Algorithms and Complexities of a few Sequence Comparison Problems

Piotr Berman*
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
Email: berman@cse.psu.edu

Bhaskar DasGupta†
Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607
Email: dasgupta@cs.uic.edu

August 29, 2005

1 Introduction

The modern era of molecular biology began with the discovery of the double helical structure of DNA. Today, sequencing nucleic acids, the determination of genetic information at the most fundamental level, is a major tool of biological research [30]. This revolution in biology has created a huge amount of data at great speed by directly reading DNA sequences. The growth rate of data volume is exponential. For instance, the volume of DNA and protein sequence data is currently doubling every 22 months [24]. One important reason for this exceptional growth rate of biological data is the medical use of such information in the design of diagnostics and therapeutics [16, 23]. For example, identification of genetic markers in DNA sequences would provide important informations regarding which portions of the DNA are significant, and would allow the researchers to find many disease genes of interest (by recognizing them from the pattern of inheritance).

*Supported by NSF grant CCR-0208821.
†Supported in part by NSF grants CCR-0206795, CCR-0208749 and IIS-0346973.

Naturally, the large amount of available data poses a serious challenge in storing, retrieving and analyzing biological information.

A rapidly developing area, *computational biology*, is emerging to meet the rapidly increasing computational need. It consists of many important areas such as information storage, sequence analysis, evolutionary tree construction, protein structure prediction, and so on [16, 23]. It is playing an important role in some biological research. For example, sequence comparison is one of the most important methodological issues and most active research areas in current *biological sequence analysis*. Without the help of computers, it is almost impossible to compare two or more biological sequences (typically, at least a few hundred character long). Applications of sequence comparison methods can be traced back to the well-known *Human Genome Project* [29], whose objective is to decode this entire DNA sequence and to find the location and ordering of genetic markers along the length of the chromosome. These genetic markers can be used, for example, to trace the inheritance of chromosomes in families and thereby to find the location of disease genes. Genetic markers can be found by finding DNA polymorphisms, *i.e.*, locations where two DNA sequences "spell" differently. A key step in finding DNA polymorphisms is the calculation of the *genetic distance*, which is a measure of the correlation (or similarity) between two genomes.

In this chapter, we discuss computational complexities and approximation algorithms for a few sequence similarity problems. We assume that the reader is familiar with the basic concepts of exact and approximation algorithms [15, 28], basic computational complexity classes such as P and NP [17, 20, 26] and basic notions of molecular biology such as DNA sequences [18, 27].

2 Local and Global Sequence Alignments

The alignment of two sequences $\vec{a} \equiv \vec{a_m} = (a_1, \ldots, a_m)$ and $\vec{b} \equiv \vec{b_n} = (b_1, \ldots, b_n)$ is a matrix with two rows, each containing the symbols of the respective sequence that are interspaced with *gap* characters. The symbols in a column are aligned with each other, and this forms a direct form of comparison. The underlying assumption is that the aligned symbols have the same origin or the same function, and, at the very least,

those pairs of symbols that do share origin or the function — and not for every symbol we can find such a correspondence — will get aligned. In an alignment, columns with gaps contain the symbols for which we have not found any correspondence. In the case of biological sequences, the term "common origin" means a shared correspondence with the sequence of the evolutionary ancestor. The alterations of the ancestral sequence are *mutations*, which can be viewed as *editing operations*. There are many possible mutations, but the most frequent ones are the *small scale* mutations: substitutions of single symbols, insertions and deletions; see Figure 1.1 for an illustration. Large scale mutations in which portions of the sequence are duplicated and/or moved to other regions cannot be represented with this type of simple alignments. However, when we want to compare the sequences that were subjected to such large-scale mutations we still have to find simple alignments first. Let (m, n) denote the size of such an alignment problem.

There are two basic kinds of alignments: *global* [27], where the rows contain the entire sequences, and *local* [25], which are global alignments of *substrings*, *i.e.*, *contiguous* subsequences. To find the best global alignment, we give some cost to each editing operations and then we search for an alignment with the minimum cost. The costs should be chosen in such a way that the best alignments — consistent with our *a priori* biological information — become the alignments with the minimum cost. A desirable goal is to have an appropriate formulation of the alignment method that such that (a) it is easy to compute, (b) it has few arbitrary parameters, and (c) in the test cases, the correct alignments have the minimum cost.

In aligning DNA sequences, good results were obtained with so-called *affine* gap penalty; the cost of substitution of a column with symbols a_i, b_j is $\sigma(a_i, b_i)$ for some given function σ and the cost of a block of k insertions (or deletions) is $\alpha + k\beta$, where α is the *gap opening* cost and β is *gap closing* cost.

To obtain a recursive formula for the minimum global alignment cost of two sequences, we view the editing (alignment) operation as one that has three states: state 0 for substitutions, state 1 for deletions and state 2 for insertions. Next, we formulate a generic subproblem: find $C(i, j, s)$, the minimum cost of aligning $\vec{a_i} = (a_1, \ldots, a_i)$ with $\vec{b_j} = (b_1, \ldots, b_j)$, such that the *final* state is s. These subproblems can be solved with the following recurrence:

$C(i,j,s) = \infty$ if $i < 0$ or $j < 0$

$C(0,0,0) = 0$

$C(i,j,0) = \min\{C(i-1,j-1) + \sigma(a_i, b_j),\ C(i,j,1),\ C(i,j,2)\}$ if $i \geq 0,\ j \geq 0,\ i+j > 0$

$C(i,j,1) = \min\{C(i-1,j,1) + \beta,\ C(i,j,0) + \alpha,\ C(i,j,2) + \alpha\}$ if $i \geq 0,\ j \geq 0$

$C(i,j,2) = \min\{C(i,j-1,2) + \beta,\ C(i,j,0) + \alpha,\ C(i,j,1) + \alpha,\ \}$ if $i \geq 0,\ j \geq 0$

A very similar recurrence describes the minimum cost alignment; the base cases describe starting the alignment process, additions in the formula correspond to adding new columns to the alignment, and other recursive cases refer to transitions between different kinds of columns.

One can convert this recursive definition to a dynamic programming algorithm by introducing an array to store the values of $C(*,*,*)$; it is convenient to think about a rectangular matrix in which each entry is a triple. Those entries can be computed in an order consistent with the recursive dependence, *e.g.*, column by column.

Several improvements were subsequently made to this simple algorithm. In the computation of the cost, we need to store only two columns (or rows) at any given time. This reduces the memory requirements from $\Theta(mn)$ to $\Theta(n)$, thus increasing the lengths of sequences that can be handled in the internal memory of a computer. By combining this approach with divide-and-conquer method one can also compute the minimum cost alignment itself using $\Theta(n)$ space. First we run the memory-efficient cost computation for $\vec{a}$ and $\vec{b}_{n/2}$ which gives us $C(i, n/2, s)$ for every $i \leq m$ and $s \leq 2$. Next we run such a computation for reversed $\vec{a}$, *i.e.*, $(a_m, a_{m-1}, \ldots, a_1)$ and reversed $\vec{b}_{n/2}$, *i.e.*, $(b_{n/2+1}, \ldots, b_n)$; denote by $\bar{C}(i, n/2, s)$ the correspondingly computed quantities. To faithfully reverse the editing process, we must also "reverse" the recursive relationships; hence $\bar{C}(m, n, 0) = 0$, while the recurrence relationships are reversed as well, *e.g.*,

$$\bar{C}(i,j,0) = \min\left\{\bar{C}(i+1,j+1,0) + \sigma(a_{i+1}, b_{j+1}),\ \bar{C}(i,j,1) + \alpha,\ \bar{C}(i,j,2) + \alpha\right\}.$$

Now for every i and s we can check the minimum alignment cost that is obtained under the condition that the editing process goes through configuration $(i, n/2, s)$: $C(i, n/2, s) + \bar{C}(i, n/2, s)$. After finding the best pairs of i and s we know the middle column of the optimum alignment and we can find the left and the

right part of the alignment by recursive application of that method. One can see that the overall running time is roughly doubled: after computing for $\Theta(mn)$ steps we are solving problems with sizes $(i, n/2)$ and $(n-i, n/2)$, so the initial computations in the children calls take roughly half of the time needed for the initial computations of their parent. Note that the doubling of the number of operations combined with a very significant decrease in memory needs often means that we avoid the use of disk memory during the computation, which translates into much smaller clock time.

The global alignment problem is very intuitively related to the process of evolutionary changes (or editing), but it may poorly represent the actual problems encountered when we compare biological sequences. First, the fragments may be scattered, and are separated either by the results of duplications and rearrangements, or by poorly conserved regions. When a region in a sequence is functionally important, most of the mutations are detrimental to the organism and do not survive the selection process. Conversely, when a region has no biological function, this effect is very weak. Finding strongly *conserved regions* is one of the main tools in identifying genes and regulatory sites. Therefore often we need to solve *local alignment problem* rather than the global one.

The first methodological step in defining the local alignment problem is the modification of the cost function. Note that we can alter an alignment by replacing an insertion column and a deletion column with a single substitution column. Therefore the minimum cost global alignment will not change if we change the gap extension cost from β to $\beta+\gamma$ and the substitution cost from $\sigma(a,b)$ to $\sigma(a,b)-2\gamma$ (because all alignments will experience the same cost change). We need to choose the constant γ in such a way that the alignments that we "like" have negative cost and the alignments that we "do not like" have positive cost.

After such a change in the cost definition, the best local alignment has the minimum cost, and this cost is negative. Once we find such an alignment with cost, say, $-A$, we say that A is its *score*; the full name of this optimization problem is the *maximum score local alignment* problem.

It may seem that now we need to consider a vastly larger number of possibilities to find the optimum solution. In actuality, it suffices to use a very small alteration of the global alignment algorithm. First, we add zero as one more term to the minimum that defines $C(i,j,0)$. Using zero means that we decided to start

the local alignment from a_{i+1} and b_{j+1}. Second, the optimum solution is not $C(m, n, 0)$ but the minimum of all $C(i, j, 0)$'s. Because the new algorithm is so similar to the previous one, we can use exactly same trick to reduce the memory usage from $\Theta(mn)$ to $\Theta(n)$.

So far, we ignored the issue why we "like" certain local alignments. A typical application is the problem of identifying *exons*, the expressed parts of the genes. Exons are separated by introns that are not expressed, *i.e.*, transcribed into proteins. Mutations within exons are in vast majority detrimental since they may cut short so-called reading frame which destroys a large part of the coded protein, interfere with RNA folding process which has similar consequences, or change the aminoacid sequence in the coded protein which may interfere with its biological function. Surviving mutations of introns are much more frequent. Therefore when we align DNA of two species and we properly choose the column costs, vast majority of exons that are common to both species will form long local alignments with positive scores. In that context one has to address the following problem: *several exons may form a single local alignment if they are separated by short introns.*

A reasonable heuristic is to set a parameter X and break the local alignments by removing those regions that have score below $-X$. More precisely, a fragment of the alignment is X-*normal* if (1) every prefix and every suffix has a non-negative score, (2) no sub-fragment has a score below $-X$ and (3) it is not contained in a larger fragment with the properties (1) and (2). Our goal is to find all X-normal fragments. Since this is a heuristic, it is good to be able to inspect quickly the results for all possible X's. This problem has surprisingly simple and efficient solution [31] which we discuss below.

The input to this problem is a (long) local alignment A with positive score. We can partition it into (a) blocks of substitution columns with positive score and (b) blocks of substitution columns with negative scores and gaps (insertions or deletions); subsequently we create sequence $\vec{A}$ of scores of these parts. Note that in $\vec{A}$ the signs of the entries alternate. For the sake of convenience, add $-\infty$ at the beginning and at the end of $\vec{A}$. From now on, we identify (alignment) fragments with ranges of positions in $\vec{A}$, *i.e.*, a fragment of A consisting of parts with scores $a_i, \ldots, a_j$ is identified with integer range $[i, j]$.

Define a *normal drop* as a fragment such that its *every prefix and suffix* has a negative score. A *normal*

rise is a fragment that is X-normal for some $X > 0$. One can observe the following:

Lemma 1.1 *If A is a normal rise and B is either a normal rise or a normal drop, then $A \cap B \in \{A, B, \emptyset\}$.*

As a result, we can create an ordered tree of all normal rises in which siblings of a node/fragment are disjoint subsets of the parent, ordered from left to right by their position in the alignment. Moreover, if $[a, b]$ and $[c, d]$ are consecutive siblings then $[b + 1, c - 1]$ is a normal drop. These normal drops can be added to the tree of normal rises. We call it a *fragment tree*: parent fragment is the union of children fragments. Now our goal is to construct a fragment tree that contains all normal rises. Our task will be easier after we add more nodes so it becomes a ternary tree.

We annotate each node/fragment with a pair (s, d) where s is the score and d is the maximum drop, *i.e.*, negative of the minimum score of a subfragment.

Suppose first that a node p had exactly three children — it cannot have less because a rise has to have at least two rise children and they have to be separated by a drop. Then we have the following rule:

$$(s_0 - d + s_1, d) \ \rightarrow \ (s_0, d_0)(-d, d)(s_1, d_1) \text{ where } d_0, d_1 < d$$

.

Now suppose that a node p annotated with (s, d) has more than three children; we will indicate how to add a parent to three of them so that the number of children of p drops by two. Consider a child with the minimum absolute value of its score, say a; it can be the first only if the second child is a drop $(-a, a)$ and then we focus on the second child; similarly we can assume it is not the last child. Consider the annotations of that child together with its left and right siblings.

Case: $(s_0, d_0), (-a, a), (s_2, d_2)$. Because $a \leq s_0, s_1$ we can coalesce the three siblings into a rise with drop value $b = \max\{d_0, a, d_2\}$. If $b < d$ then we have a contradiction with the assumption that the tree contains all normal rises, as the three siblings must have an ancestor that is a descendant of p and is b-normal; hence $b = d = a$. Thus we can add a parent that is almost d-normal, the exception being that it can be extended to the right. We can alter the previous rule so it becomes consistent with this case:

$$(s_0 - d + s_1, d) \ \rightarrow \ (s_0, d_0)(-d, d)(s_1, d_1) \text{ where } d_0 \leq d, \ d_1 < d$$

Case: $(-d_0, d_0), (s_1, d_1), (-d_2, d_2)$. One can see that we can combine these three children into a single normal drop, and the rule is

$$(-d_0 + s_1 - d_2, d_0 - s_1 + d_2) \ \rightarrow \ (-d_0, d_0), (s_1, d_1), (-d_2, d_2) \text{ where } d_0, d_2 > s_1$$

Note that we can add a node using this rule for the first applicable triple of siblings. Our two rules describe create a very simple context-free grammar after we add terminating rules

$$(s, 0) \ \rightarrow \ s, \qquad (-d, d) \ \rightarrow \ -d.$$

and the desired tree of fragments is the parsing tree for the sequence $\vec{A}$. Such a parsing tree can be computed in linear time using a deterministic push-down automata (*e.g.*, see [1, 21]).

3 Nonoverlapping Local Alignments

As we have already seen, a fundamental problem in computational molecular biology is to elucidate similarities between sequences and a cornerstone result in this area is that given two strings of length p and q, there are local alignment algorithms that will score pairs of substrings for "similarity" according to various biologically meaningful scoring functions and we can pull out all "similar" or high scoring substring pairs in time $O(pq + n)$ where n is the output size [27]. Having found the high scoring substring pairs, a global description of the similarity between two sequences is obtained by choosing the disjoint subset of these pairs of highest total score. This problem is in general referred to as the "non-overlapping local alignment" problem. We also mention a more general "d-dimensional version" of this problem involving $d > 2$ sequences, where we score d substrings, one from each sequence, with a similarity score and the goal is to select a collection of disjoint subsets of these d-tuples of substrings maximizing the total similarity.

A natural geometric interpretation of the problem is via selecting a set of "independent" rectangles in the plane in the following manner [4]. Each output substring pair being represented as a rectangle; Figure 1.2 shows a pictorial illustration of the relationship of a rectangle to local similarity between two fragments of two sequences. This gives rise to the following combinatorial optimization problem. We are a given a set S

of n positively weighted axis parallel rectangles. Define two rectangles to be independent if for each axis, the projection of one rectangle does not overlap that of another. The goal is to select a subset $S' \subseteq S$ of *independent* rectangles from the given set of rectangles of total maximum weight. The *unweighted* version of the problem is the one in which the weights of all rectangles are identical. In the d-dimensional version, we are given a set of positively weighted axis parallel d-dimensional hyper-rectangles[1] such that, for every axis, the projection of a hyper-rectangle on this axis does not enclose that of another. Defining two hyper-rectangles to be independent if for every axis, the projection of one hyper-rectangle does not overlap that of another; the goal is to select a subset of *independent* hyper-rectangles of total maximum weight from the given set of hyper-rectangles.

The non-overlapping local alignment problem, including its special case as defined by the IR problem described in Section 3.2, is known to be NP-complete. The best known algorithm for the general version of the nonoverlapping local alignment problem is due to [8] who provide a $2d$-approximation for the problem involving d-dimensional hyper-rectangles. In the sequel, we will discuss two important special cases of this problem that are biologically relevant.

3.1 The Chaining Problem

The chaining problem is the following special case [18, page 326]. A subset of rectangles is called a *chain* if no horizontal or vertical line intersects more than one rectangle in the subset in the subset and if the rectangles in the subset can be ordered such that each rectangle in this order is below and to the right of its predecessor. The goal is to find a chain of maximum total similarity. This problem can be posed as finding the longest path in a directed acyclic graph and thereby admits an optimal solution in $O(n^2)$ time where n is the number of rectangles. However, using a sparse dynamic programming method, the running time can be further improved to $O(n \log n)$ [22].

[1] A d-dimensional hyper-rectangle is a Cartesian product of d intervals.

3.2 The Independent Subset of Rectangles (IR) Problem

In this problem, first formulated by [4], for each axis, the projection of a rectangle on this axis does not enclose that of another; this restriction on the input is biologically justified by a preprocessing of the input data (fragment pairs) to eliminate violations of the constraint. See Figure 1.3 for an pictorial illustration of the problem.

Consider the graph G formed from the given rectangles in which there is a node for every rectangle with its weight being the same as that of the rectangle and two nodes are connected by an edge if and only if their rectangles are *not* independent. It is not difficult to see that G is a 5-claw free graph [4] and the IR problem is tantamount to finding a *maximum-weight* independent set in G. Many previous approaches have used this connection of the IR problem to the 5-claw free graphs to provide better approximation algorithms by giving improved approximation algorithms for d-claw free graphs. For example, using this approach, Bafna et al. [4] provided a polynomial time approximation algorithm with a performance ratio[2] of $\frac{13}{4}$ for the IR problem and Halldórsson [19] provided a polynomial time approximation algorithm with a performance ratio of $2+\varepsilon$ (for any constant $\varepsilon > 0$) for the unweighted version of the IR problem[3]. The current best approximation algorithm for the IR problem is due to Berman [9] via the same approach which has a performance ratio of $\frac{5}{2}+\varepsilon$ (for any constant $\varepsilon > 0$).

Many of the abovementioned algorithms essentially start with an arbitrary solution and then allows small improvements to enhance the approximation quality of the solution. In contrast, in this section we review the usage of a simple greedy two-phase technique to provide an approximation algorithm for the IR problem with a performance ratio of 3 that runs in $O(n \log n)$ time [12, 13]. The two-phase technique was introduced in its more general version as a multi-phase approach in the context of real-time scheduling of jobs with deadline in [10, 11]; we review the generic nature of this technique in Section 3.2.1. Although this approximation

[2]The performance ratio of an approximation algorithm for the IR problem is the ratio of the total weights of rectangles in an optimal solution to that in the solution provided by the approximation algorithm.

[3]For this and other previous approximation algorithms with an ε in the performance ratio, the running time increases with decreasing ε, thereby rendering these algorithms impractical if ε is small. Also, a straightforward implementation of these algorithms will run in at least $\Omega(mn)$ time.

algorithm does not improve the worst-case performance ratios of previously best algorithms, it is simple to implement (involving standard simple data structures such as stacks and binary trees) and runs faster than the algorithms in [4, 9, 19].

3.2.1 The Local-ratio and Multi-phase Techniques

The multi-phase technique was introduced formally in the context of real-time scheduling of jobs by the investigators in [10, 11]. Informally and very briefly, this technique works as follows:

(a) We maintain a stack **S** containing objects that are tentatively in the solution. **S** is initially empty before the algorithm starts.

(b) We make $k \geq 1$ *evaluation passes* over the objects. In each evaluation pass:

- we inspect the objects in a specific order that is easy to compute (*e.g.*, rectangles in the plane in the order of their right vertical side),
- depending on the current content of **S**, the contents of **S** during the previous passes as well as the attributes of the current object, we compute a score for the object,
- we push the object to **S** if the score is above a certain threshold

(c) We make one *selection pass* over the objects in **S** in a specific order (typically, by popping the elements of **S**) and select a subset of the objects in **S** that satisfy the feasilibilty criteria of the optimization problem under consideration.

Closely related to the two-phase version of the multi-phase technique, but somewhat of more general nature, is the *local-ratio* technique. This technique was first developed by Bar-Yehuda and Even [7] and later extended by Berman et al. [3] and Bar-Yehuda [6]. The crux of the technique is as follows [5]. Assume that given an n-dimensional vector $\vec{p}$, our goal is to find a n-dimensional *solution* vector $\vec{x}$ that maximizes (respectively, minimizes) the inner product $\vec{p} \cdot \vec{x}$ subject to some set $\mathcal{F}$ of *feasibility constraints* on $\vec{x}$. Assume that we have decomposed the vector $\vec{p}$ to two vectors $\vec{p_1}$ and $\vec{p_2}$ with $\vec{p_1} + \vec{p_2} = \vec{p}$ such that, for some $r \geq 1$ (respectively, $r \leq 1$), we can find a solution vector $\vec{x}$ satisfying $\mathcal{F}$ which r-approximates $\vec{p_1}$ and $\vec{p_2}$, that is,

which satisfies both $\vec{p_1} \cdot \vec{x} \geq r \cdot \max_{\vec{y}}\{\vec{p_1} \cdot \vec{y}\}$ and $\vec{p_2} \cdot \vec{x} \geq r \cdot \max_{\vec{y}}\{\vec{p_2} \cdot \vec{y}\}$ (respectively, $\vec{p_1} \cdot \vec{x} \leq r \cdot \min_{\vec{y}}\{\vec{p_1} \cdot \vec{y}\}$ and $\vec{p_2} \cdot \vec{x} \leq r \cdot \min_{\vec{y}}\{\vec{p_2} \cdot \vec{y}\}$). Then, $\vec{x}$ also r-approximates $\vec{p}$. This allows a given problem to be recursively broken down in subproblems from which one can recover a solution to the original problem. The local-ratio approach makes it easier to extend the results to a larger class of problems, while the multi-phase approach allows to obtain better approximation ratios in many important special cases.

The multi-phase technique was used in the context of job scheduling in [10, 11] and in the context of opportunity-cost algorithms for combinatorial auctions in [2]. We will discuss the usage the two-phase version of the multi-phase approach in the context of the IR problem [12, 13] in the next section. In some cases, it is also possible to explain the multi-phase or the local-ratio approach using the primal-dual schema; for example, see [5].

3.2.2 Application of the Two-Phase Technique to the IR Problem

The following notations and terminologies are used for the rest of this section. An interval $[a, b]$ is the set $[a, b] = \{x \in \mathbb{R} : \ a \leq x \leq b\}$. A rectangle R is $[a, b] \times [c, d]$ for some two intervals $[a, b]$ and $[c, d]$, where $\times$ denotes the Cartesian product. The weight of a rectangle R is denoted by $w(R)$. We assume that the reader with familiar with standard techniques and data structures for the design and analysis of algorithms such as in [15].

Let $R_1, R_2, \ldots, R_n$ be the n input rectangles in our collection, where $R_i = X_i \times Y_i$ for some two intervals $X_i = [d_i, e_i]$ and $Y_i = [f_i, g_i]$. Consider the intervals $X_1, X_2, \ldots, X_n$ formed by projecting the rectangles on one axis and call two intervals X_i and X_j independent if and only if the corresponding rectangles R_i and R_j are independent. The notation $X_i \simeq X_j$ (respectively, $X_i \not\simeq X_j$) is used to denote if two intervals X_i and X_j are independent (respectively, not independent).

To simplify implementation, we first sort the set of numbers $\{d_i, e_i \mid 1 \leq i \leq n\}$ (respectively, the set of numbers $\{f_i, g_i \mid 1 \leq i \leq n\}$) and replace each number in the set by its rank in the sorted list. This does not change any feasible solution to the given problem; however, after this $O(n \log n)$ time preprocessing we can assume that $d_i, e_i, f_i, g_i \in \{1, 2, \ldots, 2n\}$ for all i. This assumption simplifies the design of data structures

for the IR problem.

Now, we adopt the two-phase technique on the intervals $X_1, X_2, \ldots, X_n$. The precise algorithm is shown in Figure 1.4. The solution to the IR problem consists of those rectangles whose projections are returned in the solution at the end of the selection phase.

We first prove the correctness and running time of Algorithm TPA-IR. To show that the algorithm is correct we just need to show that the selected rectangles are mutually independent. This is obviously ensured by the final selection phase.

It takes $O(n \log n)$ time to create the list $\mathbf{L}$ by sorting the endpoints of the n rectangles. It is easy to see that, for each interval $X_i = [d_i, e_i]$ $(1 \leq i \leq n)$, the algorithm performs only a constant number of operations, which are elementary except for the computation of TOTAL(X_i) in the evaluation phase. We need to show how this function can be computed in $O(\log n)$ time for each X_i. Note that $X_i \not\simeq X_j \equiv (X_i \cap X_j \neq \emptyset) \vee (Y_i \cap Y_j \neq \emptyset)$. Since the intervals are considered in non-decreasing order of their endpoints, there is no interval in $\mathbf{S}$ with an endpoint to the right of e_i when TOTAL(X_i) is computed. As a result, when the computation of TOTAL(X_i) is needed, $X_i \not\simeq X_j$ for an X_j currently in stack provided *exactly* one of the following two conditions is satisfied:

(a) $e_j \geq d_i$,

(b) $(e_j < d_i) \wedge (Y_i \cap Y_j \neq \emptyset)$.

Since for any two intervals $Y_i = [f_i, g_i]$ and $Y_j = [f_j, g_j]$, such that neither interval encloses the other, $[f_i, g_i] \wedge [f_j, g_j] \neq \emptyset$ is equivalent to either $f_i \leq f_j \leq g_i$ or $f_i \leq g_j \leq g_i$ but not both, it follows that for the purpose of computing TOTAL(X_i) it suffices to maintain a a data structure $\mathcal{D}$ for a set of points in the plane with coordinates from the set $\{1, 2, \ldots, 2n\}$ such that the following two operations can be performed:

Insert(v, x, y)**:** Insert the point with coordinates (x, y) (with $x, y \in \{1, 2, \ldots, 2n\}$) and value v in $\mathcal{D}$. Moreover, if Insert(v, x, y) precedes Insert(v', x', y'), then $y' \geq y$.

Query(a, b, c)**:** Given a query range (a, b, c) (with $a, b, c \in \{1, 2, \ldots, 2n\} \cup \{-\infty, \infty\}$), find the sum of the values of all points (x, y) in $\mathcal{D}$ with $a \leq x \leq b$ and $y \geq c$.

For example, finding all X_j's currently in stack with $e_j \geq d_i$ is equivalent to doing Query$(-\infty, \infty, d_i)$.

For notational simplicity, assume that $n = 2^k$ for some positive integer k. We start with a skeleton rooted balanced binary tree T with $2n$ leaves (and of height $O(\log n)$) in which each node will store a number in $\{1, 2, \ldots, 2n\}$. The i^{th} leaf of T (for $1 \leq i \leq 2n$) will store the point (i, y), if such a point was inserted. With each node v of T, we also maintain the following:

- a list L_v of the points stored in the leaves of the subtree rooted at v, sorted by their second coordinate. Additionally, each entry (x, y) in the list also has an additional field $sum_{x,y}$ storing the sum of all values of all points in the list to the left of (x, y) including itself, that is, the sum of values of all points (x', y') in L_v with $y' \leq y$.
- a value s_v equal to the sum of values of all points in L_v.

Initially $L_v = \emptyset$ and $s_v = 0$ for all $v \in T$ and building the skeleton tree thus obviously takes $O(n)$ time.

To implement Insert(v, x, y), we insert the point (x, y) at the x^{th} leaf of T and update L_v and s_v for every node v on the unique path from the root to the x^{th} leaf. Since Insert(v, x, y) precedes Insert(v', x', y') implies $y' \geq y$, we simply append (x, y) to the existing L_v for every such v. It is also trivial to update $sum_{x,y}$ (from the value of $sum_{x',y'}$ where (x', y') was the previous last entry of the list) and s_v in constant time. Since there are $O(\log n)$ nodes on the above unique path, we spend $O(\log n)$ time. In a similar manner, we can also implement Query(a, b, c) in $O(\log n)$ time.

Now, we prove the performance ratio of Algorithm TPA-IR. Let B be a solution returned by Algorithm TPA-IR and A be any optimal solution. For a rectangle $R \in A$, let us define the *local conflict number* β_R to be the number of those rectangles in B that were *not* independent of R *and* were examined no earlier than R by the evaluation phase of Algorithm TPA-IR and let $\beta = \max_{R \in A} \beta_R$.

First, we show that Algorithm TPA-IR has a performance ratio of β. Consider the set of intervals $\mathbf{S}$ in the stack at the end of the evaluation phase. Let $W(A) = \sum_{R_i \in A} w(R_i)$ and $V(\mathbf{S}) = \sum_{(v, d_i, e_i) \in \mathbf{S}} v$. It was shown in Lemma 3 of [10] that the the sum of the weights of the rectangles selected during the selection phase is at least $V(\mathbf{S})$. Hence, it suffices to show that $\beta V(\mathbf{S}) \geq W(A)$.

Consider a rectangle $R_i = X_i \times Y_i \in A$ and the time when the evaluation phase starts the processing of $X_i = [d_i, e_i]$. Let TOTAL$'([d_i, e_i])$ and TOTAL$''([d_i, e_i])$ be the values of TOTAL$([d_i, e_i])$ before and after the processing of X_i, respectively.

If $w(R_i) <$ TOTAL$'([d_i, e_i])$, then X_i is not pushed to the stack and TOTAL$''([d_i, e_i]) =$ TOTAL$'([d_i, e_i])$. On the other hand, if $w(R_i) \geq$ TOTAL$'([d_i, e_i])$, then X_i is pushed to the stack with a value of $w(R_i) -$ TOTAL$'([d_i, e_i])$, as a result of which TOTAL$''([d_i, e_i])$ becomes at least TOTAL$'([d_i, e_i]) + (w(R_i) -$ TOTAL$'([d_i, e_i])) = w(R_i)$. Hence, in either case TOTAL$''([d_i, e_i]) \geq w(R_i)$.

Now, summing up over all R_i's and using the definition of β and TOTAL$''([d_i, e_i])$ gives

$$
\begin{aligned}
& W(A) \\
= & \sum_{R_i \in A} w(R_i) \\
\leq & \sum_{R_i = [d_i, e_i] \times Y_i \in A} \text{TOTAL}''([d_i, e_i]) \\
\leq & \sum_{((v,a,b) \in \mathbf{S}) \wedge (b \leq e_i) \wedge ([a,b] \not\supseteq [d_i, e_i])} v \qquad \text{(by definition of TOTAL}''([d_i, e_i]) \\
\leq & \ \beta \sum_{(v,a,b) \in \mathbf{S}} v \qquad \text{(by definition of } \beta) \\
= & \ \beta V(\mathbf{S})
\end{aligned}
$$

Now, we can show that the performance ratio of Algorithm TPA-IR is 3 by showing that For the IR problem, $\beta = 3$. First note that $\beta = 3$ is possible; see Figure 1.5. Now we show that $\beta > 3$ is impossible. Refer to Figure 1.5. Remember that rectangles in an optimal solution contributing to β must not be independent of our rectangle R and must have their right vertical right on or to the right of the vertical line L. Since rectangles in an optimal solution must be independent of each other, there can be at most one optimal rectangle crossing L (and, thereby conflicting with R in its projections on the x-axis). Any other optimal rectangle must lie completely to the right of L and therefore may conflict with R in their projections on the y-axis only; hence there can be at most two such rectangles.

3.2.3 Concluding Remarks

Algorithm TPA-IR makes a pass on the projections of the rectangles on the x-axis in a nondecreasing order of the endpoints of the projections. Can we improve the performance ratio if we run TPA-IR separately on the projections on the x-axis in left-to-right and in right-to-left order of endpoints and take the better of the two solutions? Or, even further, we may try running Algorithm TPA-IR two more times separately on the projections on the y-axis in top-to-bottom and in bottom-to-top order and take the best of the four solutions. Figure 1.6 shows that even then the worst case performance ratio will be 3. We already exploited the planar geometry induced by the rectangles for the IR problem to show that $\beta \leq 3$. Further research may be necessary to see whether we can exploit the geometry of rectangles more to design simple approximation algorithms with performance ratios better than 2.5 in the weighted case or better than 2 in the unweighted case.

For the d-dimensional version, Algorithm TPA-IR can be applied in an obvious way to this extended version by considering the projections of these hyper-rectangles on a particular axis. It is not difficult to see that $\beta \leq 2d - 1$ for this case [14], thus giving a worst-case performance ratio of $2d - 1$. Whether one can design an algorithm with a performance ratio that increases less drastically (*e.g.*, sublinearly) with d is still open.

References

[1] A. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] K. Akcoglu, J. Aspnes, B. DasGupta and M.-Y. Kao. *Opportunity Cost Algorithms for Combinatorial Auctions*, in Applied Optimization: Computational Methods in Decision-Making, Economics and Finance, E. J. Kontoghiorghes, B. Rustem and S. Siokos (editors), Kluwer Academic Publishers, pp. 455-479, September 2002.

[3] V. Bafna, P. Berman and T. Fujito. *Constant ratio approximation of the weighted feedback vertex set*

problem for undirected graphs, Int. Symp. on Algorithms and Computation, LNCS 1004, 1995, pp. 142-151.

[4] V. Bafna, B. Narayanan and R. Ravi. *Nonoverlapping local alignments (Weighted independent sets of axis-parallel rectangles*, Discrete Applied Mathematics, 71, pp. 41-53, 1996.

[5] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. S. Naor, and B. Schieber. *A unified approach to approximating resource allocation and scheduling*, Proceedings of the 32nd Annual ACM Symposium on Theory of Computing, 2000, pp. 735–744.

[6] R. Bar-Yehuda. *One for the price of two: a unified approach for approximating covering problems*, Algorithmica, 27 (2), pp. 131-144, 2000.

[7] R. Bar-Yehuda and S. Even. *A local-ratio theorem for approximating the weighted vertex cover problem*, Annals of Discrete Mathematics, 25, 1985, pp. 27-46.

[8] R. Bar-Yehuda, M. M. Halldórsson, J. Naor, H. Shachnai and I. Shapira. *Scheduling split intervals*, 14th ACM-SIAM Symposium on Discrete Algorithms, pp. 732-741, 2002.

[9] P. Berman. *A $d/2$ approximation for maximum weight independent set in d-claw free graphs*, proceedings of the 7th Scandinavian Workshop on Algorithmic Theory, Lecture Notes in Computer Science, 1851, Springer-Verlag, July 2000, pp. 214-219.

[10] P. Berman and B. DasGupta. *Improvements in Throughput Maximization for Real-Time Scheduling*, proceedings of the 32nd Annual ACM Symposium on Theory of Computing, May 2000, pp. 680-687.

[11] P. Berman and B. DasGupta. *Multi-phase Algorithms for Throughput Maximization for Real-Time Scheduling*, Journal of Combinatorial Optimization, Vol. 4, No. 3, September 2000, pp. 307-323.

[12] P. Berman and B. DasGupta. *A Simple Approximation Algorithm for Nonoverlapping Local Alignments (Weighted Independent Sets of Axis Parallel Rectangles)*, in Biocomputing, Volume 1, Panos M. Pardalos and Jose Principe (editors), Kluwer Academic Publishers, pp. 129-138, June 2002.

[13] P. Berman, B. DasGupta and S. Muthukrishnan. *Simple Approximation Algorithm for Nonoverlapping Local Alignments*, 13th ACM-SIAM Symposium on Discrete Algorithms, January 2002, pp. 677-678.

[14] M. Chlebík and J. Chlebíková. *Approximation Hardness of Optimization Problems in Intersection Graphs of d-dimensional Boxes*, proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 267-276, 2005.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*, The MIT Press, 2001.

[16] K. A. Frenkel. The human genome project and informatics, *Communications of the ACM*, 34 (11), pp. 41-51, 1991.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness,* W. H. Freeman, 1979.

[18] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge Univ Press, 1997.

[19] M. M. Halldórsson. *Approximating discrete collections via local improvements*, proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms, January 1995, pp. 160-169.

[20] D. Hochbaum. *Approximation Algorithms for NP-hard Problems*, PWS publishers, 1996.

[21] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.

[22] D. Joseph, J. Meidanis and P. Tiwari. *Determining DNA sequence similarity using maximum independent set algorithms for interval graphs*, 3rd Scandinavian Workshop on Algorithm Theory, LNCS 621, pp. 326-337, 1992.

[23] E. S. Lander, R. Langridge and D.M. Saccocio. *Mapping and interpreting biological information*, Communications of the ACM, 34 (11), pp. 33-39, 1991.

[24] W. Miller, S. Scbwartz, and R. C. Hardison. *A point of contact between computer science and molecular biology*, IEEE Computational Science and Engineering, Spring 1994, pp. 69-78.

[25] S. Needleman and C. Wunsch. *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, J. Mol. Biol., 48(3), pp. 443-453, 1970.

[26] C. H. Papadimitriou. *Computational Complexity*, Addison-Wesley; reading, MA, 1994.

[27] T. F. Smith and M. S. Waterman. *The identification of common molecular sequences*, Journal of Molecular Biology, 147, 1981, pp. 195-197.

[28] V. Vazirani. *Approximation Algorithms*, Springer-Verlag, July 2001.

[29] J. C. Venter *et al. The sequence of the human genome*, Science, 291, pp. 1304-1351, 2001.

[30] M.S. Waterman. *Sequence alignments*, in Mathematical Methods for DNA Sequences, M.S. Waterman (ed.), CRC, Boca Raton, FL, 1989, pp. 53-92.

[31] Z. Zhang, P. Berman, T. Wiehe and W. Miller. *Post-processing long pairwise alignments*, Bioinformatics 15(12), pp. 1012-1019, 1999.

```
A  A  C  C  –  –  –  G  T  A  G
A  A  –  –  T  T  T  G  A  A  C
1  2  3  4  5  6  7  8  9  10 11
```

Figure 1.1: Column classification in an alignment: $1-2$ and $8-11$ are substitutions, while $3-7$ are gaps, gaps $3-4$ are deletions and $4-7$ are insertions.

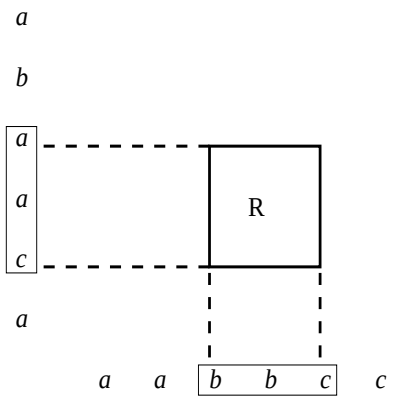

Figure 1.2: The rectangle R captures the local similarity (match) between the fragments aac and bbc of the two sequences; weight of R is the strength of the match.

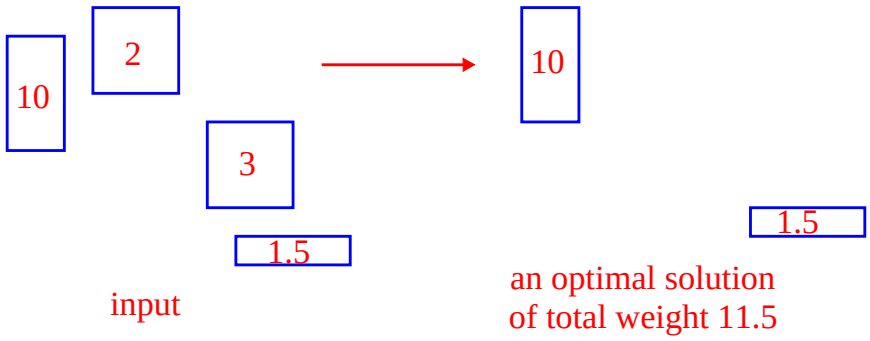

Figure 1.3: An illustration of the IR problem

(* definitions *)

a *triplet* (α, β, γ) is an ordered sequence of three values α, β and γ;

L is sequence that contains a triplet $(w(R_i), d_i, e_i)$

for every $R_i = X_i \times Y_i$ with $X_i = [d_i, e_i]$;

L is sorted so the values of e_i's are in non-decreasing order;

S is an initially empty stack that stores triplets;

TOTAL(X_j) returns the sum of v's of those triplets $(v, a, b) \in$**S**such that $[a, b] \not\simeq X_j$;

(* evaluation phase *)

for (each $(w(R_i), d_i, e_i)$ from **L**)

{

$v \leftarrow w(R_i) -$ TOTAL($[d_i, e_i]$);

if ($v > 0$) push((v, d_i, e_i),**S**);

}

(* selection phase *)

while (**S** is not empty)

{

$(v, d_i, e_i) \leftarrow$ pop(**S**);

if ($[d_i, e_i] \simeq X$ for every interval X in our solution)

insert $[d_i, e_i]$ to our solution;

}

Figure 1.4: Algorithm TPA-IR: Adoption of the two-phase technique for the IR problem.

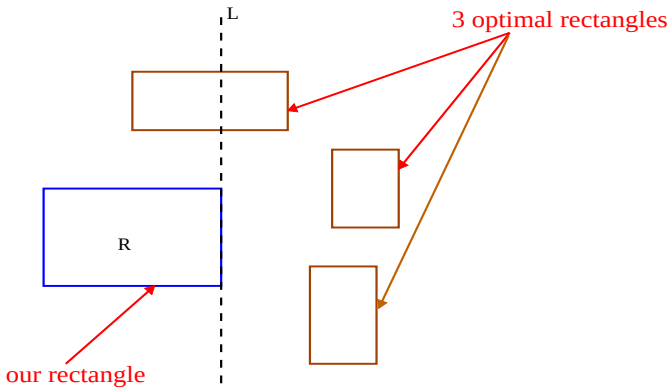

Figure 1.5: A tight example for Algorithm TPA-IR showing $\beta = 3$ is possible.

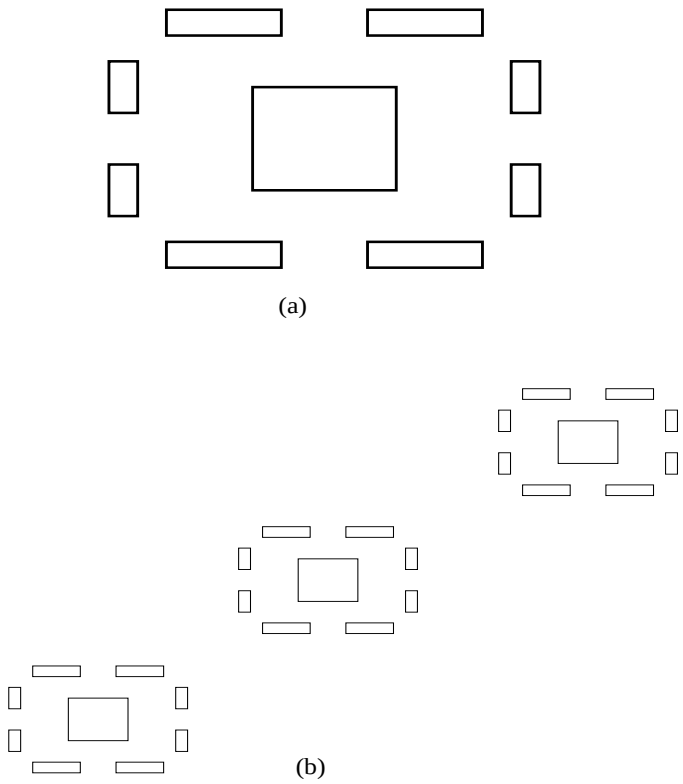

Figure 1.6: (a) The basic block of 9 rectangles such that the four runs of TPA-IR will always select R, whereas an optimal solution will select 3 of the remaining 8 rectangles. (b) The basic block repeated $\frac{n}{9}$ times such that different copies do not interfere in their projections on either axis, resulting in a performance ratio of 3 even for the best of the 4 runs.

Index