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

Colin Meek Jignesh M. Patel
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109, USA
{meek, jignesh}@eecs.umich.edu

Abstract

The analysis of sequence data is central in the increasingly important area of functional proteomics and genomics. A common type of query that is posed by scientists in these areas is to take an input query sequence and search against a large protein or gene data set finding targets that match the input query sequence. Such queries are commonly evaluated using local-alignment algorithms to find approximate matches. The current set of search tools used for this purpose employ heuristics to improve the speed of the search. However, such heuristics can sometimes miss out matching targets. Given the high-stakes involved in these biological research projects, it is natural to wonder if an algorithm could be designed that is guaranteed not to miss any results given the query specification.

There are a few know algorithms for identifying *all* local alignments, but so far these have been considered to be too computationally intensive to be used for searching large biological sequences. The most popular of these *accurate* search algorithms is the Smith-Waterman algorithm. In this paper, we take on the challenge of designing an *accurate and efficient* algorithm for evaluating local-alignment searches against large biological data sets. We propose a novel algorithm called OASIS that uses an A*-style search algorithm, which is driven by a suffix tree index built on the input data set. We experimentally evaluate the proposed algorithm using a number of motif queries against the entire Swiss Protein data set. We demonstrate that OASIS is an order-of-magnitude faster than the Smith-Waterman algorithm in most cases. Consequently, we believe that it may now be possible to use accurate search tools for evaluating local-alignments against actual biological data sets. Finally, OASIS also has the nice property of returning target sequences in the order of the matching score, which allows its use in an online setting in which the user can abort a query after viewing partial results.

1 Introduction

In the new and emerging area of functional genomics and proteomics life science researchers often have to query biological sequences. Frequently, the scientist has a protein or nucleotide sequence and they want to check if this sequence has been seen before, or postulate the function of a sequence based on similarities to known sequences. While there are various ways of finding matching sequences, local alignment is frequently used: using some scoring metric for alignments, we can determine the likelihood that two sequences are related. In the late 1980s when computer scientists started investigating algorithms for fast local alignment, it was too expensive to answer local alignment queries by matching the query sequence against each sequence in the database. However, a lot has changed since that time. Marching in line with Moore's law, processors have continued an exponential increase in computational speed, and main memory has become quite inexpensive. As a result, it is now reasonable to expect that scientists have, or will soon have GHz processor machines on their desktops with a GB or more of main memory.

A popular tool for local sequence alignment, BLAST [1], performs a linear search for strong local alignments using discrete finite automata (DFA). The DFA is designed to accept *q*-grams within a certain edit distance from query subsequences, with the restriction that insertions and deletions are disallowed [15]. All accepted *q*-grams are extended to the left and the right to determine whether they meet the search criteria, using some ceiling on the probability that such a match would occur by chance. This heuristic filter has been shown to result in very few false negatives, but does not have 100% precision and recall. Given the high-stakes and cost involved in many life sciences experiment, 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 answers.

While a number of refinements of the BLAST algorithm have been proposed and continue to be proposed, these techniques focus on improving the speed of the basic BLAST algorithm. In this paper, we take a novel approach to this problem. We recognize that with current hardware configurations large main-memory configurations are common. 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 is designed to *accurately* evaluate local-alignment matches over multiple sequences. Employing a best-first search, it identifies alignments in score order, making it ideal for *online* results generation. A suffix tree that is built on the data set enables a sub-linear scan for potential targets, by simply "walking" down the tree. We directly identify maximum score local alignments, using a best-first (A*) suffix tree traversal guided by intermediate results that are produced by using the Smith-Waterman (S-W) dynamic programming approach [19] to compute string edit distances between the input query and suffixes of the data set[19].

The most space efficient suffix trees have a space utilization of roughly 12.5 bytes per base pair (bp) [14]. Given a query of length m, and a database containing sequences with a sum length of n, suffix trees allow for the identification of exact matches in $O(m)$ time. When approximate matches are allowed, the performance can degrade to $O(mn^2)$ time, although we will demonstrate that this upper-bound is too liberal for extremely large databases! The other accurate local-alignment algorithm (S-W) runs in $O(mn)$ time under all conditions. Our algorithm scales fairly well to approximate queries, but its performance nonetheless degrades with respect to S-W as queries become less specific, and query lengths increase. However, a large class of queries that are posed by scientist involve short query sequences. For example, queries against protein data sets using peptide motifs as query sequences is frequently used to find proteins that match specific peptides that are observed in an experiment. Such peptide motifs are much shorter in length than the actual protein, and are commonly used in querying biological data sets [8]. OASIS is especially suitable for such cases, as is demonstrated in this paper.

Traditionally, the use of suffix trees has been written-off primarily because suffix trees are hard to represent efficiently on disk, and consequently on large data sets, the performance of algorithms using suffix trees is very poor. However, we argue that for querying on biological sequences current trends in hardware will allow a large portion of the suffix tree nodes to be resident in main-memory. We back up our claims by actual experiments on an actual complete data set (the popular Swiss Protein data set) on a machine that was priced less that \$3K a year ago.

An astute reader may point out that in the last few years the size of the biological sequence data has been doubling every sixteen months. Processor speeds and memory sizes are keeping up with this doubling of data set sizes, and unlike Moore's law it is unclear if this doubling of data sizes is likely to continue for a prolonged period of time (much of the increase in the data sizes in recent years have come from the public agencies doing a better job of integrating data from multiple sources). Furthermore, the OASIS algorithm is very easy to parallelize, and can easily be run on clusters of (cheap) PCs. While we don't explore parallelism in this paper, it is our plan to explore this in the future.

The remainder of this paper is organized as follows: we will begin by describing some existing work in this area, describe the bases for the OASIS algorithm, outline our strategy for the construction and representation of the underlying index structure in secondary memory, describe the algorithm itself, and finally present results of experiments on a complete protein database.

2 Related Work

We now review some existing work in this area. An effective (but non-accurate) system is QUASAR [4]. 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. [6] and [3] have applied suffix trees to the problem of searching unstructured sequence data, in both cases with reference to genetic sequences. To our knowledge, no study of the tradeoff point between linear and suffix tree searches has been conducted, with respect to approximate matching. Approximate matching is a critical component of sequence analysis [7], so while suffix trees have been proven to be extremely efficient for exact matching, their performance does degrade when closeness requirements are relaxed. This follows from the intuition that approximate matching requires branching in the search space.

Trees provide the greatest benefit for hierarchically organized searches. In the case of approximate string matching, there is no direct hierarchy in the sense that we cannot simply descend the tree to an answer. The degree of branching in the search for matches increases as we allow for larger edit distance between the query and the target. At some point, the overhead (space and time) of the tree structure negates the advantage of a hierarchically organized search. Similar problems have been addressed for multi-dimensional indexing, where the tree structure gracefully incorporates linear storage and search techniques as the data requires it [2], but there is no clear extension to the string edit-distance domain: searching a hyper-rectangle would result in too many false hits, and restricts us to a single *q* value, without the possibility of insertions and deletions. It is possible to index a 4-gram “TACG” if we map it to a coordinate in 4-d space, $\langle$ T,A,C,G $\rangle$. Allowing for insertions and deletions, we would need to search using hyper-rectangles corresponding to various interpretations of dimensionality, since insertions and deletions would effectively expand and collapse dimensions in the search space.

3 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 (for an example, see Figure 1). We will use this notation throughout. Q is an m-length query, where q_i is the i^{th} symbol. Similarly, T is an n-length target, where t_j is the j^{th} symbol. In our complexity analyses, m and n will always have this meaning.

There are several alignment operations:

- *Replacement* with either the same symbol (see label 1 in Figure 1) or a different symbol (see label 2 in

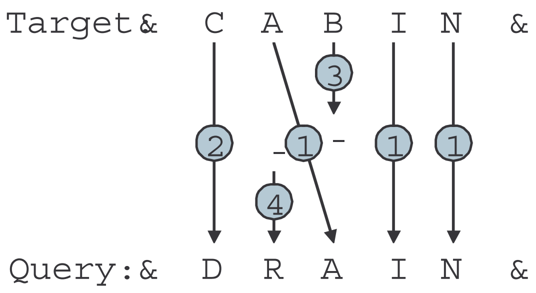

Figure 1: Local alignment example

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 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 $\alpha \to \beta$, where insertions are represented as $- \to \beta$ and deletions as $\alpha \to -$. The score of an operation is denoted $S(\alpha \to \beta)$. An edit matrix S (see Table 1) stores the scores, so the cost of an operation $\alpha \to \beta$ is simply $S_{\alpha,\beta}$. The Smith-Waterman algorithm [19] finds the local alignment between two sequences with the maximum possible score using a dynamic programming approach that runs in $O(mn)$ time.

	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 table

3.1 Smith-Waterman Algorithm

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. Note that we can induce that best possible score from previous entries in the matrix:

$$G_{i,j} = \max \left\{ \begin{array}{cc} 0 & We can “start over'' (new local alignment) \\ G_{i-1,j-1} + s(q_i \to t_j) & Replacement \\ G_{i-1,j} + s(q_i \to -) & Insertion \\ G_{i,j-1} + s(- \to t_j) & Deletion \end{array} \right\} \quad (1)$$

The cost of the algorithm is the cost of building the matrix G, which has $m \cdot n$ entries. Consider for example a query $Q =$ “TACG” against a target $T =$ “AGTACGCCTAG”. Table 2 outlines the execution of S-W, using the edit matrix in Table 1, where $\nwarrow$ indicates that a replacement yielded 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 can uncover the alignment:

$$\begin{array}{cccccc} T:... & \text{T} & \text{A} & \text{C} & \text{G} & ... \\ & \downarrow & \downarrow & \downarrow & \downarrow & \\ Q: & \text{T} & \text{A} & \text{C} & \text{G} & \end{array}$$

4 Generalized Suffix Tree Structure

In this section, we now outline the basics of the suffix tree, the basic data structure of the OASIS algorithm. Suffix trees are a special case of A^+ trees. These are, essentially, trees in which the edges are labeled with strings from $A^*\{\epsilon\}$, where A is a finite alphabet. Each node can have only one outgoing edge labeled with aw where $a \in A$, which means the branching degree of the tree is at most $|A|$, the cardinality of the symbol alphabet. The *path* to a node is described by the concatenation of the strings labeling the edges traversed. For instance, in Figure 2 $path(\texttt{5A}) =$ “AG”.

A string w is said to be a *word* of an A^+ tree if there exists a node k such that $path(k) =$ “wu”. A suffix tree T on a string t is an A^+ tree in which every subsequence of t is a word of T. One final definition is required: a *compact* suffix tree is one in which every node is either the root, a branching node or a leaf. In practice, this

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

Table 2: G matrix built using S-W

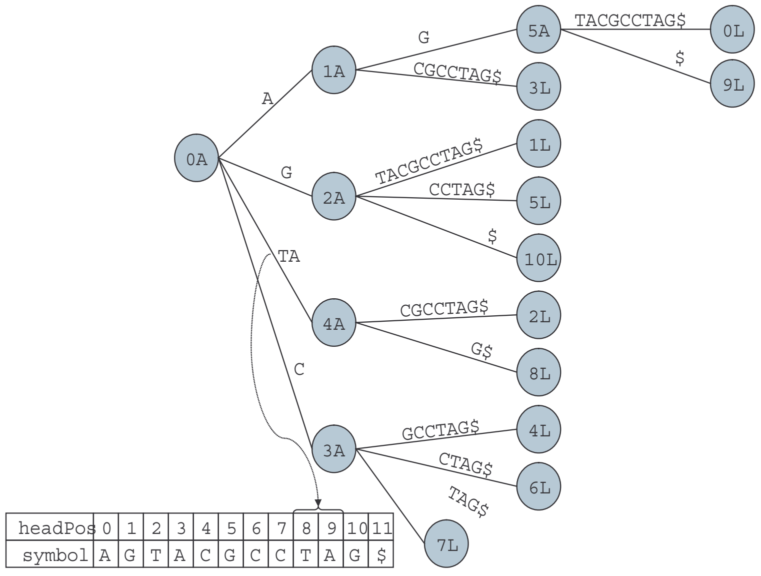

Figure 2: Suffix tree on the sequence "AGTACGCCTAG"

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. Note that in general, sequences are appended with a terminal symbol, e.g. $.

4.1 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 possible, since the tree must contain a path beginning with all subsequences. If the query is consumed, this means that a match has been found. 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 `4A`, and the subsequent two symbols on the arc `2L`. This indicates that a match exists beginning at position 2 in T (more on the naming conventions for nodes later in this paper.) In general, once a match has been found, its location(s) in the target sequence can be identified by descending to its successor leaf/leaves.

4.2 Tree Construction and Representation

In general, the construction and traversal of suffix trees results in "random-like access" [9]. Efficient in-memory construction methods [16, 20] do not adapt well to secondary-memory [3, 12]. An existing secondary-memory construction technique [12] constructs sub-trees stemming from fixed-length prefixes of each suffix in memory, by making one pass through the sequence data for each subtree. [3] supports multiple sequences by building trees for each in memory, and then merging them. Another trie storage method, developed initially for PATRICIA tries [17] is the "index fabric" [5, 6]. It provides excellent scalability and access patterns, but is optimized for XML data, and may not effectively handle the redundancy of suffix representations.

We also construct our trees in multiple passes, but use a hashing function over sequence *q*-grams to determine which branches of the tree are built in each pass. We use a hash function with n bins, where n is the number of passes that we will make over the input data set. In each pass, we determine from the first q symbols of every suffix whether that suffix is to be written in the current pass. In this way we avoid swapping to disc by storing only the relevant tree branches in memory for each pass. In practice, accessing a suffix tree results in many random accesses accesses. To minimize this phenomenon, we then reorganize the tree on disk such that siblings are stored contiguously, since OASIS must explore all children of a node when the node is expanded.

The strings labeling the arcs of a tree are generally referred to indirectly using pointers to the original sequence. To reduce the number of random accesses incurred by these pointer fetches, we store some of this data with the actual node: as many symbols as can be stored in 4 bytes using Huffman coding to conserve space [18]. We need only fetch from the original sequence when the relevant symbols are not stored along with the node.

4.3 Multi-sequence

A fairly trivial modification of the structure allows for the storage of multiple sequences: we can simply concatenate multiple sequences, separated by the terminal symbol. Note that in practice, a match is associated with the position in the *full* concatenated sequence, based on the position in the leaf array. We simply store the offsets of each sequence, and based on the position, identify the target sequence using a weighted binary search through the offset table. This tends to work well in practice, since the lengths of individual sequences is fairly consistent. For instance, given a structure storing x sequences and a position p, with n equal to the sum length of the sequences, we start the search at sequence $\lfloor p \cdot x/n \rfloor$, refining the range for the search as we go.

5 A^* Heuristic Search

We will now offer a brief overview of A^* search, a key aspect of our approach. A^* is a technique for bounding and directing branching searches, a "best-first" strategy wherein nodes showing the greatest promise are expanded first. In this instance, we are interested in maximizing score. Each node has associated with it a current score, and an expected score. At each stage in the search, the node with the highest expected score is expanded.

5.1 Expected Score

In the search, each node n has some score associated with its path from the root, denoted $g(n)$. In addition, we can in many cases make heuristic predictions about the value of the best possible path from that node to the goal, denoted $h(n)$. h should be an *optimistic* heuristic, in the sense that it should never underestimate the actual best value. The expected value of a node is then simply $f(n) = g(n) + h(n)$. Our heuristic is an example of a monotonic heuristic, which means that the f-value can never increase, only decrease. Note also that at the goal state, $f(n) = g(n)$, since at this point completely accurate predictions are of course available!

5.2 The Algorithm

The successors of the root are first pushed onto a priority-queue, where the ordering is according to f-value. At every step, the node with the highest rank is expanded, and its successor pushed on the queue. Based on the monotonicity assumption, we know that if a goal node is at the top of the queue, no other solution can ever have a higher f value. Extending the process, we can identify in order of score the best all goal states.

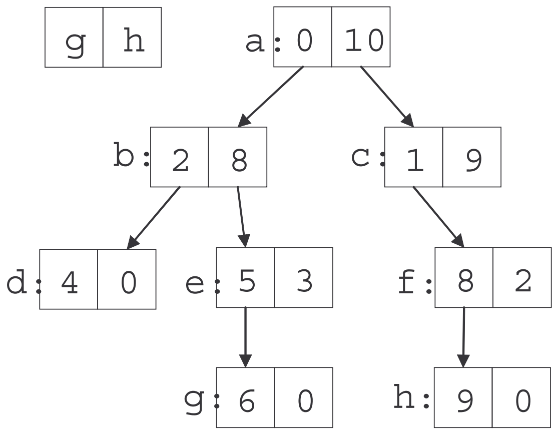

Figure 3: Sample search tree for A^* search, leaves are goal nodes

Consider the search tree in Figure 3, where the goal states are the leaf states. The algorithm trace is as follows, where goal states are shown in bold:

Step	Current Node	Q (node/f-value)	Results
1	a	{b/10,c/10}	{}
2	b	{c/10,e/8,**d**/4}	{}
3	c	{f/10,e/8,**d**/4}	{}
4	f	{**h**/9,e/8,**d**/4}	{}
5	**h**	{e/8, **d**/4}	{**h**}
6	e	{**g**/6, **d**/4}	{**h**}
7	**g**	{**d**/4}	{**h**, **g**}
8	**d**	{}	{**h**, **g**, **d**}

Note that in case of ties, as in step 1, we give higher priority to the node with the higher g-value, since we have less confidence in the h-values.

6 OASIS

The basic idea of the algorithm is to perform an A^* search, where the search space is defined by a suffix tree, and nodes are expanded in a manner closely related to S-W. We will begin by describing the specifications for the algorithm, then describe its operation, and finally give a brief example to demonstrate its execution.

6.1 Specifications

6.1.1 Input

The algorithm takes as input a suffix tree $ST(\mathbf{T})$, where $\mathbf{T}$ is the concatenation of the sequences $T_1T_2...T_s$. A query Q is given, along with an edit score matrix S. $minScore$ is the minimum relevant score, to allow for search pruning, and since such restrictions are useful in practice, to avoid the reporting of trivial alignments. $maxReturn$ is the maximum number of alignments to return, where only one alignment is returned per sequence, as with S-W.

6.1.2 Output

The algorithm returns the starting points and scores of each alignment (i.e. the starting point of the word in $\mathbf{T}$ yielding the alignment.) The algorithm operates on-line: as matches are found, they are output in descending order of alignment score.

6.2 The Algorithm

For expositional purposes, we will describe the algorithm as it relates to A^*:

6.2.1 g-values

As we follow a path through the suffix tree, we essentially fill in the columns of the S-W matrix, G, with some key differences. This highlights one advantage of this approach: work can be shared among multiple paths with the same prefix. We do not however allow for the score to be reset to 0, because this work will be duplicated elsewhere: resetting to 0 means considering a different starting alignment with respect to $\mathbf{T}$, when all such alignments are considered along some path in $ST(\mathbf{T})$. The g-value associated with a particular node is simply the maximum value in the last filled-in column of G. In practice, we maintain only this last column along with the node.

To save time and space, we only expand and store the portions of that column with *strictly positive values*, with the exception of the root node, where all values are initialized to zero. We call this *alignment pruning*. Again, we do this to avoid duplicating work, since equivalent or better scoring paths can be identified *starting after* these alignments through other paths. Essentially, this mirrors the "starting over" process used by S-W. For instance, consider a partial local alignment between a portion of a query $qq_{i+1}...q_j$ and the suffix $t_kt_{k+1}...t_l$. If the maximum alignment score beginning at $q_i \to t_k$ and ending at $q_j \to t_l$ is negative, then we can guarantee that the score of any alignment between $q_i...q_j...$ and $t_k...t_l...$ will have a lower score than the alignment between $q_{j+1}...$ and $t_{l+1}....$ We explicitly consider all starting alignments for the query in the column expansion, and because of the fundamental properties of the suffix tree, we consider all starting alignments, since every suffix of the database exists along some rooted path through the tree.

6.2.2 h- and f-values

We generate an $m+1$-length vector H, where H_i $(0 \leq i \leq m)$ represents the maximum possible alignment score of $q_{i+1}q_{i+2}...q_m$ with any arbitrary target, again where the query $Q = q_1q_2...q_m$. Calculating these values is trivial, assuming non-positive values for insertions and deletions. H_m is set to zero, since the relevant portion of the query is null. We can then inductively calculate the remaining values inductively: $H_{i-1} = H_i +$ the maximum score for the replacement of q_{i-1}.

Returning to our earlier example, $Q =$ "TACG" and the edit matrix in Table 1, each symbol is optimally replaced by itself, yielding a score of 1, and we have $H = \{4, 3, 2, 1, 0\}$. This matrix gives an upper limit on how much the remainder of the query can contribute to the score. Consider the G matrix in Table 2. If we take the fourth column (under the symbol A), and add the values in the H vector, we get:

G	H	$F = G + H$
-	4	-
0	3	3
2	2	4
1	1	2
0	0	0

Note that the second row predicts the maximum possible (and actual) alignment score extending from q_2 and t_4, equal to 4. The expected score f of a node is the maximum F value associated again with the last filled column, with the caveat that we do not consider f-values of pruned alignments. If this f-value is below $minScore$, the entire node can immediately be pruned (*node pruning*).

6.2.3 Goal state

Defining the goal state is not straightforward. Clearly leaf nodes are candidates, and are tagged as *accepted* if they are not pruned. There is another important case however: if the maximum expected score ever drops below or equal to the maximum *actual* score seen anywhere along the ancestor path, then a node is also marked as *accepted*. This is because such paths can, by an optimistic heuristic, never yield a stronger alignment than the one already found. It is advantageous to identify such situations as early as possible so that costly node expansions can be avoided. Accepted nodes are pushed onto the priority queue, with their expected scores set equal to the maximum score seen along the path. Accepted states are effectively *goal* states, in the conventional language of A^* search, since they describe the maximum alignment score along a particular path through the suffix tree.

When a node tagged as *accepted* is at the top of the priority queue, all of its successor leaves are written out, provided they refer to sequences for which no maximum score alignment has already been found.

6.3 Example Execution

Next we illustrate the execution of the algorithm using an example. We return to the target and suffix tree in Figure 2, and the query "TACG", with a minimum alignment score of $minScore = 1$ and using the edit matrix in Table 1. The priority queue is initialized with the root node:

	G	H	F	$nodePtr =$ `0A`
	-	4	4	$maxScore = 0$
T	0	3	3	$f = 4$
A	0	2	2	$g = 0$
C	0	1	1	
G	0	0	0	

The root node is the only node to use H_0, which equals 4 in this example. This is because it is the only node for which no part of the query has been consumed, and thus all of $q_1 q_2 ... q_m$ can contribute to the expected score. Otherwise, we might prune the root, although it has a feasible alignment using the entire query. We now examine `0A`'s children: `1A`, `2A`, `4A`, `3A`.

6.3.1 Expanding `0A` (`1A`, `2A`, `4A`, `3A`)

Node `1A`

	previous G col	G	H	F	$nodePtr =$ `1A`
	-	A			←Target ($headPos = 3$)
	-	-	4	-	$maxScore = 1$
T	0	↖-1	3	2	$f = 3$
A	0	↖1	2	3	$g = 1$
C	0	↑0	1	1	$tag = VIABLE$
G	0	↖-1	0	-1	

The expansion of the node `1A` is shown above. For clarity, we show the previous G column, or in other words the last column from the parent's G table. As mentioned, the expansion corresponds to an extension of the G table using the symbols along the node's incoming arc. For instance, in this example the G column corresponds to the initial alignment scores along the path beginning with the symbol `A`. As such, the "previous" column is initialized to all zeros. The first two values in the G column refer to the replacement scores (indicated by ↖) of $A \to T$ and $A \to A$. The third value is the result of an insertion (↑). Which information is stored in memory for each node? We keep the variables $maxScore, f, g, tag$ and $nodePtr$, which represent the maximum score seen along the path so far, the maximum expected score, the maximum score in the current expansion, and a pointer to the suffix tree node respectively. We also maintain all relevant (i.e. positive) entries in G, in this case only the second row value (1) along with a bitmap indicating which entries are represented, to conserve space. In this case, $G_{relevant} =$ '$0100'$ and $G = \{1\}$. This node is tagged as $VIABLE$ because $f \geq minScore$.

Node `2A`

	previous G col	G	H	F	$nodePtr =$ `2A`
	-	G			←Target ($headPos = 5$)
	-	-	4	-	$maxScore = 1$
T	0	↖-1	3	2	$f = 1$
A	0	↖-1	2	1	$g = 1$
C	0	↖-1	1	0	$tag = ACCEPTED$
G	0	↖1	0	1	

The expansion if node `2A`, shown above, highlights an important issue: $f = 1$, although $F_1 = 2$! This is, again, because non-positive alignments are pruned. Clearly, if this path were to be expanded, it would have a lower score than the alignment omitting this first negative-score replacement. This node is tagged as $ACCEPTED$ at this point, since $maxScore = f$, and therefore no further expansion can be fruitful.

Node `4A`

	previous G col	G		H	F	$nodePtr = $ `4A`
	-	T	A			$\leftarrow$Target ($headPos = 8 - 9$)
	-	-	-	4	-	$maxScore = 2$
T	0	$\nwarrow$1	$\leftarrow$0	3	3	$f = 4$
A	0	$\uparrow$0	$\nwarrow$2	2	4	$g = 2$
C	0	$\nwarrow$-1	$\uparrow$1	1	2	$tag = VIABLE$
G	0	$\nwarrow$-1	$\uparrow$0	0	0	

The expansion of the node `4A` (shown above) is the first case where multiple columns of G are expanded within a single node expansion. Notice that the table is filled in using the S-W procedure (on the subsequence "TA") as described in Section 3.1. Despite this, we still maintain only the last column of the G matrix. The second type of alignment pruning is shown here: note that although the alignment in the third row has a positive score ($G_3 = 1$), there is no point in continuing in that direction since its maximum possible score ($H_3 = 2$) has already been achieved along this path ($G_2 = maxScore = 2$).

Node `3A`

	previous G col	G	H	F	$nodePtr = $ `3A`
	-	C			$\leftarrow$Target ($headPos = 6$)
	-	-	4	-	$maxScore = 1$
T	0	$\nwarrow$-1	3	2	$f = 2$
A	0	$\nwarrow$-1	2	1	$g = 1$
C	0	$\nwarrow$1	1	2	$tag = VIABLE$
G	0	$\uparrow$0	0	0	

When we reach the point of expanding node `3A` (shown in the table above), the priority queue contains the following entries, in $(nodePtr/f)$ pairs: $Q = \{(\texttt{4A}/4), (\texttt{1A}/3), (\texttt{3A}/2), (\texttt{2A}/1)\}$. Note that although `2A` 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: We now examine the queue and pick the node that is at the front of the queue. In our example, the node `4A` is at the front of the queue and it's children, (`2L`, `8L`), are expanded next.

6.3.2 Expanding `4A` (`2L`, `8L`)

Node `2L`

	previous G col	G			H	F	$nodePtr$ = `2L`
	A	C	G	...			←Target ($headPos = 3 - 5$)
	-	-	-	-	4	-	$maxScore = 4$
T	-	-	-	-	3	-	$f = 4$
A	2	←1	←0	-	2	3	$g = 4$
C	-	↖3	←2	-	1	3	$tag = ACCEPTED$
G	-	↑2	↖4	-	0	4	

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 above), we reach an accept state ($f = g$) in the second row, and finish. Note that we no longer need to maintain the G information about this node.

Node `8L`

	previous G col	G		H	F	$nodePtr$ = `8L`
	A	G	\$			←Target ($headPos = 9 - 10$)
	-	-	-	4	-	$maxScore = 2$
T	-	-	-	3	-	$f = 2$
A	2	←1	-	2	-	$g = 2$
C	-	↖1	-	1	-	$tag = ACCEPTED$
G	-	↑0	-	0	-	

The case of node `8L`, shown above, illustrates one of the unusual cases where we realize in *retrospect* that an accept state has been reached. When we reach a terminal symbol, \$, no further expansion is possible, so we simply set all values to the maximum value seen along the path, which was uncovered in the earlier expansion of node `4A`.

Search Termination Now, the priority queue contains $Q = \{(\texttt{2L}/4), (\texttt{1A}/3), (\texttt{8L}/2)(\texttt{3A}/2), (\texttt{2A}/1)\}$. 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.

7 Experimental Evaluation

In this section we present an experimental evaluation of the OASIS algorithm. We also compare the performance of OASIS with the Swiss-Waterman (S-W) algorithm, which is the traditional *accurate* local alignment algorithm.

7.1 Experimental Setup and Implementation Details

All experiments were run on a Linux 2.4.13 machine with 896 MB of memory, a 1.70 GHz Intel Xeon processor, and a Fujitsu MAN3367MP hard drive with a SCSI interface and a 40 GB capacity.

We implemented both OASIS and S-W in C++. Both code sets were compiled with full optimization using gcc. For OASIS, we limited main memory usage to 256MB for these experiments, using a simple LRU buffer pool system. The implementation used a block size of 2K to swap portions of the suffix tree index structure between main memory and disk as necessary.

7.2 Data Set and Queries

For the data set in these experiments, we used the *entire* Swiss protein database [13]. Samples of four proteins in this data set are shown in Figure 4. This data set is an annotated collection of proteins with low redundancy, and is commonly used in practice. The database contains over 100K sequences, totaling roughly 40M protein symbols, ranging in length from 7 to 2048 symbols. The proteins are represented as sequences, consisting of symbols representing the 20 amino acids, or residues.

```
>gi|2495000|sp|Q63931|CCKR_CAVPO CHOLECYSTOKININ TYPE A RECEPTOR (CCK-A RECEP-
TOR) (CCK-AR)
MDVVDSLFVNGSNITSACELGFENETLFCLDRPRPSKEWQPAVQILLYSLIFLLSVLGNTLVITVLIRNKRMRTVTNIFL
LSLAVSDLMLCLFCMPFNLIPSLLKDFIFGSAVCKTTTYFMGTSVSVSTFNLVAISLERYGAICKPLQSRVWQTKSHALK
VIAATWCLSFTIMTPYPIYSNLVPFTKNNNQTGNMCRFLLPNDVMQQTWHTFLLLILFLIPGIVMMVAYGLISLELYQGI
KFDAIQKKSAKERKTSTGSSGPMEDSDGCYLQKSRHPRKLELRQLSPSSSGSNRINRIRSSSSTANLMAKKRVIRMLIVI
VVLFFFLCWMPIFSANAWRAYDTVSAERHLSGTPISFILLLSYTSSCVNPIIYCFMNKRFRLGFMATFPCCPNPGTPGVRG
EMGEEEEGRTTGASLSRYSYSHMSTSAPPP
>gi|1708198|sp|P80487|HHP_THICU HETEROTROPH-SPECIFIC PROTEIN
AADDVTVVIGSAAPMSGPQ
>gi|13878750|sp|Q9CDN0|RS18_LACLA 30S ribosomal protein S18
MAQQRRGGFKRRKKVDFIAANKIEVVDYKDTELLKRFISERGKILPRRVTGTSAKNQRKVVNAIKRARVMALLPFVAEDQN
>gi|13878816|sp|Q9CI15|TIG_LACLA Trigger factor (TF)
MTVSFEKTSDTKGTLSFSIDQETIKTGLDKAFNKVKANISVPGFRKGKISRQMFNKMYGEEALFEEALNAVLPTAYDAAV
KEAGIEPVAQPKIDVAKMEKGSDWELTAEVVVKPTVSLGDYKDLTVEVEATKEVSDEEVETRLTNSQNNLAELVVKETAA
ENGDTVVIDFVGSVDGVEFEGGKGSNHSLELGSGQFIPGFEEQLVGTKAGETVEVKVTFPENYQAEDLAGKEALFVTTVN
EVKAKELPELDDELAKDIDEEVETLDELKAKFRKELEESKAEAYNDAVETAAIEAAVANAEIKEIPEEMIHEEVHRAMNE
FLGGMQQQGISPEMYFQITGTSEDDLHKQYEADADKRVRTNLVIEAIAAAENFTTSDEEVKAEIEDLAGQYNMPVEQVEK
LLPVDMLKHDIAMKKAVEVIATTAKVK...
```

Figure 4: Portion of swissprot database

For our experiments, we used a query workload consisting of 500 queries. The queries were randomly selected from the `ProClass` database of functional motifs [11]. The `ProClass` Database is a non-redundant database which organizes the Swiss Protein data base according to family relationships (organizing similar proteins in the same family class). So our queries represent the scenario where the scientist is querying a protein database looking for short matches; such queries are frequently used to find matches to short peptide sequences [8] [1]. A simple definition of a peptide is two or more amino acids covalently linked by an amide bond between two specific groups of the amino acids (the carboxylic and the alpha-amino groups). A simpler view of the peptide is that it is a short sequence of amino acids.

The `ProClass` database contains roughly 70K motifs, ranging in length from 3 to 80 residues, with an average length of 17.

In our experiments, we used three different edit matrices: a unit edit matrix (like Table 1 but for the 20 amino acid symbols), and the BLOSUM90 and BLOSUM62 (for instance Figure 5) matrices; the BLOSUM matrices are frequently used by life science researchers. We also varied the "selectivities" of the queries. The selectivities are varied using the following technique: One of the inputs to the algorithm is $minScore$, which determines the minimum score of alignments to return. The selectivity is controlled using the *minimum score ratio*, which indicates the ratio of the maximum possible alignment score we will accept. For instance, given the example query "TACG" and unit edit distance, 4 is the maximum possible alignment score with any arbitrary sequence. Given a ratio of 0.5, we would search for alignments with score $4 \cdot 0.5 = 2$ and above. Note that a ratio of 0 returns the entire database, which is typically not a very useful query.

A suffix tree was constructed on the entire Swiss protein data sets. The suffix tree was constructed in 17 passes, resulting in a 500MB structure. This construction phase took roughly 3.5 hours.

7.3 Experimental Goals

The experiments are intended to test and demonstrate the following issues:

- What are the tradeoffs between S-W and suffix-tree local alignment approaches?
- To what extent does the edit matrix influence performance? The intuition here is that a more homogeneous matrix, i.e. one that discriminates little between symbols, will result in less differentiation between different paths, and thus a higher degree of branching in the search. We performed tests using a unit

[1]The *approximate* matches using BLAST has two versions for matching proteins, one for queries 5-15 residues longer, and the other for 15 residues or longer sequences). Similarly there are two versions of BLAST for querying nucleotide (genetic) sequences corresponding to short queries, which are 7-20 base pairs, and long queries, which are larger than 20 base pairs.

edit distance matrix (e.g Table 1), BLOSUM90, and BLOSUM62 (see Figures 5), which progressively admit greater differences between target and query. BLOSUM matrices indicate the degree of similarity between two distantly related protein sequences. The different degrees of BLOSUM matrices, e.g. 62% in BLOSUM62, indicate that sequences with 62% similarity are given the greatest weight in generating the matrix. As such, lower values favor more distantly related protein sequences[10]. These matrices are commonly used by biologists as distance metrics for the BLAST and FASTA algorithms.

- To what extent does the length of the query influence performance? With S-W, the relationship between query length and processing time is linear. With our algorithm, the influences are more complicated, as the degree of "redundancy" is highest at lower tree depths. We explore this issue in more detail in the following section.

7.3.1 The Effect of Redundancy on OASIS

OASIS exploits the redundancy found in most finite alphabet sequences. When an arc in the suffix tree is expanded, each symbol along that arc can refer to multiple symbols in the database. Of course, a single symbol in the database is also represented in multiple *positions* in the tree, since it will appear in multiple suffixes! For instance, the 'A' at $headPos = 3$ in Figure 2 exists on 4 separate paths! We will present a simplified mathematical model of the situation, to help explain the degradation of performance with longer queries, and the expected improvement of the algorithm's performance as the database size increases.

We will make two simplifying assumptions:

- Assume that the database is composed of random sequences of symbols, where symbols are selected uniformly from the alphabet A, independent of context. n is again the sum length of the database sequences.

- Assume that we are not allowing insertions and deletions, and the depth of the search is thus strictly bounded by the length of the query. Note that by depth, we mean the length of the path, e.g. node `8L` in Figure 2 has depth 4.

Given a query length m, it is clear that in a full m-depth tree search, every symbol in the database can be "visited" at most m times, since m suffixes will contain that symbol within a depth of m. Again, each symbol in the tree can refer to multiple sequences or parts of sequences. How many exactly? It turns out the answer depends on the depth in the tree. At depth one, the intuition is straightforward: given the uniformity assumption,

```
#  Matrix made by matblas from blosum62.iij
#  * column uses minimum score
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Blocks Database = /data/blocks_5.0/blocks.dat
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected =  -0.5209
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1  0 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  3  0 -1 -4
D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4  1 -1 -4
C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -3 -2 -4
Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0  3 -1 -4
E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -2 -1 -4
H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0  0 -1 -4
I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3 -3 -1 -4
L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4 -3 -1 -4
K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0  1 -1 -4
M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3 -1 -1 -4
F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3 -3 -1 -4
P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -1 -2 -4
S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0  0  0 -4
T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1  0 -4
W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -3 -2 -4
Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -2 -1 -4
V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4 -3 -2 -1 -4
B -2 -1  3  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4  1 -1 -4
Z -1  0  0  1 -3  3  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
X  0 -1 -1 -1 -2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2  0  0 -2 -1 -1 -1 -1 -1 -4
* -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4  1
```

Figure 5: BLOSUM62 Edit Matrix

precisely $\frac{1}{|A|}$ of suffixes begin with any given symbol. Extending the argument, we expect precisely $\frac{1}{|A|^2}$ of suffixes to begin with any arbitrary sequence of two symbols. The number of suffixes in the tree is equal to n, so in general, we expect each symbol at depth d in the tree to refer to $\frac{n}{|A|^d}$ different positions in the database.

Making the assumption that each m-length query will require a full m-depth search through the suffix tree, and the above relationships, we can expect to expand as much as $\frac{m \cdot |A|^m}{n}$ times as many columns as S-W using the suffix tree approach! This is mitigated by two important factors: node pruning, and the fact that in practice, the boundary of the search space (which this formula alludes to) is in practice very sparse. The critical observations here however are that the worst-case suffix tree performance should improve with respect to S-W as the database size increases, and degrade with the length of the query!

Pruning significantly improves algorithm performance of course. Although a mathematical analysis of this issue is beyond the scope of this paper, we should note that deep nodes, while more dangerous, also have a greater likelihood of being pruned, for the following reason: the f-values become more accurate as we descend the tree, since the optimistic heuristic portion has a lower and lower contribution. In essence, there is an increasing chance of pruning paths that will eventually lead to low scores as we gain more information along those paths. Experimental evaluation reveals a less drastic increase in running time as the query length increases (see Section 7.4).

On average, OASIS expands only 4.6% as many columns as S-W. In our experiments, in the worst-case twice as many columns were expanded by OASIS, though this is counteracted somewhat by alignment pruning: OASIS frequently does less work per column.

7.4 Experimental Results

In this section, we present the experimental results comparing OASIS with S-W. Since each data point in our experiment represents a collection of 500 queries, we will present the results using “whisker” plots. Figure 6 describes the components of a whisker plot.

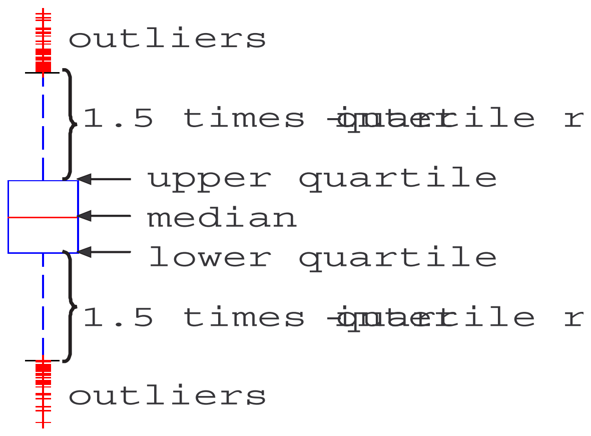

Figure 6: Whisker Plot Legend

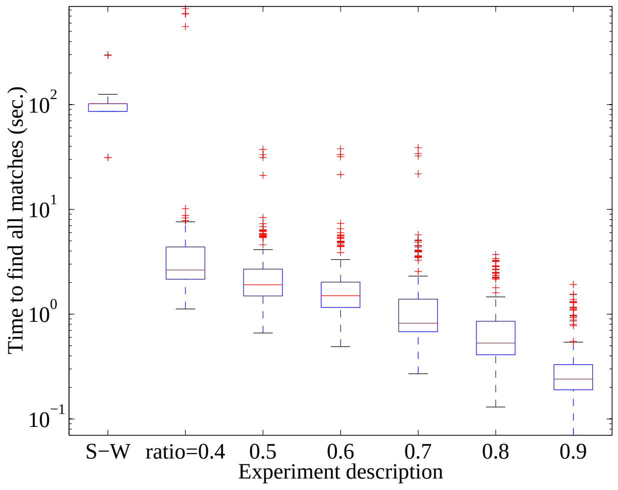

Figure 7: Unit Edit Distance: Total Running Times (Log Scale)

Figures 7 to 9 show whisker plots of the time taken to run full queries for various selectivities, with time shown on a logarithmic scale. For instance, for ratio = 0.7 all sequences with alignment scores greater than or equal to 0.7 times the maximum possible alignment score are returned. As a point of reference, we also show the running times for Smith-Waterman, which are not influenced by the selectivity, or the edit matrix for that matter.

As can be observed from these figures, **the OASIS algorithm is an order of magnitude faster than the S-W algorithms in many cases, and even a few order of magnitude faster in some cases!** As the selectivity of the query increases (larger ratios) the relative performance of OASIS over S-W increases rapidly. The less "relaxed" the edit matrix is in terms of admitting distantly related proteins (e.g. unit edit distance matrix and BLOSUM90), the higher is the relative performance improvement from using OASIS.

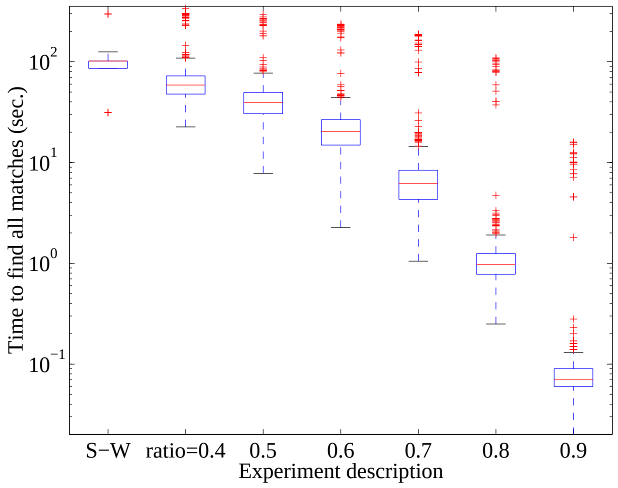

Figure 8: BLOSUM90: Total Running Times (Log Scale)

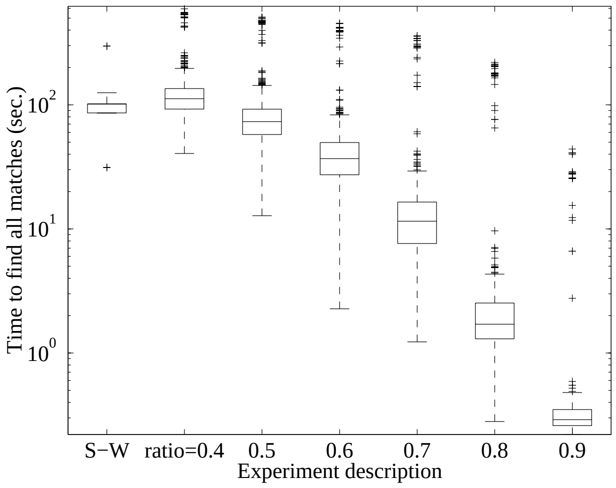

Figure 9: BLOSUM62: Total Running Times (Log Scale)

At the same time, we also see in these figures that there are "expensive" outliers for OASIS, especially for low selectivity queries, and more "relaxed" edit matrices (such as BLOSUM62). In many cases, these expensive outliers generally corresponded to queries that were larger than 40 amino acids. Consequently, a simple rule of thumb can be used is to switch to using the S-W algorithm for large query sequences (or use an approximate matching algorithm such as BLAST). Note that as per the analysis in Section 7.3.1, this tradeoff point is influenced by the size of the database, and as the size of the database increases, it is expected that this tradeoff point will move to a larger number, making OASIS the preferred choice for larger queries.

In many other cases, the expensive outliers were associated with queries producing large numbers of matches. We anticipate that in online searches, users will choose to terminate such low-discrimination queries after the top results have been returned.

7.4.1 Effect of Query Length

Figure 10 illustrates the effect of query length on running times, for a minimum score ratio of 0.7. Each point represents the mean running time for queries of each observed length. Notice that while S-W running times increase linearly with respect to query length, OASIS exhibits super-linear behavior. This speaks to the inherent tradeoffs involved in the use OASIS, or any search algorithm that makes use of a hierarchical structure to drive the search process. Fortunately, in practice, performance does not appear to degrade in the worst-case fashion according to the theoretical analysis.

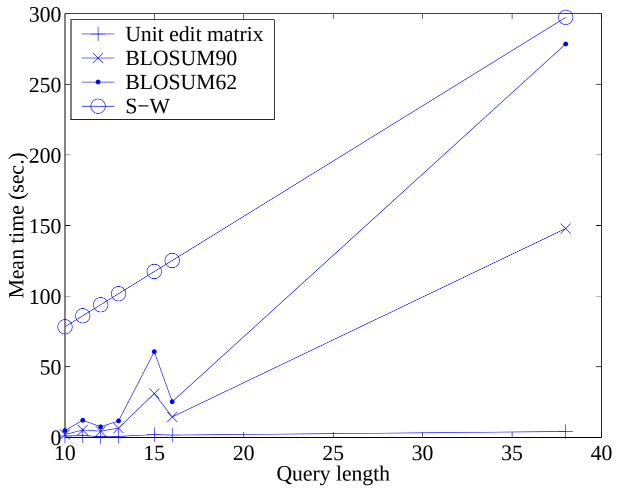

Figure 10: Mean Time by Query Length (Unit Edit Distance Matrix)

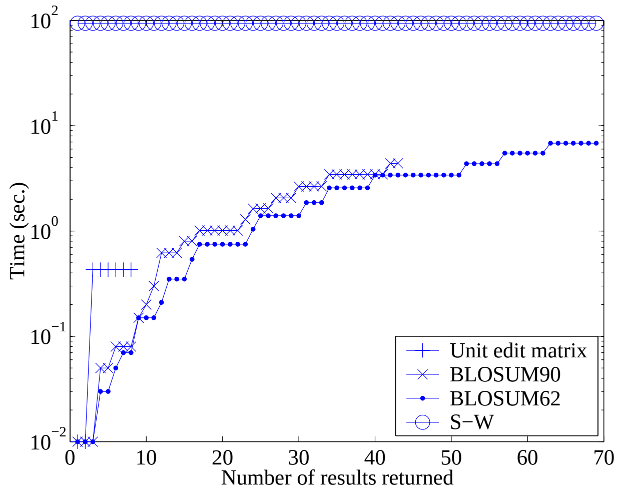

Figure 11: Times for Online Results Generation (Log scale), Query: `CLNTLGSYKCSC`

7.4.2 Online Characteristics

The results presented in the previous sections describe the *complete* running times of the algorithm: the algorithm continues running until it is certain that no branches can produce alignments scoring above the specified minimum, but returns results online in descending score order. Top results are returned far more quickly. For instance, the top scoring alignment was returned in under 0.5 seconds on average for all experiments reported in the previous sections. The top 100 database targets were returned in under 7 seconds on average. In contrast, algorithms such as S-W and the approximate BLAST searches, cannot provide this information until the full run is completed.

In this section, we now examine the online behavior of OASIS. We illustrate the online behavior of OASIS with a single query; however the behavior described here is also typical for other queries. We use the following 12 symbol long query for this experiment: `CLNTLGSYKCSC`. In this experiment, the "ratio" parameter in OASIS was set to 0.7, which produced 8, 43 and 69 matches for the unit edit distance, BLOSUM90 and BLOSUM62 cases respectively. The results using this query are shown in Figure 11. Note that there are several plateaus in the plot, resulting from accepted paths pointing to multiple sequences. As can be observed from this figure, OASIS returns the top few results very quickly, making it well suited for use in an online setting.

8 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 considered to be too computationally expensive. Scientists 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 sort sequences are frequently used in querying protein and genetic data sets, and we show that OASIS is extremely 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 uses 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 plan on adapting the tree construction algorithm to allow for incremental updates. This would allow updates to the database to incorporated into the index in a more efficient fashion that having to rebuild the entire index. We also plan on investigating techniques to further improve the performance of OASIS to answer longer queries. Finally we also plan on investigating techniques to effectively run OASIS on parallel architectures, especially when evaluating long queries.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 1990.

[2] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.

[3] 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.

[4] Stefan Burkhardt, Andreas Crauser, Paolo Ferragina, Hans-Peter Lenhof, Eric Rivals, and Martin Vingron. q -gram based database searching using a suffix array (QUASAR). In *RECOMB*, pages 77–83, 1999.

[5] B. Cooper and M. Shadmon. The index fabric. Technical report, RightOrder Inc. http://www.rightorder.com, 2001.

[6] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proceedings of the 27th VLDB Conference*, 2001.

[7] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[8] National Center for Biotechnology Information. BLAST Program Selection Guide. *http://www.ncbi.nlm.nih.gov/BLAST/producttable.html*, June 2002.

[9] Robert Giegerich and Stefan Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[10] S. Henikoff and J. Henikoff. Amino acid substitution matrices from protein blocks. In *Proc. Natl. Academy Science 89(10)*, pages 915–919, 1992.

[11] H. Huang, C. Xiao, and C. Wu. Proclass protein family database. *Nucleic Acids Research*, 2000.

[12] Ela Hunt, Malcolm P. Atkinson, and Robert W. Irving. A database index to large biological sequences. In *The VLDB Journal*, pages 139–148, 2001.

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

[14] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software - Practice and Experience*, 29(13):1149–1171, 1999.

[15] Stefan Kurtz. Foundations of sequence analysis. *citeseer.nj.nec.com/kurtz01foundations.html*, 2001.

[16] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[17] D.R Morrison. Patricia - practical algorithm to retrieve information d in alphanumeric. *Journal of the ACM*, 15:514–534, 1968.

[18] Michael Schindler. Practical huffman coding. *http://www.compressconsult.com/huffman/*.

[19] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 1981.

[20] E. Ukkonen. Constructing suffix trees on-line in linear time. In J. van Leeuwen, editor, *Proceedings of the 12th IFIP World Computer Congress*, pages 484–492, Madrid, Spain, 1992. North-Holland.