Determining DNA Sequence Similarity Using Longest Path Algorithms for Rectangle Graphs

JOÃO MEIDÂNIS,[1] DEBORAH JOSEPH,[2] and PRASOON TIWARI[3]

Computer Sciences Department, Univ. of Wisconsin-Madison, Madison WI 53706, USA.

Abstract. Motivated by the problem of predicting functionality by finding similarities in DNA and amino acid sequences, we study a particular class of two dimensional graphs and present an algorithm that finds longest paths for these graphs. The algorithm runs in $O(n \log n)$ time, where n is the number of nodes, and its implementation provides a *practical* solution to a common problem in genetic sequence comparison.

April 7, 1992, revised May 28, 1992, August 14 1992, and October 11, 1993.

---

[1]Permanent Affiliation: Computer Science Dept., State University of Campinas, Cx. Postal 6065, 13081 – Campinas – SP, Brazil. Partially supported by FAPESP, Brazil, under grant 87/0197-2.

[2]Supported by NSF PYI grant DCR-8451387.

[3]Supported by Wisconsin Alumini Research Foundation and by National Science Foundation under grant CCR-9024516

1 Introduction

Key to the success of large sequencing efforts, such as the Human Genome Program, are the mathematical and computational tools for organizing and analyzing large quantities of genetic sequence data. In this paper we focus on the problem of predicting the functionality of a DNA sequence by comparing it to a database of sequences whose functionality is known.

The algorithms that we discuss extend methods of local alignment to find consistent "families" of similarities between sequences. This leads us to study longest-path algorithms for a particular class of graphs. For this class we present an algorithm that finds a longest path in $O(n \log n)$ time, where n is the number of nodes, and its implementation provides a practical solution to a common problem in genetic sequence comparison.

The work presented in this paper is motivated by a common problem in genetic sequence comparison. The problem is to predict the functionality of a DNA sequence by comparing the sequence to a database of sequences with known functionality. This problem arises in large scale genome sequencing projects.

To understand how these database comparisons are made, we need to know a few facts about the biological mechanisms involved in protein synthesis. In a living cell, a DNA molecule is used as a template to manufacture proteins. A protein molecule can be viewed as a sequence of units called *amino acids*. There are 20 different amino acids which naturally occur in proteins across all life forms.

The translation of a DNA sequence into an amino acid sequence is done using the *genetic code*, which converts base triplets into amino acids. Since there are 4 possible bases, the number of possible triplets (also called *codons*) is 64, which shows that the genetic code is *degenerate*, i.e., distinct codons can specify the same amino acid. Also, three of the possible codons do not correspond to any amino acid and are used to stop translation.

Not all parts of a DNA molecule are used in the cell to produce proteins. The parts that do code for proteins are called *genes*. Some genes code for other products, e.g., RNA.

When the nucleotide make up of a DNA sequence is known, analysis of its biological function can take place. Questions such as "Does it contain a gene?" and "Does that gene code for a product similar to a known one?" immediately arise. Today, with the advent of the large, centralized genetic sequence databases [4, 2, 11, 17], a standard approach is to compare it to the entries in these databases, for which information about functionality is at least partially known.

Some databases contain DNA sequences only, while others contain proteins sequences only. While it might seem more natural to query DNA databases with our DNA sequence, protein databases often give better results. Reasons for this include the degeneracy of the genetic code, potential presence of introns (see below), and the higher scoring precision possible in the larger amino acid alphabet [6]. On the other hand, DNA databases are more complete and up-to-date, because DNA sequencing is so much easier than protein sequencing. A reasonable solution is to search using translated DNA databases, which combines the advantages of these databases with the search at the amino-acid level [9].

Of course, the DNA query sequence has to be translated into an amino acid sequence using the genetic code before it can be compared. This process also cuts the original sequence into *open reading frames* (ORFs), characterized by the absence of termination codons.

The next question is how to compare a translated ORF to a database entry. In recent years, many sequence comparison algorithms have been developed and implemented [1, 7, 10, 15, 16, 18, 19, 21]. These algorithms can have either a *global* or a *local* character. Global methods seek an alignment of two given sequences that maximizes a certain similarity measure. Thus, the entire length of both sequences is involved in the result, and the maximum score obtained is in some sense an average between good and poor matching regions.

In contrast, local comparison methods find similar regions with very high scores within the two sequences. In these methods, inserting gaps in the regions is either not allowed, or heavily penalized. As a result, these methods may yield a large, perhaps overlapping, collection of pairs of matching regions. Most existing implementations can be configured to produce all the matches with scores above a given threshold.

For several biological reasons, local comparison methods are more appropriate for determining sequence similarity when the goal is to predict functionality. First, similar functionality between genetic sequences can often be accurately predicted by observing that relatively short subregions (motifs) are conserved [9]. Second, in most higher organisms (eukaryotes), genes are composed of regions that are expressed, that is, translated into protein sequences (these regions are called exons), and also regions that are removed from the gene before it is translated (these regions are called introns). When comparing DNA sequences, with the goal of predicting functionality, we are interested in the similarity between exon regions, not intron regions. However, if we simply have a DNA sequence obtained from a large sequencing project, it is difficult, if not impossible, to tell which regions represent introns, and which are exons. Many methods have been proposed to aid in distinguishing between introns and exons, but currently these methods do not completely solve the problem.

Finally, local similarity methods are particularly useful for dealing with *frameshift errors*. DNA sequencing is an imperfect process and even with repeated sequencing there will very often be errors present in long query sequences. When trying to determine functionality, the type of error that causes the most problem is the erroneous insertion or deletion of a nucleotide base. This is because of the biological mechanisms in which information in a codon (three adjacent nucleotide bases in a DNA sequence) is translated to form a single amino acid in a protein sequences.

To overcome the problems caused by inserted or deleted nucleotide bases, systems such as FIND-IT and NUTSS [20, 9] begin by translating the given DNA sequence into three possible protein sequences, one for each of the possible ways of grouping bases into triplets. Each piece is then compared to all database sequences, and matches of any of the pieces with the same database entry are collected to be analyzed together. Note that these matches may come from one or more of the three possible translations (reading frames). Long local matches broken up by missing or inserted bases can be recovered by juxtaposition of matches from different reading frames.

One last problem still remains in this approach. Experience using the FIND-IT system [20] to analyze a two-minute region (a DNA sequence of more than 91,000 nucleotide bases) of the *E. coli* genome sequenced in F. Blattner's laboratory at the University of Wisconsin reveals the following pattern. Most database entries yield no matched subregions, indicating no detectable similarity to the query. Some entries provide a few matched regions with relatively high score; these can be inspected manually by experts for significant similarities. However, some entries generate a large number of overlapping matched regions. *These large collections of matched subregions are very hard to further analyze manually.* One method for circumventing this problem is to group matches with distinct, nonoverlapping regions into a small number of "families," and analyze these families as a whole. This is the approach taken by systems such as FIND-IT [20], and the setting in which our algorithm will be presented.

In this paper, we propose a natural way of constructing families by grouping individual matches together. Under this formulation, matches become rectangles in the plane and a family of matches becomes a path in a certain weighted directed acyclic graph build from these rectangles. We present an algorithm to find a path with maximum total weight using a sweep technique very common in interval graph algorithms [8, 3]. The algorithm runs in $O(n \log n)$ time and uses $O(n)$ space on graphs with n nodes.

Sobel and Martinez study a similar problem and employ a standard longest-path algorithm to find the best family [22]. Thus, their solution runs in $O(n + m)$ time (m being the number of edges), which is asymptotically slower for dense graphs.

It is interesting to note that a particular case of our problem is to find the longest increasing subsequence of a given sequence of real numbers, a problem that has been considered several times in the literature, and for which the best known algorithm runs in time $O(n \log n)$ [14].

The rest of this paper is organized as follows. In Section 2 we use the "graphic matrix" plot introduced by Maizel and Lenk [13] to rephrase the problem of combining segment pairs in terms of rectangle graphs. The algorithm itself is presented in Section 3, together with a proof of correctness and analysis of its computational resource requirements. Finally, in Section 4 we discuss some directions for future research.

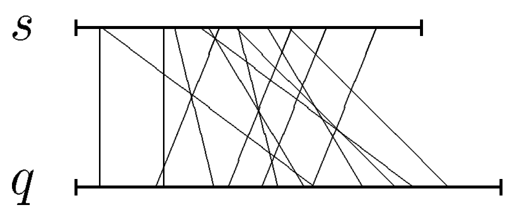

Figure 1:

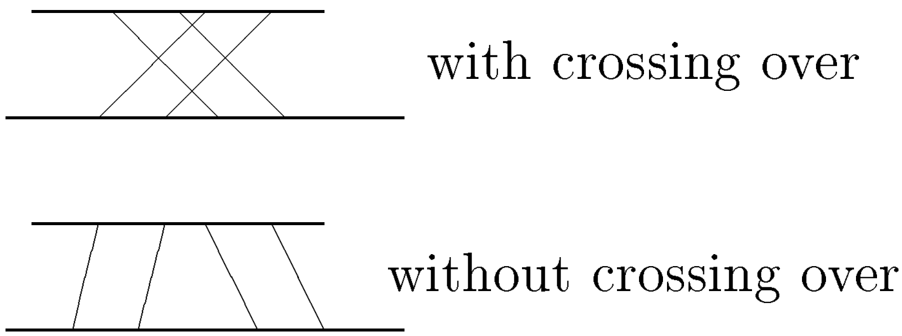

Figure 2:

2 Modeling the Problem

Suppose we performed a database search using a query sequence q and obtained, among other results, a collection of segment pairs specifying matches between q and one of the sequences s in the database (see Figure 1). Each match is accompanied by a non negative *score*. Our purpose is to form families of segment pairs and select the family with maximum total score.

Families are not just arbitrary sets of segment pairs. Two segment pairs can only be in the same family if they *do not intersect* in either q or s. In addition, we require that segment pairs *do not cross over*, in the sense that the relative order in which they appear in each sequence should be the same. Examples of segment pairs with and without cross over are illustrated in Figure 2.

To solve our problem, the first step is to represent all the segment pairs in a cartesian plane where the sequence q lays on the X-axis and the sequence s lays in the Y-axis (see

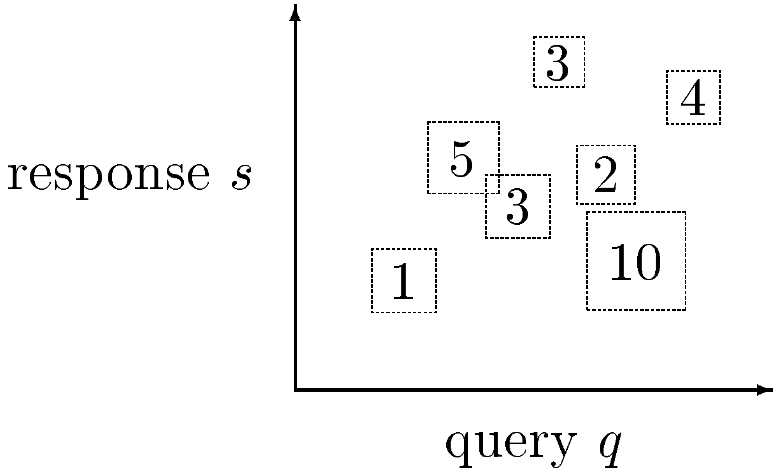

Figure 3:

Figure 3). We identify character i of sequence q with the point $(i, 0)$ and character j of sequence s with the point $(0, j)$. Each segment pair corresponds to a rectangle in the first quadrant whose projections in the axes are the portions matched in each sequence.

This construction is completely analogous to the "graphic matrix" plot considered in the literature [12, 13, 19]. The only differences are that the rectangles are actually squares in the cited references, and only their main diagonals were used to represent them.

We can now translate our conditions for grouping segment pairs into the rectangle context. It is easy to see that an admissible family of rectangles can be ordered so that each rectangle lies strictly above and to the right of the previous one.

Formally, we have the following definitions. A two dimensional interval or *rectangle* is a subset of the plane that can be written as a product of two finite closed intervals, one from each coordinate axis. Given a rectangle r, we define

$$\begin{aligned} Xmin(r) &= \text{minimum } X\text{-coordinate of any point in } r \\ Xmax(r) &= \text{maximum } X\text{-coordinate of any point in } r \\ Ymin(r) &= \text{minimum } Y\text{-coordinate of any point in } r \\ Ymax(r) &= \text{maximum } Y\text{-coordinate of any point in } r \end{aligned}$$

so that for any rectangle r we have

$$r = [Xmin(r), Xmax(r)] \times [Ymin(r), Ymax(r)].$$

In what follows, we will also assume that we are given a collection of rectangles and each rectangle r has a non negative *weight* associated with it, denoted by $Weight(r)$.

For two rectangles r and s, we say that r *precedes* s (notation: $r \to s$) when $Xmax(r) < Xmin(s)$ and $Ymax(r) < Ymin(s)$. In other words, r is located to the "southwest" of s in the coordinate plane. We may now construct a directed acyclic graph (DAG) with the rectangles as nodes and a directed edge from r to s if and only if r precedes s. We define a *rectangle graph* as being a directed acyclic graph that can be obtained from a collection of rectangles in this fashion.

A *path* in a rectangle graph is an ordered set $p = (r_1, r_2, \ldots, r_m)$ of rectangles such that $r_j \to r_{j+1}$ for $1 \leq j \leq m-1$. The rectangle r_1 is called the *head* of p and denoted by $Head(p)$ if p is not empty. The empty path has no head. For a path p we denote by $Weight(p)$ the sum of the weights of all rectangles in p.

We define an order relation among the non empty paths as follows. If p, q are non empty paths we write $p \preceq q$ to indicate the fact that $Weight(p) \leq Weight(q)$ and either $Head(p) = Head(q)$ or $Head(p) \to Head(q)$. In this case we say that q *dominates* p. The idea behind this definition is that, for any path z with suffix p we can substitute q for p in z maintaining the condition of increasing independence without decreasing its total weight. If p and q are not comparable with respect to this relation, that is, if neither $p \preceq q$ nor $q \preceq p$, we say that they are *compatible*. Thus, given two paths, they are either compatible or one of them dominates the other.

3 The Algorithm

This section contains the description of an algorithm to find the maximum weight path for a weighted rectangle graph. Before getting into the technical details, however, we will give an intuitive idea of the method used.

We use a dynamic programming approach. Instead of computing the total weights of all possible paths, we use the geometric properties of the problem to cut down the number of set that need to be considered.

As we mentioned before, we will employ a sweep technique. A choice has to be made for one of the coordinate axes to be the sweep axis. The algorithm will work with either choice. We chose the X-axis here. Our goal is to compute, for each rectangle i, the maximum weight of a path starting at i. In addition, we also record the *next* rectangle in such a maximum weight path.

After sorting all the end points of rectangles projected onto the X-axis, we process them from right to left. At each moment, when scanning coordinate x, we have a collection D of paths that represent the best possible choices for the interval $[x, +\infty]$. When we first reach a new rectangle i, we look for the best next rectangle that can be attached to i, record it as the next rectangle in a maximum weight path starting at i, and compute this maximum weight. This way we basically have to compute *one* total weight per rectangle.

Later, when we move to the left and reach the leftmost coordinate of rectangle i, we update the collection D with the new information provided by i. The key idea here is to keep in D only compatible paths, as defined in Section 2. Indeed, if $p \preceq q$ there is no point in keeping p around, since q will always be a replacement at least as good.

After all rectangles have been processed, we simply select the element in D with largest total weight.

We will assume that the input consists of n rectangles, which we identify with the integers from 1 to n, and five vectors

$$Xmin[i], Xmax[i], Ymin[i], Ymax[i], Weight[i],$$

$1 \leq i \leq n$, containing the coordinates and the weights of the rectangles.

The algorithm uses an integer array $Next[1..n]$, initially undefined, to record the best choice to the right, for each rectangle. A vector $TotalWeight[0..n]$, initially containing zeros in each entry, is used to compute total weights. Notice that $TotalWeight$ starts at index 0, and we set $TotalWeight[0] = 0$ to handle the base case nicely. Finally, we will need the set D, which starts out empty. The algorithm is presented in Figure 4.

The operations needed for the set D are performed by the routines $Best$ and $Update$. At any moment, the set D contains a collection of non empty pairwise compatible paths, represented by their heads. The operation $Best(D, y)$ returns the path head i in D which has the minimum $Ymin[i]$ greater than y, or zero if no path satisfies this condition.

Notice that for each rectangle i, a value is assigned to $Next[i]$ exactly once, and either $Next[i] = 0$ or else $i \rightarrow Next[i]$. This means that, after the execution of the algorithm is completed, the vector $Next$ defines an acyclic chain structure on the nodes, with each chain ultimately leading to zero. We introduce the notation $path(i)$ to indicate the chain of $Next$ links starting at i, that is,

$$path(i) = \begin{cases} \emptyset & \text{if} \quad i = 0 \\ (i, path(Next[i])) & \text{otherwise} \end{cases}$$

We will also sometimes write $i \in D$ meaning $path(i) \in D$.

The operation $Update(D, i)$ adds a new path, namely, $path(i)$, to D; however, we want to preserve the compatibility among members of D. If $p \preceq q$, we want to keep only q. So, $Update$ also removes from $D' = D \cup path(i)$ all elements p such that there is another $q \in D'$ with $p \preceq q$.

The rationale behind these abstract requirements is as follows. In the algorithm, we scan the X-axis from right to left processing all endpoints. When we find $Xmax[i]$ for some rectangle i we want to link it to the best path in the region $[Xmax[i], +\infty] \times [Ymax[i], +\infty]$. We keep all the best paths for regions of this form in D. Each element j in D means, "there is a path of weight $TotalWeight[j]$ extending from $Ymin[j]$ up." To find the best one for linking i to, we look for the maximum $TotalWeight$ among all elements sort all $Xmin$ and $Xmax$ together into array $Sorted[1..2n]$

$TotalWeight[0] = 0$

$D = \emptyset$

for $j \leftarrow 2n$ **downto** 1 **do**

    if $Sorted[j] = Xmax[i]$ for some i **then**

        $Next[i] \leftarrow Best(D, Ymax[i])$

        $TotalWeight[i] \leftarrow TotalWeight[Next[i]] + Weight[i]$

    else /* $Sorted[j] = Xmin[i]$ for some i */

        $Update(D, i)$

    endif

endfor

output list starting at $Best(D, -\infty)$ and following the $Next$ links

Figure 4:

$j \in D$ that have $Ymin[j] > Ymax[i]$. However, because of the compatibility condition, and the fact that we are scanning right to left in the X-axis, "maximum *TotalWeight*" is equivalent to "minimum *Ymin*," so what we need is exactly what $Best(D, Ymax[i])$ returns.

To prove the correctness of the algorithm, we will assume that the routines *Best* and *Update* behave as described above. We will see later how to implement them efficiently.

Our proof relies on the following fact. Upon termination of the algorithm, for each rectangle i, $TotalWeight[i]$ is the maximum weight of any path starting from i. To see that, fix a maximum weight path starting at each rectangle, and reason by induction on the length of such a path.

The base case is when a given rectangle i has no successors. In this case, the call $Best(D, Ymax[i])$ will return 0 and this value will be stored as $Next[i]$, causing the total weight for i to be equal to $Weight[i]$, which is correct.

For the remaining cases, let i be any rectangle with more than one rectangle in its fixed maximum path and assume that the claim is true for all rectangles with shorter maximum paths. When $Best(D, Ymin[i])$ is executed, the set D had been updated with all successors of i. Among these is the rectangle j that follows i in the fixed maximum weight path we chose in the beginning. We claim that a rectangle with at least the same total weight as j will be returned by *Best*. Indeed, if r is the returned rectangle, it has maximum total weight among all processed rectangles above i, and this includes j. On the other hand, j is part of a maximum path, so in fact $TotalWeight[j] = TotalWeight[r]$ and $TotalWeight[i]$ will be computed correctly.

Having established this claim about the total weight, it is not difficult now to verify that $path(i)$ is a maximum weight path starting at i, for each rectangle i.

3.1 Efficient Implementation of the Set D

The set D can be efficiently implemented as a *balanced search tree* [23]. This structure permits the execution of several operations in $O(\log n)$ time, where n is the number of items in the structure. For our purposes, we only need operations for search, insertion, joining, and splitting, all of which can be implemented in $O(\log n)$. If we do not insist in $O(\log n)$ time per *Update* operation, we only need search, insertion, and deletion (see Section 3.2). Furthermore, the amount of storage needed never exceeds $O(n)$.

At any given moment, D will contain a set of non-empty, mutually compatible paths. Each path is represented in D by its head, which is a rectangle i. The rest of the path can be obtained by following the *Next* links. In other words, i represents $path(i)$ as defined previously. Thus, D can be implemented as a set of rectangle numbers.

There are two keys associated to each rectangle in D: its *TotalWeight* and its *Ymin* coordinate. In a balanced search tree, elements are stored in increasing (or decreasing) order of keys. Since we have two keys, this is apparently a problem. Fortunately,

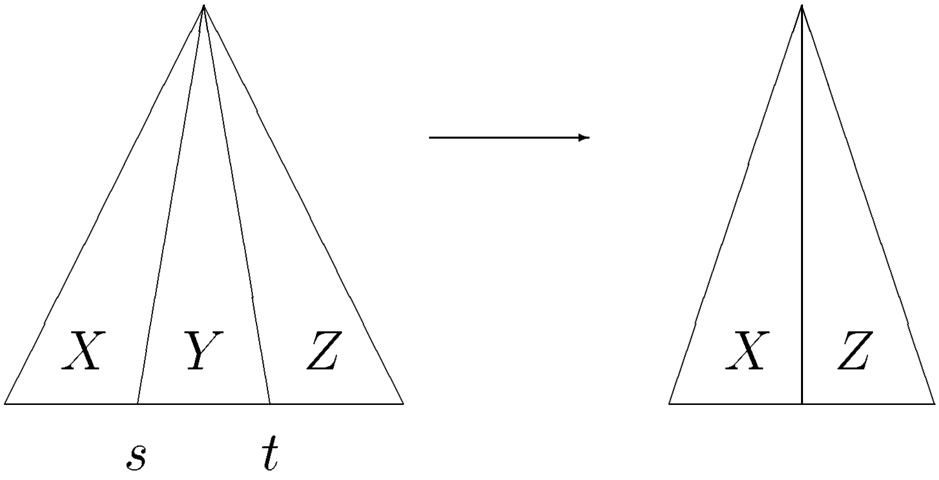

Figure 5:

because all elements in D are mutually compatible, and because we scan the ordered X-coordinates right to left, the order of ascending *TotalWeight* is the same as the order of descending *Ymin*. Hence, if we keep them sorted based on one of the keys they will automatically be sorted with respect to the other key as well.

Let's now implement our operations *Best* and *Update*. The *Best* operation is essentially a search, and can be implemented in $O(\log n)$ time [23].

In order to implement $Update(D, i)$, we need to include $path(i)$ in D and then update the augmented set D'. One of the following three cases will happen.

1. There is $p \in D$ with $path(i) \preceq p$. In this case, D is not modified at all and i is just discarded.

2. No element $p \in D$ dominates $path(i)$, but $path(i)$ does not dominate any $p \in D$ either. In this case, $path(i)$ is compatible with all the elements in D, so all we need is to insert i in the structure, according to any of the keys.

3. No $p \in D$ dominates $path(i)$, but $path(i)$ dominates some elements in D. In this case, these elements have to be removed from D and $path(i)$ inserted in their place. The key observation here is that these elements are necessarily contiguous in D. Indeed, it is easy to see that if $p, q \preceq path(i)$ and z is between p and q in D, then $z \preceq path(i)$. Hence, they can be removed in $O(\log n)$ by performing two split operations followed by a concatenate operation (see Figure 5.)

We can determine which of the three cases occurs by doing two search operations for i in D, one using the *TotalWeight* keys and the other with the *Ymin* keys. If, in any of the searches, we ever find an element in D that dominates $path(i)$ we know we are in Case (1). If, on the other hand, we ever find an element that is dominated by $path(i)$, we know we are in Case (3). Finally, if during the searches we fail to meet an element in D comparable to $path(i)$, we conclude that Case (2) occurs.

Since all these operations can be performed in $O(\log n)$ steps, the total time required by the sweep is $O(n \log n)$. As the sorting can also be performed in $O(n \log n)$ steps, the algorithm can be executed in $O(n \log n)$ steps.

3.2 Alternative Implementation of D

The implementation shown in Section 3.1 guaranteed that each call to *Best* or *Update* will require $O(\log n)$ time. If we do not insist that each call to *Update* takes $O(\log n)$, a simpler approach can be used, still maintaining an overall running time of $O(n \log n)$.

The difference lies in Case (3) of the previous section (Section 3.1). Here, instead of removing a range of consecutive elements all at once, we could just remove them one by one, taking $O(\log n)$ time per removal. With this approach, it is no longer clear that we can guarantee $O(\log n)$ time per *Update* operation. However, since each rectangle is removed from D at most once, the overall time bound for removals is still $O(n \log n)$.

4 Conclusions

We have taken a problem in computational genetics, found a formal model for it in graph theory, and presented an efficient solution to the problem using fairly simple algorithmic techniques. We believe that our work makes important contributions in two ways. First, we have not simplified the biological problem that needed to be solved – ours is an efficient solution for one of the *real* problems that systems such as FIND-IT face [20]. Second, although the interval graph model that we form for the biological problem is important for the algorithmic solution, most important for the efficient solution is the observation that the interval graph does not have to be explicitly represented using adjacency lists or matrices. Our implicit representation makes an $O(n \log n)$ algorithm possible.

Nevertheless, several interesting questions remain unanswered. The algorithm that we presented finds the family that represents the best match (with respect to the combined weight of individual matches) between the query sequence and the database sequence. In practice one would like to find all very good families. Many of the very good families may be overlapping. We would like an algorithm that efficiently finds and presents all very good families. Emphasis should be placed on developing a good method of presentation of the output.

Sobel and Martinez apply a technique due to Byers and Waterman [5] for the computation of all families with weight above a given threshold [22]. This procedure can be applied in connection with the algorithm presented here as well. However, this takes additional $O(n^2)$ time, and the authors do not address the issue of a good output presentation.

Two questions having to do with the requirements for combining segment pairs are also interesting.

The first question relates to the "below-and-to-the-left" condition. For biological reasons we required that either $r \to s$ or $s \to r$ for distinct rectangles r, s belonging to the same family. There are situations in which we can relax this condition. An appropriate model might allow a fixed number of rectangle pairs that violate the condition provided that the sequences forming the rectangles are nonoverlapping. What is the best algorithm for the maximum weight family under the new requirements?

A second question relates to our definition of adjacency in rectangle graphs. It can be argued that the requirement that projections do not intersect at all is too strong, making us miss very good families just because of a few bases meeting at the extremities. In this respect, a two dimensional version of the tolerance graphs seems to be the appropriate model. An efficient algorithm for the maximum weight increasing independent set for this class would be interesting. However, to be practical from the biological standpoint the measure of weight used for the independent set must be modified to reflect the overlap.

Acknowledgements.

This research was done in co-operation with the E. coli Genome Project, Genetics Department, University of Wisconsin – Madison. Part of the preparation of this report was done at the Aspen Center for Physics' Workshop on Recognizing Genes and Other Components of Genomic Structure, June 1992.

We also acknowledge support from FAPESP, Brazil (grant 87/0197-2), National Science Foundation (PYI grant DCR-8451387 and grant CCR-9024516), and Wisconsin Alumini Research Foundation.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. *J. of Molecular Biology*, 215:403–410, 1990.

[2] W. C. Barker, D. G. George, and L. T. Hunt. Protein sequence database. In Russell F. Doolittle, editor, *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, volume 183 of *Methods in Enzymology*, pages 31–49. Academic Press, New York, 1990.

[3] A. A. Bertossi and A. Gori. Total domination and irredundance in weighted interval graphs. *SIAM J. Discrete Math.*, 1(3):317–327, 1988.

[4] C. Burks et al. GenBank: Current status and future directions. In Russell F. Doolittle, editor, *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, volume 183 of *Methods in Enzymology*, pages 3–22. Academic Press, New York, 1990.

[5] T. H. Byers and M. S. Waterman. Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming. *Oper. Res.*, 32:1381–1384, 1984.

[6] M. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. In M. Dayhoff, editor, *Atlas of Protein Sequence and Structure*, volume 5, pages 345–352. National Biomedical Research Foundation, Silver Spring, MD, 1978. Supplement 3.

[7] O. Gotoh. Optimal sequence alignment allowing for long gaps. *Bull. Math. Biol.*, 52(3):359–373, 1990.

[8] U. I. Gupta, D. T. Lee, and J. Y.-T. Leung. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12:459–467, 1982.

[9] S. Henikoff, J. C. Wallace, and J. P. Brown. Finding protein similarities with nucleotide sequence databases. In Russell F. Doolittle, editor, *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, volume 183 of *Methods in Enzymology*, pages 111–132. Academic Press, New York, 1990.

[10] X. Huang, R. C. Hardison, and W. Miller. A space-efficient algorithm for local similarities. *Computer Applications in the Biosciences (CABIOS)*, 6(4):373–381, 1990.

[11] P. Kahn and G. Cameron. EMBL data library. In Russell F. Doolittle, editor, *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, volume 183 of *Methods in Enzymology*, pages 23–31. Academic Press, New York, 1990.

[12] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity search. *Science*, 227:1435–1441, 1985.

[13] J. Maizel and R. Lenk. Enhanced graphic matrix analysis of nucleic acid and protein sequences. *Proc. Nat. Acad. Sci. U. S. A.*, 78:7665–7669, 1981.

[14] U. Manber. *Introduction to Algorithms.* Addison-Wesley, Reading, 1989.

[15] H. M. Martinez. An efficient method for finding repeats in molecular sequences. *Nucleic Acids Research*, 11(13):4629–4634, 1983.

[16] W. Miller and E. W. Myers. Sequence comparison with concave weighting functions. *Bull. Math. Biol.*, 50(2):97–120, 1988.

[17] S. Miyazawa. DNA data bank of Japan: Present status and future plans. In George I. Bell and Thomas M. Marr, editors, *Computers and DNA*, pages 47–61. Addison-Wesley, Reading, 1990.

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

[19] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Nat. Acad. Sci. U. S. A.*, 85:2444–2448, 1988.

[20] J. Shavlik. Finding genes by case-based reasoning in the presence of noisy case boundaries. In *Proceedings of the DARPA Cased-Based Reasoning Workshop*, pages 327–338, Washington, DC, 1991.

[21] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. of Molecular Biology*, 147:195–197, 1981.

[22] E. Sobel and H. M. Martinez. A multiple sequence alignment program. *Nucleic Acids Research*, 14(1):363–374, 1986.

[23] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1983.