A Table-Driven, Full-Sensitivity Similarity Search Algorithm

Gene Myers * Richard Durbin †

July 1, 2002

In honor of Michael Waterman's 60th birthday.

Abstract

Searching a database for a local alignment to a query under a typical scoring scheme such as PAM120 or BLOSUM62 with affine gap costs, is a computation that has resisted algorithmic improvement due to its basis in dynamic programming and the weak nature of the signals being searched for. In a query preprocessing step, a set of tables can be built that permit one to (a) eliminate a large fraction of the dynamic programming matrix from consideration, and (b) to compute several steps of the remainder with a single table lookup. While this result is not an asymptotic improvement over the original Smith-Waterman algorithm, its complexity is characterized in terms of some sparse features of the matrix and it yields the fastest software implementation to date for such searches.

Key words: dynamic programming, similarity search.

1 Introduction

Consider the problem of searching a database of protein sequences looking for those that are similar to a given query sequence. By similar we mean that the query and the target sequence in question have a local alignment between a substring of each that scores above a specifiable threshold under a specifiable alignment scoring scheme. The first algorithm for this problem was given by Smith and Waterman [12] whose names have now become synonymous with the search. A Smith-Waterman search is considered a *full-sensitivity* search as it is guaranteed to find all local alignments above a threshold, whereas the popular heuristics BLAST [1] and FASTA [8] miss some matches. While its a considered a standard search, it is used less often because it is the most computationally expensive search due to its basis in dynamic programming.

Techniques for speeding up Smith-Waterman searches have primarily involved appeals to hardware parallelization. A coarse-grained parallelization is achieved by evenly partitioning the database to be searched and assigning each partition to a processor of a MIMD machine or network of workstations [7]. Fine-grained parallelization of the dynamic programming

*Informatics Research, Celera Genomics, Rockville, MD, USA (email: Gene.Myers@celera.com).
†Sanger Centre, Hinxton Hall, Cambridgeshire, UK.

algorithm using SIMD supercomputers or systolic arrays has also been reported and involves assigning an array of processing elements to the cells on an anti-diagonal of the dynamic programming matrix [9]. While performance gains scale in the number of processors, they require a correspondingly greater investment in hardware. We are concerned in this paper with trying to speed Smith-Waterman searches by improving the software, that is, by designing a cleverer algorithm. The best current implementation of Smith-Waterman search is Phil Green's SWAT code [5] which attains it speed by a direct appeal to sparsity and some machine-level coding tricks. An example of sparsity lies in the basic observation that under common scoring schemes over 60% of the dynamic programming matrix has value 0, and it is simple to avoid computing most of these "unproductive" entries. In this paper, we explore the approach of building a set of tables that allow us (1) to perform several steps of the underlying dynamic programming computation in a single lookup, and (2) to more fully exploit sparsity, often eliminating over 90% of the entries because they are unproductive (i.e. cannot be part of a local alignment above threshold).

This work focuses specifically on improving protein similarity searches as they are conducted in practice. That is we consider sequences over the 23 letter amino acid alphabet that includes the three ambiguity codes B, Z, and X, and we consider evaluating the score of alignments with (a) substitutions matrices such as the PAM120 or BLOSUM62 matrix [3, 6], and (b) affine gap penalties [4]. Recall that a gap of length n is scored $a+bn$ under the affine gap model where a and b are user-specifiable scores for (1) the introduction of a gap, and (2) for each symbol in the gap, respectively. Accommodating such gap scores complicates the underlying dynamic programming computation and the design of the corresponding speedup tables. But the greatest difficulty arises from the fact that the scoring schemes are designed to extract very weak signals from the background and this makes it impossible to easily eliminate whole-sale regions of the dynamic programming matrix as is done in some recent algorithms [2, 11] for approximate string matching involving comparatively stringent identity matches as their match criterion. Moreover, the large range of integer scores and alphabet size preclude a direct application of the Four-Russians table-based approach [13] that computes the dynamic programming matrix by consulting a universal table of solutions to all possible $u \times v$ sub-matrices. Specifically, for a uniform gap cost scoring scheme such a table would involve $((2g+\sigma+1)|\Sigma|)^{u+v}$ entries where g is the gap penalty, σ is the maximum substitution score, and $|\Sigma|$ is the size of the underlying alphabet. For a realistic combination of $g = 8$ and the PAM120 scoring scheme, where $\sigma = 12$, the Four-Russians table would involve 667^{u+v} entries. Thus even a 2×1 table would be impossibly large (296 million entries) and this does not even consider the further complication of affine gaps costs.

After carefully studying the characteristics of the underlying dynamic programming matrix and after much experimentation, we arrived at the approach we present below. Typically, this approach examines and computes only 4% of the underlying dynamic programming matrix and can do so 2 or 3 database symbols at a time using only a constant amount of time per entry examined. The steps per entry are not complex so it is the case that we have significantly reduced the number of instructions executed per database comparison, probably on the order of 30-fold over the basic Smith-Waterman algorithm. Nonetheless, the resulting algorithm is only anywhere from break-even to twice as fast as the tuned SWAT program depending on the scoring scheme and sequences involved. The reason for this phenomenon is based in the poorer caching behavior of table-based algorithms: the tables are large and the reference pattern into the tables is relatively random implying many cache misses which slow memory references by a factor as high as ten. Despite this our algorithm is still the

fastest software-based algorithm for Smith-Waterman searches. For searches involving more selective scoring schemes, the improvement of our algorithm over implementations based on dynamic programming should widen.

2 Preliminaries

We begin the development of our algorithm with the case of uniform gaps and extend it to affine gaps at the last. Our problem is formally as follows. We are given:

1. an alphabet Σ over which sequences are composed,
2. a $|\Sigma| \times |\Sigma|$ substitution matrix S giving the score, $S[a][b]$, of aligning a with b,
3. a uniform gap penalty $g > 0$,
4. a query sequence $Query = q_1 q_2 \ldots q_P$ of P letters over alphabet Σ,
5. a target sequence $Target = t_1 t_2 \ldots t_N$ of N letters over Σ, and
6. a threshold $T > 0$.

The problem is to determine if there is an alignment between a substring of $Query$ and a substring of $Target$ whose score is greater than or equal to threshold T. It should be noted that while not formally necessary, the problem makes little practical sense unless the substitution matrix S consists of both positive and negative scores and is negatively biased (see, for example, Figure 2).

We assume that the reader is familiar with the *edit graph* formulation [10] of sequence comparison where the problem of finding a high-scoring local alignment between $Query$ and $Target$ is mapped to one of finding a high-scoring path in an edge-weighted, directed graph whose vertices are arranged in a $(P+1) \times (N+1)$ array as illustrated in Figure 1. Briefly reviewing its construction, each diagonal substitution edge from $(i-1, j-1)$ to (i, j) models aligning q_i and t_j and has weight or score $S[q_i][t_j]$, each horizontal deletion edge from $(i-1, j)$ to (i, j) models leaving q_i unaligned and has weight $-g$, and each vertical insertion edge from $(i, j-1)$ to (i, j) models leaving t_j unaligned, also with weight $-g$. There is then a one-to-one correspondence between local alignments of say $Query[g \ldots i]$ and $Target[h \ldots j]$ and paths from $(g-1, h-1)$ to (i, j) where the score of the alignment equals the weight or score of the path. Thus, in the edit graph framework, our problem is to determine if there is a path in the graph whose score is T or greater.

The first observation is that it suffices to limit our attention to *prefix-positive* paths. A prefix of a path is a sub-path consisting of some initial sequence of its edges. A path is prefix-positive if the score of every non-empty prefix of the path is positive. Given an arbitrary path of positive score T, one obtains a prefix-positive sub-path of score T or greater by eliminating the prefix (if any) with the most negative score. Thus if there is a path of score T or greater in the edit graph of $Query$ and $Target$ then there is a prefix-positive path of score T or greater. While we will not use it here, one could go on to define suffix-positive paths and argue that it suffices to consider only paths that are both prefix- and suffix-positive.

Now consider the sequence of edges in a prefix-positive path. Let S denote a substitution edge, I an insertion edge, and D a deletion edge. Then the sequence of edge types must be in the regular language $S(I|S|D)^\star$ as the first edge must be positive and only substitution edges

score so. We may equivalently write this as $SD^\star((S|I) D^\star)^\star$ which in plain words asserts that a path can be partitioned into sub-paths such that each subpath is a substitution or insertion followed by a possibly empty series of deletions, save for the first which must begin with a substitution. We define the *length* of a path to be the number of substitution and insertion edges within it, or equivalently, to be the number of target symbols in its corresponding alignment. Thus a path from (g, h) to (i, j) has length $j - h$.

The basic dynamic programming algorithm for Smith-Waterman searches involves computing $C(i, j)$, the score of a heaviest or maximal path ending at vertex (i, j). It is not difficult to observe the following recurrence [12]:

$$C(i,j) = \max\{0, C(i-1,j-1) + S[q_i][t_j], C(i-1,j) - g, C(i,j-1) - g\}$$

The 0 term models the possibility of the null-path that begins and ends at (i, j), the other three terms capture the possibilities of reaching (i, j) by a maximal path whose last edge is a substitution, deletion, or insertion, respectively. Note that the recurrence is true regardless of whether we consider $C(i, j)$ to be the score of a maximal path ending at (i, j) or that of a maximal prefix-positive path to (i, j) as these are the same.

For the purposes of illustration we will use, except where noted otherwise, the example of S being the PAM120 substitution matrix shown in Figure 2 and $g = 12$. In this case, one observes that, on average, 62% of the $(P + 1) \times (N + 1)$ matrix of C-values is 0 when a random protein is compared against another[1]. It is possible to take advantage of the sparsity of non-zero values by arranging a computation which spends time proportional to the number of such entries in the C-matrix. We sketch such an approach in the remainder of this section as a preliminary exercise.

Consider a set p of index-value pairs $\{(i, v) : i \in [0, P]\}$ where each index is to a column of the edit graph. In what follows, the pairs will model the column and score of a path ending in a given row of the edit graph. We call such collections *path sets*. The following operators on path sets will be used to succinctly describe forthcoming algorithms. Given two path sets p and q let their merged union, $\cup^{max}$ be

$$p \cup^{max} q = \{(i, \max u) : (i, u) \in p \cup q\}$$

[1]We assumed an average length of 300 residues and generated each "random" protein via a series of Bernouilli trials that selected residues with the frequency with which they are found in Version ??? of Genbank.

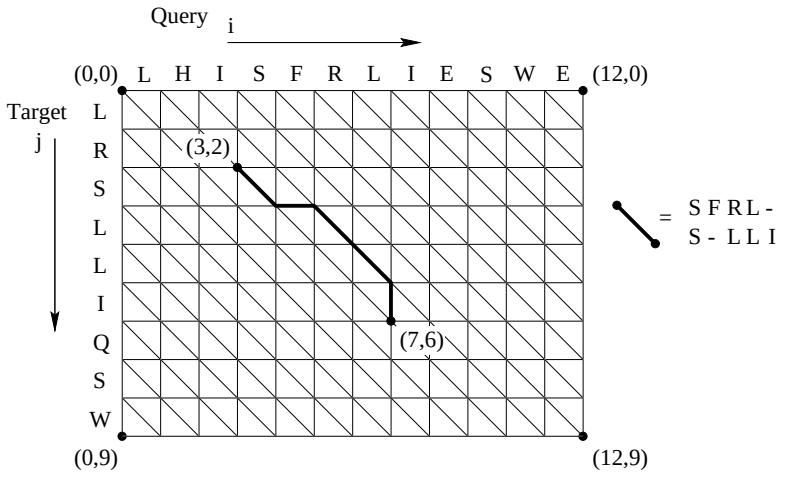

Figure 1: A sample edit graph and local alignment.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X
A	3																						
R	-3	6																					
N	0	-1	4																				
D	0	-3	2	5																			
C	-3	-4	-5	-7	9																		
Q	-1	1	0	1	-7	6																	
E	0	-3	1	3	-7	2	5																
G	1	-4	0	0	-5	-3	-1	5															
H	-3	1	2	0	-4	3	-1	-4	7														
I	-1	-2	-2	-3	-3	-3	-3	-4	-4	6													
L	-3	-4	-4	-5	-7	-2	-4	-5	-3	1	5												
K	-2	2	1	-1	-7	0	-1	-3	-2	-2	-4	5											
M	-2	-1	-3	-4	-6	-1	-4	-4	-4	1	3	0	8										
F	-4	-4	-4	-7	-6	-6	-6	-5	-2	0	0	-6	-1	8									
P	1	-1	-2	-2	-3	0	-1	-2	-1	-3	-3	-2	-3	-5	6								
S	1	-1	1	0	-1	-2	-1	1	-2	-2	-4	-1	-2	-3	1	3							
T	1	-2	0	-1	-3	-2	-2	-1	-3	0	-3	-1	-1	-4	-1	2	4						
W	-7	1	-5	-8	-8	-6	-8	-8	-5	-7	-5	-5	-7	-1	-7	-2	-6	12					
Y	-4	-6	-2	-5	-1	-5	-4	-6	-1	-2	-3	-6	-4	4	-6	-3	-3	-1	8				
V	0	-3	-3	-3	-2	-3	-3	-2	-3	3	1	-4	1	-3	-2	-2	0	-8	-3	5			
B	0	-2	3	3	-6	0	2	0	1	-3	-5	0	-4	-6	-2	0	-1	-7	-4	-3	4		
Z	-1	-1	0	2	-7	4	3	-2	1	-3	-3	-1	-3	-6	-1	-2	-2	-7	-5	-3	1	5	
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2: PAM120 Scoring Matrix.

That is, for index-value pairs in p and q with the same index, keep the pair with the greatest value. Furthermore for a path set p and symbol $a \in \Sigma$, let

$$Advance(p, a) = \{(i, v - g) : (i, v) \in p\} \ \cup^{max} \ \{(i + 1, v + S[q_{i+1}][a]) : (i, v) \in p\}$$

Informally, $Advance(p, a)$ extends the paths in p with a substitution or insertion edge presuming the next target sequence letter is a. To capture the effect of extending by a (possibly empty) sequence of deletion edges, let

$$Delete_\tau(p) = \cup^{max}\{(i + \Delta, v - \Delta g) : (i, v) \in p \text{ and } v - \Delta g > \tau\}$$

Finally, the path set $Start(a)$ will define the set of paths that begin with a positive substitution edge on target letter a:

$$Start(a) = \{(i, v) : v = S[q_i][a] > 0\}$$

The value of $Start(a)$ is computed in an $O(P|\Sigma|)$ preprocessing step for each value of $a \in \Sigma$ and stored in a table for use during the search proper.

With these definition in hand we now turn to the problem of computing the non-zero values of the C-matrix row by row. Let Pos_j be the path set $\{(i, v) : v = C(i, j) > 0\}$ that models the non-zero values in row j. A positive entry in row j of the edit graph is the value of a prefix-positive path that is either (a) a maximal prefix-positive path to row $j - 1$ followed by an insertion or substitution edge to row j followed by a possibly empty sequence of deletion edges in row j, or (b) a positive weight substitution edge from row $j - 1$ to row j followed by a possibly empty sequence of deletion edges in row j. It thus directly follows that

$$Pos_j = \begin{cases} Delete_0(Advance(Pos_{j-1}, t_j) \cup^{max} Start(t_j)) & \text{if } j \geq 1 \\ \emptyset & \text{if } j < 1 \end{cases}$$

Thus we have a computable recurrence for Pos_j in terms of Pos_{j-1}. All that remains is to examine how efficiently the computation can be done.

Once a table of $Start$ values has been precomputed, one can perform any number of searches against targets in a database in $O(N + Pos)$ *search* time per target, where $Pos = \Sigma_j |Pos_j|$ is the number of non-zero entries in the C-matrix. With each path set modeled as an index-ordered list, it is an easy exercise to compute each intermediate set in the recurrence for Pos_j in time proportional to the sum of the sizes of its input and output path sets. $|Start(t_j)| \leq |Pos_j|$ and $|Advance(Pos_{j-1}, t_j)| \leq 2|Pos_{j-1}|$, so their $\cup^{max}$-merge can be computed in $O(|Pos_{j-1}| + |Pos_j|)$ time. A traversal of this list, culling non-positive entries and propagating delete edges, accomplishes the effect of $Delete_0$ in the same time, because the size of the result is $|Pos_j|$. Simply checking the Pos_j list for an entry greater than or equal to T completes the computation in the desired time.

In practice, the algorithm is implemented most efficiently by modeling path sets as a stack of non-zero positions and an $O(N)$ array explicitly modeling a matrix row. This is explained in Appendix A. In the practical case of comparison under PAM120 scores with a uniform gap penalty of 12, this simple algorithm results in only 38% of the matrix being computed on average. However, the increased logic required to compute each entry requires about 2.74 times longer than the basic dynamic programming algorithm, making our sparse algorithm take about 7% more time on average.

3 The Start and Extension Tables

Entries that are zero in the dynamic programming matrix are clearly "unproductive" in that a prefix-positive path does not pass through such a vertex. In this section we lever this basic idea further by building tables that help us avoid expending effort on additional unproductive vertices. The size of our tables and the degree to which they screen the matrix will be determined by two small parameters k_S and k_E which in practice might typically be set to 4 and 3, respectively. Let the length of a path or alignment be the number of symbols in the target subsequence it aligns. Consider a vertex x in row j of the edit graph of $Query$ vs. $Target$ and the following definitions:

1. x is a *seed* if there exists a prefix-positive path to x that is of length k_S, i.e., that starts in row $j - k_S$.

2. x is *startable* if there exists a positive path to x whose prefix of length k_S is a seed.

3. x is *extendable* if there exists a positive path through x to row $j + k_E$ whose suffix of length k_E is prefix positive, i.e., it has a prefix-positive "extension" of length k_E beyond x.

4. x is *productive* if it is both startable and extendable.

Let $Seed$, $Startable$, $Extensible$, and $Productive$ be the sets of vertices that are seeds, are startable, are extendable, and are productive, respectively. Note that $Seed$ is a proper subset of $Startable$ as the suffix of a long prefix-positive path that spans the last k_S rows need not be prefix-positive.

In this section we show how to build tables for a given query that with an $O(1)$ lookup deliver the startable vertices for any target substring of length k_S, and indicate if a given vertex is extendable given any ensuting target substring of lenght k_E. Using these tables allows an algorithm that compare the query to a target in $O(N + |Productive| + |Seed|)$

k_S	*Startable*	*Seed*
1	37.8%	22.1%
2	24.1%	11.2%
3	17.2%	6.9%
4	12.7%	4.6%

k_E	*Extensible*
1	23.4%
2	15.6%
3	12.7%
4	10.3%

Productive		k_S			
		1	2	3	4
k_E	1	23.4%	16.1%	11.8%	9.2%
	2	15.6%	11.3%	8.5%	6.9%
	3	12.7%	9.4%	6.9%	5.3%
	4	10.3%	7.9%	6.6%	4.5%

Table 1: Sparsity of Table-Computed Vertex Sets.

time. While the times to build the tables will be significant, keep in mind, that they need only be built once for a given query and then may be used to scan the entire database. Also note during the following development that the algorithm of the previous section is just a special case where $k_S = 1$ and $k_E = 0$. Table 1 gives the expected percentage of the matrix that is in each of the sets *Seed*, *Startable*, *Extensible*, and *Productive* for a variety of values of k_S and k_E in the case where S is the PAM120 scoring matrix and $g = 12$.

3.1 Start Trimming

Limiting dynamic programming to the startable vertices requires a table $Start(w)$ where w ranges over the $|\Sigma|^{k_S}$ possible sequences of length k_S. The entry for a given sequence w is defined as follows:

$$Start(w) = \{\ (i, v) : \exists \text{ prefix-positive path from row 0 to vertex } (i, k_S) \text{ in the edit graph of } Query \text{ vs. } w, \text{ and } v \text{ is the score of a maximal such path } \}$$

The entire table may be computed efficiently using the simple sparse algorithm of the previous section for figuring the path set for each entry w and by factoring the computation of entries that share a common prefix as detailed in the following recurrence:

$$\begin{aligned} Start(\varepsilon) &= \{(i, 0) : i \in [0, P]\} \\ Start(wa) &= Delete_0(Advance(Start(w), a)) \end{aligned}$$

Effectively $Start(w)$ is computed for every sequence of length k_S *or less*, with the path set for wa being computed incrementally from that for w. This reduces the worst-case time for computing the entire table *Start*, $O(P|\Sigma|^{k_S})$, by a factor of k_S over the brute-force algorithm. Let α be the expected percentage, $|Seed|/(N{+}1)(P{+}1)$, of vertices that are seed vertices in a comparison, for example, as reported in Table 1. It follows by definition that the expected size of the path set for an entry of the *Start* table is αP and thus that the entire table occupies $O(\alpha P|\Sigma|^{k_S})$ space in expectation. With care, only 4 bytes are required for each index-value pair in a path set and the table can be realized as an array of pointers to sequentially allocated path sets giving a concrete estimate of $(4\alpha P{+}4)|\Sigma|^{k_S}$ bytes on a 32-bit address machine. Efficiently looking up an entry w is simply a matter of converting w to an

integer index in the range $[0, |\Sigma|^{k_S}-1]$ in the usual fashion. Returning to the time to compute the table, note that there is a greater degree of sparsity in this computation over that of a search comparison as $Start(w)$ becomes sparser geometrically as w gets longer. Hence, of the $(P+1)(|\Sigma|^{k_S+1}-1)/(|\Sigma|-1)$ dynamic programming entries that would be computed in the worst case computation of the entire $Start$ table, only a small percentage, β, are actually computed. Indeed β is $O(\alpha)$ in expectation as the deepest level of the recursion for $Start$ dominates the cost. So in expectation $Start$ is computed in time proportional to its size.

With the $Start$ table in hand, one can easily limit the dynamic programming computation of $Query$ vs. $Target$ to the startable vertices in each row as follows. Let $Stable_j$ be the path set $\{(i,v) : (i,j)$ is a startable vertex and $v = C(i,j)\}$. It follows directly from the definition of a startable vertex and the structure of prefix-positive paths that:

$$Stable_j = \begin{cases} Delete_0(Advance(Stable_{j-1}, t_j) \cup^{max} Start(Target[j-(k_S-1)\ldots j])) & \text{if } j \geq k_S \\ \emptyset & \text{if } j < k_S \end{cases}$$

Incrementally converting each successive subsequence $Target[j-(k_S-1)\ldots j]$ of the target into an integer index into the $Start$ table takes constant time, so the above recurrence outlines an $O(N + |Startable|)$ algorithm for detecting local alignments at or above threshold T.

3.2 Extension Trimming

We now turn to the development of a table that eliminates vertices that are not extendable. For a given sequence w of length k_E and a column index in the range $[0, P]$, let:

$$Extend(i,w) = min\{\ drop(p) :\ p \text{ is a path from } (i,0) \text{ to row } k_E \text{ in the edit graph of } Query \text{ vs. } w\ \}$$

$$\text{where } drop(p) = -\min\{\ score(q) :\ q \text{ is a prefix of } p\ \} \geq 0.$$

That is, $Extend(i,w)$ gives the least decrease in score that will be encountered along a path from column i when the next k_S target symbols are w. Thus (i,j) is an extendable vertex if and only if $C(i,j) > Extend(i, Target[j+1\ldots j+k_E])$.

Computing the $Extend$ table involves the following variation of the basic dynamic programming algorithm. Let $E_w[0\ldots P]$ be a $(P+1)$-vector of numbers for which $E_w[i] = -Extend(i,w)$. It is a straightforward exercise in induction to verify the following recurrences:

$$\begin{aligned} E_\varepsilon[i] &= 0 \\ E_{aw}[i] &= min(0, max(E_w[i+1] + S[q_{i+1}][a], E_w[i] - g, E_{aw}[i+1] - g)) \end{aligned}$$

As for the $Start$ table, a factor of k_E in efficiency is gained by factoring the computation of $Extend(i,w)$ for different w but this time for sequences sharing a common *suffix*. Working "backwards" by extending suffixes is required in this case as we are interested in optimal paths *starting* in a given column as opposed to ending in one. Overall the computation of the $Extend$ table takes $O(P|\Sigma|^{k_E})$ time and the table occupies the same amount of space. In practice, the values in the $Extend$ table are small enough to fit in a single byte so that the table occupies exactly $(P+1)|\Sigma|^{k_E}$ bytes.

Once the $Extend$ table is computed it can be used to prune vertices that are not extendable during the search computation. The point at which to do so is during the extension of path sets along deletion edges. To this end define:

$$Delete_E(p,w) = \cup^{max}\{(i+\Delta, v-\Delta g) : (i,v) \in p \text{ and } v - \Delta g > Extend(i+\Delta, w)\}.$$

Since $Extend(i,w) \geq 0$ for all i and w it follows that $Delete_E(p,w) \subseteq Delete_0(p)$ for all p and w. One must trim vertices that are not extendable during the deletion extension of an index-value pair, as opposed to afterwards, in order to avoid potentially wasting time on a series of vertices that are all not extendable along a chain of deletion edges. Stopping at the first vertex that is not extendable in such a chain is correct as $Extend(i,w)-g \leq Extend(i+1,w)$ implies that all vertices reached from it by virtue of a deletion are also not extendable.

Let $Prod_j$ be the path set $\{(i,v) : (i,j)$ is a productive vertex and $v = C(i,j)\}$. Then by definition it follows that:

$$Prod_j = \begin{cases} Delete_E(A\_S(j), Target[j+1 \ldots j+k_E]) & \text{if } j \geq k_S \\ \emptyset & \text{if } j < k_S \end{cases}$$

$$\text{where} \quad A\_S(j) = Advance(Prod_{j-1}, t_j) \cup^{max} Start(Target[j-(k_S-1) \ldots j])$$

Computing the indices into the *Start* and *Extend* tables can easily be done incrementally at an overhead of $O(N)$ time over the course of scanning *Target*. The time to compute $Prod_j$ from $Prod_{j-1}$ is as follows. *Advance* takes time $O(|Prod_{j-1}|)$. The merge of this result with the *Start* table path set takes time proportional to the sum of the lengths of the two lists, i.e., $O(|Prod_{j-1}| + |Seed_j|)$, where $Seed_j$ is the set of seed vertices in row j. Finally computing $Delete_E$ takes no longer than the sum of the size of the path set given to it and the size of the path set that results, i.e., $O(|Prod_{j-1}| + |Seed_j| + |Prod_j|)$. Thus the overall time to search *Target* is $O(N + \Sigma_j|Prod_j| + \Sigma_j|Seed_j|) = O(N + |Productive| + |Seed|)$ as claimed at the outset.

A subtlety ignored to this point is that alignments of score T or more can end on non-productive vertices when T is sufficiently small, specifically when $T \leq max(\sigma(k_S-1), g(k_E-1))$, where σ is the largest entry in the scoring matrix S. This happens in two ways: when the alignment is of length less than k_S and so does not give rise to a seed vertex, and when the last vertex of the alignment is followed by such negative scoring edges that all extensions drop to 0 in less than k_E rows. How to make the algorithm detect these cases at no extra asymptotic cost is shown in Appendix B. For now we note that to create such instances T has to be set to an unrealistically small value. For example, for our running example of PAM120 and $g = 12$, with $k_S = k_E = 3$, T would have to be 24 or less, and typically users never set T below 50.

Table 2 shows the time and space to produce the tables, and the search time for given choices of k_S and k_E. Time and space for both tables increase geometrically with k, with the factor for the *Start* table being somewhat less because the number of seeds per entry is also decreasing geometrically. For running time, one notes that times generally decrease as the k parameters increase save that when k_E goes from 2 to 3, times actually increase in columns for larger k_S, so that the best overall time is obtained with $k_S = 4$ and $k_E = 2$. The reason for this is due to memory caching behavior on todays hardware. As processor speeds have gotten ever faster, memory speeds have not, so that for todays machines any memory cache miss typically incurs slowdowns of more than a factor of 10 over accesses that are in cache.[2] As the *Start* and *Extension* tables get larger, it becomes more and more likely that an access into the table will not be in the cache as the access pattern to these tables is more or less random. Thus the observed behavior: even though only 1/20th of the d.p. matrix is being computed for $k_S = 4$ and $k_E = 2$, only a factor of 2.6 is gained in time over the naive algorithm. Nonetheless, this is not an inconsequential performance gain.

[2]All timing experiments were performed on a T22 IBM laptop with 256Mb of memory and a 1Ghz Pentium III processor with 256Kb cache, running SuSE Linux 7.1.

Start k_S	*Preprocessing* *Time(secs)*	*Space* *(Kb)*
1	.00027	5
2	.0019	62
3	.025	892
4	.38	13,390

Extend k_E	*Preprocessing* *Time(secs)*	*Space* *(Kb)*
1	.00053	7
2	.0081	159
3	.19	3,650
4	4.3	83,952

Search *Time(secs)*		k_S = 1	2	3	4
k_E	off	201	127	96	81
	1	177	115	94	82
	2	132	91	79	72
	3	118	92	87	78

vs. Dynamic Programming: 187

Table 2: Times and space for sparse SW-searches on queries of length 300 and a database of 100,000 300 residue entries.

4 A Table-Driven Scheme for Dynamic Programming

In addition to restricting the SW computation to productive vertices, one can further develop a "jump" table that captures the effect of *Advance* and *Delete* over $k_J > 0$ rows. For each sequence of k_J potential row labels w and each column position i, an entry $Jump(i, w)$ gives the change in score that can be obtained at each column position $k \geq i$. That is, $Jump(i, w) = \{(k, u) : \text{the maximal path from } (0, i) \text{ to } (k_j, k) \text{ is } u \text{ in the edit graph of } w \text{ versus } Query\}$.

As formulated the table *Jump* is unmanageably large at $O(\Sigma^{k_J} P^2)$ space. However, this is greatly improved by realizing that one need only record those (k, u) for which $u > -(T-1)$. One will never jump from a vertex of score T or more as the algorithm can quit as soon as it encounters such a vertex. Jumping from a vertex of score less than T with a change of $-(T-1)$ or more leads to a non-positive entry and so need not be considered. We can thus formulate the computations of the *Jump* table as:

$$\begin{aligned} Jump(i, \varepsilon) &= Delete_{-(T-1)}(\{(i, 0)\}) \\ Jump(i, wa) &= Advance(Delete_{-(T-1)}(Jump(i, w)), a) \end{aligned}$$

If we truly wish to compute only critical entries on every other k_J^{th} row of the dynamic programming matrix, then care must be taken to model all possible seed paths that can lead to the next row to be computed. For example, if $k_J = 2$, so that only every even row is actually computed, and $k_S = 4$, then one might miss seeds that end in an odd row. The required fix is to modify the $Start(w)$ table so that it models all seeds of length $k_S-(k_J-1)$ to k_S that align to a suffix of w. While this decreases the sparsity of the table, the loss is offset by the gain in being able to skip rows. So the *Start* table is now defined by the recurrence:

$$\begin{aligned} Start(\varepsilon) &= \{(i, 0) : i \in [0, P]\} \\ Start(wa) &= \begin{cases} Delete_0(Advance(Start(w), a)) \cup^{max} Start(\varepsilon) & \text{if } |w| < k_J - 1 \\ Delete_0(Advance(Start(w), a)) & \text{otherwise} \end{cases} \end{aligned}$$

The *Jump* table considers all paths from the current row to a future one that begin with a series of deletion edges. Moreover, if there is a path with score T or more that ends with

a deletion, there is always one with such score that ends on a substitution, as deletion edges have negative score. Thus it is no longer necessary in the search algorithm to explicitly compute entries in a given row that can obtained by deletion edges from another entry in the same row. So we can save time in the main loop by not having to invoke $Delete_E$, but rather letting the application of the $Jump$ table take care of its effect. Moreover, we can save a little space in the $Jump$ and $Start$ tables by removing entries in a path set that can be obtained through deletion from other entries in the path set. Specifically, define

$$\begin{aligned} Nodels(p) &= p - \{(i,v) : \exists (k,u) \in p \text{ s.t. } k < i \text{ and } v \leq u-(i-k)g\} \\ Jump'(i,w) &= Nodels(Jump(i,w)) \\ Start'(w) &= Nodels(Start(w)) \end{aligned}$$

Unlike the algorithms of the proceeding sections, the tactic of letting the $Jump$ table handle deletions implies that the set of entries, $Cand_j$, explicitly computed for every k_J^{th} row is a subset of the productive entries in the row and a superset of the entries not producible by deletions. That is, $Nodels(Prod_j) \subseteq Cand_j \subseteq Prod_j$. To compute $Cand_j$, one first applies the $Jump$ table to the entries of $Cand_{j-k_J}$ and then merges these with all the seeds of lengths between $k_S-(k_J-1)$ and k_S ending on row j. Finally, this aggregate set of entries on row j are checked for extendability. Putting this together formally in recurrences:

$$Cand_j = Trim_E(\begin{array}{c} JUMP(Cand_{j-k_J}, Target[j-(k_J-1)\ldots j]) \\ \cup^{max} \; Start'(Target[j-(k_S-1)\ldots j]) \end{array} , Target[j+1\ldots j+k_E])$$

$$\text{where} \quad JUMP(p,w) = \{(i+k, v+u) : (i,v) \in p \,\text{and}\, (k,u) \in Jump'(i,w) \text{ and } v+u > 0\}$$

$$\text{and} \quad Trim_E(p,w) = \{(i,v) \in p : v > Extend(i,w)\}$$

Efficiently implementing $JUMP$ involves organizing the index-value pairs for a given entry $Jump(i,w)$ as a list ordered on the value of the pairs, not their index. Thus when jumping from an entry (i,v) one processes the offsets (k,u) in order until $v+u \leq 0$ at which point no further elements of the list need be considered.

The remaining detail is to modify the check for paths scoring T or more. One should, of course examine every entry in $Cand_j$, but since one is skipping k_J-1 rows, it is necessary to check if there is an extension of each of these that has score over T in the skipped rows. To this end let $Peak(i,w)$ be the maximum score acheived on all possible paths starting at $(i,0)$ in the matrix of $Query$ versus w. Like the analysis for the $Extend$ table, except easier, $Peak(i,w) = P_w[i]$ where:

$$\begin{aligned} P_\varepsilon[i] &= 0 \\ P_{aw}[i] &= max(0, P_w[i+1] + S[q_{i+1}][a], P_w[i] - g, P_{aw}[i+1] - g)) \end{aligned}$$

To test for paths of score T or greater, it simply suffices to check if $v + Peak(i, Target[j+1\ldots j+(k_J-1)]) \geq T$ for each (i,v) in $Cand_j$. The total space for the $Peak$ table is $O(|\Sigma|^{k_J-1}P)$.

It is no longer possible to give a statement of the complexity of the resulting algorithm in simple terms as $JUMP(p,w)$ potentially involves enumerating a position of the underlying dynamic programming matrix several times. It simply becomes an empirical question as to whether in practice this inefficiency is offset by the fact that one no longer needs to consider deletion sequences and that one skips rows when $k_J > 1$. In Table 3 one sees that the time and space for producing the jump table increase geometrically with the highest reasonable

$Jump'$ k_J	$Preprocessing$ $Time(secs)$	$Space$ (Kb)
1	0.0031	32
2	0.098	1,175
3	2.18	35,786

$Seed$		k_S			
$Vertices$		1	2	3	4
k_J	1	22.1%	11.2%	6.9%	4.6%
	2		27.3%	15.4%	9.7%
	3			31.9%	18.4%

$Start'$		k_S			
$Space(Kb)$		1	2	3	4
k_J	1	5	62	880	14,020
	2		156	1,949	28,356
	3			3,966	52,677

$Search\ Time\ (secs)$

$k_J = 1$		k_S			
		1	2	3	4
k_E	1	130	88	76	63
	2	102	76	75	62
	3	112	92	88	79
	4	102	89	85	78

$k_J = 2$		k_S		
		2	3	4
k_E	1	134	110	94
	2	126	107	87
	3	89	80	72
	4	78	72	62

$k_J = 3$		k_S	
		3	4
k_E	1	102	85
	2	84	71
	3	94	72
	4	64	56

Table 3: Times and space for sparse, jump-based SW-searches on queries of length 300 and a database of 100,000 300 residue entries.

value of k_J being 3. In the second row of tables, one sees that for every row skipped there is a corresponding increase in the number of verices that are seeds and in the size of the $Start$ tables. If $Seed(s, j)$ is the fraction of seeds for $k_S = s$ and $k_J = j$, then one sees that $Seed(s, j) \leq \Sigma_{t=s-j+1}^{s} Seed(t, 1)$ as a seed for a given k_S is often a seed for $k_S - 1$. Running time is a complex function of the three k parameters due to the stochastic nature of competition for the cache as the parameters increase. In particular, increasing k_J doesn't win, *unless* one is running with large values of k_S and k_E. The basic insight is that the cost of the table lookups per row is such that skipping rows only wins if the number of entries per row is made small enough with the use of the $Start$ and $Extension$ tables. Overall, on our test machine, the best performance was acheived with $k_S = k_E = 4$ and $k_J = 3$. At 56 seconds for the search, this is 3.33 times faster than the our implementation of the original Smith-Waterman algorithm.

5 The Case of Affine Gap Costs

We now turn to the case where a gap of length n has an affine cost $g+hn$ for user specificiable constants $g, h > 0$. This variation of the problem was first solved by Gotoh who observed that one simply needed to maintin separate reccurences for the best path ending at (i, j) with an insertion, $I(i, j)$, and with a deletion $D(i, j)$. The resulting recurrence system is as follows :

$$\begin{aligned} C(i,j) &= \max\{0, C(i-1,j-1) + S[q_i][t_j], D(i,j), I(i,j)\} \\ D(i,j) &= \max\{D(i-1,j), C(i-1,j) - g\} - h \\ I(i,j) &= \max\{I(i,j-1), C(i,j-1) - g\} - h \end{aligned}$$

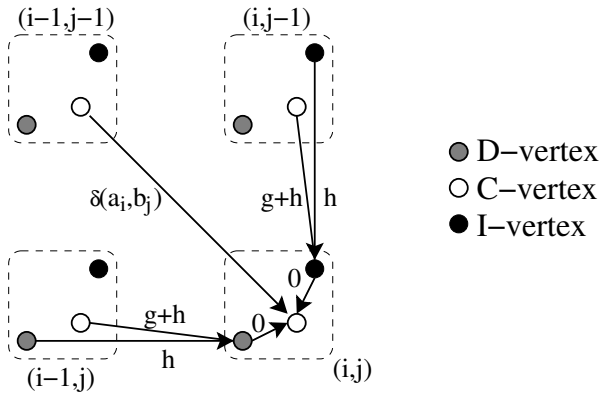

Figure 3: Affine edit graph edge and vertex structure.

An edit graph construction follows immediately from these recurrences where now at each coordinate (i, j) there are three types of vertices – C, D, and I – corresponding to the three recurrences. Figure 3 shows the structure of the graph and the weights of its edges. Just as in the uniform gap case, any path through this graph models an alignment and our objective is to determine if there are paths of score greater than τ.

Some simple observations follow directly from examining the dependencies and terms in the recurrences. First note that to compute the j^{th} row from the $(j-1)^{st}$ requires knowing only the vectors of C and I values in row $j-1$, and not on the D values in that row. The second observation is that if $I(i,j) \leq C(i,j) - g$ then the I value at vertex (i,j) need not be recorded as any maximal path through its I-vertex will have score less than the maximal path passing through the corresponding C-vertex. Similarly if $C(i,j) = I(i,j)$ (it can never be that $C(i,j) < I(i,j)$) then the C value at (i,j) need not be recorded as any maximal path through its C-vertex will have score less then the maximal path passing through the corresponding I-vertex.

To generalize the uniform case to the affine case, it suffice to generalize path sets for a given row to a set of index-value-*type* triples, $p = \{(i, v, t) : i \in [0, P] \text{ and } t \in \{C, I\}\}$. A triple (i, v, C) denotes that the C-vertex in column i can acheive value v, and a triple (i, v, I) denotes the same for the I-vertex. With this definition in place, the basic constructors – $\cup^{max}$, *Advance*, and *Delete* – are easily extended as follows:

$$p \cup^{max} q = \{(i, \max u, t) : (i, u, t) \in p \cup q\}$$

$$\begin{aligned} Advance(p, a) &= \{(i, v-(g+h), I) : (i, v, C) \in p\} \cup^{max} \{(i, v-h, I) : (i, v, I) \in p\} \\ &\quad \cup^{max} \{(i+1, v + S[q_{i+1}][a], C) : (i, v, ?) \in p\} \end{aligned}$$

$$Dvalues(p) = \cup^{max}\{(i+\Delta, w, C) : (i, v, ?) \in p \text{ and } w = v-(g+\Delta h) \text{ and } \Delta > 0\}$$

$$\begin{aligned} Delete_\tau(p) &= \{(i, v, C) : (i, v, C) \in Dvalues(p) \text{ and } v > \tau\} \\ &\quad \cup^{max}\{(i, v, t) : (i, v, t) \in p \text{ and } v > \tau\} \end{aligned}$$

Since all other tables – *Start*, *Jump*, and *Extend* – can be defined in terms of these basic functions the overall feasibility of the generalization follows.

Nonetheless there are a few details that need to be addressed. First, jumping from an I-term penalizes extensions beginning with an insertion less than from the corresponding C-term. Therefore, there actually need to be two jump tables, $Jump_C$ and $Jump_I$, one for each type of path set tuple. The difference is simply in the basis of the recurrence for each as follows:

$$\begin{aligned} Jump_C(i, \varepsilon) &= Delete_{-(T-1)}(\{(i, 0, C)\}) \\ Jump_I(i, \varepsilon) &= Delete_{-(T-1)}(\{(i, 0, I)\}) \end{aligned}$$

Search Time (secs)

$k_J = 1$		k_S			
		1	2	3	4
k_E	1	149	129	109	95
	2	144	124	117	104
	3	145	129	113	104
	4	131	113	106	100

$k_J = 2$		k_S		
		2	3	4
k_E	1	194	145	112
	2	164	126	107
	3	139	113	93
	4	115	98	86

vs. Dynamic Programming: 288

Table 4: Times and space for sparse, jump-based SW-searches with affine gaps on queries of length 300 and a database of 100,000 300 residue entries.

The same issue arises for extension values following a C- or I-term. However, in this case we found that it was in practice not worthwhile to develop two tables but simply to use the lower cutoff afforded by the two cases.

Finally, just as in the earlier treatment one should remove terms that can be obtained by deletion from another term (i.e. see $Nodels$). In addition, one should eliminate, wherever possible, C-terms for which the corresponding I-term has the same value, and I-terms for which the corresponding C-value is g or larger than it, as per the observations made immediately after the introduction of the recurrences. Capturing this in equations:

$$\begin{aligned} Nodels(p) &= p - \{(i,v,C) : (i,w,C) \in Dvalues(p) \text{ and } w \geq v\} \\ Reduce(p) &= p - \{(i,v,I) : (i,w,C) \in p \cup Dvalues(p) \text{ and } w - g \geq v\} \\ &\quad - \{(i,v,C) : (i,v,I) \in p\} \\ Jump'(i,w) &= Reduce(Nodels(Jump(i,w))) \\ Start'(w) &= Reduce(Nodels(Start(w))) \end{aligned}$$

This then completes a sketch of the modifications needed to extend our algorithm to the case of affine gap costs.

We now turn to the empirical performance of the algorithm. All measurements where with the PAM120 matrix and the affine gap cost $8 + 4n$. The time and space to build the $Extend$ table is as before whereas the $Start$ and two $Jump$ tables increase by fixed fractions to the extent that it is no longer possible to build $Jump$ tables for $k_J = 3$ in under 100Mb of memory. In table 4 run times are given as a function of k_S and k_E for k_J set to 1 and to 2. As in in the uniform gap case, time does not improve with larger k_J unless the other two parameters are large enough to make each row sufficiently sparse. The best overall time occurs when $k_S = k_E = 4$ and $k_J = 2$, and in this case the search time is 3.34 times faster than a tightly coded implementation of the basic dynamic programming algorithm.

In practice today, the code considered best by the community is Phil Green's SWAT program [5] that implements the Smith-Waterman algorithm with affine gap costs. In addition to a query specific scoring matrix, that we also used in our dynamic programming baseline, this code packs values into words and uses some basic properties of the recurrence to reduce the operation counts at each cell. In order to get a sense of how our algorithm performs against this standard program and on real, as opposed to simulated datasets, we ran a series of searches against a 3 million residue subset of the PIR database, with four different proteins — a periodic clock protein of length 173 (*pcp*), a lactate dehydrogenase of length 319 (*dehydro*), a cGMP kinase of length 670 (*kinase*), and a growth factor of length 1210 ($gfactor$). We ran our algorithm with several settings of (k_S, k_E, k_J) as shown in tables

Space (*Mb*)	*Ours* (3, 1, 1)	*Ours* (4, 1, 1)	*Ours* (4, 3, 2)	*Ours* (4, 4, 2)
pcp	1.1	7.8	22.3	68.6
dehydro	1.8	17.2	47.5	132.9
kinase	3.1	31.8	94.9	274.2
gfactor	5.6	54.3	166.6	—

Times	*Total* = *Setup* + *Search*				*Total*
(*secs*)	*Ours* (3, 1, 1)	*Ours* (4, 1, 1)	*Ours* (4, 3, 2)	*Ours* (4, 4, 2)	*Swat*
pcp	4.3 = .0 + 4.3	4.3 = .3 + 4.0	5.0 = .8 + 4.2	8.5 = 4.7 + 3.8	6.6
dehydro	12.4 = .0 + 12.4	11.8 = .6 + 11.2	12.3 = 1.9 + 10.4	18.9 = 9.6 + 9.3	14.1
kinase	29.3 = .1 + 29.2	24.6 = 1.1 + 23.5	21.4 = 4.0 + 17.4	36.4 = 21.2 + 15.2	28.9
gfactor	59.2 = .2 + 59.0	48.4 = 1.8 + 46.6	36.4 = 7.4 + 29.0	*Not run*	51.5

Table 5: Times and space of our algoriithm and SWAT on 4 different queries against a real protein database of 3 million residures under the PAM120 matrix and gap costs $8 + 4n$.

Space (*Mb*)	*Ours* (3, 1, 1)	*Ours* (4, 1, 1)	*Ours* (4, 3, 2)	*Ours* (4, 4, 2)
pcp	1.4	9.7	31.0	77.4
dehydro	2.3	22.1	66.7	152.1
kinase	4.5	40.0	131.2	318.6
gfactor	7.8	67.7	230.4	—

Times	*Total* = *Setup* + *Search*				*Total*
(*secs*)	*Ours* (3, 1, 1)	*Ours* (4, 1, 1)	*Ours* (4, 3, 2)	*Ours* (4, 4, 2)	*Swat*
pcp	5.3 = .0 + 5.3	5.3 = .3 + 5.0	5.8 = 1.1 + 4.7	9.5 = 5.2 + 4.3	7.4
dehydro	26.4 = .1 + 26.3	24.7 = .7 + 24.0	19.9 = 2.5 + 17.4	26.8 = 11.5 + 15.3	15.7
kinase	56.2 = .2 + 56.0	50.8 = 1.4 + 49.4	34.1 = 5.4 + 28.7	50.3 = 24.7 + 25.6	34.1
gfactor	98.6 = .3 + 98.3	87.4 = 2.3 + 85.1	55.8 = 9.6 + 46.2	*Not run*	60.5

Table 6: Times and space of our algoriithm and SWAT on 4 different queries against a real protein database of 3 million residures under the BLOSSUM62 matrix and gap costs $8 + 2n$.

5 and 6 and the SWAT program whose performs depends only on the scoring matrix and gap costs. We ran the series of trials with the PAM120 matrix and a gap cost of $8 + 4n$, and again with the BLOSUM62 matrix and a gap cost of $8 + 2n$. The timings for our program involve a setup time to build the tables that is only a function of the query, and a search time that is directly proportional to the size of the database. We report both these numbers and their total in the tables. *Swat* takes neglible startup time and runs in time linear in the size of the database. For the PAM120 scheme, our algorithm was consistently faster than SWAT for the parameter tuple $(4, 3, 2)$ and occasionaly more so for other choices. Moreover, search speed was the fastest for $(4, 4, 2)$ implying this combination will eventually give the best performance on a large a database, and is 1.5 to 2 times faster than SWAT's search speed. For the BLOSSUM62 scheme, our algorithm basically takes the same amount of total time with the search speed ranging from a small fraction faster to 1.5 times faster than that of SWAT's search speed. The basic reason for the lesser performance of our algorithm in the BLOSSUM62 case is do to the small per symbol deletion penalty of 2 versus 4 in the PAM120 case. This lower gap extension penaly reduces the effectiveness of the *Extension* table trimming and so more marginal entries remain alive.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. A basic local alignment search tool. *J. Mol. Biol.* 215 (1990), 403-410.

[2] W.I. Chang and E.L. Lawler. Sublinear expected time approximate matching and biological applications. *Algorithmica* 12 (1994), 327-344.

[3] M.O. Dayhoff, W.C. Barker, and L.T. Hunt. Establishing homologies in protein sequences. *Methods in Enzymology* 91 (1983), 524-545.

[4] O. Gotoh. An improved algorithm for matching biological sequences. *J. Molec. Biol.* 162 (1982), 705-708.

[5] P. Green. "`http://www.genome.washington.edu/phrap.docs/swat.html`" (1994).

[6] S. Henikoff and J.G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA* 89 (1992), 2264-2268.

[7] X. Huang. A space-efficient parallel sequence comparison algorithm for a message-passing multi-processor. *Int. J. Parallel Prog.* 18, 3 (1989), 223-239.

[8] D.J. Lipman, and W.R. Pearson. Rapid and sensitive protein similarity searches. *Science* 227 (1985), 1435-1441.

[9] D.P. Lopresti. P-NAC: A systolic array for comparing nucleic acid sequences. *Computer* 20, 7 (1987), 98-99.

[10] E. Myers. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 2 (1985), 251-266.

[11] E. Myers. A sublinear algorithm for approximate keyword matching. *Algorithmica* 12, 4-5 (1994), 345-374.

[12] T.F. Smith and M.S. Waterman. Identification of common molecular sequences. *J. Mol. Biol* 147 (1981), 195-197.

[13] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica* 15, 1 (1996), 50-67.

6 Appendix

6.1 Detecting Short High-Scoring Alignments

As noted in the paper proper, it is possible for there to be an alignment of length less than k_S that has score T or more when $T \leq \sigma(k_S - 1)$ (recall σ is the maximum entry in S). To handle this rare scenario it suffices to build a table $Short$ indexed by sequences w of length k_S that indicates if an alignment of score T or more exists between w and $Query$. It is easy

to compute *Short* as a byproduct of computing *Start* and it occupies only $|\Sigma|^{k_S}$ bytes in practice. A recurrence for computing *Short* is as follows:

$$\begin{aligned} Short(\varepsilon) &= \textbf{false} \\ Short(wa) &= Short(w) \textbf{ or } (\exists (i,v) \in Start(wa), v \geq T) \end{aligned}$$

Note carefully, that the table *Short* needs to be checked for every row, regardless of the use of a *Jump* table, so that $O(N)$ time is taken checking for short matches during a search.

6.2 Detecting Hits Ending on Non-Productive Vertices

An alignment or hit of score T or more can end on a non-productive vertex when all possible extensions of the alignment of length less than k_E drop by score more than T. Moreover, in the *Jump* table version of the algorithm productive vertices on skipped rows are not checked. Indeed, we treated this later possibility in the main text by introducting a table $Peak(w, i)$. To treat the case first case, simply requires that the table be over strings w of length $k_E - 1$ (if $k_E > k_J$). The fix is then to check every productive candidate (i, v) in a row j to see if $v + Peak(Target[j+1 \ldots j+(k_E-1)], i) \geq T$.

6.3 Implementation Nuances

Within the body of the paper, values in a given row are modeled by path sets consisting of a column-ordered list of column,value pairs. While one can implement the functions such as *Advance* on such a representation in linear time, we find that in practice a better representation is to have a 0-terminated list of all columns, c, in the path set (not necessarily ordered) and a $(P+1)$-element array, D for the row of the dynamic programming matrix where the corresponding value is in each column position within the array, and all other values are set to 0. For example, suppose $P = 10$ and we have the path set $p = \{(3, 10), (5, 3), (8, 6)\}$. Then a representation would be $c = \{5, 8, 3, 0\}$ and $D = [0, 0, 0, 10, 0, 3, 0, 0, 6, 0, 0]$. The reason this representation provides a better implementation than the path set representation, is that the core operation $\cup^{max}$ does not require a merge of lists, but simply uses the setting of the D array to see if there is a collision between two path sets.