree representation, symbol and leaf accesses are by their nature random, since they are ordered according to the original database sequence. We are currently experimenting with an alternative leaf representation to alleviate this problem, so that leaves are stored contiguously with the internal nodes.

4.6 Online Behavior

The results presented in the previous sections describe the *complete* running times of all algorithms. However unlike S-W and BLAST, OASIS has an online behavior, and produces results incrementally with scores in descending order. As a result, the top results are returned far more quickly. In this section, we now examine the online behavior of OASIS.

We illustrate the online behavior of OASIS with a single protein query; however the behavior described here is typical of other queries. We use a 13 symbol protein query from the Motif collection (DKDGDGCITTKEL). In this experiment, the selectivity was set to $E = 20,000$. With this setting, OASIS identifies 5899 viable alignments, and BLAST identifies 5437 viable alignments. Figure 9 plots the times at which each result is returned by OASIS. Note that the top results are returned very quickly, with the first 40 results being returned in under 4/100ths of a second.

5 Related Work

We now review some existing work in this area. An efficient, but inaccurate, system for local alignment matching is QUASAR [8]. Based on suffix arrays, it achieves a performance gain over BLAST in searches for "strongly similar DNA sequences" by filtering out sections of the database not likely to generate any useful matches. The technique proposed by Kahveci and Singh [18] indexes the substrings of a database according to background statistical properties using a "wavelet" approach. For a given query string, it then identifies regions that are able to produce a strong global alignment. After filtering, the authors report that between 5-50% of the database remains. However, the effectiveness of the approach with standard protein substitution matrices (e.g. BLOSUM62 or PAM30) is not explored. A similar approach has been adopted by Ooi et

al. [30], in which a bitmap index is used to filter out "irrelevant subsequences". Sahinalp et al. [33] have observed that compression distances in strings is almost metric, and leverage this property to use VP-trees to prune out large portions of the database when evaluating string proximity searches. OASIS differs from these approaches in that it uses its index structure to directly evaluate alignments rather than simply identifying strong regions. There is a filtering component however, in the sense that OASIS allows us to consider only a small subset of the database.

BLAST [1–3] and FASTA [32] are are commonly used for similarity searching on biological sequences. Both tools employ heuristics to speed up their search. There has been considerable work on refinements of the BLAST algorithm [3, 20, 24, 39], and these techniques focus primarily on improving the speed and the sensitivity of the basic BLAST algorithm. However, none of these have the same sensitivity as S-W. Recent work [4, 10] has also considered using suffix trees for searching unstructured sequence data. Both of these works refer to the application of the suffix trees to genetic sequences, but do not detail any approximate matching strategy. Other groups [29] have used lookup index structures to identify exact matches, and thus achieve considerable performance gains on searches for exact similarity on clean data. Chavez and Navarro [9] develop an algorithm operating on a suffix tree that finds all matches within a certain "edit distance", or number of edits and symbol substitutions. For the PAM and BLOSUM matrices used in protein sequence analysis, edit distance provides a very loose lower-bound on the the actual alignment score, since certain residues are substituted with high likelihood.

A* is a widely used technique, and has been proposed as a method for solving *multiple* sequence alignment [19, 21], which finds alignments between two or more sequences. These papers investigate how to pick the heuristic in the A* algorithm for multiple-alignment searches. Neither of these methods have been implemented, or compared with existing tools like S-W or BLAST for the simpler case of local-alignments. OASIS only considers local-alignments, and hence can pick simpler heuristics. Furthermore, the search in OASIS is driven by a suffix tree, which results in significant pruning of the search space.

In general, the construction and traversal of suffix trees results in "random-like access" [14] for a number of efficient in-memory construction methods [25, 38]. Since, even the most efficient suffix tree representations require 12.5 bytes per symbol [22], for large data sets the suffix trees need to reside on disk. In recent years, a number of solutions have been proposed for efficiently construct suffix trees on disk [4, 10, 16, 17, 31]. Specifically for biological sequences, a recent paper by Hunt et al. [16] proposed a secondary-memory algorithm, which we have used in our implementation.

Suffix trees have also been applied for aligning whole genomes [11, 12], and for exploring repeated structures in genomic sequences [23]. OASIS differs from these approaches as it targets a different problem, namely local-alignment searches.

6 Conclusion and Future Work

We have introduced a new algorithm called OASIS which improves upon the performance of the existing state-of-the-art for *accurate* local sequence alignment. The existing accurate local-alignment algorithm, the Smith-Waterman (S-W) algorithm, is rarely used since it is very computationally expensive. Consequently, when searching large biological sequence databases, life science researchers often settle for approximate search algorithms, which though finely tuned to avoid missing good matches, do not guarantee that a good match will *never* be missed. In this paper we show that the OASIS algorithm is often an order-of-magnitude or more faster than the S-W algorithm when the query is a short sequence. Such short sequences are often used in querying biological sequence data sets, and OASIS is very effective in these cases.

Besides not missing any matches, OASIS, also has the property of returning result tuples in decreasing order of the matching scores. Consequently, OASIS can also be used in an online mode, where the scientist may want to abort the query after seeing the top few results.

As part of future work, we will investigate techniques to allow incremental updates to the suffix-tree disk structure. We are also planning on investigating techniques to further improve the performance of OASIS for answering long queries, and extending our current implementation to include affine gap penalties. Finally, we plan on investigating the application of OASIS for evaluating local-alignment matches in other domains, such as identifying closely matching musical pieces based on a few hummed notes [5, 26].

Acknowledgements

We would like to thank Jim Gray, Don Huddler, Ela Hunt, and H. V. Jagadish for providing valuable feedback on earlier versions of this paper. Colin Meek was supported by the National Science Foundation under grant IIS-0085945. Other support for this work was provided in part by a fellowship from the University of Michigan, and research gift donations from IBM and Microsoft.

References

[1] S. Altschul and W. Gish. Local Alignment Statistics. *Methods Enzymol*, 266:460–480, 1996.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[3] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

[4] P. Bieganski, J. Riedi, J. V. Carlis, and E. F. Retzel. Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 35–44, 1994.

[5] W. P. Birmingham, B. Pardo, C. Meek, and J. Shifrin. The MusArt Music-Retrieval System: An Overview. *D-Lib Magazine*, 8(2), 2002.

[6] BLAST Program Selection Guide, National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/BLAST/producttable.html, February 2003.

[7] BLAST download site: http://www.ncbi.nlm.nih.gov/BLAST, 2003.

[8] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram Based Database Searching Using a Suffix Array (QUASAR). In *RECOMB*, pages 77–83, 1999.

[9] E. Chávez and G. Navarro. A Metric Index for Approximate String Matching. In *LATIN*, pages 181–195, 2002.

[10] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *VLDB*, pages 341–350, 2001.

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

[12] A. Delcher, A. Phillippy, J. Carlton, and S. Salzberg. Fast Algorithms for Large-scale Genome Alignment and Comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

[13] C. Dwan. Speedup at What Cost? An Evaluation of Heuristic vs. Complete Homology Search Techniques. *Presented at Bioinformatics Technology Conference, Tucson, Arizona*, 2002.

[14] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, 19(3):331–353, 1997.

[15] H. Huang, C. Xiao, and C. Wu. ProClass Protein Family Database. *Nucleic Acids Research*, 28(1):273–276, 2000.

[16] E. Hunt, M. P. Atkinson, and R. W. Irving. A Database Index to Large Biological Sequences. In *VLDB*, pages 139–148, 2001.

[17] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. One-dimensional and Multi-dimensional Substring Selectivity Estimation. *VLDB Journal*, 9(3):214–230, 2000.

[18] T. Kahveci and A. K. Singh. An Efficient Index Structure for String Databases. In *VLDB*, pages 351–360, 2001.

[19] K. Kelly and P. Labute. The A* search and Applications to Sequence Alignment. *http://www.chemcomp.com/article/astar.htm*, 1996.

[20] W. J. Kent. BLAT: The BLAST-like Alignment Tool. *Genome Research*, 12(4):656–664, 2002.

[21] H. Kobayashi and H. Imai. Improvement of the A* Algorithm for Multiple Sequence Alignment. *Genome Informatics*, 9:120–130, 1998.

[22] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software - Practice and Experience*, 29(13):1149–1171, 1999.

[23] S. Kurtz, J. V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: The Manifold Applications of Repeat Analysis on a Genomic Scale. *Nucleic Acids Research*, 29(22):4633–4642, 2001.

[24] B. Ma, J. Tromp, and M. Li. PatternHunter: Faster and More Sensitive Homology Search. *Bioinformatics*, 18(3):440–445, 2002.

[25] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[26] C. Meek and W. Birmingham. Johnny Can't Sing. In *ISMIR*, pages 124–132, 2002.

[27] D. Morrison. PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[28] H. Nash and D. Blair. Comparing Algorithms for Large-scale Sequence Analysis. In *BIBE*, pages 89–96, 2001.

[29] J. Ogasawara and S. Morishita. Practical Software for Aligning ESTs to Human Genome. In *CPM*, pages 1–16, 2002.

[30] B. C. Ooi, H. H. Pang, H. Wang, L. Wong, and C. Yu. Fast Filter-and-Refine Algorithms for Subsequence Selection. In *IDEAS*, pages 243–255, 2002.

[31] S. Park, W. W. Chu, J. Yoon, , and J. Won. Similarity Search of Time-Warped Subsequences Via a Suffix Tree. *Information Systems*, 28(7):867–883, 2003.

[32] W. R. Pearson and D. J. Lipman. Improved Tools for Biological Sequence Comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[33] C. Sahinalp, M. Tasan, J. Macker, and M. Ozsoyoglu. Distance Based Indexing for Sequence Proximity Search. In *ICDE*, 2003.

[34] P. Sellers. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

[35] E. Shpaer, M. Robinson, D. Yee, J. Candlin, R. Mines, and T. Hunkapiller. Sensitivity and Selectivity in Protein Similarity Searches: A Comparison of Smith-Waterman in Hardware to BLAST and FASTA. *Genomics*, 38:179–191, 1996.

[36] T. Smith and M. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[37] SWISS-PROT Database, European Bioinformatics Institute, http://www.ebi.ac.uk/swissprot/, 2002.

[38] E. Ukkonen. Constructing Suffix Trees On-line in Linear Time. In *Proceedings of the 12th IFIP World Computer Congress*, pages 484–492, 1992.

[39] H. E. Williams and J. Zobel. Indexing and Retrieval for Genomic Databases. *IEEE TKDE*, 14(1):63–78, 2002.