Sequence Alignment using FastLSA

Kevin Charter, Jonathan Schaeffer, Duane Szafron
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1

Abstract ***For two strings of length*** m ***and*** n ***(***$m \leq n$***), optimal sequence alignment (as a function of the alignment scoring function) takes time and space proportional to*** $m \times n$ ***to compute. The time actually consists of two parts: computing the score of the best alignment (calculating*** $(m+1) \times (n+1)$ ***values), and then extracting the alignment (by reading the computed values). The space requirement is usually prohibitive. Hirschberg's algorithm reduces the space needs to roughly*** $2 \times m$***, but doubles the cost of computing and extracting the alignment. This paper introduces the FastLSA algorithm that is adaptive to the amount of space available. At one extreme, it uses linear space, while at the other it uses quadratic space. Based on the memory resources available, the algorithm saves the maximum amount of information to achieve the lowest extraction cost. The algorithm is shown to be analytically and experimentally superior to Hirschberg's algorithm.***

Keywords: sequence alignment, DNA, protein, Hirschberg's algorithm, dynamic programming, linear space.

1 Introduction

Sequence alignment is one of the fundamental operations performed in computational biology research. It is at the heart of the Human Genome Project, where sequences are compared to gather evidence for a common function or biological origin. The goal is to produce the best alignment for a pair of DNA or protein sequences (represented as strings of characters). A good alignment has zero or more gaps inserted into the sequences to maximize the number of positions in the aligned strings that match. For example, consider aligning the sequences "ATTGGC" and "AGGAC". By inserting gaps ("-") in the appropriate place, the number of positions where the two sequences agree can be maximized:

```
ATTGG-C
A--GGAC
```

Here, the aligned sequences match in four positions. Algorithms for efficiently solving this type of problem are well known and are based on dynamic programming. Aligning the sequences "ATTGGC" and "AGGAC" reduces to finding the maximum cost path through an array of size $m + 1$ and $n + 1$ ($m = 5$, $n = 6$, adding an extra row and column to include the gap). Given an array of size $O(m \times n)$, it takes $O(m \times n)$ time to compute the array cost entries, and then $O(m + n)$ time to identify the maximum-cost path in the matrix. In this paper, algorithms that are based on storing the complete matrix are called full matrix algorithms.

Unfortunately, $O(m \times n)$ space can be prohibitive, especially given that a DNA sequence can be millions of characters long. Even aligning two (small) sequences of 10,000 requires a prohibitive amount of storage (100,000,000 entries). Hirschberg was first to report a way of doing the computation using linear space [1]. Less storage means that some values must be recomputed. It is a space-time tradeoff: the computation costs double but the space overhead drops to being linear in the length of the strings.

In summary, there are two extremes for sequence alignment: full matrix, which minimizes the computational complexity, and linear space, which minimizes the storage requirements. Neither of these algorithms accommodates the real-world situation where you have more memory than needed for a linear space algorithm, but not enough to do a full matrix computation.

This paper introduces the FastLSA (Fast Linear-Space Alignment) algorithm. Linear-space alignment algorithms (such as Hirschberg's) do not take advantage of additional memory that might be available. By recursively subdividing the problem into $k \geq 2$ pieces using storage that is bounded by $2 \times k \times n$, FastLSA uses the the additional storage to reduce the execution time. In the limit, where $k = m$, the algorithm becomes equivalent to a full matrix algorithm. FastLSA can be viewed as being two generalizations of Hirschberg's algorithm:

1. Recognizing that the best sequence alignment is likely a diagonal path through the matrix. Thus, subdividing the problem vertically and horizontally makes sense for this class of applications.

2. Generalizing the recursion. Since all matrix entries must be visited at least once, recursively partitioning the data into $k \geq 2$ pieces allows additional storage to be used to reduce repeated computations.

This paper briefly describes the FastLSA algorithm and assesses its execution performance. Hirschberg's algorithm ends up recomputing roughly $(m+1) \times (n+1)$ values, while a full matrix algorithm has no recomputations. FastLSA with $k = 2$ recomputes only half as many values as Hirschberg's algorithm. Higher values of k give additional savings. In the limit ($k = m$), FastLSA does no recomputations. The experimental results closely mirror the analytical results, showing that FastLSA out-performs Hirschberg's algorithm on DNA sequence alignment.

2 Sequence Alignment

A simple model of sequence alignment illustrates the basic idea. The following scoring metric is simplistic; more complex scoring functions are used in practice. If two aligned sequences have identical values in the same column, then this will have a score of +2. If the values in a column differ then the score is -1. If a column contains a gap, then a penalty of -2 is imposed. The alignments are built assuming that both sequences do not have a gap in the same column. The best alignment is the one that has the maximal score. The alignment path is usually not unique. Our example alignment,

```
ATTGG-C
A--GGAC
```

has a score of $(+2 \times 4) + (-1 \times 0) + (-2 \times 3) = +2$.

The basic sequence alignment algorithm assumes that the entire dynamic programming matrix is in memory. It consists of two parts: FindScore to build the matrix of values and FindPath to traverse the matrix and identify the path(s) that lead to the maximal score. Since $O(m \times n)$ storage is used, this is a full matrix algorithm. The storage makes it possible to produce the alignment path (FindPath) by a linear traversal across the matrix.

Figure 1 shows the dynamic programming matrix built for the alignment of "ATTGC" and "AGGAC". In the FindScore phase, the table values are filled in from right-to-left and bottom-to-top. Between the table entries, the values that get propagated horizontally to the left, vertically up, and diagonally up-left are shown. Going left corresponds to inserting a gap in the left-hand-side sequence. Going up inserts a gap in the sequence at the top. Going diagonally up-left corresponds to "matching" the corresponding letters (no gap is inserted). Each table entry is the maximum of

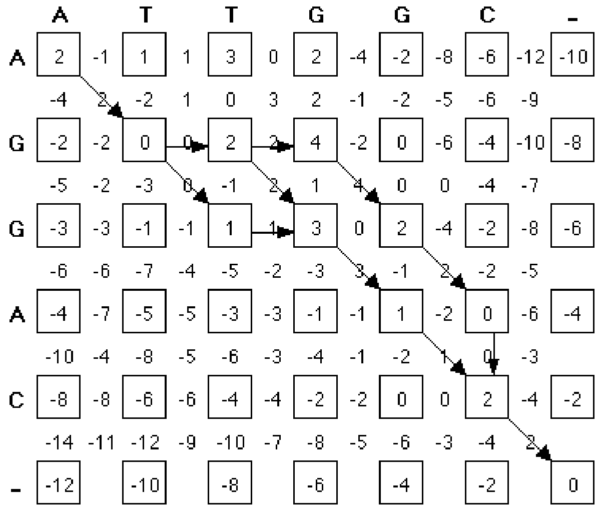


Figure 1: Alignment Matrix for "ATTGGC" and "AGGAC".

the three incoming values. The answer in the top left-hand corner (+2) confirms that the alignment given previously leads to the maximum score. The arrows show the optimal path back through the matrix. Note that there are three optimal alignments according to our metric, each giving the score of +2. In addition to the alignment given above, the following are also optimal:

```
ATTGGC    ATTGGC
A-GGAC    AG-GAC
```

There are a variety of sequence alignment algorithms based on Figure 1, of which Needleman-Wunsch [2] and Smith-Waterman [3] are two of the most important. The time and space concerns led to the invention of faster algorithms that were not as thorough in their alignment scoring. These algorithms, such as the well known Basic Local Alignment Search Tool (BLAST) [4], attempt to find high-scoring substring matches. The fast algorithms are often used first to see if there is an interesting relationship between the sequences. If a match is found then a more computationally expensive algorithm (such as Needleman-Wunsch or Smith-Waterman) is used to get a better quality answer.

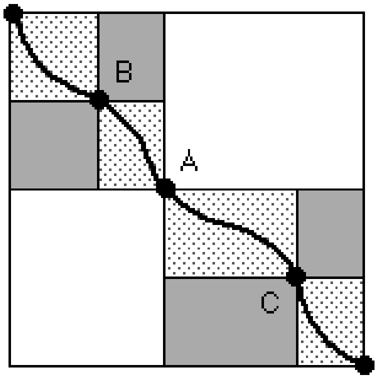
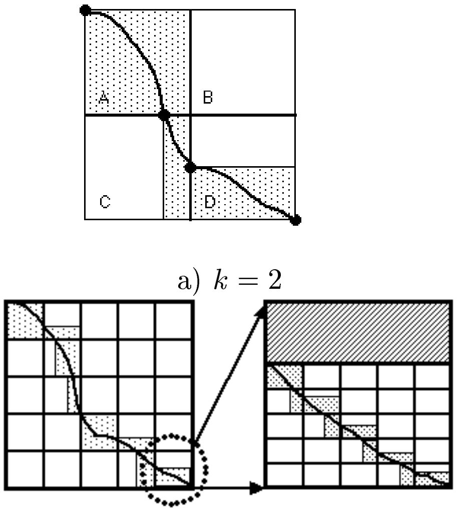


Figure 2: Hirschberg's Algorithm.

Hirschberg was the first to observe that the same computation can be done using linear space [1]. The FindScore component is easily modified to use linear space since it uses only the current and previous row at any time. Actually, it has been shown that only the last row is needed [5]. The FindPath component is harder, since it uses the results in the reverse order to which they were computed. Hirschberg's algorithm reduces the space requirements by using a recursive divide-and-conquer procedure, as shown in Figure 2. Consider string q of length m and string s of length n. The algorithm divides q in half: $q1$ $[1..m/2]$ and $q2$ $[m/2..m]$. $q1$ is aligned with s using a linear-space FindScore algorithm, saving only the last row of the computation. This alignment is the upper-left box in Figure 2. The alignment has proceeded from the upper-left corner to the middle dividing line. It then aligns the reverse of $q2$ with the reverse of s using a linear-space FindScore algorithm, saving only the last row of the computation. This alignment is from the bottom right-hand corner to the middle dividing line. The first alignment computes the scores from the start of the sequence to the midpoint; the second does it from the end of the sequence to the midpoint. From the two saved rows, the algorithm can determine where the alignment path crosses q's midpoint (c). This is point A in Figure 2. The problem now is reduced to solving two simpler problems: align $q1$ with $s[1..c]$ and $q2$ with

$s[c+1..n]$. The recursion ends when the length of a sequence to be aligned is one.

The time complexity for Hirschberg's algorithm is $O(n^2)$. The space complexity is $2 \times n = O(n)$. Chao, Hardison and Miller provide a nice overview of linear-space sequence alignment algorithms [6]. A new reduced-spaced sequence alignment algorithm uses bidirectional search [7]. This algorithm has the nice property that it does not need to consider the entire $m \times n$ matrix, eliminating portions of it that probably cannot be part of the best alignment. However the program runs two to three times slower than Hirschberg's algorithm [8].



a) $k = 2$

b) $k = 5$

Figure 3: Subdividing the Sequence Alignment Matrix.

3 FastLSA Algorithm

Given two strings of length m and n ($m \leq n$), and R units of memory, what is the most cost-effective way to do a sequence alignment? If $R \geq m \times n$, then the full matrix algorithm can be used since everything will be resident in memory (note that you may not want to do this anyway because cache effects may result in poor performance). If this is not the case, then a reduced memory variant must be used. Hirschberg's algorithm works and only uses $2 \times m$ memory. It does not address the issue of what to do if you have more memory available.

Hirschberg's algorithm recursively divides the problem in half, by saving row information. However, if the problem is bisected both row-wise *and* column-wise, and both the row and column boundaries are saved, then the alignment can be computed while recalculating fewer values.

Consider the column and row bisection in Figure 3a. Computing 3/4 of the matrix (quadrants B, C, and D) allows the bisection boundary information (the thick horizontal and vertical lines in the matrix) to be saved. Now the last quarter of the problem (quadrant A) can be recursively solved. When it is completed, the optimal path will either go from quadrant A to quadrant B, C or D (C in the figure). However, the saved column information allows the algorithm to quickly compute the information needed to extend the path to quadrant D. To find the path through quadrant C requires consideration of only the dotted portion of that region. Hirschberg's algorithm, in contrast, would require the entire row from the right-hand side of the matrix to the path to be recomputed. Here, a little information can save a lot of recalculation. Hirschberg's algorithm requires calculating, on average, each value in the matrix twice. Figure 3a illustrates that FastLSA offers significant savings; only the dotted regions are recomputed. Each of these regions can, in turn, be solved recursively.

The algorithm is not restricted to bisection. The FastLSA algorithm recursively divides each dimension of the matrix into $k \geq 2$ subproblems. Figure 3b illustrates this idea for $k = 5$. Initially the entire matrix is computed, saving 4 rows and 4 columns during the computation (as seen on the left-hand side of Figure 3b). Once the top-left region is (recursively) computed, the optimal path will extend into one of three $(m/k) \times (n/k)$ boxes (to the immediate right, immediately below, or diagonally down). Each box has the necessary row and column information to localize the amount of recalculation that needs to be done. The dotted regions are the recursive subproblems where recomputations need to be performed. Hence, in Figure 3b, the number of areas (subproblems) that need to be solved is k in the best case (the k boxes on the diagonal) and $2k-1$ in the worst case. In this example, 8 subproblems are needed. Each of these subproblems can, in turn, be solved recursively using the same procedure. For example, the right-hand side of Figure 3b shows an enlargement of the bottom-right corner of the matrix given on the left-hand side of the figure. The part of the region that is still relevant to the solution is divided into 5, both horizontally and vertically. The top shaded portion of the region is ignored since it has been proven that the optimal path being followed cannot go through it.

An analysis of FastLSA's performance gives the following results for aligning two sequences of length n:

- Space: $S(n, n) \approx 2kn$ (assuming $n >> k$).
- Expected number of values recomputed: $E(m, n) \approx m \times n/k$.

For $k = 2$, FastLSA can be expected to recompute roughly $E(m, n) = 0.5 \times m \times n$ values, about half of the recomputation cost of Hirschberg's algorithm. This comes at the cost of using double the storage of Hirschberg's algorithm. Depending on the length of the sequences being aligned, this extra storage is usually not an issue. Increasing k reduces the time complexity by increasing the space usage. In the limit, $k = m$, FastLSA becomes equivalent to the full matrix algorithm with no recomputations.

4 Experimental Results

FastLSA has been integrated into the commercial sequence alignment code for the BioTools product GeneTool (www.biotools.com). Experiments were conducted using a database of 3,171 sequences (a publicly available subset of GenBank sequences). For each value of $k = 2, 3, ...11$, five runs were performed aligning a sequence against each member of the database using different alignment scoring parameters. The average execution times and number of recomputed values over these runs are reported. Timings were done on a 200 MHz PC with 128 MB of RAM running Linux.

Figure 4 compares algorithms based on the number of values recomputed. Full matrix is at a constant of 0 recomputations, and Hirschberg's algorithm required $0.93 \times m \times n$ recomputations in our experiments (better than the theoretical value of 1.0). FastLSA is illustrated for $k = 2$ through 11. Both the theoretical (expected case $E(m, n)$) and experimental results are shown. The analytical result is a good predictor of performance, even though it is an upper bound based on unrealistic assumptions (that both matrices are the same size). The standard deviation is small, ranging from 0.041 for $k = 2$ to 0.015 for $k = 11$.

Figure 5 compares the recomputation execution time for FastLSA and Hirschberg. The FastLSA(1) time is an extrapolation of FastLSA(2)'s time. FastLSA's actual execution times reduce by a faster rate than predicted by $E(m, n)$. The execution overhead of FastLSA (saving values; recursing; deciding when recomputations are needed) appears to be more than offset by positive cache effects due to using less memory. Performance levels off at $k = 11$, where the overhead of using linear storage is within a few percent of full matrix performance.

Full matrix results are not shown in Figure 5, since the excessive space requirements degrade performance due to swapping and cache effects (alignments of two sequences with sizes greater than 5,000 do not fit comfortably in RAM). Even small alignments, such as sequences of length 500, have poor cache performance. Except for small sequences, full matrix algorithms run slower than linear space algorithms, even though they compute fewer values!

The anomaly in Figure 5 is the poor performance of Hirschberg's algorithm. Our implementation produces the correct results, but the

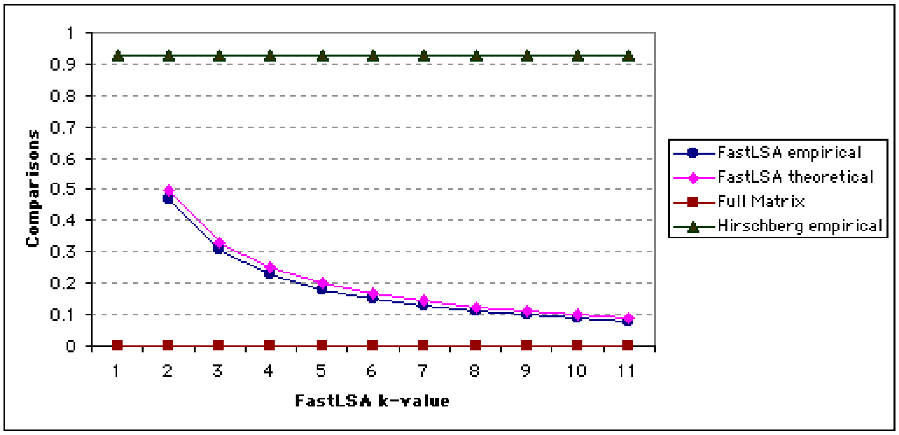


Figure 4: Number of Recomputations.

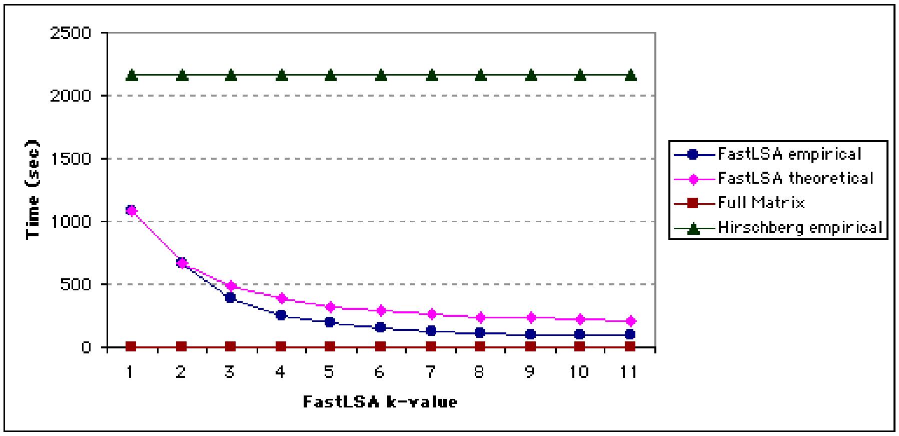


Figure 5: Execution Time.

times are surprisingly bad. A major reason for this is the cache. FastLSA uses memory like a stack, continually pushing, popping, and reading from the top of the memory area allocated to the algorithm. This gives very good cache locality. It also has a smaller cache working set then does Hirschberg's algorithm because the intervals it works with are at least a factor of k smaller. In contrast, Hirschberg's algorithm continually overwrites its entire allocated memory. When it solves a subproblem, the algorithm continually overwrites a single row in memory. Each overwrite goes sequentially from the first to the last element in the row. If the space requirements for the row are too big to fit into the cache, then each row access is likely no longer in the cache.

5 Acknowledgments

We would like to thank BioTools Inc. for kindly making their source code available to us. Ian Parsons integrated FastLSA into the BioTools products.

Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] D. Hirschberg. A linear space algorithm for computing maximal common subexpressions. *Communications of the ACM*, 18(6):341–343, 1975.

[2] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[3] T. Smith and M. Waterman. Identification of common molecular sequences. *Journal of Molecular Biology*, 197:723–728, 1981.

[4] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[5] E. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[6] K. Chao, R. Hardison, and W. Miller. Recent developments in linear-space alignment methods: A survey. *Journal of Computational Biology*, 1(4):271–291, 1994.

[7] R. Korf. Divide-and-conquer bidirectional search. In *International Joint Conference on Artificial Intelligence*, pages 1184–1189, 1999.

[8] R. Korf, 1999. Private communication, September.