EFFICIENT ALGORITHMS FOR SEQUENCE ANALYSIS WITH CONCAVE AND CONVEX GAP COSTS

David A. Eppstein

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
1989

ABSTRACT

EFFICIENT ALGORITHMS FOR SEQUENCE ANALYSIS WITH CONCAVE AND CONVEX GAP COSTS

David A. Eppstein

We describe algorithms for two problems in sequence analysis: sequence alignment with gaps (multiple consecutive insertions and deletions treated as a unit) and RNA secondary structure with single loops only. We make the assumption that the gap cost or loop cost is a convex or concave function of the length of the gap or loop, and show how this assumption may be used to develop efficient algorithms for these problems. We show how the restriction to convex or concave functions may be relaxed, and give algorithms for solving the problems when the cost functions are neither convex nor concave, but can be split into a small number of convex or concave functions. Finally we point out some sparsity in the structure of our sequence analysis problems, and describe how we may take advantage of that sparsity to further speed up our algorithms.

CONTENTS

FIGURES

ACKNOWLEDGEMENTS

I would like to thank: my advisor, Zvi Galil, for much help and encouragement; my co-authors, Raffaele Giancarlo and Pino Italiano, without whom this research would not have begun, and for the espresso; the other faculty and students at Columbia; and last but far from least, Diana, for providing me with the motivation to finish.

This research was supported in part by an NSF student fellowship, and by NSF grants DCR-85-11713, CCR-86-05353, and CCR-88-14977.

1. INTRODUCTION

We consider algorithms for two problems in sequence analysis. The first problem is sequence alignment, and the second is the prediction of RNA structure. Although the two problems seem quite different from each other, their solutions share a common structure, which can be expressed as a system of dynamic programming recurrence equations. These equations also can be applied to other problems, including text formatting and data storage optimization.

We use a number of assumptions about the problems in order to provide efficient algorithms. The primary assumption is that of concavity or convexity. The recurrence relations for both sequence alignment and for RNA structure each include an energy cost, which in the sequence alignment problem is a function of the length of a gap in either input sequence, and in the RNA structure problem is a function of the length of a loop in the hypothetical structure. In practice this cost is taken to be the logarithm, square root, or some other simple function of the length. For our algorithms we make no such specific assumption, but we require that the function be either convex or concave. We also give algorithms for functions that are neither convex nor concave, but which can be broken up into a small number of pieces each of which is convex or concave.

The second assumption is that of sparsity. In the sequence alignment problem we need only consider alignments involving some sparse set of exactly matching subsequences; analogously, in the RNA structure problem we need only consider structures involving some sparse set of possible base pairs. We show how the algorithms for both problems may be further sped up by taking advantage of this sparsity rather than simply working around it.

Although the problems we consider have practical applications, we consider their analysis from the perspective of theoretical computer science. In particular,

we compare the merit of different algorithms based on formulae for the longest possible time they can take on inputs of a given length (worst case analysis), rather than on experimental timing information. Further, we ignore for the most part the constant factors in these formulae; thus we say that the time for some algorithm on inputs of length n is $O(f(n))$, meaning there exists a constant c such that the time is less than $cf(n)$.

Because the sequence analysis problems we are interested in need to be solved for very long sequences, and because in many cases our algorithms are better than previous ones by as much as an order of magnitude (factor of n), differences among the constant factors in our time bounds will typically be overwhelmed by differences in the non-constant parts of the bounds. However these constants are not entirely unimportant, and we endeavor to keep them small. Further, we point out which versions of our algorithms are most suitable for practical implementation, and which are of more purely theoretical interest.

1.1. Motivation

Our primary motivation in developing the sequence analysis algorithms we present is their application to molecular biology, although the same the same sequence analysis procedures also have important uses in other fields. The reasons for the particular interest in molecular biology are, first, that it is a growing and important area of scientific study, and second, the lengths of biological sequences are such that the need for efficient methods of sequence analysis is acute.

The development and refinement of efficient techniques for DNA sequencing [60, 51] has contributed to a boom in nucleic acid research. A wealth of molecular data is presently stored in several data banks (reviewed by Hobish [28]), which are loosely connected with each other. The biggest of these banks are GenBank [11] and the EMBL data library [23]. As of 1986, GenBank contained 5,731 entries with a total of more than 5 million nucleotides. Most of these data banks double in size every 8-10 months, and there are currently no signs that this growth is slowing down. On the contrary, the introduction of automatic sequencing techniques is expected to accelerate the process [48]. The existence of these nucletide and amino acid data banks allows scientists to compare sequences of molecules and find similarities between them. As the quantity of data grows, however, it is becoming increasingly difficult to compare a given sequence, usually a newly determined one, with the entire body of sequences within a particular data bank.

The current methods of RNA structure computation have similar limitations. In this case one needs only perform computations on single sequences of RNA, rather than performing computations on entire data banks at once. However the cost of present methods grows even more quickly with the size of the problem than does the cost of sequence comparison, and so at present we can only perform structure computations for relatively small sequences of RNA.

Present algorithms for biological sequence analysis tend to incorporate sophisticated knowledge of the domain of application, but their algorithmic content is limited. Typically they act by computing a set of simple recurrences using dynamic programming in a matrix of values. Our algorithms solve the same problems, using assumptions about the physical constraints of the domain that are either the same as previous assumptions or less restrictive than them. However by using more sophisticated algorithmic techniques we can take better advantage of the physical constraints of the problems to derive more efficient methods for sequence analysis.

1.2. Definitions of Convexity and Concavity

All our algorithms use in some way or another the assumption that some cost function in the recurrence being solved satisfies a concavity or convexity condition.

The usual definition for a function $g(x)$ to be concave is that for any $a \leq b$ and $0 \leq t \leq 1$,

$$g(ta + (1-t)b) \leq tg(a) + (1-t)g(b). \tag{1}$$

I.e., the function $g(x)$ on the interval $[a, b]$ always falls on or below the line between the function's values on the endpoints of the interval. An equivalent definition for continuous functions is that, for each $a \leq b$, $g(a)+g(b) \geq 2g((a+b)/2)$. This clearly follows from inequality 1, and conversely inequality 1 can be recovered from this special case by approximating $ta + (1-t)b$ using binary search. Similar definitions can be given for convexity, or we can simply define $g(x)$ to be convex exactly when $f(x) = -g(x)$ is concave.

For our purposes, all cost functions will be functions of pairs of numbers (x, y), with $x \leq y$. Since the usual definitions of convexity or concavity apply only to functions of one variable, we need to explain the convexity and concavity conditions we use. we say that $w(x, y)$ is *concave* when it satisfies the *quadrangle inequality*:

$$w(i, j) + w(i', j') \leq w(i', j) + w(i, j'), \tag{2}$$

whenever $i \leq i' \leq j \leq j'$. Similarly, $w(x, y)$ is *convex* when it satisfies the inverse quadrangle inequality, which can be found by replacing $\leq$ with $\geq$ in inequality 2. Equivalently $w(x, y)$ is convex when $f(x, y) = -w(x, y)$ is concave.

An important case of such two dimensional convex or concave functions is that $w(x, y)$ is really some function $g(y - x)$ of the difference between the two variables. For such g satisfying the quadrangle inequality, and for $0 \leq a \leq b$, let $x = (a + b)/2$

and $y = x - a = b - x = (b-a)/2$. Then $0 \le y \le x \le b$ and

$$\begin{aligned} g(a) + g(b) &= w(y,x) + w(0,b) \\ &\ge w(y,b) + w(0,x) \\ &= 2g(x). \end{aligned}$$

So the concavity of $w(x,y)$ by our definition implies that of $g(y-x)$ by the usual definition. Conversely assume g is concave and let $i \le i' \le j \le j'$. Then there exists t between 0 and 1 such that $j' - i' = t(j - i') + (1-t)(j' - i)$. Further, $j - i = (j - i') + (j' - i) - (j' - i') = (1-t)(j - i') + t(j' - i)$. So

$$\begin{aligned} w(i,j) + w(i',j') &= g(j-i) + g(j'-i') \\ &\le (tg(j-i') + (1-t)g(j'-i)) + ((1-t)g(j-i') + tg(j'-i)) \\ &= g(j-i') + g(j'-i) \\ &= w(i',j) + w(i,j'), \end{aligned}$$

and the concavity of g implies that of w. Thus our choice of terminology is justified.

The quadrangle inequality was introduced by Monge [54], and revived by Hoffman [29], in connection with a planar transportation problem. More recently, F. Yao [81] considered the following recurrence relations:

$$\begin{aligned} C[i,i] &= 0 \\ C[i,j] &= w(i,j) + \min_{i<k\le j} C[i,k-1] + C[k,j], \text{ for } i < j. \end{aligned}$$

Yao proved that if the weight function w satisfies the quadrangle inequality then the obvious $O(n^3)$ algorithm can be sped up to $O(n^2)$. A corollary of this result is an $O(n^2)$ algorithm for computing optimum binary search trees, an earlier remarkable result of Knuth [36]. However her techniques do not seem to apply to the sequence analysis problems we consider. Recently, the quadrangle inequality has seen use in a number of other dynamic programming algorithms for sequence

analysis [3, 17, 27, 35, 53, 78], and it is this set of algorithms that we improve and generalize, as well as developing new algorithms for the solution of further sequence analysis problems.

2. SEQUENCE ALIGNMENT

The first sequence analysis problem we study is that of sequence alignment. We only consider alignments between pairs of sequences; computation of alignments between three or more sequences has also been studied [4, 8, 9, 22, 63] but the problem of simultaneously aligning more than a small constant number of sequences is NP-complete in general [20, 47].

Sequence alignment seeks to compare the two input sequences, either to compute a measure of similarity between them or to find some core sequence that each shares characteristics from. Sequence alignment is an important tool in a wide variety of scientific applications [65, 74]. In molecular biology, the sequences being compared are proteins or nucleotides. In geology, they represent the stratigraphic structure of core samples. In speech recognition, they are samples of digitized speech.

The first important use of such comparisons is to find a common subsequence or consensus sequence for the two input sequences. For instance, in geology and paleontology, an important discovery was a thin layer of iridium in many core samples from around the world, at a point corresponding to the time at which the dinosaurs became extinct. In molecular biology, a common structure to a set of sequences could lead to an elucidation of the function of those sequences.

A related use of sequence alignment is an application to computer file comparison, made by the widely used *diff* program [6]. Here the input sequences are lines from text files. The program tries to find a large subsequence of the two line sequences, so that the remaining unmatched lines, which are output as the set of differences between the two files, are small in number.

The other use of sequence alignment is, as we have said, to compute a measure of similarity between the sequences. This can be used for instance in molecular

biology, to compute the taxonomy of evolutionary descent of a set of species, or the genetic taxonomy of a set of proteins within a species [14]. It can also be used to group proteins by common structure, which also typically has a high correlation with common function. For other fields of application the uses of such a measure are similar.

2.1. Longest Common Subsequences and Edit Distance Computation

The simplest definition of an alignment is a matching of the symbols of one input sequence against the symbols of the other so that the total number of matched symbols is maximized. In other words, it is the longest common subsequence between the two sequences [7, 26, 30]. However in practice this definition leaves something to be desired. One would like to take into account the process by which the two sequences were transformed one to the other, and match the symbols in the way that is most likely to correspond to this process. For instance, in molecular biology the sequences being matched are either proteins or nucleotides, which share a common genetic ancestor, and which differ from that ancestor by a sequence of mutations. The best alignment is therefore one that could have been formed by the most likely sequence of mutations.

More generally, one can define a small set of *edit operations* on the sequences, each with an associated cost. For molecular biology, these correspond to genetic mutations, for speech recognition they correspond to variation in speech production, and so forth. The alignment of the two sequences is then the set of edit operations taking one sequence to the other, having the minimum cost. The measure of the distance between the sequences is the cost of the best alignment.

The first set of edit operations to be considered consisted of substitions of one symbol for another (point mutations), deletion of a single symbol, and insertion of a single symbol. It turns out that the minimum cost set of edit operations transforming one string x to another string y can be computed by a simple recurrence:

$$C[i,j] = \min\{C[i-1,j-1] + s(i,j), C[i-1,j] + f(i), C[i,j-1] + g(j)\}. \quad (3)$$

Since we will be dealing with generalizations of this recurrence throughout this thesis, in the solution of more general sequence alignment problems, let us explain in a little more detail its meaning and derivation.

In this equation $C[i, j]$ measures the cost of the minimum set of edit operations transforming string x_i to y_j, where x_i consists of the first i symbols of string x and similarly y_j consists of the first j symbols of string y. Clearly if $i = 0$ the only possibility is to insert all the symbols of y_j, so we initialize the first row of the matrix to the insertion cost of each such substring. Similarly for $j = 0$ we use the deletion cost of substring x_i. Otherwise, both $i > 0$ and $j > 0$, and the recurrence above is well defined at position (i, j).

In any set of operations taking x_i to y_j, the last symbol of y_j has to appear somehow. Either it is inserted by an edit operation, or it is carried across or substituted from a symbol in x_i. In the latter case, if it does not come from the last symbol of x_i, then since no operation transposes symbols that last symbol itself must be deleted. Thus, in the minimum cost set of edit operations taking x_i to y_j, there are three possibilities:

(1) The last two positions i and j of each substring are aligned with each other; either they match and no edit operation need take place, or they are unequal and a substitution must be performed; in either case the total cost can be represented as $C[i-1, j-1] + s(i, j)$, where $s(i, j)$ is either zero, for an exact match between the two positions, or the cost of a substitution between the two, otherwise.

(2) The symbol in the last position i of x_i is deleted. Then the total cost is $C[i-1, j] + f(i)$, where f measures this deletion cost.

(3) The symbol in the last position j of y_j is inserted. Then the total cost is $C[i, j-1] + g(j)$, where g measures this insertion cost.

Thus each possible case of a minimum cost is included in the minimization of recurrence 3. Conversely, each choice in the minimization gives rise to a possible set of edit operations. Thus the recurrence correctly computes the minimum cost alignment. If x has length m and y has length n, there will be mn values of the

recurrence to compute, and each takes constant time to compute, so the total time is $O(mn)$.

This algorithm was independently discovered many times by researchers from a variety of fields [13, 24, 45, 55, 58, 59, 61, 69, 70]. Further, one can find the subsequence of the first string having the best alignment with the second string in the same time, by slightly modifying the initial conditions for the first row and column of the matrix.

Extensive study has been made of the alignment problem solved above, and a number of algorithms have been given which improve the above time bound when there are only a small number of edit operations in the minimum cost alignment for the two input sequences [16, 18, 40, 41, 42, 43, 44]. Unfortunately, for the applications we are interested in the number of edit operations is typically some large fixed fraction of the lengths of the input sequences, and so this approach does not yield any improvement in asymptotic time bounds.

If only the cost of the best sequence is desired, the $O(mn)$ dynamic programming algorithm for the sequence alignment problem need take only linear space: if we compute the values of the dynamic programming matrix in order by rows, we need only store the values in the single row above the one in which we are performing the computation. However if the sequence itself is desired, it would seem that we need $O(mn)$ space, to store pointers for each cell in the matrix to the sequence leading to that cell. Hirschberg [25] showed that less storage was required, by giving an ingenious algorithm that computes the edit sequence as well as the alignment cost, in space $O(n)$, and remaining within the time bound of $O(mn)$.

Approaches other than dynamic programming are also possible. Masek and Paterson [50] have designed an algorithm that can be thought of as a finite automaton of $O(n)$ states, which finds the edit distance between the pattern and the strings in the database in $O(mn/\log n)$ time. This time bound arises from the assumption

that each word of memory can hold $O(\log n)$ bits of information (as it must, to be able to store a pointer into the input strings).

Further speed-ups are also possible using this automata-theoretic approach; in particular one can construct a finite automaton for a string x that determines the minimum cost edit sequence between x and substrings of a string y input to the automaton. Such an automaton would then run in linear time. However this automaton has an exponential number of states; in particular each state can be equated with a possible column of the matrix solving recurrence 3. Thus the linear time bound arises from loopholes in the theoretical model of computation, and gives no improvement in practice. In contrast, the algorithms in this thesis are true improvements, and do not take advantage of such tricks.

2.2. Generalized Edit Operations and Gap Costs

Above we discussed sequence alignment with the operations of point substitution, single symbol insertion, and single symbol deletion. A number of further extensions to this set of operations have been considered. If one adds an operation that transposes two adjacent symbols, the problem becomes NP-complete [20, 71]. If one is allowed to take circular permutations of the two input sequences, before performing a sequence of substitutions, insertions, and deletions, the problem can be solved in time $O(nm \log n)$ [33].

Another important extension to the set of operations is the following. We call a consecutive set of deleted symbols in one sequence, or inserted symbols in the other sequence, a *gap*. With the operations and costs above, the cost of a gap is the sum of the costs of the individual insertions or deletions which compose it. However, in molecular biology for example, it is much more likely that a gap came about through one mutation that deleted all the symbols in the gap, than that many individual mutations combined to create the gap. Similar motivations apply to other applications of sequence alignment. Therefore we would like to allow gap insertions or deletions to combine many individual symbol insertions or deletions, with the cost of a gap insertion or deletion being some function of the length of the gap.

Because of the motivation above, the most likely choices for gap costs are convex functions of the gap lengths. With such functions the cost of a long gap will be less than the sums of the costs of any partition of the gap into smaller gaps, so that as desired the best alignment will treat each gap as a unit. However it is also natural to consider other classes of gap cost function. We call the sequence alignment problem with gap insertions and deletions the *gap sequence alignment problem*, and similarly we name special cases of this problem by the class of cost

functions considered, e.g. the *convex sequence alignment problem*, etc.

Experimental results by Fitch and Smith [15] indicate that, for biological sequence alignment, the cost of a gap may depend on its endpoints (or location) and on its length. Thus, if we denote the cost of a potential gap between symbols i and j by $w(i, j)$, we have

$$w(i, j) = f_1(i) + f_2(j) + g(j - i) \tag{4}$$

for some functions f_1, f_2, and g.

As we have said, the original sequence alignment problem treats the cost of a gap as the sum of the costs of the individual symbol insertions or deletions of which it is composed. Therefore if the gap cost function is some constant times the length of the gap, the gap sequence alignment problem can be solved in time $O(mn)$. This can be generalized to a slightly wider class of functions, the *linear* or *affine* gap cost functions. These functions satisfy equation 4, with $g(j - i) = c \cdot (j - i)$ for some constant c. A simple modification of the solution to the original sequence alignment problem also solves the linear sequence alignment problem in time $O(mn)$ [21].

The gap sequence alignment problem can again be solved by a simple dynamic programming algorithm [65]. The algorithm is similar to that for the original sequence alignment problem; however where in recurrence 3 we needed only consider insertions and deletions of the final position of each substring, here we must allow insertions and deletions of arbitrary lengths. Thus the recurrence for solving this problem becomes

$$C[i, j] = \min\{C[i - 1, j - 1] + s(i, j), F[i, j], G[i, j]\}, \tag{5}$$

where

$$F[i, j] = \min_{0 \le \ell < i} C[\ell, j] + w(\ell, i) \tag{6}$$

$$G[i, j] = \min_{0 \le \ell < j} C[i, \ell] + w'(\ell, j). \tag{7}$$

Here $C[i,j]$ as before calculates the best alignment between substrings x_i and y_j. But now instead of a single expression for a possible deletion, we have recurrence 6, which calculates the best alignment ending in a deletion of some terminal substring of x_i. And similarly recurrence 7 calculates the best alignment ending in an insertion of some terminal substring of y_j. The functions w and w' compute the cost of each such deletion or insertion respectively.

Thus the computation of each entry in the dynamic programming matrix depends on all the previous entries in the same row or column, rather than simply on the adjacent entries in the matrix. The time bound for solving the recurrence is then $O(mn\max(m,n))$. This method was discovered by Waterman et al. [76], based on earlier work by Sellers [66]. But this time bound is an order of magnitude more than that for non-gap sequence alignment, and thus is useful only for much shorter sequences.

Galil and Giancarlo [17] noted that recurrence 6 for fixed j, and recurrence 7 for fixed i, can be expressed in a form which generalizes that of the *least weight subsequence* problem, which had previously been applied to text formatting [38, 27] and optimal layout of B-trees [27]:

$$E[j] = \min_{0 \le i < j} D[i] + w(i,j) \tag{8}$$

Here $D[i]$ depends in some simple way on the corresponding value $E[i]$. In the sequence alignment computation, for example in row r of recurrence 7, $D[i] = C[r,i]$ and $G[r,i] = E[i]$; thus $D[i]$ is the minimum of $E[i]$ and two other values; in the previously considered least weight subsequence problem $D[i]$ is taken to be equal to $E[i]$.

The computation of recurrence 5 involves the solution to $O(n+m)$ such problems, one for each row and column of the matrix. The *cost function* w is the cost of an insertion or a deletion, depending on which of equations 6 or 7 we are using recurrence 8 to solve.

In fact, for gap cost functions of the form given by equation 4, the recurrence above can be simplified to

$$E[j] = \min_{0 \le i < j} D[i] + g(j - i). \tag{9}$$

In this new form, the $f_1(i)$ and $f_2(j)$ components of the cost function $w(i, j)$ are hidden in the two arrays D and E, and in the computation by which $E[i]$ is computed from $D[i]$. More explicitly, if we are calculating for example recurrence 7 for row r, using recurrence 9, then $G[r, i] = E[i] + f_2(i)$ and $D[i] = C[r, i] + f_1(i)$.

If we could speed up the solution to recurrence 8, or its specialization recurrence 9, we would then have a corresponding speedup in the solution of recurrence 5, and thus in the computation of sequence alignment with gaps.

Without further assumption we cannot take less than $O(n^2)$ time to compute n values of recurrence 8, so there is no such speed up. Indeed, we must examine all possible values of $w(k, i)$. However if we make assumptions about the nature of w we may be able to avoid this examination, and speed up the computation of the recurrence. Galil and Giancarlo [17] considered the gap sequence alignment problem for both convex and concave cost functions. They gave an algorithm for solving recurrence 8 with such functions in time $O(n \log n)$. For many simple convex and concave functions, a binary search step in their algorithm can be eliminated, resulting in a linear time solution to the recurrence. As a result they solved both the convex and concave sequence alignment problems in time $O(mn \log n))$, or $O(mn)$ for many simple functions. Miller and Myers [53] independently solved the same problem in similar time bounds.

Wilber [78] pointed out a resemblance between the least weight subsequence problem and a matrix searching technique that had been previously used to solve a number of problems in computational geometry [2]. He used this technique in an algorithm for solving the least weight subsequence problem in linear time. His

algorithm also extends to the generalization of the least weight subsequence problem expressed in recurrence 8 above. However in its application to sequence alignment, we require the simultaneous solution to many copies of the recurrence, with the value of $D[k]$ in each copy depending on the partial solutions of other copies. For such interleaved computations, Wilber's analysis breaks down and his algorithm can not be used for concave sequence alignment.

Klawe and Kleitman [35] studied a similar problem for the convex case; they gave an $O(n\alpha(n))$ algorithm, also using matrix searching, which can be used in an $O(mn\alpha(n))$ algorithm for the convex sequence alignment problem. Here α is a very slowly growing function, the functional inverse of the Ackermann function. Klawe [34] also gave a simpler algorithm, again based on matrix searching, which solves both the convex and concave problems (although she only mentions the convex case); this algorithm takes the somewhat worse time bound of $O(n \log^* n)$, where $\log^* n = \min\{i : \log^{(i)} n \leq 2\}$. All of these matrix searching algorithms, while theoretically very interesting, have less importance for practical solution to sequence alignment problems, because the constant factors in their time bounds are very large.

2.3. Sparse Sequence Alignment

All of the alignment algorithms above take a time which is at least the product of the lengths of the two input sequences. This is not a big problem when the sequences are relatively short, but the sequences used in molecular biology can be very long, and for such sequences these algorithms can take more time than the computing power available for their computation.

Wilbur and Lipman [79, 80] proposed a method for speeding these computations up, at the cost of a small loss of accuracy, by only considering matchings between certain subsequences of the two input sequences. Wilbur and Lipman's algorithm first selects a small number of *fragments*, where each fragment is a pair of substrings, one from each input string, having exactly the same sequence of symbols. Denote by M the number of fragments. The algorithm chooses all possible fragments with a certain fixed length; this can be done in time $O(n+m+M)$ using standard string matching techniques.

An alignment is then defined to be a sequence of fragments, with the following properties. First define the *diagonal* of a fragment to be the difference between the positions its two substrings occupy in the input strings. If adjacent fragments in the alignment have the same diagonal, we call that pair of fragments a *mismatch*; the substrings of the second fragment are required to start after the substrings of the first fragment in the respective input strings. Other pairs of fragments are called *gaps*; the substrings of the second fragment are required to start after the ends of the corresponding substrings of the first fragment. Thus the substrings of mismatched fragments may overlap, but the substrings of gaps may not.

The cost of a mismatch is some function of the distance between the starts of the substrings. This corresponds to substitutions in the original alignment problem. The cost of a gap is a similar function, plus a function of the distance between the

diagonals of the two fragments. This distance is the *length* of the gap, and this second function corresponds to the gap length cost function of the gap sequence alignment problem.

We call the alignment problem defined as above the *fragment alignment* problem, and as before also include in the name of the problem any restriction on the class of gap cost functions allowed. It would also be possible to consider general classes of mismatch cost functions; however we will assume that the mismatch function is always linear. If the fragments are taken to be single symbols, the problem will be exactly the gap sequence alignment problem; if the number of matching symbols is small this representation of the problem may lead to a more efficient solution than the non-sparse algorithms. If the fragments are taken to have lengths greater than 1, the fragment alignment approximates the gap alignment problem, but M will become smaller and so the solution time will be decreased.

Wilbur and Lipman [79, 80] used a simple dynamic programming algorithm to solve the fragment alignment problem in time $O(n + m + M^2)$. They first selects a small number of *fragments*, where each fragment is a triple (i, j, k) such that the k-tuple of symbols at positions i and j of the two strings exactly match each other; that is, $x_i = y_j$, $x_{i+1} = y_{j+1}$, ..., $x_{i+k-1} = y_{j+k-1}$. For instance, we might choose all possible matching fragments with some fixed length k; such a set of fragments can be found in time $O(n+m+M)$ using standard string matching techniques (see [12] for details).

A fragment (i', j', k') is said to be *below* (i, j, k) if $i + k \leq i'$ and $j + k \leq j'$; i.e. the substrings in fragment (i', j', k') appear strictly after those of (i, j, k) in the input strings. Equivalently we say that (i, j, k) is *above* (i', j', k'). The *length* of fragment (i, j, k) is the number k. The *forward diagonal* of a fragment (i, j, k) is the number $j - i$, and the *back diagonal* is $i + j$.

An *alignment* of fragments is defined to be a sequence of fragments such that,

if (i, j, k) and (i', j', k') are adjacent fragments in the sequence, either (i', j', k') is below (i, j, k) on a different forward diagonal (a *gap*), or the two fragments are on the same forward diagonal, with $i' > i$ (a *mismatch*). The cost of an alignment is taken to be the sum of the costs of the gaps, minus the number of matched symbols in the fragments. The number of matched symbols may not necessarily be the sum of the fragment lengths, because two mismatched fragments may overlap. Nevertheless it is easily computed as the sum of fragment lengths minus the overlap lengths of mismatched fragment pairs. The cost of a gap is some function of the distance between forward diagonals $g(|(j - i) - (j' - i')|)$.

When the fragments are all of length 1, and are taken to be all pairs of matching symbols from the two strings, these definitions coincide with the usual definitions of sequence alignments. When the fragments are fewer, and with longer lengths, the fragment alignment will typically approximate fairly closely the usual sequence alignments, but the cost of computing such an alignment may be much less.

The method given by Wilbur and Lipman [79, 80] for computing the least cost alignment of a set of fragments is as follows. Given two fragments, at most one will be able to appear after the other in any alignment, and this relation of possible dependence is transitive; therefore it is a partial order. They process fragments in the order of any topological sorting of this order. Some such orders are by rows (i), columns (j), or back diagonals $(i + j)$.

For each fragment, the best alignment ending at that fragment is taken as the minimum, over each previous fragment, of the cost for the best alignment up to that previous fragment together with the gap or mismatch cost from that previous fragment. The mismatch cost is simply the length of the overlap between two mismatched fragments; if we are computing the alignment for fragment (i, j, k) and the previous fragment is $(i - x, j - x, k')$ then this length can be computed as $\max(0, k' - x)$. From this minimum cost we also subtract the length of the new

fragment; thus the total cost of any alignment includes a term linear in the total number of symbols aligned. Formally, we have

$$C(i,j,k) = -k + \min \begin{cases} \min\limits_{(i-x,j-x,k')} C(i-x, j-x, k') + \max(0, k'-x) \\ \min\limits_{(i',j',k') \text{ above } (i,j,k)} C(i',j',k') + g(|(j-i)-(j'-i')|) \end{cases} \tag{10}$$

The naive dynamic programming algorithm for this computation, given by Wilbur and Lipman [79, 80], takes time $O(M^2)$. If M is sufficiently small, this will be faster than many other sequence alignment techniques. However, M may possibly be as large as n^2, in which case this technique would be significantly worse than the non-sparse dynamic programming technique.

The *fastp* program [46], based on Wilbur and Lipman's algorithm, is in daily use by molecular biologists, and improvements to the algorithm are likely to be of great practical importance. Eppstein et al. [12] studied the problem for linear gap cost functions, and gave an $O(n + m + M \log\log \min(M, nm/M))$ algorithm. (To avoid problems when nm is close to M, we define $\log x$ to be $\log_2(2 + x)$ here and throughout this thesis; thus $\log x \geq 1$ always.) This time bound degrades gracefully, in that when M approaches nm the bound remains smaller than $O(mn)$.

A related set of results have been discovered for the longest common subsequence problem. This can be considered a special case of sequence alignment, in which the cost of point substitution operations is made prohibitively high. Therefore it can also be solved with the Wilbur-Lipman sparse sequence alignment technique, by letting the set of fragments be simply the single-symbol matches between the two input sequences. However the problem is simpler than the general sparse sequence alignment problem. Hunt and Szymanski showed that, if the number of matches M is small enough, the use of sparsity could lead to an efficient longest common subsequence algorithm. Their technique takes time $O(M \log n + n \log \sigma)$, where σ is the size of the input alphabet, and without loss of generality $n \geq m \geq \sigma - 1$. For

biological sequences $\sigma = 4$ and the $\log \sigma$ term vanishes.

Apostolico and Guerra [7] showed that the longest common subsequence problem can be made even more sparse, by only considering *dominant matches* [26]; they reduced the time bound to $O(d \log(nm/d) + n \log \sigma + m \log n)$. Different versions of their algorithm instead take time $O(M \log \log n + n \log \sigma)$ or $O(d \log \log n + n\sigma)$. Eppstein et al. [12] improved these bounds to $O(d \log \log \min(d, nm/d) + n \log \sigma)$.

2.4. New Results

We present a number of new algorithms for sequence alignment with gaps.

First, recall that, as described above, Wilber [78] gave a linear time algorithm for the concave least weight subsequence problem which extends to a linear time solution of recurrence 8. However, we noted that this solution is unsuitable for the application of the recurrence to sequence alignment. Thus the fastest known computation of sequence alignment with concave gap costs took time $O(mn \log^* n)$, using an algorithm of Klawe [34]. In chapter 4, we show how the concave case of recurrence 8 may be solved in linear time, as in Wilber's algorithm, but in a form suitable for use in sequence alignment. Thus we can find a minimum cost alignment with concave gap costs in time $O(nm)$.

Next, in chapter 7, we consider gap cost functions $w(x, y)$ in the form of equation 4, and for which the gap length term $g(y-x)$ can be broken into s intervals, such that in each interval the function is either convex or concave. We call such functions *mixed convex and concave*. We solve the mixed sequence alignment problem in time $O(nms\alpha(n/s))$. No previous solution was better than the $O(mn \max(m, n))$ algorithm for arbitrary cost functions; the new time bound is never worse than this, and when $s < n$ it will be substantially better.

Finally, in chapter 8, we solve the the concave fragment alignment problem in time $O(n+M \log M)$, and the convex problem in time $O(n+M \log M\alpha(M))$. These time bounds greatly improve the previous best known $O(M^2)$ algorithm of Wilbur and Lipman [79, 80]. As stated above, a similar improvement had been found for linear cost functions [12]; however the algorithms achieving the new bounds are quite different from those for the linear case.

3. RNA STRUCTURE

RNA molecules are among the primary constituents of living matter. RNA is used by cells to transport genetic information between the DNA repository in the nucleus of the cell and the ribosomes which construct proteins from that information. It is also used within the process of protein construction, and may also have other important functions.

An RNA molecule is a polymer of nucleic acids, each of which may be any of four possible choices: adenine, cytosine, guanine, and uracil. Thus an RNA molecule can be represented as a string over an alphabet of four symbols, corresponding to the four possible nucleic acid bases. In practice the alphabet may need to be somewhat larger, because of the sporadic appearance of certain other bases in the RNA sequence. This string or sequence information is known as the *primary structure* of the RNA. The primary structure of an RNA molecule can be determined by gene sequencing experiments.

However, in an actual RNA molecule, hydrogen bonding will cause further linkages will form between pairs of bases. Adenine typically pairs with uracil, and cytosine with guanine. Other pairings, in particular between guanine and uracil, may form, but they are much more rare. Each base in the RNA sequence will pair with at most one other base. Paired bases may come from positions of the RNA molecule that are far apart in the primary structure. The set of linkages between bases for a given RNA molecule is known as its *secondary structure*.

The *tertiary structure* of an RNA molecule consists of the relative physical locations in space of each of its constituent atoms, and thus also the overall shape of the molecule. The tertiary structure is determined by energetic (static) considerations involving the bonds between atoms and the angles between bonds, as well as kinematic (dynamic) considerations involving the thermal motion of atoms. Thus the

tertiary structure may change over time; however for a given RNA molecule there will typically be a single structure that closely approximates the tertiary structure throughout its changes.

The tertiary structure of an RNA molecule determines how the molecule will react with other molecules in its environment, and how in turn other molecules will react with it. Thus the tertiary structure controls enzymatic activity of RNA molecules as well as the splicing operations that take place between the time RNA is copied from the parent DNA molecule and the time that it is used as a blueprint for the construction of proteins.

Because of the importance of tertiary structure, and its close relation to molecular function, molecular biologists would like to be able to determine the tertiary structure of a given RNA molecule. Tertiary structures can be determined experimentally, but this requires complex crystalization and X-ray crystalography experiments, which are much more difficult than simply determining the sequence information of an RNA molecule. Further, the only known computational techniques for determining tertiary structure from primary structure involve simulations of molecular dynamics, which require enormous amounts of computing power and therefore can only be applied to very short sequences.

Because of the difficulty in computing tertiary structures, biologists have resorted to the simpler computation of secondary structure, which also gives some information about the physical shape of the RNA molecule. Secondary structure computations also have their own applications: by comparing the secondary structures of two molecules with similar function one can determine how the function depends on the structure. In turn, a known or conjectured similarity in the secondary structures of two sequences can lead to more accurate computation of the structures themselves, of possible alignments between the sequences, and also of alignments between the structures of the sequences [62].

3.1. Secondary Structure Assumptions and the Structure Tree

A perfectly accurate computation of RNA secondary structure would have to also include a computation of tertiary structure, because the secondary structure is determined by the tertiary structure. As we have said this seems to be a hard problem. Instead, a number of assumptions have been made about the nature of the structure. An energy is assigned to each possible configuration allowed by the assumptions, and the predicted secondary structure is the one having the minimum energy.

The possible base pairs in the structure are usually taken to be simply those allowed by the possible hydrogen bonds among the four RNA bases; that is, a base pair is a pair of positions (i, j) where the bases at the positions are adenine and uracil, cytosine and guanine, or possibly guanine and uracil. We write the bases in order by their positions in the RNA sequence; i.e. if (i, j) is a possible base pair, then $i < j$. Each pair has a binding energy determined by the bases making up the pair.

Define the *loop* of a base pair (i, j) to be the set of bases in the sequence between i and j. The primary assumption of RNA secondary structure computation is that no two loops cross. In other words, if (i, j) and (i', j') are base pairs formed in the secondary structure, and some base k is contained in both loops, then either i' and j' are also contained in loop (i, j), or alternately i and j are both contained in loop (i', j'). This assumption is not entirely correct for all RNA [49], but it works well for a great majority of the RNA molecules found in nature.

A base at position k is *exposed* in loop (i, j) if k is in the loop, and k is not in any loop (i', j') with i' and j' also in loop (i, j). Because of the non-crossing assumption, each base can be exposed in at most one loop. We say that (i', j') is a *subloop* of (i, j) if both i' and j' are exposed in (i, j); if either i' or j' is exposed then by the

Figure 3.1.1. Secondary structure for $\mathrm{tRNA}_{\mathrm{F}}^{\mathrm{Met}}$ of *Anacystis nidulans*, after [64].

non-crossing assumption both must be. An example secondary structure fitting the non-crossing assumption is illustrated in figure 3.1.1.

Therefore the set of base pairs in a secondary structure, together with the subloop relation, forms a forest of trees. Each root of the tree is a loop that is not a subloop of any other loop, and each interior node of the tree is a loop that has some other subloop within it. A number of names have been given to the various possible configurations of loops and subloops [64]. We define a *hairpin* to be a loop with no subloops, that is, a leaf in the loop forest, and we define a *single loop* or *interior loop* to be a loop with exactly one subloop. Any other loop is called a *multiple loop*. A single loop such that both pairs of the subloop are adjacent to the pairs of the outer loop, so that only the pairs of the subloop are exposed in the outer loop, is

called a *stacked pair*. A single loop such that one base of the subloop is adjacent to a base of the outer loop is called a *bulge*. Some of these configurations can be seen in the figure, which contains four regions of stacked pairs, three of them containing a hairpin, and the fourth containing a multiple loop with three subloops.

As we have said, each base pair in an RNA secondary structure has a binding energy which is a function of the bases in the pair. We also include in the total energy of the secondary structure a *loop cost*, which is a function of the length of the loop. This length is simply the number of exposed bases in the loop. The loop cost may also depend on the type of the loop; in particular it may differ for hairpins, stacked pairs, bulges, single loops, and multiple loops.

With these definitions one can easily compute the total energy of a structure, as the sum of the base pair binding energies and loop costs. The optimal RNA secondary structure is then that structure minimizing the total energy.

3.2. Computation of Secondary Structure

With the definitions above, the optimum secondary structure can be computed by a three-dimensional dynamic program with matrix entries for each triple (i, j, k), where i and j are positions in the RNA sequence (not necessarily forming a base pair) and k is the number of exposed bases in a possible loop containing i and j. This computation takes time $O(n^4)$; the algorithm was discovered by Waterman and Smith [75].

Clearly, this time bound is so large that the computation of RNA structure using this algorithm is feasible only for very short sequences. Therefore, one needs further assumptions about the possible structures, or about the energy functions determining the optimum structure, in order to perform secondary structure computatation in a more reasonable time bound.

A particularly simple assumption is that the energy cost of a loop is zero or a constant, so that one need only consider the energy contribution of the base pairs in the structure. Nussinov et al. [56] showed how to compute a structure maximizing the total number of base pairs, in time $O(n^3)$; this algorithm was later extended to allow arbitrary binding energies for base pairs, while keeping the same time bound [57].

A more natural assumption is that the cost of a loop is a linear function of its length, rather than being completely arbitrary. This can again be solved in time $O(n^3)$, using a somewhat more complicated dynamic programming technique [64].

Instead of restricting the possible loop cost functions, one could restrict the possible types of loops. In particular, an important special case of RNA secondary structure computation is the computation of the best structure with no multiple loops. Such structures can be useful for the same applications as the more general RNA structure computation. Single loop RNA structures could be used to con-

struct a small number of pieces of a structure which could then be combined to find a structure having multiple loops; in this case one sacrifices optimality of the resulting multiple loop structure for efficiency of the structure computation. Also, the recurrences for computing more general RNA structures include terms for possible single loops, so speeding up single loop structure minimization would also speed up those other computations, at least by a constant factor.

The single loop secondary structure computation can again be expressed as a dynamic programming recurrence relation [73, 64]. We will be examining this recurrence throughout this thesis, so we present it here:

$$C[p,q] = \min_{p<p'<q'<q} G[p',q'] + g((p'-p)+(q-q')) \tag{11}$$

In this recurrence, p and q are a possible base pair for which we want to compute the minimum energy single loop. Let n be the length of the input RNA sequence; then both C and G are $n \times n$ matrices, for which we want to compute the values using the above recurrence. The total energy contribution of this loop will be stored in $G[p,q]$. The value of $C[p,q]$ will be the minimum over all possible subloops (p',q') of the subloop energy, together with the term $g((p'-p)+(q-q'))$ giving the cost of the loop itself as a function of the length. Then $G[p,q]$ will be calculated from $C[p,q]$ by including the binding energy given by pairing p and q; if the bases are unable to pair then this will be reflected by giving them an infinite binding energy. $G[p,q]$ also includes a minimization over simpler possible structures pairing p and q, including hairpins and bulges. For RNA structure computations which include the possibility of multiple loops, $C[p,q]$ will still be computed as above, and the terms for possible multiple loop structures will be minimized into $G[p,q]$.

To simplify the presentation of our algorithms, we will deal with recurrence 11 in a slightly modified form, after a change of variables. In particular, let $E[i,j] = C[n-i+1,j]$ and $D[i,j] = G[n-i+1,j]$. Further let $w(x,y) = g(y-x)$. Then

recurrence 11 becomes

$$E[i,j] = \min_{\substack{0 \le i' < i \\ 0 \le j' < j}} D[i',j'] + w(i'+j', i+j). \tag{12}$$

There is one restriction on the minimization of recurrence 11 missing here, which is that $p' < q'$; but this can be taken care of by letting $G[i,j] = +\infty$ when $i+j > n+1$.

A naive algorithm for solving this relation would seem to require time $O(n^4)$, as for the multiple loop RNA structure computation, but the space requirement is reduced from $O(n^3)$ to $O(n^2)$. In fact the time for solving the recurrence can also be reduced, to $O(n^3)$, as was recently shown by Waterman and Smith [75]. But this is still more time than one would want to spend for the computation. To achieve even faster single loop secondary structure computation, we can again restrict our attention to linear cost functions. With this restriction, the time for the RNA structure computation can be reduced to $O(n^2)$ [32].

For both the multiple loop and single loop RNA structure computation, it seems reasonable to consider a class of loop energy cost functions that is not completely general, and yet is somewhat wider than the class of linear functions. In particular, concave and convex functions are a natural choice, and seem to provide a good fit with the actual physical energies of the RNA molecules. However, no algorithm was known that sped up RNA structure computation, either for single loops or for multiple loops, by assuming convexity or concavity.

3.3. Sparseness in Secondary Structure

The recurrence relations that have been defined for the computation of RNA structure are all indexed by pairs of positions in the RNA sequence (and possibly also by numbers of exposed bases). For many of these recurrences, the entries in the associated dynamic programming matrix include a term for the binding energy of the corresponding base pair. If the given pair of positions do not form a base pair, this term is undefined, and the value of the cell in the matrix must be taken to be $+\infty$ so that the minimum energies computed for the other cells of the matrix do not depend on that value, and so that in turn no computed secondary structure includes a forbidden base pair.

Further, for the energy functions that are typically used, the energy cost of a loop will be more than the energy benefit of a base pair, so base pairs will not have sufficiently negative energy to form unless they are stacked without gaps at a height of three or more. Thus we could ignore base pairs that can not be so stacked, or equivalently assume that their binding energy is again $+\infty$, without changing the optimum secondary structure. This observation is similar to that of sparse sequence alignment, in which we only include pairs of matching symbols when they are part of a longer substring match.

These factors combine to greatly reduce the number of possible pairs, which we denote M, to a value much less than the upper bound of n^2. If we required base pairs to form even higher stacks, this number would be further reduced. The computation and minimization in this case is taken only over positions (i, j) which can combine to form a base pair. Such problems can still be solved with the algorithms listed above, by giving a value of $+\infty$ to $D[i, j]$ at the missing positions. However, the complexity measures of the previous algorithms for the problem do not depend on the number of possible base pairs, but only on the length of the input sequence.

Thus a further way of speeding up algorithms for the computation of RNA secondary structure would be to take more careful advantage of the existence of the missing positions, rather than simply working around them. Algorithms using this approach would take time proportional to some function of the number of possible base pairs, rather than the possibly much greater number of pairs of positions in the sequence.

Concurrently with the work in this thesis, Eppstein et al. [12] showed that the sparse single loop RNA structure problem for linear loop energy cost functions could be solved in time $O(n + M \log\log \min(M, n^2/M))$. When M is close to n^2 this time bound degenerates to the $O(n^2)$ bound for the non-sparse algorithm of Kanehise and Goad [32]; when M is less than n^2 the sparse algorithm can provide a significant improvement to the previous bounds. No other such sparse RNA structure computation algorithms were previously known.

3.4. New Results

First, in chapter 5, we give a data structure for solving a dynamic minimization problem resembling the generalized least weight subsequence problem described in chapter 2. This data structure will prove useful in several of our RNA structure computation algorithms.

Next, in chapter 6, we show how to compute single loop RNA secondary structure, for convex or concave energy costs, in time $O(n^2 \log^2 n)$. For many simple cost functions, such as logarithms and square roots, we show how to improve this time bound to $O(n^2 \log n \log\log n)$. The best previous algorithm, due to Waterman and Smith [75], took $O(n^3)$ time. These results have recently been improved by Aggarwal and Park [3], who gave an $O(n^2 \log n)$ algorithm; however they use matrix searching techniques, which lead to a high constant factor in the time bound, so our algorithms may be better in practice.

Then, in chapter 7, we show how to compute single loop RNA secondary structures with mixed convex and concave costs, in time $O(nms \log n\alpha(n/s))$. Our algorithm for this problem is based on Aggarwal and Park's technique for the convex and concave problems, and again uses matrix searching techniques. A variant of the algorithm, without matrix searching, runs in time $O(nms \log n \log(n/s))$.

Finally, in chapter 9, we show how to solve the sparse convex or concave single loop RNA structure problem in time $O(n + M \log M \log\min(M, n^2/M))$. For many simple functions this time bound can be further improved to $O(n + M \log M \log\log\min(M, n^2/M))$. Both bounds are never worse than those of the best known algorithms for the non-sparse problem; when M is less than n^2 the sparse algorithms may provide a big improvement over the corresponding non-sparse techniques. The time bound resembles that of $O(n+M \log\log\min(M, n^2/M))$ given by Eppstein et al [12] for the sparse problem with linear costs, but the algorithmic

techniques used are quite different. No previous algorithms for sparse computation of RNA structure were known.

4. THE CONCAVE LEAST WEIGHT SUBSEQUENCE PROBLEM

Recall that, in chapter 2, we showed how the problem of aligning two sequences of length n and m, with some gap cost function w, may be reduced to solving $m + n$ copies of the following recurrence:

$$E[j] = \min_{0 \le i < j} D[i] + w(i, j), \tag{13}$$

We will not here make any further assumptions about D, except that each value $D[i]$ can be computed in some simple way once the corresponding value of $E[i]$ is known. The obvious dynamic programming algorithm for this recurrence takes time $O(n^2)$; if we speed up this computation we will achieve a corresponding speedup in the computation of the modified edit distance. We will see in chapter 7 another application of this problem, to both sequence alignment and the computation of RNA secondary structure. We also use the problem again in chapter 8 as part of a sparse sequence alignment algorithm.

Two dual cases of recurrence 13 have previously been studied. In the *concave* case, the gap length cost function w satisfies the *quadrangle inequality*:

$$w(i, j) + w(i', j') \le w(i', j) + w(i, j'), \tag{2}$$

whenever $i \le i' \le j \le j'$. In the *convex* case, the weight function satisfies the inverse quadrangle inequality, found by replacing $\le$ by $\ge$ in equation 2.

For both the convex and the concave cases, good algorithms have recently been developed. Hirschberg and Larmore [27] assumed a restricted quadrangle inequality with $i \le i' < j \le j'$ in inequality 2 that does not imply the inverse triangle inequality. They solved the "least weight subsequence" problem, with $D[j] = E[j]$, in time $O(n \log n)$ and in some special cases in linear time. They used this result to derive improved algorithms for several problems. Their main application is an

$O(n \log n)$ algorithm for breaking a paragraph into lines with a concave penalty function. This problem had been considered by Knuth and Plass [38] with general penalty functions. Galil and Giancarlo [17] discovered algorithms for both the convex and concave cases which take time $O(n \log n)$, or linear time for some special cases. Miller and Myers [53] independently discovered a similar algorithm for the convex case. Aggarwal et al. [2] had previously given an algorithm which solves an offline version of the concave case, in which D does not depend on E, in time $O(n)$; Wilber [78] extended this work to an ingenious $O(n)$ algorithm for the online concave case; however as we shall see Wilber's algorithm has shortcomings that make it inapplicable to the sequence alignment problem. Klawe and Kleitman [35] extended the algorithm of Aggarwal et al. to solve the convex case in time $O(n\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. Klawe [34] also gave a simpler $O(n \log^* n)$ algorithm for the convex case; although she does not mention the fact, her algorithm also provides a solution to the concave case in the same time bound.

In this chapter we concentrate on the concave case of recurrence 13. As we have said, Wilber already gave a linear time algorithm for this case, so one would think that nothing remains to be said. But as we shall see, Wilber's algorithm is unsuited to our applications.

For the reduction given in the introduction from sequence matching to recurrence 13, and also for the applications of this recurrence with mixed convex and concave functions given in chapter 7, we interleave the solutions of a number of separate convex or concave copies of recurrence 13. I.e. many separate such problems will be solved, but the values of $D[i]$ used in the minimization for some of the problems will depend on the values of $E[j]$ computed in other problems. Because of this, we require an additional property of any solutions we use: each value $E[j]$ must be known before the computation of $E[j+1]$ begins.

Instead, Wilber's $O(n)$ time algorithm for the concave case of recurrence 13

guesses a block of values of $E[j]$ at once, then computes the corresponding values of $D[j]$ and verifies from them that the guessed values were correct. If they were incorrect, they and the values of $D[j]$ need to be recomputed, and the work done computing and verifying the incorrect values can be amortized against progress made. But if Wilber's algorithm is interleaved with other computations, and an incorrect guess is made, those other computations based on the incorrect guess will also need to be redone, and this extra work can no longer be amortized away.

Therefore, we now present a modification to Wilber's algorithm that allows us to use it in interleaved computations. Thus we achieve an $O(mn)$ time bound for sequence alignment with concave gap costs; we shall also see further applications of this problem in later chapters.

4.1. Wilber's Algorithm

First let us sketch Wilber's algorithm for the concave least weight subsequence problem. The algorithm depends on the following three facts. First, let $i_1 < i_2 < j_1 < j_2$, and let $P[j] = \min_{i_1 \le i \le i_2} D[i] + w(i,j)$ for $j_1 \le j \le j_2$ and for some concave function w. Then the values of $P[j]$ can be computed in time $O((i_2 - i_1) + (j_2 - j_1))$, using the algorithm of Aggarwal et al. [2]. Second, in recurrence 13, let $i(j)$ be the value of i such that $D[i] + w(i,j)$ supplies the minimum value for $E[j]$ and again let w be concave. Then for $j' > j$, $i(j') \ge i(j)$. And third, if we extend w to be equal to $+\infty$ for (i,j) with $i \ge j$, it remains concave.

The algorithm proceeds as follows. Assume that we know the values of $D[j]$, $E[j]$, and $i(j)$ for $j \le k$. Let $D[j] = d(E[j])$ be the function taking values of $E[j]$ computed in the recurrence to the corresponding values of $D[j]$ used for later values of the recurrence. Let $p = \min(2k - i(k) + 1, n)$. Define a *stage* to be the following sequence of steps, which Wilber repeats until all values are computed:

(1) Compute $P[j] = \min_{i(k) \le i \le k} D[i] + w(i,j)$ for $k < j \le p$ using the algorithm of Aggarwal et al. $P[j]$ may be thought of as a guess at the eventual value of $E[j]$.

(2) Compute $Q[j] = d(P[j])$ for $k < j \le p$. I.e. we perform the computation that would be used to compute the values of $D[j]$, if $E[j]$ were in fact equal to $P[j]$.

(3) For each j with $k < j \le p$, compute $R[j] = \min_{k < i < j} Q[i] + w(i,j)$ using the algorithm of Aggarwal et al. $R[j]$ substitues the values of $Q[j]$ into the recurrence to verify that the guessed values of $P[j]$ were correct.

(4) Let h be the least j such that $R[j] < P[j]$, or p if no such index exists. Then $E[j] = P[j]$ for $k < j \le h$. If $h = p$, we know that all guesses were correct. Otherwise, we know that they were correct only through h; then $E[h+1] = R[h+1]$ and $i(h+1) > k$. In either case, start a new stage with k updated to

reflect the newly verified values of $E[j]$ and $D[j]$.

A proof that the above algorithm in fact correctly computes recurrence 13 is given in Wilber's paper. We now give the time analysis for this algorithm; a similar analysis will be needed for our modification to the algorithm. The total time for each stage is $O((p-k)+(k-i(k))=O(k-i(k))$. If $h=n$ we are done, and this final stage will have taken time $O(n)$. If $h=p\neq n$ then we will have computed $p-k=2k-i(k)+1-k=k-i(k)+1$ new values, so the time taken is matched by the increase in k. And if $h\neq p$, then $i(h+1)>k$ and $i(h+1)-i(k)>k-i(k)$, so the time taken is matched by the increase in $i(k)$. Neither k nor $i(k)$ decrease, and both are at most n, so the algorithm takes linear time.

However as we have seen this may not hold when we interleave its execution with other computation. In particular, the analysis above depends on step 2 of each stage, the computation of $Q[j]=d(P[j])$, taking only constant time for each value computed; but if we interleave several computations this step may take much longer. The problem is that $d(x)$ may not be a simple function depending only on x, but instead it may use the value of x as part of one of the other interleaved dynamic programs; and if we supply $P[j]$ as the value of x instead of the correct value of $E[j]$, this may cause incorrect work to be done in the interleaved programs. This incorrect computation will need to be undone, if the value of $P[j]$ turns out not to equal $E[j]$, and therefore the time taken to perform it can not be amortized against the total correct work performed. Instead we now describe a way of performing each stage in such a way that we only need to compute $d(E[j])$, for actual values of $E[j]$.

4.2. The New Algorithm

We introduce a new variable, c, corresponding to the role of $i(k)$ in Wilber's algorithm, and an array $A[j]$ which stores the already-computed "influence" of $D[i]$, for $i < c$, on future values of $E[j]$. That is, for all j from 1 to n,

$$A[j] = \min_{0 \le i < c} D[i] + w(i, j). \tag{14}$$

Actually, equation 14 will not hold as written above; instead we guarantee the slightly weaker condition that, if index i supplies the minimum of $D[i] + w(i, j)$ in the computation of $E[j]$, then either $i \ge c$ or $E[j] = A[j]$.

Initially $c = 0$ and all values of A are $+\infty$; clearly equation 14 holds for these initial conditions. As in Wilber's algorithm, let k be the greatest index such that $D[k]$ is known; initially $k = 0$. Finally let $p = 2k - c + 1$; c is always at most k so $p > k$. We proceed as follows.

(1) Compute $P[j] = \min(A[j], \min_{c \le i \le k} D[i] + w(i, j))$ for $k < j \le p$ using the algorithm of Aggarwal et al. As in Wilber's algorithm, we compute here our guess at the values of $E[j]$. Wilber's analysis applies here to show that the algorithm of Aggarwal et al. can be used, taking time $O((k - c) + (p - k))$.

(2) For each i with $k < i < p$, compute

$$B[i] = \max_{i < j \le p} P[j] - w(i, j)$$

using the algorithm of Aggarwal et al. Here we differ from Wilber's algorithm; instead of plugging our guesses into the function $d(x)$, we compute the bounds $B[i]$ directly from the guesses.

(3) While $k < p$, increase k by 1, let $E[k] = P[k]$, and compute $D[k] = d(E[k])$. If $k = p$, start the next stage at step 1. If not and $D[k] < B[k]$, stop and go to step 4. Otherwise, continue the loop in this step.

(4) We have found k to be the least index with $D[k] < B[k]$. For $k < j \leq p$, let $A[j] = P[j]$. Set $c = k$, and start a new stage at step 1.

The algorithm can be visualized as in figure 4.2.1. The figure depicts a matrix, with columns numbered by j and rows numbered by i. The value at position (i, j) of the matrix is $D[i] + w(i, j)$. Positions below the diagonal are not used by the algorithm, and no value is defined there. Then the goal of the computation is to compute the minimum value in each column. As in Wilber's algorithm, rows are indexed starting from 0 but column numbers can start from 1, since $D[0]$ is defined and used in the minimization but $E[0]$ is not defined. The values in any row of a matrix are not known until the minimum in the corresponding column has been computed.

At each stage, the minima in all columns up to and including column k have been computed, and so the values in all rows up to k are computable. The contribution of the values in rows above (but not including) row c to the minimization for each column j has been computed into $A[j]$. Step 1 extends this computation of the contribution to include area (1) of the figure, i.e. rows c through k and columns $k + 1$ through p. The remaining steps test the values so computed, to see whether they are the actual minima in their columns. If so, k can be advanced to p. Otherwise, one of the columns in the range from $k+1$ through p has a minimum in a row from k to p, and by concavity none of the values in area (2) of the figure will be the minimum in their columns. So in this case, we have computed the influence between rows c and k, and we can advance c.

More formally, we have the following lemmas.

Lemma 1. If, in the computation of a stage, for some i it is the case that $B[i] \leq D[i]$, and assuming the values of D computed in all previous stages were correct, then for all j, $i < j \leq p$, $D[i] + w(i, j) \geq P[j]$.

Proof: $P[j] - w(i, j) \leq B[i]$ by the computation of B. So if $B[i] \leq D[i]$, then

Figure 4.2.1. State in the computation of recurrence 13.

clearly the desired inequality holds. •

Lemma 2. If, in the computation of a stage, we encounter a row i with $B[i] > D[i]$, and assuming the values of D computed in all previous stages were correct, then there exists a column j with $i < j \leq p$, such that $D[i] + w(i, j) < P[j]$.

Proof: Let j be the column supplying the maximum value of $B[i]$, i.e. $B[i] = P[j] - w(i, j)$. Then $P[j] > D[i] + w(i, j)$. •

Lemma 3. For any j, $A[j] \geq \min_{0 \leq i < j} D[i] + w(i, j)$.

Proof: $A[j]$ is always taken to be a minimum over some such terms, so it can never be smaller than the minimum over all such terms. •

Next we show that $A[j]$ encodes the minimization over rows above row c, so the total minimum is the better of $A[j]$ and the minimimum over later rows.

Lemma 4. Each stage computes the correct values of D and E. Further, after the computation of a given stage, for each index j,

$$\min_{0 \le i < j} D[i] + w(i,j) = \min(A[j], \min_{c \le i < j} D[i] + w(i,j)), \tag{15}$$

Proof: We prove the lemma by an induction on the number of stages; thus we can assume that it held prior to the start of the stage. By k and c here we mean the values held at the end of the stage; let k' denote the value held by k at the start of the stage, and similarly let c' denote the value of c at the start of the stage.

We first prove the assertion that E and D are computed correctly. In a given stage, we compute these values for indices j with $k' < j \le k$. In particular, $E[j] = P[j]$ and $D[j] = d(E[j])$ for those indices. Recall that $P[j]$ was computed as

$$P[j] = \min(A[j], \min_{c \le i \le k'} D[i] + w(i,j)).$$

Further, for $i < k$, $D[i] \ge B[i]$, or else we would have stopped the loop in step 3 earlier. Therefore by lemma 1, for each row i with $k' < i < j$, $P[j] \le D[i] + w(i,j)$, so these additional rows can not affect the minimization for $E[j]$, and by the induction hypothesis of equation 15 $E[j]$ is in fact computed correctly.

Now we show that equation 15 also holds. Clearly if the stage terminates with $k = p$, it remains true, because c and A remain unchanged. Otherwise, $c = k$ is the least row such that $B[c] > D[c]$. By lemma 2, there exists a column j with $c < j \le p$, such that

$$D[c] + w(c,j) < P[j] \le \min_{0 \le i \le c'} D[i] + w(i,j).$$

By lemma 1, for $c' < i < c$, $P[j] \le D[i] + w(i,j)$, so

$$D[c] + w(c,j) < \min_{0 \le i < c} D[i] + w(i,j).$$

But then by concavity, for every $j' > j \geq p$,

$$D[c] + w(c, j') < \min_{0 \leq i < c} D[i] + w(i, j').$$

So

$$\begin{aligned} \min_{0 \leq i < j'} D[i] + w(i, j') &= \min_{c \leq i < j'} D[i] + w(i, j') \\ &= \min(A[j'], \min_{c \leq i < j'} D[i] + w(i, j')), \end{aligned}$$

where the last part of the above equation holds because of lemma 3. For $k < j' \leq p$, the values of $D[i] + w(i, j')$ for $c' \leq i \leq k'$ have already been computed in step 1 and incorporated into $A[j']$ in step 4. And since for each $i < c$, $D[i] \geq B[i]$, we know by lemma 1 that $D[i] + w(i, j') \geq P[j'] = A[j']$ so these rows can not affect the minimum. Therefore the equation is true for all columns. •

It remains to show that the bounds B computed in step 2 can be computed using the algorithm of Aggarwal et al. Recall that B is defined by the recurrence $B[i] = \max_{i < j \leq p} P[j] - w(i, j)$. To hide the dependence of j on i, define $f(i, j)$ to be $P[j] - w(i, j)$ if $i < j$, or $-\infty$ otherwise. Then $B[i] = \max_{k+1 < j \leq p} f(i, j)$. The problem is to find the maxima on the rows of the matrix implicitly determined by the function $f(i, j)$. The algorithm of Aggarwal et al. [2] can do this in time $O(p - k)$. It uses as an assumption that, for any four positions $i < i'$ and $j < j'$, if $f(i, j') \geq f(i, j)$, then $f(i', j') \geq f(i', j)$; i.e. in any submatrix, as we progress down the rows of the submatrices, the row maxima move monotonically to the right. Define a matrix of values having this property to be *totally monotonic*. It is the assumption of monotonicity that we must prove, in order to justify the use of the algorithm of Aggarwal et al.

Lemma 5. Let $f(i, j)$ be defined as above. Then the matrix of values of $f(i, j)$ for $k < i < p$ and $k + 1 < j \leq p$ is totally monotone.

Proof: First, if $(i' \geq j)$, then $f(i',j') \geq f(i',j) = -\infty$ and the conclusion holds. So we may assume that $i < i' < j < j'$. But then

$$\begin{aligned} f(i,j) + f(i',j') &= P[j] + P[j'] - w(i,j) - w(i',j') \\ &\geq P[j] + P[j'] - w(i',j) - w(i,j') \\ &= f(i',j) + f(i,j') \end{aligned}$$

by the quadrangle inequality. So if we assume $f(i,j') \geq f(i,j)$, then for the above inequality to hold $f(i',j') \geq f(i',j)$ and the matrix is monotone. •

A similar proof of the monotonicity of $D[i] + w(i,j)$ (with a definition of monotonicity for column minima instead of row maxima) holds for the use of the algorithm of Aggarwal et al. in computing the values of F in step 1. However that proof was given by Wilber for the analogous step in his algorithm, so we omit it here.

Now that we have determined the correctness of the algorithm, let us determine the amount of time it takes.

Theorem 1. Recurrence 13 can be solved in time $O(n)$, where n is the number of values of E solved for.

Proof:

Let k and c denote the values of the variables at the end of a given stage, with k' and c' holding the values before the stage. The time for the stage is then $O((p-k') + (k'-c')) = O(k'-c')$. If, after the stage, $k = n$, we are done and the stage took time $O(n)$. If the stage finished without finding any $D[i] < B[i]$, then $k = p$ so $k - k' = 2k' - c' + 1 - k' = k' - c' + 1$ and the time spent is balanced by the increase in k. Otherwise, $c - c' = k - c' > k' - c'$ and the time spent is balanced by the increase in c. Both k and c are monotonically increasing and both are bounded by n. Thus as before the new algorithm takes linear time. But note that now we

only calculate the values of $D[j]$ corresponding to actual computed values of $E[j]$; thus the algorithm can be safely interleaved with other computations, without loss of time. •

Corollary 1. The sequence alignment problem with concave gap costs can be solved in time $O(mn)$.

Proof: Recall that, in chapter 2, we showed how the sequence alignment problem can be reduced to m least weight subsequence problems of size n (one for each row of the sequence alignment dynamic programming matrix) together with n problems of size m (for the columns of the matrix). Thus the total time is $O(mn + nm) = O(mn)$. •

4.3. Submatrices of the Least Weight Subsequence Problem

We have seen that recurrence 13, for concave cost functions, can be computed in linear time, even when the computation must be interleaved with other similar computations. As an immediate consequence, the edit distance problem for concave gap costs can be solved in time $O(mn)$. We will also apply our solution to the concave case of the recurrence in chapter 7, where we discuss mixed convex and concave cost functions for both sequence alignment and RNA structure computation.

In chapter 8, we again use the least weight subsequence problem, as part of an algorithm for sparse sequence alignment. In that case we need a solution to what looks like a more general problem; however the problem turns out to be no harder than the least weight subsequence problem. We show the needed identity here.

Recall that the least weight subsequence problem can be thought of in terms of finding the column minima in the matrix of values $D[i]+w(i,j)$. Define a *submatrix* of the problem to be a subset X of the row indices, and another subset Y of the column indices, such that we are only interested in finding, for each column in Y, the minimum among the values in the rows in X. To relate this to the original least weight subsequence problem, we now want to solve the recurrence

$$E[j] = \min_{\substack{i \in X \\ i<j}} D[i] + g(i,j), \tag{16}$$

for only those $j \in Y$. For the original problem we required that each $D[j]$ be computable from $E[j]$; here $E[j]$ may never be computed so we instead require that each $D[j]$ be computable whenever all $E[i]$ are known for $i \leq j$.

Lemma 6. Let X and Y be the rows and columns of a submatrix of a concave least weight subsequence problem. Then the solution of recurrence 16 for X and Y can be solved in time $O(|X| + |Y|)$.

Proof: Let $i_0, i_2, \ldots i_m$ be the sequence of indices appearing in either X or Y; then clearly $m \leq |X| + |Y|$. Let $E'[j] = E[i_j]$, and let $D'[j] = D[i_j]$, or $D'[j] = +\infty$ if $j \notin X$. Finally let $w'(x, y) = w(i_x, i_y)$. Then recurrence 16 can be rewritten

$$E'[j] = \min_{0 \leq i < j} D'[j] + w'(i, j),$$

which is exactly a least weight subsequence problem of size m. Some extra values of $E[j]$ are calculated besides those for $j \in Y$, but this cannot cause any problem. Further, it is not hard to see that w' is concave exactly when w is, because the missing rows and columns cannot affect the quadrangle inequality for the remaining indices. The solution of the reduced problem takes time $O(m)$, as does the reduction from the submatrix to the reduced problem, so the submatrix problem can be solved in time $O(m) = O(|X| + |Y|)$. •

5. A DYNAMIC MINIMIZATION DATA STRUCTURE

We now describe a data structure to solve a minimization with dynamically changing input values. We will later use this data structure in our solutions of the RNA structure computation with convex or concave costs, as well as in the sparse RNA structure problem. The data structure may also have independent interest of its own. We consider the following equation:

$$E[i] = \min_j D[j] + w(i, j). \tag{17}$$

Each of the indices i and j are taken from the set of integers from 0 through some bound n. The minimization for each $E[i]$ depends on all values of $D[j]$, not just those for which $j < i$. The cost function $w(i, j)$ is assumed to be either convex or concave. The values of $D[j]$ will initially be set to $+\infty$. At any time step, one of the following two operations may be performed:

(1) Compute the value of $E[i]$, for some index i, as determined by equation 17 from the present values of $D[j]$.

(2) Decrease the value of $D[j]$, for some index j, to a new value that must be smaller than the previous value but may otherwise be arbitrary.

This problem clearly has a strong resemblance to the least weight subsequence problem considered in the previous chapter. In particular, that problem can be solved by the minimization of this chapter as follows. Initially set all values of $D[j]$ to $+\infty$. Then, for values of i in ascending order from 0 to n, compute $E[i]$ from the minimization, use the computed value of $E[i]$ to compute the value $D[i]$ should have, and set $D[i]$ to that value by decreasing it from its previously infinite value. Thus the least weight subsequence problem could be solved by the methods described in this chapter. However because the dynamic minimization above allows more general sequences of operations, the time per operation will be more than if we were only solving the least weight subsequence problem.

In particular, we will give a data structure for this problem that will take $O(\log n)$ amortized time per operation. For simple cost functions, this time can be reduced to $O(\log \log n)$ amortized time per operation. To contrast this with the least weight subsequence problem, we have seen in chapter 4 that there for the concave case each value can be computed in constant amortized time; and in the convex case the algorithm of Klawe and Kleitman [35] computes each value in $O(\alpha(n))$ amortized time.

However, the dynamic minimization solved here solves a more general class of problems than the least weight subsequence problem. We will show in later chapters how it may be applied to the efficient computation of RNA secondary structure, both with and without sparsity assumptions.

Our algorithm for the general dynamic equation above is similar to the least weight subsequence algorithm of Galil and Giancarlo [17]. However we will later see how these techniques can be combined with the matrix searching algorithms of Aggarwal et al. [2] to provide further reductions in the theoretical time bounds for computation of a sequence of operations.

5.1. Partition into Intervals

We first show that we need only consider concave cost functions; the convex case will turn out to be essentially the same.

Lemma 7. If $w(i,j)$ is convex, then $w'(i,j) = w(i, n-j+1)$ is concave.

Proof: Let $f(j) = n - j + 1$. Then f maps the interval $1 \ldots n$ into itself. If $j < j'$, then clearly $f(j') < f(j)$. Therefore, if the inverse quadrangle inequality holds for $w(i,j)$, the inequality formed by reversing the order of j and j' holds for $w'(i,j) = w(i, f(j))$. But this is the same as the quadrangle inequality for $w'(i,j)$. •

Corollary 2. The dynamic minimization problem defined by equation 17, for convex weight functions $w(i,j)$, can be solved as a concave problem by reversing the order of the second index j.

From now on in this chapter we will assume without loss of generality that $w(i,j)$ is concave. Our algorithm is based on the following fundamental fact:

Lemma 8. For any i, j, and j', with $j < j'$, if $D[j] + w(i,j) \geq D[j'] + w(i,j')$, then for all $i' > i$, $D[j] + w(i',j) \geq D[j'] + w(i,j')$. Conversely, if $D[j] + w(i,j) \leq D[j'] + w(i,j')$, then for all $i' < i$, $D[j] + w(i',j) \leq D[j'] + w(i',j')$.

Proof: By the quadrangle inequality, $w(i,j') + w(i',j) \geq w(i,j) + w(i',j')$. Subtracting $w(i,j') + w(i',j') + D[j'] - D[j]$ from both sides and rearranging gives

$$(D[j] + w(i,j)) - (D[j'] + w(i,j')) \leq (D[j] + w(i',j)) - (D[j'] + w(i',j')).$$

But by assumption $(D[j] + w(i,j)) - (D[j'] + w(i,j'))$ is positive, and therefore $(D[j] + w(i',j)) - (D[j'] + w(i',j'))$ must also be positive and the first statement holds. The proof of the converse statement is similar. •

For specificity, let us break ties in favor of the smaller index. That is, we say that $D[j]$ is better than $D[j']$ at i if either $D[j] + w(i,j) < D[j'] + w(i,j')$, or $j < j'$.

Corollary 3. At any given time, the values of $D[j]$ supplying the minima for the positions of $E[i]$, with ties broken as above, partition the possible indices i into a sequence of intervals. If $j < j'$, if i is in the interval in which $D[j]$ is best, and i' is in the interval in which $D[j']$ is best, then $i < i'$.

Thus our algorithm need simply maintain the interval in which each value $D[j]$ is best, and a search structure of the interval boundaries, in which the interval containing a given point i can be looked up. Such a search structure can be maintained at a cost of $O(\log n)$ per modification or search, using any form of balanced binary trees [5, 37, 67]. If we use the *flat tree* data structure of van Emde Boas [68], this time can be reduced to $O(\log \log n)$. Thus it remains to show how to decrease a given value of $D[j]$, while maintaining the partition above and performing only $O(1)$ search tree operations.

In fact we may need to perform more than $O(1)$ search tree operations when we reduce a value of $D[j]$, because many other values of $D[j']$ may have their corresponding intervals reduced to nothing and thus will need to be removed from the search tree. We avoid that difficulty by, whenever we insert a value of $D[j]$ in the search tree, charging the operation with the time required to later delete it. In this way, each reduction will perform $O(1)$ non-charged search tree operations, and will be charged for $O(1)$ further operations which may occur in the future. The total is $O(1)$ operations per reduction, but the time bounds become amortized over the lifetime of the data structure rather than worst case per operation.

We call an index j into the array $D[j]$ *live* if, for some $E[i]$, $D[j]$ supplies the minimum in equation 17. As well as finding the interval containing a given index i into array E, we also need to search for the first live index before a given index j into array D. This can be done by maintaining another search tree or flat tree containing the live indices.

Let $R[j]$ be the rightmost (greatest) index in the interval corresponding to

index j, and similarly let $L[j]$ be the leftmost index. For brevity, let $C(i, j)$ stand for $D[j] + w(i, j)$.

As in the algorithm of Galil and Giancarlo [17], we make use of a subroutine $border(j, j')$. This will always be called with $j < j'$; it returns the greatest index i such that $C(i, j) \leq C(i, j')$. If no such index exists, it returns 0. For arbitrary cost functions, lemma 8 can be used to derive a binary search routine that finds $border(j, j')$ in time $O(\log n)$. For many functions, $border(j, j')$ can be calculated directly as the root of a functional equation; we say that such a function has the *closest zero property*. Hirschberg and Larmore [27], and later Galil and Giancarlo [17] used this property to derive more efficient algorithms for the problems they solved. Most simple functions that are likely to be seen in practice, such as logarithms and square roots, have the closest zero property.

5.2. The Reduction Algorithm

The steps performed to reduce the value of $D[j]$ are as follows:

```
begin
    repeat
        find j' < j as large as possible with j' live;
        if no such j' exists then L[j] ← 0; break;
        else if C(L[j'], j) < C(L[j'], j) then begin
            L[j] ← L[j'];
            make j' no longer live;
        end;
    end;
    if j' still exists then begin
        L[j] ← border(j', j);
        R[j'] ← L[j] − 1;
    end;

    repeat
        find j' > j as small as possible with j' live;
        if no such j' exists then R[j] ← n; break;
        else if C(R[j'], j) < C(R[j'], j) then begin
            R[j] ← R[j'];
            make j' no longer live;
        end;
    end;
    if j' still exists then begin
        R[j'] ← border(j, j');
        L[j] ← R[j'] + 1;
    end;
    if L[j] ≤ R[j] and j is not in the search structure then
        add j to the search structure;
end
```

Clearly a change in $D[j]$ can only affect the borders between it and its neighbors in the interval partition, and not any of the borders between unchanged values. Each iteration of the two loops above removes a point j' from the set of live points, exactly when the decrease in $D[j]$ expands the corresponding interval to cover the

remaining interval of j'; that is, when j' no longer supplies the minimum at any point. The remaining steps fix the borders of the intervals between j and any remaining neighbors. It can be seen, using lemma 8, that the resulting partition is exactly that described by corollary 3. Thus the algorithm correctly solves the dynamic minimization problem we are interested in.

Theorem 2. The data structure above can be implemented to take $O(\log n)$ time, or $O(\log \log n)$ for functions with the closest zero property.

Proof: The time for each reduction can be split into the time per iteration of the loops, and the remaining time. The loop time for an iteration deleting point j', as we have said, will be charged when we insert j' rather than when we delete it. This time is one search tree operation per iteration. Thus the time possibly charged to j will be one search tree operation. The remaining time consists of at most 4 search tree operations, to remove the old interval boundaries from the search tree and insert the new ones, and possibly to add j to the list of live indices. We also make two calls to the *border* subroutine. The total amortized time per operation is $O(\log n)$, or for simple functions $O(\log \log n)$. •

5.3. Homogeneous Sequences of Operations

We have shown how to solve our dynamic minimization problem at a cost of $O(\log n)$ steps per operation, or even $O(\log \log n)$ for simple functions. We do not know how to reduce these bounds further. However, we can achieve better time bounds for a sequence of operations, when all operations are of the same type (either reductions of values of D, or computations of values of E). We call such a sequence of operations *homogeneous*. Then as we shall show, k homogeneous operations can be performed in time $O(k \log n/k)$. For example, n such operations would take time $O(n \log n/n) = O(n)$, so the amortized time per operation would be reduced to a constant. If the cost function is simple, the time for a homogeneous sequence of k operations can be reduced to $O(k \log \log n/k)$. We first consider the simple case.

In the data structure of the previous section, in place of the flat trees of van Emde Boas, we use Johnson's improved flat trees [31]. This is again a structure in which one can insert, delete, and search for points numbered from 0 to n. However, whereas flat trees take time $O(\log \log n)$ per operation, improved flat trees take time $O(\log \log G)$, where G is the length of the gap between points in the structure containing the point being searched for, inserted, or deleted. More importantly for our analysis, a sequence of k operations, all of one type (insertions, deletions, or searches), can be performed in total time $O(k \log \log n/k)$. For insertions and deletions this follows from Johnson's analysis of his algorithm.

For a sequence of searches in order by the positions being searched for, the time bound follows because consecutive searches in the same gap can be detected in constant time, by simply comparing each new search point with the endpoints of the gap containing the previous search point. Therefore we need pay the $O(\log \log D)$ cost at most once per gap. The cost of the sequence of search operations is therefore $\sum_{i=1}^{k} O(\log \log D_i)$. Because the function $f(x) = \log \log x$ is convex, any sum

$\sum_{i=1}^{k} f(x_i)$ is maximized when the x_i are all equal, and so the sum is bounded by $kf(\sum_{i=1}^{k} x_i/k)$. In particular, the cost of the search sequence can be reduced to

$$O(k \log\log(\sum_{i=1}^{k} D_i/k)) = O(k \log\log n/k).$$

For a sequence of k searches in non-sorted order we could use another instance of Johnson's flat trees to sort the search points, in time $O(k \log\log n/k)$, and then perform the searches in sorted order as above; or if the sequence was not known in advance we could simply insert a dummy point in the flat tree recording the search result, so that future searches giving the same result will see a smaller gap, and then after the last search we could delete all the inserted dummy points, to leave the data structure clean for the next sequence of operations. In fact in all our applications of the data structure, each sequence of searches can in fact be made to be already in sorted order, so no such handling will be needed.

Let us state the result so far as a theorem:

Theorem 3. The data structure solving equation 17 for simple cost functions can be implemented to take $O(k \log\log n/k)$ time for each homogeneous sequence of operations.

For non-simple functions, we must also take into account the binary searches required to compute $border(i, j)$ when including new values from A into the data structure. Assume that k such computations need be performed for a given column. If all the binary searches occurred in disjoint intervals of the range from 0 to n, the total time would be $O(k \log n/k)$ by a similar analysis to that for simple functions. To force the search intervals to be disjoint, we first find the borders among the points being inserted.

In particular, we need to solve an instance of equation 17, in which there are k new values of $D[j]$ given. By corollary 3 of the first section, each value of $D[j]$

supplies the minima for $E[i]$ with i in some interval of the range from 0 to n, and further these intervals appear in the same order as the positions of $D[j]$. Clearly, $border(i, j)$ for a newly added point j need only be computed within the interval in which $D[j]$ is better than the other new points. Further, all computations of $border(i, j)$ have at least one of the indices i or j being a newly added point. If, given the set of new values to be inserted, we can compute the partition of $[0 \ldots n]$ into intervals in which each of these values is best, guaranteed to exist by corollary 3, we can use this partition to perform each $border(i, j)$ computation in a disjoint interval, and therefore the total time for these computations for k new points will be $O(k \log n/k)$.

The algorithm of Aggarwal et al. [2] can find the minima at all n points, and therefore the boundaries between the intervals, in time $O(k + n)$. That of Galil and Giancarlo [17], which uses binary searches to find interval borders as in the data structure of the first section, but which needs only a stack instead of a more complicated search structure, can find the boundaries in time $O(k \log n)$. We combine these two algorithms to achieve a bound of $O(k \log n/k)$, which is what we are trying to achieve.

This is done as follows. We first select the points $E[n/k]$, $E[2n/k]$, etc, and find for each of these points which value of $D[j]$ supplies the minimum. This computation involves only k points $E[i]$, and so we can solve it in time $O(k)$ using the algorithm of Aggarwal et al. The remaining points in the range from 0 to n are divided up by this computation into k segments, each of length n/k. For each boundary between values $D[j-1]$ and $D[j]$, we know from the above computation which segments it falls in. If the two endpoints of an interval both have the same value of $D[j]$ supplying their minima, there can be no boundary within that interval. Otherwise, if $D[i]$ is the left minimum and $D[j]$ the right minimum, the segments will contain only those boundaries of intervals corresponding to positions between i and j.

Thus for each segment we can perform a binary search, as used in the computation of Galil and Giancarlo, for the boundaries that may fall within that interval. Each binary search is thus limited to a range of n/k points, and so it will take time $O(\log n/k)$. Each value of $D[j]$ is involved in the computations for at most two segments, those to the left and to the right of the segment border points for which it supplies the minima. If $D[j]$ does not supply the minimum for any segment border point, it will be involved in the computation for exactly one segment. The time for a segment containing b boundaries will be $b \log n/k$, and so the total time for computing boundaries between new point intervals is $O(k \log n/k)$ as desired.

Once we have computed the boundaries between the intervals of the new points being inserted, we can insert the points into the data structure as before, computing $border(i, j)$ by a binary search that stays within the interval of the point being inserted. The sum of the interval lengths is bounded by n, so the time for insertion is bounded by $O(k \log n/k)$. Thus we have shown the following theorem:

Theorem 4. The data structure solving equation 17 for arbitrary convex or concave cost functions can be implemented to take $O(k \log n/k)$ time for each homogeneous sequence of operations.

In chapter 6, we use the $O(\log n)$ and $O(\log \log n)$ single-operation bounds for the solution of equation 17 to efficiently compute single loop RNA secondary structures. Later, in chapter 9, we show how this data structure, and the bounds for homogeneous sequences of operations, can be used to take advantage of sparsity for even faster RNA structure computation.

6. RNA STRUCTURE WITH CONVEX OR CONCAVE COSTS

In this chapter we examine the following recurrence, which can be considered as a two-dimensional generalization of the least weight subsequence problem of chapter 4:

$$E[i,j] = \min_{\substack{0 \le i' < i \\ 0 \le j' < j}} D[i',j'] + w(i'+j', i+j) \tag{12}$$

As with the least weight subsequence problem, D is taken to be some simple function of E. It may be that D is not defined for certain pairs (i,j); in the case of RNA structure this occurs when two bases do not pair. To account for this we give that position of D the special value $+\infty$, which is taken to be larger than all real numbers. E may of course also take this value. We are given as initial values $D[i,0]$ and $D[0,j]$ for all i and j; we require the solution of the recurrence for $1 \le i \le n$ and $1 \le j \le n$; in the application of this recurrence to sequence analysis n will be the length of the input sequence.

The recurrence can be solved by the obvious dynamic program in time $O(n^4)$; fairly simple techniques suffice to reduce this time to $O(n^3)$ [75]. In this paper we present a new algorithm, which when w is convex solves recurrence 12 in time $O(n^2 \log^2 n)$. For many common choices of w, a more complicated version of the algorithm solves the recurrence in time $O(n^2 \log n \log\log n)$. Similar techniques can be used to solve the concave case of recurrence 12.

The recurrence above has an important application to the computation of RNA secondary structure [64, 75]. After a simple change of variables, one can use it to solve the following recurrence:

$$C[p,q] = \min_{p < p' < q' < q} G[p',q'] + g((p'-p) + (q-q')) \tag{11}$$

Recurrence 11 has been used to calculate the secondary structure of RNA, with the assumption that the structure contains no *multiple loops*, that is, loops having

more than one subloop nested within them [64]. Thus $C[p, q]$ gives the minimum energy loop in which p and q form a base pair, and the minimization is over all possible loops interior to p and q. Our algorithm computes this structure in worst case time $O(n^2 \log^2 n)$, under the realistic assumption that the energy function w of a loop is a convex function of the number of exposed bases in that loop.

It is possible to calculate RNA secondary structure with multiple loops, but this seems to require time $O(n^3)$ for linear energy functions, or $O(n^4)$ for general functions [75]. Secondary structure computation without multiple loops is less useful, because most known RNA structures in fact contain such loops. However, the multiple loop recurrences contain 11 to calculate the energy of loops without multiple subloops, and so speeding up this case will also speed up the more general computation, if not asymptotically then at least in the constant factor. It is also possible that our algorithm could be used to find plausible single loop substructures, which could be connected together heuristically to form a more general RNA structure.

For computations on long sequences, the space requirements of the algorithms can be at least as important as the time; if one algorithm requires more time than another it can simply be run longer, but if an algorithm requires more memory than is available, it can not run at all. The previous RNA structure algorithms require space $O(n^2)$, except for the most general $O(n^4)$ algorithm which takes $O(n^3)$. A naive implementation of the algorithms we describe would take $O(n^2 \log n)$, but this can without much difficulty be reduced to $O(n^2)$, matching the other algorithms.

6.1. Contention Within a Diagonal

In recurrence 12, we call each of the points (i', j') that may possibly contribute to the value of $E[i, j]$ *candidates*. We consider the computation of $E[i, j]$ as a contest between candidates; the *winner* is the point (i', j') with the minimum value of $D[i', j'] + w(i' + j', i + j)$. If we can find a way of eliminating many candidates at once, we can use this to reduce the time of an algorithm for solving recurrence 12.

We say that two points (i, j) and (i', j') in the matrices D or E are on the same *diagonal* when $i + j = i' + j'$. By the *length* of a diagonal we mean the number of points on it; e.g. the longest diagonal in the matrix has length n rather than $n\sqrt{2}$. We say that (k, l) is in the *range* of (i, j) when $k > i$ and $l > j$; that is, when point (i, j) is a candidate for the value of $E[k, l]$.

In this section we describe a way of eliminating candidates within the same diagonal. Using these methods, any given point (i, j) need only compare the values of candidates from different diagonals; there will be only one possible choice for the winning candidate from any given diagonal. In the next section we describe how to compare candidates from different diagonals in order to achieve our time bounds.

In what follows we will assume that the region below a diagonal is a right triangle, having as its hypotenuse the diagonal below the given one, and having as its opposite right angled corner the point (n, n). In fact this region need not be triangular, but if we pretend that our matrices D and E are at the bottom right corner of $2n \times 2n$ matrices we can extend the region to a triangle of the given form (figure 6.1.1). This extension will not change the time bounds of our algorithms.

We denote rectangles by their upper left and lower right corners; that is, by the rectangle extending from (i, j) to (i', j') we mean the set of points (x, y) such that $i \leq x \leq i'$ and $j \leq y \leq j'$. The range of a point (i, j) is the rectangle extending from $(i + 1, j + 1)$ to (n, n) (figure 6.1.2).

Figure 6.1.1. Making the region below a diagonal triangular.

Figure 6.1.2. Ranges of points are rectangles extending to (n, n).

Lemma 9. If (i, j) and (i', j') are on the same diagonal, and if $D[i, j] \leq D[i', j']$, then for all (k, l) in the range of both points, $D[i, j] + w(i + j, k + l) \leq D[i', j'] + w(i' + j', k + l)$. In other words, (i', j') need not be considered as a candidate for

those points in the range of (i, j).

Proof: Immediate from the assumption that $i + j = i' + j'$. •

Given a point (i, j) on some diagonal, define the *upper bound* of (i, j) to be the point (i', j') with $i' < i$, with $D[i', j'] \leq D[i, j]$, and with i' as large as possible within the other two constraints. If there is no such point, take the upper bound to be (n, n). Informally, the upper bound is the closest point above (i, j) on the diagonal that has a lower value than that at (i, j). Similarly, define the *lower bound* to be the point (i'', j'') with $i'' > i$, with $D[i'', j''] < D[i, j]$, and with i'' as small as possible, or (n, n) if there is no such point. Note the asymmetry in the above inequalities—we resolve ties in favor of the point that is further toward the top right corner of the matrix.

Observe that, if we order points (i, j) from a single diagonal lexicographically by the pair of values $(D[i, j], i)$ then the result is a well-ordering of the points such that, if (i', j') is a bound of (i, j), then $(i', j') < (i, j)$ in the well-ordering.

Define the *domain* of (i, j) to be the rectangular subset of the range of (i, j), extending from $(i+1, j+1)$ to (i'', j'), where (i', j') is the upper bound of (i, j) and (i'', j'') is the lower bound.

Lemma 10. Each point (i, j) of a given diagonal need only be considered as a candidate for the domain of (i, j). The domains for all points on the diagonal are disjoint and together cover the set of all points below the diagonal.

Proof: If a point below row i'' is within the range of (i, j), it is also within the range of (i'', j''), so (i, j) will never win the competition there. Similarly, if a point in the range of (i, j) is after column j' it will be won by (i', j'), or by some other point that is even better.

For any given two points on the diagonal, either one is a bound of the other or there is a bound of one of the two points between the two. Therefore no two

Figure 6.1.3. Domains of the points from a single diagonal.

domains can overlap. To see that all points below the diagonal are contained in some domain, first note that each such point is contained in the range of some point (i, j). Then the only way it can be removed from that point's domain is if it is also contained in the range of one of the bounds of (i, j). But because of the well-ordering described above we can not continue this process of taking bounds of bounds forever; therefore there must be some point on the diagonal containing the original point in its domain. •

An illustration of the partition of the region below a diagonal into the domains for the points on the diagonal is given in figure 6.1.3.

Lemma 11. The domains for each of the points on a diagonal having m total points can be found in time $O(m)$.

Proof: We process the points (i, j), all on the diagonal except for a dummy point (n, n), in order by increasing values of i. We maintain a stack of some of the previously processed points; for each point in the stack, the point below it in the stack is that point's upper bound. Each point that we have already processed, but that is no longer in the stack, will already have had its domain computed. No lower bound has yet been reached for any of the points still on the stack. Initially the stack contains (n, n), which is a lower bound for some points on the diagonal but which is itself not on the diagonal.

To process a point (i, j), we look at the point i', j' at the top of the stack. If $(i', j') \neq (n, n)$ and $D[i, j] < D[i', j']$, then (i, j) is a lower bound for (i', j'), so we can pop (i', j') from the stack, compute its domain from this lower bound and the upper bound found at the next position on the stack, and repeat with the point now at the top of the stack. Otherwise, (i, j) is not a lower bound for any stacked points, but (i', j') can be seen to be an upper bound for (i, j), so we push (i, j) on the stack. Finally, when all points have been processed, the points remaining on the stack have (n, n) as their lower bound, so we may pop them one at a time and compute their domains as before.

Each point is pushed once and popped once, so the total time taken by the above algorithm is linear. As we have seen the processing of each point maintains the required properties of the stack, so the algorithm correctly computes the upper and lower bounds, and therefore also the domains. •

We now give a more formal description of the algorithm described above. We denote the stack by S. Each position p on the stack consists of three components: $V(p)$, $I(p)$, and $J(p)$. $V(p)$ is the value of D at the point indexed by $I(p)$ and $J(p)$. The stack contains a dummy member at its bottom, which is marked by having a V value of $-\infty$. We use $k = i + j$ to denote the number of the diagonal for which we are computing the domains.

Then the domain computation algorithm can be expressed in pseudo-code as follows:

```
begin
    push (−∞, n, n) onto S;
    for i ← max(0, k − n − 1) to min(k, n − 1) do
        begin
        j ← k − i;
        while V(top) > D[i, j] do begin
            domain(I(top), J(top)) ←
                    rectangle from
                    (I(top) + 1, J(top) + 1)
                    to (i, J(top − 1));
            pop S
        end;
        push (D[i, j], i, j) onto S
    end;
    while V(top) > −∞ do begin
        domain(I(top), J(top)) ←
            rectangle from
             (I(top) + 1, J(top) + 1)
             to (n, J(top − 1));
          pop S
      end
end
```

6.2. Contention Among Diagonals

In the previous section we described a method for quickly resolving the competition among candidates within a single diagonal; here we will add to this an algorithm for resolving competition among candidates from different diagonals. We will reduce the minimization of recurrence 12 to the minimum of only two values, which can then be solved directly. The algorithm of the previous section for competition between diagonals works for any cost function, but here we require the cost function to be convex or concave.

Recall that the data structure of chapter 5 can solve equations of the form

$$E[i] = \min_j D[j] + w(i,j). \tag{17}$$

with the two possible operations of looking up the value of $E[i]$ for some index i, and decreasing the value of $D[j]$, for some index j. Each such operation takes time $O(\log n)$; if the cost function w is simple, each operation takes time $O(\log \log n)$.

We keep a separate such data structure for each row and column of the matrix of the original problem; thus denote the matrices for row i as $E_i^R[j]$ and $D_i^R[j]$, and similarly denote the matrices for column j as $E_j^C[i]$ and $D_j^C[i]$. We will maintain the property that, at the time we compute the value of $E[i,j]$ in recurrence 12, it will be found as the minimum of the two values $E_i^R[i+j]$ and $E_j^C[i+j]$. Thus each such computation takes the time for two data structure operations, plus a constant amount of work to combine the two values.

It remains to show, once $E[i,j]$ and $D[i,j]$ have been computed for some point (i,j), how the data structure can be updated to maintain the invariant, for those points at which $D[i,j]$ supplies the minimum in recurrence 12. We will compute these values, and update the data structure, a diagonal at a time. First we use the previously described algorithm to compute the domains for each point on the diagonal. Each domain is a rectangle; we cut it into strips, which will be intervals

Figure 6.2.1. Cutting domains into strips.

either of rows or columns. We choose whether to cut the domain into row intervals or column intervals according to which direction results in the fewer strips. We show in figure 6.2.1 three such domains already cut into strips.

We further maintain for each point (i, j) a list $L(i, j)$ of strips beginning at point (i, j). When we cut the domains of a diagonal into strips, we simply add each strip to the appropriate list. Then, when we process the diagonal containing point (i, j), which is the first time at which the strips in $L(i, j)$ can influence the computation, we use the strips to update D_i^R and D_j^C. In particular, if a strip in $L(i, j)$ was formed from the domain of point (i', j'), we reduce $D_i^R[i' + j']$ to the minimum of that value and $D[i', j']$. Similarly we reduce $D_j^C[i'+j']$ to $\min(D_j^C[i'+j'], D[i'+j'])$.

It can be seen that, when we calculate the value of $E[i, j]$ for each point (i, j), the minimum computed by $E_i^R[i+j]$ will be that of recurrence 17 for those strips that

have already been included in D_i^R. Similarly $E_j^C[i+j]$ will perform the minimization over all strips that have been included in D_j^R. But any such strip is not included until it can influence the value of $E[i,j]$, and so whatever $D[i',j']$ supplies the minimum will necessarily have $i' < i$ and $j' < j$.

Further, the domain of true minimum of recurrence $E[i,j]$ must by lemma 10 contain (i,j). If the domain is cut into row interval strips, then the strip containing (i,j) must begin at (i,j') with $j' \leq j$, and so the strip will already have been included in the data structure by the time we compute $E[i,j]$. Similarly, if the domain were cut into column strips, we would have included the appropriate strip in D_j^C.

We have thus shown that the true minimum is included in our computation of $E[i,j]$, and only points containing (i,j) in their ranges are included. Therefore our algorithm in fact correctly computes each $E[i,j]$. It remains to give a time bound for the algorithm. First let us compute a bound on the number of strips formed when we cut the domains.

Lemma 12. The total number of strips produced from the domains of the points on a single diagonal is $O(n \log n)$.

Proof: Assume without loss of generality, as in the previous section, that the region to be cut into domains and then strips is a triangle, rather than having its corners cut off. The length of the diagonal of the triangle is at most $2n$.

Let $T(m)$ be the largest number of strips obtainable from a triangle having m elements on the diagonal. As in the proof of lemma 11, the point on the diagonal having the least value has a rectangular domain extending to the corner of the triangle, leaving two smaller triangular regions to be divided up among the remaining diagonal points. Let us say the sides of this outer rectangular domain are $i+1$ and $j+1$; then i and j are the diagonal lengths of the smaller triangles, and $i+j = m-1$. The number of strips formed by this outer domain is the smaller of $i+1$ and $j+1$;

without loss of generality we will assume this to be $i+1$. Then

$$T(m) = \max_{\substack{i+j=m-1 \\ i \le j}} T(i) + T(j) + i + 1. \tag{18}$$

Now assume inductively that $T(k) \le ck \log k$ for $k < m$ and some constant $c \ge 1$. Let i and j be the indices giving the maximum in equation 18; note that $i < m/2$. By induction, $T(m) = O(m \log m)$ for all m. But the number of strips for any diagonal is certainly no more than $T(2n) = O(n \log n)$. •

But now we are done, because each strip produces constant overhead work in the maintenance of the corresponding strip list $L(i,j)$, and one value reduction in the data structure D_i^R or D_j^C.

Theorem 5. The above algorithm computes the values of $E[i,j]$ in recurrence 12, when the cost function is either convex or concave, in total time $O(n^2 \log^2 n)$, or in time $O(n^2 \log n \log\log n)$ for simple weight functions.

Proof: By lemma 12, we create $O(n^2 \log n)$ total strips, the processing for each of which takes time $O(\log n)$ (or $O(\log\log n)$ for simple functions). Further we compute $O(n^2)$ values of $E[i,j]$, each again taking time $O(\log n)$ or $O(\log\log n)$. Computing the domains of each point takes time $O(n^2)$. The total time is dominated by that for the strips, which is $O(n^2 \log^2 n)$ or $O(n^2 \log n \log\log n)$. •

We should note that the algorithm as described above takes space $O(n^2 \log n)$. The dominant term is that for the lists of strips $L(i,j)$; all other space is $O(n^2)$. The strip list space can be reduced to $O(n^2)$ also, at no cost in time, by delaying the division of domains into strips. In the reduced space algorithm each entry of $L(i,j)$, rather than being a strip, will be a rectangle, initially the domain of some point (i',j'). When we come to adding the strips to the data structure for row i, we cut just the single strip from the rectangle in the list, and if more strips remain to be cut from the rectangle we move the reduced rectangle to list $L(i+1,j)$. Similarly

if we are cutting the rectangle into columns we process the corresponding strip, and move the reduced rectangle to $L(i, j + 1)$. In this way the space required for the lists is not more than the total number of domains created, which is $O(n^2)$.

6.3. Summary

We have described algorithms that speed up a family of dynamic programs, used for prediction of RNA structure.

A recent paper by Aggarwal and Park [3] improves our bounds, for both the convex and concave cases, to $O(n^2 \log n)$. As in our algorithms, the space bound is $O(n^2)$. This improvement, while very interesting theoretically, may not be so useful for practical applications to RNA folding programs, because it uses matrix searching techniques [2], which lead to large constant factors in the time bounds.

In chapter 7, we show how to further extend the efficient computation of RNA structure, to the case in which the cost function is neither convex nor concave, but in which it can be split up into a small number of intervals in each of which it is either convex or concave.

In chapter 9, we show how to improve the results both of this chapter and of Aggarwal and Park's algorithm, by taking advantage of sparsity in the RNA structure computation (the infinities in the dynamic programming matrices alluded to at the start of this chapter).

7. MIXED CONVEX AND CONCAVE COST FUNCTIONS

In chapter 2, we showed how the sequence alignment problem with a gap cost function $w(i, j)$ may be solved using linearly many copies of the recurrence

$$E[j] = \min_{0 \le i < j} D[i] + w(i, j). \tag{13}$$

Further, experimental evidence shows that $w(i, j)$ may be assumed to be of the form

$$w(i, j) = f_1(i) + f_2(j) + g(j - i). \tag{19}$$

I.e. the cost of a gap is taken to be the sum of three terms, one each for the cost of breaking the sequence at the two endpoints of the gap, and one term $g(j-i)$ for the dependence of the cost on the length of the gap. For such functions, recurrence 13 may be simplified to

$$E[j] = \min_{0 \le i < j} D[i] + g(j - i). \tag{20}$$

In other words, we have abstracted the portions of $w(i, j)$ depending only on the endpoints i and j into arrays D and E, and out of the cost function $w(i, j)$ which now is simply a function $g(j - i)$ of the length of the gap.

The obvious dynamic programming algorithm for recurrence 13 (and recurrence 20) takes time $O(n^2)$; if we speed up this computation we will achieve a corresponding speedup in the computation of the modified edit distance.

It is not possible, without further assumption, to solve recurrence 13 more efficiently than the obvious $O(n^2)$ algorithm, because we must look at each of the possible values of $w(i, j)$. Thus in chapter 4, we required the assumption that w is concave (satisfies the quadrangle inequality) in order to achieve a linear time algorithm to solve the recurrence. Similarly the assumption that w is convex leads to a $O(n\alpha(n))$ time algorithm [35].

However, this argument does not apply to recurrence 20, because $g(j - i)$ only has linearly many values to be examined. Therefore we would like to speed up the

computation of recurrence 20, without assuming anything about the convexity or concavity of g. It is not now known whether such a solution is possible. In this chapter we provide a partial solution, by broadening the class of cost functions g for which an efficient solution is possible. In particular we allow g to be *piecewise concave and convex*, with s alternations. More precisely, we assume that we can find indices $c_1, c_2, \ldots c_{s-1}$ such that g is concave in the range $[1, c_1]$, convex in the range $[c_1, c_2]$, and so on. For such functions we solve recurrence 13 in time $O(ns\,\alpha(n/s))$, and therefore we also solve the modified edit sequence problem in time $O(n^2 s\,\alpha(n/s))$. Note that these times are never worse than the times of $O(n^2)$ and $O(n^3)$ for the naive dynamic programming solutions; when s is small our times will be much better than the naive times. Our algorithms use as subroutines the previous solutions to the concave and convex cases of recurrence 13; if these solutions are improved it is likely that our algorithms would also be sped up.

7.1. Mixed Cost Sequence Alignment

Let us consider again recurrence 20:

$$E[j] = \min_{0 \le i < j} D[i] + g(j - i).$$

We assume that there exist indices $c_1, c_2, \ldots c_{s-1}$ such that g is concave in the range $[0, c_1]$, convex in the range $[c_1, c_2]$, and so on. By examining adjacent triples of the values of g, we can easily divide the numbers from 0 to n into such ranges in linear time; therefore from now on we assume that $c_1, c_2, \ldots c_{s-1}$ are given. Also define $c_0 = 0$ and $c_s = n$.

We now form s functions g_1, g_2, $\ldots g_{s+1}$ as follows. Let $g_p(x) = g(x)$ if $c_{p-1} \le x \le c_p$; for other values of x let $g_p(x) = +\infty$. Then

$$E[j] = \min_{1 \le p \le s} E_p[j], \tag{21}$$

where

$$E_p[j] = \min_{0 \le i < j} D[i] + g_p(j - i). \tag{22}$$

Our algorithm proceeds by solving recurrence 22 independently for each g_p, and then using equation 21 to find $E[j]$ as the minimum of the s results obtained. We use as subroutines the algorithms mentioned in the introduction for solving recurrence 13 when g is convex or concave.

It turns out that concave segments (i.e. g_p for odd p) remain concave on the entire range $[1, n]$, and therefore we could apply the algorithm of the previous section directly to them. The solution for convex segments is more complicated, because in this case the added infinities do interfere with convexity. Further, even if convexity held and we applied Klawe and Kleitman's algorithm in the straightforward way, we would only achieve a time of $O(n\alpha(n))$ for each segment; the bound we wish to achieve is $O(n\alpha(n/s))$. By a more complicated process we may solve both the concave and convex segments in such a way that the amortized time per segment

is bounded by the formula above; however any individual segment p may have a solution time that is higher or lower than that bound, depending on its width $a_p = c_p - c_{p-1}$.

Now fix some segment p, either convex or concave. Then $E_p[j]$ depends on those values $D[i]$ such that $c_{p-1} \leq j - i \leq c_p$. Thus if we consider a matrix of pairs (i, j), the pairs such that $E_p(j)$ depends on $D[i]$ form a diagonal strip of width a_p. We solve the recurrence for g_p by dividing this strip into right triangles, of width and height a_p, pointing alternately to the upper right and lower left, and having as their hypotenuses alternately the diagonal borders of the strip determined by c_{p-1} and c_p (figure 7.1.1) We solve the recurrence independently on each triangle, and piece together the solutions to obtain the solution of g_p for the entire strip.

In particular, let the *upper triangle* U_t be the pairs (i, j) for which $j < c_p + (t-1)a_p$, $i \geq (t-1)a_p$, and $c_{p-1} \leq j - i$. Similarly let the *lower triangle* L_t be the pairs (i, j) for which $j \geq c_p + (t-1)a_p$, $i < ta_p$, and $j - i \leq c_p$. Then for any fixed j, all pairs (i, j) such that $E_p[j]$ depends on $D[i]$ may be found in the union of the upper and lower triangles containing j. More formally,

$$\begin{aligned} E_p[j] &= \min_{0 \leq i < j} D[i] + g_p(j-i) \\ &= \min_{c_{p-1} \leq j-i \leq c_p} D[i] + g_p(j-i) \\ &= \min\{X_p[j], Y_p[j]\}, \end{aligned}$$

where

$$\begin{aligned} X_p[j] &= \min_{j - c_{p-1} \geq i \geq \lceil (j-c_p+1)/a_p \rceil a_p} D[i] + g_p(j-i) \\ &= \min_{(i,j) \in U_{\lceil (j-c_{p-1}+1)/a_p \rceil}} D[i] + g_p(j-i) \end{aligned} \tag{23}$$

and

$$\begin{aligned} Y_p[j] &= \min_{\lceil (j-c_p+1)/a_p \rceil a_p > i \geq j - c_p} D[i] + g_p(j-i) \\ &= \min_{(i,j) \in L_{\lceil (j-c_p+1)/a_p \rceil}} D[i] + g_p(j-i). \end{aligned} \tag{24}$$

Thus we can compute the values of E_p by solving the values of X_p and Y_p

Figure 7.1.1. Strip for segment p divided into triangles.

within each upper and lower triangle. The computation within upper triangle U_t, corresponding to equation 23, can be expressed as follows:

$$X_p[j] = \min_{(t-1)a_p \leq i \leq j - c_{p-1}} D[i] + g_p(j-i), \tag{25}$$

which is exactly the same form as recurrence 13, except on a problem of size a_p instead of n. Further, all values of $g_p(j-i)$ in the recurrence are in the range $[c_{p-1} \leq j - i \leq c_p]$, so g_p is consistently either convex or concave in this range. Therefore, we can solve recurrence 25 in time $O(a_p \alpha(a_p))$ by using the Klawe and Kleitman's algorithm or our modified version of Wilber's algorithm.

The computation of Y_p in lower triangle L_t, which may be written

$$Y_p[j] = \min_{j - c_p \leq i < t a_p} D[i] + g_p(j-i), \tag{26}$$

is however not of the appropriate form. In particular, for the least value of j in the triangle, $Y_p[j]$ depends on all of the values of $D[i]$ for i in the triangle; succeeding values of j use successively fewer values of $D[i]$. However, observe that this pattern of usage implies that all values of $D[i]$ will be known before any value of $Y_p[j]$ from the triangle need be computed. Therefore, this is an offline problem. Because of this offline nature of the problem, we need not compute the values of Y_p in order by j; in fact we will compute them in reverse order, to transform the corresponding search matrix to upper triangular form. Actually this step is not necessary, as the relevant algorithms can be applied directly to the lower triangles, but it simplifies the presentation.

Let $j' = c_p + ta_p - j$, and let $i' = ta_p - i$. Then equation 26 can be rewritten

$$Y_p[c_p + ta_p - j'] = \min_{1 \le i' \le j'} D[ta_p - i'] + g_p(c_p + i' - j'), \tag{27}$$

which is now the same form as that of recurrence 20. Finally note that if $g_p(j - i)$ is a convex function of $j - i$, then $g_p(c_p + i' - j')$ is also a convex function of $j' - i'$; and similarly if g_p is concave it remains so under the change of variables. Thus by reversing the order of evaluation we have transformed equation 26 into a form that can be solved by Klawe and Kleitman's or Wilber's algorithms. Again the time taken for the solution is $O(a_p\alpha(a_p))$.

Each segment is composed of at most $O(n/a_p)$ upper and lower triangles; since the time for solving each triangle is $O(a_p\alpha(a_p))$, the total time to compute E_p for segment p is $O(n\alpha(a_p))$. The time for computing all segments is $\sum_{i=1}^{s} O(n\alpha(a_i))$, which, because $\sum a_p = n$ and by the convexity of the inverse Ackermann function, we can simplify to $O(ns\,\alpha(n/s))$. The time for combining the values from all segments is $O(ns)$. Therefore the total time for the algorithm is $O(ns\,\alpha(n/s))$. Thus we have shown the following facts:

Theorem 6. Recurrence 20, for g mixed convex and concave with s segments, can

be solved in time $O(ns\,\alpha(n/s))$.

Corollary 4. The sequence alignment problem for gap cost functions of the form $w(i,j) = f_1(i) + f_2(j) + g(j-i)$, with g mixed with s segments, can be solved in time $O(mns\,\alpha(n/s))$.

7.2. Mixed Cost RNA Structure

As we saw in chapter 3, a dynamic program similar to that used above for sequence comparison has also been used for predicting the secondary structure of RNA. The recurrence is as follows:

$$E[i,j] = \min_{\substack{0 \le i' < i \\ 0 \le j' < j}} D[i',j'] + w(i'+j', i+j). \tag{12}$$

As before, $D[i,j]$ may be easily computed from $E[i,j]$. Also as before, we assume that w is a function only of the difference between the two diagonals: $w(i'+j', i+j) = g((i+j)-(i'+j'))$.

The naive dynamic programming solution takes time $O(n^4)$, and a simple observation of Waterman and Smith reduces this to $O(n^3)$ [75]. In chapter 6 we showed that, if w is either convex or concave, this time can be further reduced to $O(n^2 \log^2 n)$. Aggarwal and Park [3] used different methods to reduce this time to $O(n^2 \log n)$. We now show that these results can be extended to piecewise convex and concave functions. If the number of segments in g is s, recurrence 12 can be solved in time $O(n^2 s \log n\, \alpha(n/s))$. For small s, this bound is much better than Waterman and Smith's time of $O(n^3)$. Our algorithm follows in outline that of Aggarwal and Park.

We solve recurrence 12 using divide-and-conquer techniques. Along with $E[i,j]$ we maintain another array $W[i,j]$, initially $+\infty$ at all cells. At all times $W[i,j]$ will be the minimum over some points (i',j') with $i' < i$ and $j' < j$ of $D[i',j'] + g((i+j)-(i'+j'))$. At each recursive level we will divide the pairs of indices (i,j) into two sets; if (i',j') is in the first set and (i,j) is in the second, with $i' < i$ and $j' < j$, then after that level the minimization for $W[i,j]$ will include the value for (i',j'). The divide and conquer will ensure that all pairs (i',j') with $i' < i$ and $j' < j$ will eventually be included in $W[i,j]$, at which time we can simply take $E[i,j] = W[i,j]$.

Each level of the recursion proceeds in three phases. First, we compute recursively $E[i,j]$ for $0 \leq j \leq n/2$. Second, we compute $W[i,j]$ for $n/2 < j \leq n$, using the formula

$$W[i,j] = \min_{\substack{0 \leq i' < i \\ 0 \leq j' \leq n/2}} D[i',j'] + g((i+j)-(i'+j')). \tag{28}$$

In the lower levels of the recursion, we take $W[i,j]$ as the minimum of its previous value and the above quantity. Finally, we compute recursively $E[i,j]$ for $n/2 < j \leq n$ and combine the recursive computation with the values of $W[i,j]$ computed above. In each recursive call, we switch the roles of i and j so that each index will be halved at alternate levels of the recursion; thus the dynamic programming matrix seen at each level of the recursion remains square.

Lemma 13. The above algorithm sketch correctly computes $E[i,j]$ for each (i,j).

Proof: By induction on the number of levels in the recursion. $E[i,j]$ is computed correctly in the first recursive call by induction. In the second half of the matrix, half of the possible values which we are minimizing over are supplied by $W[i,j]$, and the other half are supplied by the recursion, so $E[i,j]$ is again computed correctly. •

Thus all that remains is to show how to solve recurrence 28. Fix i, and let $W_i[j] = W[i,j]$. Then the recurrence can be rewritten

$$\begin{aligned} W_i[j] &= \min_{\substack{0 \leq i' < i \\ 0 \leq j' \leq n/2}} D[i',j'] + w(i'+j', i+j) \\ &= \min_{d} \min_{\substack{0 \leq i' < i \\ i'+j'=d \\ 0 \leq j' \leq n/2}} D[i',j'] + g((i+j)-(i'+j')) \\ &= \min_{d} Z[i,d] + g(i+j) - d), \end{aligned} \tag{29}$$

where

$$Z[i,d] = \min_{\substack{0 \leq i' < i \\ i'+j'=d \\ 0 \leq j' \leq n/2}} D[i',j']. \tag{30}$$

For a fixed diagonal d, equation 30 can be solved in time $O(n)$ using a prefix computation [39]. In particular $Z[i,d] = \min\{Z[i-1,d], D[i-1,d-i+1]\}$, so successive values of Z can be computed in constant time per value. Therefore all values of Z for the top level of the recursion can be computed in total time $O(n^2)$.

The remaining computation is recurrence 29. This follows a similar form to that of recurrence 13, and can be solved by the same methods. In fact the problem is offline (the values of Z on the right side of the equation do not depend on the values of W_i on the left) and so for convex or concave cost functions, the recurrence can be solved in linear time by the algorithm of Aggarwal et al. [2]. This observation is the heart of the $O(n^2 \log n)$ algorithm of Aggarwal and Park [3] for the convex and concave cases of the RNA structure computation. For our case, g is neither convex nor concave, but mixed. As in the previous section, recurrence 29 can be solved by dividing the matrix of pairs (d,j) into diagonal strips, and the strips into triangles. This leads to a time of $O(ns\,\alpha(n/s))$ for solving each instance of recurrence 29, and a total time for all such recurrences of $O(n^2 s\,\alpha(n/s))$.

Theorem 7. Recurrence 12 for $w(i,j) = g(j-i)$ a mixed convex and concave function with s segments can be solved in time $O(n^2 s \log n\,\alpha(n/s))$.

Proof: We have seen that the time spent performing the computations at the outer level of our recursive algorithm is $O(n^2 s\,\alpha(n/s))$. We may compute the total time for the algorithm by expanding two recursive levels at once, one halving j and the next halving i, so that we return to the same square shape of the dynamic programming matrix at lower levels of the recursion. Let $T(n)$ be the complexity of solving the problem for an $n \times n$ dynamic programming matrix. Then $T(n)$ consists of the time for solving the outer level of the recursion and the two problems at the next level, together with $T(n/2)$ for each of the four subproblems two levels down

in the recursion. This gives the recurrence

$$T(n) = 4T(n/2) + O(n^2 s\,\alpha(n/s)),$$

and solving for $T(n)$ gives $T(n) = O(n^2 s\,\log n\,\alpha(n/s))$. •

8. SPARSE SEQUENCE ALIGNMENT

In this chapter we are concerned with the comparison of two sequences, of lengths n and m, which differ from each other by a number of mutations. An alignment of the sequences is a non-crossing matching of positions in one with positions in the other, such that the number of unmatched positions (insertions and deletions) and matched positions with the symbol from one sequence not the same as that from the other (point mutations) is kept to a minimum. This is a well-known problem, and a standard dynamic programming technique solves it in time $O(nm)$ [55]. In a more realistic model, a sequence of insertions or deletions would be considered as a unit, with the cost being some simple function of its length; sequence comparisons in this more general model can be solved in time $O(n^3)$ [76]. The cost functions that typically arise are convex; for such functions this time has been reduced to $O(nm \log n)$ [17, 53] and even $O(nm\alpha(n))$, where α is a very slowly growing function, the functional inverse of the Ackermann function [35]. In chapter 4we saw that, for concave cost functions, the time can be further reduced to $O(mn)$, matching the time bound when only single-symbol insertions and deletions are considered.

Since the time for all of these methods is quadratic or more than quadratic in the lengths of the input sequences, such computations can only be performed for fairly short sequences. Wilbur and Lipman [79, 80] proposed a method for speeding these computations up, at the cost of a small loss of accuracy, by only considering matchings between certain subsequences of the two input sequences. In particular, their algorithm finds the best alignment in which each matched pair of symbols is part of a contiguous sequence of at least k matched symbols, for some fixed number k.

8.1. Finding Fragments

Let the two input sequences be denoted $x_1x_2 \ldots x_m$ and $y_1y_2 \ldots y_n$. Also, let σ be the size of the alphabet of symbols composing x and y. For biological sequence comparison $\sigma = 4$. In general we may typically assume $\sigma \leq m+1$, because at most m different symbols can be used in sequence x, and all other symbols appearing in y must either mismatch or be inserted. However if different symbols have different costs associated with their insertion, deletion, or substitution, σ may be larger, as large as $m+n$.

Wilbur and Lipman's algorithm first selects a small number of *fragments*, where each fragment is a triple (i, j, k) such that the k-tuple of symbols at positions i and j of the two strings exactly match each other; that is, $x_i = y_j$, $x_{i+1} = y_{j+1}$, ..., $x_{i+k-1} = y_{j+k-1}$. We will denote by M the number of fragments considered by the algorithm. In particular, an important special case is that we take all possible fragments of length k. We now show how to find all such fragments, in time $O((n+m)\log\sigma + M)$. For biological sequences, $\log\sigma = O(1)$ (we add two new endmarker symbols to the alphabet, so $\sigma = 6$) and the time becomes $O(n+m+M)$.

The method of fragment generation described in this section is not new; it uses a standard tool of string matching algorithms, the suffix tree. The reader is referred to [52, 77] for a definition of suffix tree of a string z, as well as for an algorithm that constructs it in $O(|z|\log\sigma)$ time.

We build the suffix tree for string $x\$_1y\$_2$, where $\$_1$ and $\$_2$ are two different end markers which match no symbol of x and y. Each leaf ℓ_i of the tree corresponds to a suffix of the string, starting from position i in the string. Further, every node in the tree has a string associated with it, which is the common prefix of all suffixes corresponding to leaves below the node in the tree. In particular, given two leaves ℓ_i and ℓ_j corresponding to positions in x and y, the least common ancestor of the

leaves corresponds to the maximal common prefix of the two leaves, which is the maximum common substring of the two strings starting at positions i and j.

Thus to accomplish our goal we need only find each node u of the tree with the length $l(u)$ of the corresponding string satisfying $l(u) \geq k$; and for each such node find all pairs i and j with $i \leq n$ and $j > n$ (so that i corresponds to a position in x, and j to a position in y), and with u the least common ancestor in the tree of ℓ_i and ℓ_j. The first part of this task, finding nodes with long enough correeponding substrings, is easily accomplished with a pre-order traversal of the suffix tree. We mark these nodes, so that we can quickly distinguish them from nodes with corresponding substrings that are too short.

Next observe that a node u is the least common ancestor of ℓ_i and ℓ_j if, and only if, ℓ_i and ℓ_j descend from different children of u. Thus to enumerate the desired substrings corresponding to u, we need simply take each pair v and w of children of u, such that $v \neq w$, and list pairs (i, j) with ℓ_i a descendant of v with $i \leq n$ and ℓ_j a descendant of w with $j > n$. To speed this procedure we should consider only those v having descendants ℓ_i meeting the condition above, and similarly for w; in this way each pair of children considered generates at least one substring, except for the pairs v, v of which there are linearly many in the tree.

To be able to perform the above computation, at the time we consider node u we must have for each of its children two lists of their descendant leaves, corresponding to positions in the two input strings. By performing a post-order traversal of the tree, we can list the substrings corresponding to each node u as above, and then merge the lists of leaves at the children of u to form the lists at u ready for the computation at the parent of u.

Thus to summarize the generation of matching substrings, we first compute a suffix tree; next we perform a pre-order traversal to eliminate those nodes corresponding to suffixes that are too short; and finally we perform a post-order traversal,

maintaining lists of leaves descended from each node, to generate pairs of positions corresponding to the desired common substrings. The generation of the suffix tree and the pre-order traversal each take time $O((n+m)\log\sigma)$. The post-order traversal and maintenance of descendant lists also takes time $O(m+n)$, and the generation of pairs of leaves corresponding to common substrings takes time $O(M)$. Thus the total time for these steps is $O((n+m)\log\sigma + M)$. For biological sequences $\sigma = 6$ and the time becomes $O(n+m+M)$.

8.2. Aligning Fragments

A fragment (i', j', k') is said to be *below* (i, j, k) if $i + k \leq i'$ and $j + k \leq j'$; i.e. the substrings in fragment (i', j', k') appear strictly after those of (i, j, k) in the input strings. Equivalently we say that (i, j, k) is *above* (i', j', k'). The *length* of fragment (i, j, k) is the number k. The *forward diagonal* of a fragment (i, j, k) is the number $j - i$. This differs from the *back diagonals* $i + j$ used for the RNA structure computation in chapter 6; here we will use both forward and back diagonals.

An *alignment* of fragments is defined to be a sequence of fragments such that, if (i, j, k) and (i', j', k') are adjacent fragments in the sequence, either (i', j', k') is below (i, j, k) on a different forward diagonal (a *gap*), or the two fragments are on the same forward diagonal, with $i' > i$ (a *mismatch*). The cost of an alignment is taken to be the sum of the costs of the gaps, minus the number of matched symbols in the fragments. The number of matched symbols may not necessarily be the sum of the fragment lengths, because two mismatched fragments may overlap. Nevertheless it is easily computed as the sum of fragment lengths minus the overlap lengths of mismatched fragment pairs. The cost of a gap is some function of the distance between forward diagonals $g(|(j - i) - (j' - i')|)$.

When the fragments are all of length 1, and are taken to be all pairs of matching symbols from the two strings, these definitions coincide with the usual definitions of sequence alignments. When the fragments are fewer, and with longer lengths, the fragment alignment will typically approximate fairly closely the usual sequence alignments, but the cost of computing such an alignment may be much less.

The method given by Wilbur and Lipman for computing the least cost alignment of a set of fragments is as follows. Given two fragments, at most one will be able to appear after the other in any alignment, and this relation of possible dependence is transitive; therefore it is a partial order. We process fragments in

the order of any topological sorting of this order. Some such orders are by rows (i), columns (j), or back diagonals ($i + j$).

For each fragment, the best alignment ending at that fragment is taken as the minimum, over each previous fragment, of the cost for the best alignment up to that previous fragment together with the gap or mismatch cost from that previous fragment. The mismatch cost is simply the length of the overlap between two mismatched fragments; if the fragment whose alignment is being computed is (i, j, k) and the previous fragment is $(i - x, j - x, k')$ then this length can be computed as $\max(0, k' - x)$. From this minimum cost we also subtract the length of the new fragment; thus the total cost of any alignment includes a term linear in the total number of symbols aligned. Formally, we have

$$C(i,j,k) = -k + \min \begin{cases} \min\limits_{(i-x,j-x,k')} C(i-x, j-x, k') + \max(0, k'-x) \\ \min\limits_{(i',j',k') \text{ above } (i,j,k)} C(i',j',k') + g(|(j-i)-(j'-i')|) \end{cases} \tag{10}$$

The naive dynamic programming algorithm for this computation, given by Wilbur and Lipman, takes time $O(M^2)$. If M is sufficiently small, this will be faster than many other sequence alignment techniques. However we would like to speed the computation up to take time linear or close to linear in M; this would make such computations even more practical for small M, and it would also allow more exact computations to be made by allowing M to be larger.

8.3. Division into Subproblems

Eppstein et al. [12], in related work, showed how to perform the computation of recurrence 10 for linear functions $g(x)$ in time $O(n+m+M \log\log \min(M, nm/M))$. However their techniques do not extend to convex or concave cost functions. Here we consider the problem of solving the recurrence when g is either convex or concave. In this section we show how we may use divide and conquer to produce a number of subproblems, the solution of which in an appropriate order will solve the original problem. In the next section we show that such an order may be found simply by processing the points in the order of the back diagonals they lie on.

We consider recurrence 10 as a dynamic program on points in a two-dimensional matrix. Each fragment (i, j, k) gives rise to two points, (i, j) and $(i+k-1, j+k-1)$. We compute the best alignment for the fragment at point (i, j); however we do not add this alignment to the data structure of already computed fragments until we reach $(i+k-1, j+k-1)$. In this way, the computation for each fragment will only see other fragments that it is below. We compute separately the best mismatch for each fragment; this is always the previous fragment from the same diagonal, and so this computation can easily be performed in linear time. From now on we will ignore the distinction between the two kinds of points in the matrix, and the complication of the mismatch computation.

As in the RNA structure computation of chapter 6, each point has a range consisting of the points below and to the left of it. However for this problem we divide the range into two portions, the *left influence* and the *right influence*. The left influence of (i, j) consists of those points in the range of (i, j) which are below and to the left of the forward diagonal $j - i$, and the right influence consists of the points above and to the right of the forward diagonal. Within each of the two influences, $g(|p-q|) = g(p-q)$ or $g(|p-q|) = g(q-p)$; i.e. the division of the range

in two parts removes the complication of the absolute value from the cost function.

Our algorithm for this problem can be viewed as a novel application of the Bentley-Saxe dynamic-to-static reduction [10]: we perform two such reductions, in two different orders, one for each type of eighth-plane piece of the fragment point ranges. The differing order leaves the problem dynamic, but the reduction instead can be imagined as removing the vertical or horizontal boundaries of the pieces, leaving only the forward-diagonal boundaries. The reduced subproblem can then be solved with matrix-searching techniques.

We first cut the domains of each point into right and left pieces, as described above. We divide the points into subproblems, and then proceed to compute the values of the recurrence at each point. Each value we compute is derived from the subproblems containing the given point, and once we have computed this value we apply it in the subproblems depending on it. The order in which we will compute the values at each point will be by back diagonals. This order is is symmetric with respect to the two kinds of pieces, so without loss of generality from now on we need only consider the subproblems derived from the right pieces, i.e. those eighth-plane pieces which are bordered on two sides by rows and forward diagonals.

We use divide and conquer to produce the subproblems into which we divide the computation. Thus, let us consider first the right influences only. We now describe how we might solve the problem if we didn't have to worry about including the left incluences in the computation.

The points corresponding to the fragments in the problem are arranged in a dynamic programming matrix with m rows and n columns. We divide the points into two sets: set A consists of those points above row $m/2$, and set B consists of those points below row $m/2$. Then, the problem could be solved by the following three steps:

(1) Solve recurrence 10 for set A recursively.

(2) Compute the minimum in recurrence 10 for those points (i, j, k) in B, where the points (i', j', k') used to compute the minimization are restricted to those points in A.

(3) Complete the computation of recurrence 10 for set A recursively.

This algorithm depends on the fact that no point in A is below any point in B, because of the way the sets were chosen. Thus step 1 can be solved without consideration of set B. Similarly step 2 computes all possible influence of the values at points in set A on the values in set B, so again step 3 can be carried out without further consideration of set A. Thus we have reduced the original problem to two similar problems (steps 1 and 3) with m reduced to half the original value, together with one problem (step 2) in which we know that each point in A occurs on a row above those containing all points in B.

In each subproblem, the points in A are those above some row and the points in B are those below the same row. Recall that we are restricting our attention here to right influences only. Thus the minimization for point (i, j) in B depends on the value at a point (i', j') in A exactly when $j' - i' < j - i$. Thus we order the points in the subproblems by the numbers of their forward diagonals. Such an order can be maintained by initially bucket sorting all points, and then splitting the sorted list at each level of the recursion.

For each point x in set B on diagonal d, $E[x]$ is calculated as the minimum of $D[y] + g(d' - d)$ for point y in set A on a previous diagonal d'. Then all other points on diagonal d will have the same value of $E[x]$, so we might as well index E by diagonals instead of by points. Similarly, y is the point on diagonal d' having the least value of $D[y]$ among all such points in set A, then no other point on the same diagonal can ever supply the minimum, so we will compute the same results if, for each diagonal, we throw out all points except the one with the least value of $D[y]$, and use this value in indexing D by diagonals instead of by points. So each

reduced subproblem can be written as the recurrence

$$E[d] = \min_{d' < d} D[d'] + g(d' - d).$$

But this is exactly the least weight subsequence problem considered in chapter 4. More precisely, we consider only those diagonals actually containing points in either A or B, and the problem is a submatrix of the least weight subsequence problem, as described at the end of that chapter.

If g is concave, so is the cost function of the reduced subproblem, and a problem with k points may be solved in time $O(k)$. If g is convex, we may use the algorithm of Klawe and Kleitman [35] to solve the problem in time $O(k\alpha(k))$; here α is the inverse Ackermann function, a very slowly growing function. Both the convex and concave methods are based on the matrix searching algorithm of Aggarwal et al. [2]. Another possible solution to these least weight subsequence problems is to use the algorithms of Galil and Giancarlo [17]. These solve both the convex and concave problems in time $O(t \log t)$, or for functions with the closest zero property in time $O(t)$. However they are much simpler than the matrix searching algorithms, and so even the $O(t \log t)$ version of the algorithms will likely be better than matrix searching in practice.

8.4. Combining Subproblem Solutions

From the previous section, it would appear that we have solved the fragment alignment problem already. But this is not so, because we only considered the problem for the right pieces of the point influences. The left pieces will also be reduced by a similar divide and conquer, again to least weight subsequence problems. However, for the left pieces the divide and conquer is by columns of the dynamic programming matrix. Thus the order of computation of the two types of pieces is incompatible, so we cannot perform both recursions simultaneously to solve the recurrence for both types of pieces at once.

Instead, we perform the two recursions separately, in each case setting up a data structure representing the state of each subproblem, but not solving any subproblems. Each point will be in set A for $O(\log n)$ subproblems and set B for $O(\log n)$ subproblems. Then we perform the actual computation in a third order, advancing the state of each subproblem as the values needed for the computation become available and combining the results of the subproblems to form the solution to recurrence 10. We will show that this may be done by considering the points (i, j) in order by back diagonals $i + j$.

The actual order in which the subproblems receive the values of $D[x]$ for points in set A will be more arbitrary than that described above, as will be the order in which the values that have been determined within the subproblem for points in set B are requested by the main program. However the forward diagonals totally order the points by their dependence on each other. The subproblem solution proceeds by saving each given value of $D[x]$ until all previous values in the dependence order are known, and then computing as many derived values as possible with the known values and saving these derived values until the main program asks for them. In this way each subproblem solution operates asynchronously of the main program.

All we require is that, whenever the main program asks for the subproblem's value at a point x in set B, all values $D[y]$ for points y on previous forward diagonals of set A will have already been given to the subproblem.

When we split the computation into subproblems, we also keep for each point a list of the subproblems for which that point is in set A; thus when the point's value is computed we need only look at the list to determine which subproblems can proceed in their computation. Along with these subproblem computations, we also proceed as we have said along back diagonals; for each point on a given back diagonal we compute the value as the minimum of the $O(\log n)$ values from the subproblems for which the point is in set B, and then include the computed value in the computations for which the point is in set A.

Let us now summarize the outline of the sparse alignment algorithm in pseudo-code:

```
begin
    find sparse set X of fragments;
    divide-and-conquer by rows to produce
        subproblems for right influences;
    divide-and-conquer by columns for left influences;
    for diag ← 2 to 2n do
        for x ∈ X with row(x) + column(x) = diag do begin
            E[x] ← +∞;
            for subproblem S with x ∈ B(S) do
                E[x] ← min(E[x], value at x in S);
            compute D[x] from E[x];
            for subproblem S with x ∈ A(S) do
                include value of D[x] in S;
        end
    end
end
```

It remains to show that, when the back diagonal computation reaches each point, the subproblems giving the point's value will all have computed their separate

minimizations for that point, so that the total value for that point can in fact be computed. In terms of the pseudo-code above, we need to show that each subproblem S with $x \in B(S)$ is ready to supply the value at x when the computation reaches the back diagonal containing point x.

For clarity of explanation, assume the subproblem S is one involving right influences; the assertion for left influence subproblems follows by symmetry. If a point (i, j) in set $B(S)$ for some subproblem S depends on the value at a point (i', j') in set $A(S)$, then clearly $i' < i$ and $j' - i' < j - i$. But then $j' + i' = (j' - i') + 2i' < (j - i) + 2i = j + i$; that is, the back diagonal containing $(i' + j')$ appears before that containing (i, j). Because we process points in order by back diagonals, $D[(i', j')]$ will already have been computed and included in subproblem S. Therefore all subproblem results will in fact be computed in time for them to be combined by the back diagonal computation, and the algorithm correctly computes recurrence 10.

Theorem 8. The problem of sequence alignment from a sparse set of fragments can be solved in time $O(n + m + M \log M \alpha(M))$ for convex gap cost functions, and time $O(n + m + M \log M)$ for concave functions.

Proof: As we have said, the time for each subproblem of size t is $O(t\alpha(t))$ in the convex case, and $O(t)$ in the concave case. Each point is in $O(\log n)$ subproblems, so the divide and conquer adds a logarithmic factor to these time bounds, giving $O(n + m + M \log M)$ in the concave case, and $O(n + m + M \log M \, \alpha(M))$ in the convex case. •

If we use the algorithms of Galil and Giancarlo, the bound for fragment alignment with simple functions is $O(n + m + M \log M)$, for both the convex and concave cases. For arbitrary convex and concave functions the time rises to $O(n + m + M \log^2 M)$. However the latter algorithms do not use matrix searching and are

therefore likely to be more efficient in practice.

9. SPARSE RNA STRUCTURE COMPUTATION

As we described in chapter 3, the following recurrence has been used to predict RNA structure:

$$E[i,j] = \min_{\substack{0 \le i' < i \\ 0 \le j' < j}} D[i',j'] + w(i'+j', i+j). \tag{12}$$

The value of $E[i,j]$ is the cost of the best singly-nested loop pairing bases i and j; $D[i,j]$ is then the best overall structure pairing the two bases, which may be a hairpin, single loop, or other structure.

The obvious dynamic programming algorithm solves recurrence 12 for sequences of length n in time $O(n^4)$ [64]; this can be improved to $O(n^3)$ [75]. When w is a linear function of the distance between back diagonals $(i'+j') - (i+j)$, another easy dynamic program solves the problem in time $O(n^2)$ [32]. Here we consider instead the case that the cost function is either convex or concave. In chapter 6we described a $O(n^2 \log^2 n)$ algorithm for such costs; this has recently been improved to $O(n^2 \log n)$ [3]. However we would like to reduce these times further, and so it would seem that we need to use some further assumptions about the problem. As in the previous chapter, the assumption we use is sparsity.

If a given pair of positions do not form a base pair, no structure pairing the two can exist, and the value of the cells in the matrix must be taken to be $+\infty$. Thus the minimum energies computed for the other cells of the matrix will not depend on that value, and so in turn no computed secondary structure will include a forbidden base pair.

Further, for the energy functions that are typically used, the energy cost of a loop will be more than the energy benefit of a base pair, so base pairs will not have sufficiently negative energy to form unless they are stacked without gaps at a height of three or more. Thus we could ignore base pairs that can not be so stacked, or

equivalently assume that their binding energy is again $+\infty$, without changing the optimum secondary structure.

This is analogous to the previous chapter, in which we showed how to compute sequence alignments efficiently from a sparse set of fragments. Here we wish, instead of computing sequence alignments, to compute RNA structure as in recurrence 12. Again we need only consider a sparse set of fragments, which here may be taken as pairs of substrings that may form stacked pairs of some given height. However, because here base pairs must be stacked to form a stable structure, we can consider a sparse set of fragments without affecting the optimality of the computed structure. Nevertheless, we might restrict our attention to longer fragments, gaining still more time at some expense of accuracy.

In fact the set of fragments may be computed exactly as in the previous chapter, by finding common substrings of the input sequence x together with another string x^*, formed by reversing the order of the symbols in x and replacing each symbol by its complementary base pair. Thus, if there are M such fragments, finding them takes time $O(n + m + M)$.

Eppstein et al. [12] described how to solve recurrence 12 for a sparse set of fragments when w is linear. Their algorithm takes time $O(M \log\log \min(M, n^2/M))$, and is closely related to their algorithm for sparse sequence alignment with linear gap costs. Here we instead assume that w is either convex or concave. In this case, neither the techniques for the linear case, nor the alignment algorithm of the previous chapter can be used. We instead derive a different algorithm, that for both the convex and concave cases takes time $O(M \log M \log \min(M, n^2/M))$. For simple cost functions this can be further reduced to $O(M \log M \log\log \min(M, n^2/M))$.

9.1. Computing the Structure

Each point (possible base pair) may be considered as having a *range of influence* consisting of the region of the dynamic programming matrix below and to the right of it. Thus the range of each point is a quarterplane with vertical and horizontal boundaries. We first effectively remove the horizontal boundaries, leaving half-planar ranges, at a logarithmic cost in execution time. This is done as follows.

We solve the problem by a divide and conquer recursion on the rows of the dynamic programming matrix. For each level of the recursion, having t points in the subproblem for that level, we choose a row r such that the numbers of points above r and below r are each at most $t/2$. Such a row must always exist, and it can easily be found in linear time. Thus we can partition the points of the problem into three sets: those above r, those on r, and those below r. In fact it would be possible to partition the points into only two sets, by including the first half of the points on r among the points below r, and including the second half of the points on r among the points above r. However the correctness of the algorithm is easier to see with the three-part division; and since the best way of computing the two-part division seems to be by first computing the three-part division, we might as well just use the three part division.

Within each level of the recursion, we will need the points of each set to be sorted by their column number. This can be achieved by initially bucket sorting all points, and then at each level of the recurrence performing a pass through the sorted list to divide it into the three sets. Thus the order we need will be achieved at a linear cost per level of the recurrence.

We note that for any point above or on r, the minimum value in equation 12 only depends on the values of other points above r. For points below r, the value of equation 12 is the minimum between the values from points above r, and the

points below r. Thus we can compute all the minima by performing the following steps: (1) solve the problem above r by a recursive invocation of our algorithm, (2) use the values given by this solution to solve the problem for the points on r, (3) compute the influence of the points above or on r, on the values of the points below r, and (4) recursively solve the problem below r.

This divide and conquer technique is similar to the dynamic-to-static reduction of Bentley and Saxe [10]; it differs from the RNA structure algorithm of Aggarwal and Park [3] in that we divide only by rows, and not by columns. It does not seem possible to modify the algorithm of Aggarwal and Park to run in time depending on the sparsity of the problem, because at each level of their recursion they compute a linear number of matrix search problems, the size of each of which does not depend on the sparsity of the problem.

The problem remaining after our recursion is as follows. We are given a set A of points above a certain row of the matrix, and a set B of points below the row. Both sets are sorted by column number. The values of the points in A are known, and we want to know their contributions to the minimizations for each of the points in B. Each level of the divide and conquer recursion computes the solution to two such problems, one with A the points above row r and B the points on row r, and a second with A the points above or on row r and B the points below row r.

We now write a recurrence equation for the reduced subproblem:

$$E[i,j] = \min_{\substack{(i',j')\in A \\ 1\le j'<j}} D[i',j'] + w(i'+j', i+j). \tag{31}$$

The crucial difference between this and equation 12 is that now, the requirement that $i' < i$ has been subsumed by the separation into sets A and B. In other words, the horizontal boundaries of the quarter-planar regions of influence have been removed, leaving only the vertical boundaries. Thus the range of influence of each point in A is the subset of B to the right of the point, and points in A are

totally ordered by inclusion of the ranges. We use the total order to add the points of A to the data structure described in chapter 5, so that when we process each point of B exactly the points that influence it will have been added to the data structure.

In particular, we process the points of A and B in order by their column numbers. The details of this processing will be given below. Within a given column, we first process the points of B, and then the points of A. By proceeding along the sorted lists of points in each set, we need only spend time on columns that actually contain points, so there will be no time loss determining which points to process next. Clearly, if we use this order, then whenever we process a point (i, j) from the set B, the points (i', j') of A that will have been processed will be exactly those with $j' < j$.

We process points by maintaining a copy of the data structure of chapter 5. Recall that the data structure maintains a matrix of values $D[y]$, initially all $+\infty$. At each step, the algorithm may either decrease one such value, or it may answer a query of the form

$$E[x] = \min_y D[y] + w(y, x). \tag{32}$$

It is easily seen that equation 32 is like equation 31, but with points (i, j) replaced by the numbers $i + j$ of their diagonals, and with the requirement that $j' < j$ removed. As we have described above, this last requirement will be taken care of by the order in which we process the points.

To process a point (i, j) from A, with value v, we let $y = i + j$ be the number of the diagonal containing the point, and reduce $D[y]$ to $\min(D[y], v)$. To process a point (i, j) from B, we let $x = i + j$ be the number of the diagonal containing the point, and compute the influence of the points in A on the value at (i, j) to be $E[x]$ as in equation 32.

This completes the solution of equation 31, and thus the solution of recur-

rence 12. To summarize, the algorithm solving the recurrence can be written in pseudo-code as follows:

```
procedure RNA(x, y):
begin
    find sparse set X of possible base pairs from the two strings;
    sort X by column numbers;
    let arrays E and D be indexed by members of X;
    for x ∈ X do E[x] = +∞;
    Recurse(X);
end
```

The recursive subprocedure called above solves the problem within the set of points given, assuming the influences of previous points have already been included in the computation. The input set of points is assumed sorted by column numbers, and the splitting of that set into subsets A, B, and C must maintain that sorted order. Note that there is no call to $Recurse(B)$ because the points in B, being all on the same row, cannot influence each other.

```
procedure Recurse(X):
begin
    let j be a row with at most |X|/2 points above and below it;
    let A be the points above row j in X;
    let B be the points on row j;
    let C be the points below row j;
    if A ≠ ∅ then begin
        Recurse(A);
        Influence(A, B);
    end;
    for x ∈ B do
        compute D[x] from E[x];
    if C ≠ ∅ then begin
        Influence(A ∪ B, C);
        Recurse(C);
    end;
end
```

Finally we give pseudo-code for computing the influence of one set of points on another, in which all the actual work of solving the recurrence is performed. Again, the input sets are sorted by column.

```
procedure Influence(A, B):
begin
    let X = A ∪ B, maintaining sorted order; if A and B both have
        points in the same column let those of B come first;
    prepare a data structure solving F[i] = min_j G[j] + w(j, i);
    for x ∈ X in order do begin
        d ← row(x) + column(x);
        if x ∈ A then G[d] ← min(G[d], D[x])
        else E[x] ← min(E[x], F[d]);
    end;
end
```

Before we give the time bound, let us first note that the time per data structure operation can be taken to be $O(\log M)$ or $O(\log \log M)$ rather than $O(\log n)$ or $O(\log \log n)$. This is because we need only consider diagonals of the dynamic programming matrix that actually contain some of the M points in the sparse problem. We number these non-empty diagonals in order by their real diagonal numbers; it is easily seen that this change does not affect the convexity or concavity of the cost function. However closest zero functions have the complication that the computation of $border(x, y)$ is defined in terms of actual diagonal numbers. Therefore we need to translate actual diagonal numbers into the nearest non-empty diagonals; a table to perform this translation can be created in $O(n + M)$ time, and used throughout the algorithm.

Theorem 9. The RNA structure computation of recurrence 12, for a sequence of length n, with M possible base pairs, and convex or concave cost functions, can be performed in time $O(n + M \log^2 M)$. For cost functions with the closest zero property, the computation can be performed in time $O(n + M \log M \log \log M)$.

Proof: Denote the number of points processed at a given level of the recurrence by t. Then the time taken at that level is $O(t)$, together with $O(t)$ operations from the data structure. The time per data structure operation is either $O(\log M)$ or $O(\log \log M)$, as described above. The latter version also requires $O(M)$ preprocessing time to set up the flat tree search structures; however the same structures can be re-used at different levels of the recursion and so this setup time need only be payed once. The divide and conquer adds another logarithmic factor to this bound. We also need to compute the possible base pairs and bucket sort them, in a preprocessing stage taking time $O(t)$. The total time to solve recurrence 12 is $O(n + M \log^2 M)$ for arbitrary convex or concave functions. For simple functions the time can be reduced to $O(n + M \log M \log \log M)$. •

9.2. Faster Computation for Intermediate Sparsity

In the introduction we promised a time bound for the RNA structure computation of $O(n + M \log M \log \min(M, n^2/M))$ for arbitrary convex or concave loop cost functions, and $O(O(n + M \log M \log \log \min(M, n^2/M))$ for simple cost functions. Yet in the previous section the bounds we gave were only $O(n + M \log^2 M)$ or $O(n + M \log M \log \log M)$. Here we describe how to improve our algorithms to run within the time bounds we claimed. We assume without loss of generality that $n < M$; otherwise, the bounds given in the previous section reduce to those here.

First let us examine the algorithm for simple functions. The algorithm for arbitrary functions is similar but requires a few more ideas. Our algorithms will be similar to those of the previous section, but the divide and conquer scheme will be different. Instead of dividing only by rows, we divide alternately by rows and columns, similarly to the divide and conquer technique used in the non-sparse RNA structure algorithm of Aggarwal and Park [3]. More precisely, at even levels of the divide and conquer recurrence we divide the dynamic programming matrix at some row i as before; however we choose i to be the center of the matrix rather than the center of the sparse set of points in the matrix. At odd levels we similarly divide by columns. In this way, each level of the recursion performs a computation in a matrix that is either square or close to square; there can be $O(\log n)$ levels before the recursion bottoms out at single points.

In terms of the pseudo-code given in the previous section, we need two versions of *Recurse*, one that divides by rows and one that divides by columns, each of which calls the other. We also need two versions of *Influence*, one to be called by each version of *Recurse*. All of these procedures keep the sets of points they handle in two sorted orders: sorted by rows, and sorted by columns. Unlike the code of the previous section, we divide into only two sets A and C; the line of division

between them will be halfway between two actual rows or columns, and so there is no in-between set B. Further this line of division is chosen by halving the number of columns in the sets, instead of halving the number of points.

As before, we compute the values of the points in A recursively, compute the influence of these values on those of the points in C, and then finish the computation of the values in C recursively. In the description that follows we assume that the current level in the divide and conquer recursion is even, so that as in the previous section the division between A and C occurs on a row boundary; the computation for odd levels is similar.

In the previous section, we computed the value D from E for each point when it was part of set B. Because here there is no such set, we must do so at another time; in particular we do so when the recursion bottoms out, and all points are on a single row or column. In this way the value is computed exactly once for each point, before it is needed.

Thus the pseudo-code for the procedures can be written as follows. We have merged the *Influence* procedure in with *Recurse*, because it would have been called only in one place. We only show one of the two mutually recursive procedures; the other can be found by replacing rows by columns and vice versa.

```
procedure RecurseColumn(X):
begin
    let i and k be the first and last rows occurring in X;
    if i = k then
        for x ∈ X do
            compute D[x] from E[x];
    else begin
        j ← ⌈(i + k)/2⌉;
        let A be the points above row j in X;
        let C be the points on or below row j;
        RecurseRow(A);
        let F[i] = min_j G[j] + w(j, i) be solved by the data structure;
```

```
            for x ∈ X in order by columns do begin
                d ← row(x) + column(x);
                if x ∈ A then G[d] ← min(G[d], D[x])
                else E[x] ← min(E[x], F[d]);
            end;
            RecurseRow(C);
        end;
    end
```

Recall that we can implement the data structure of chapter 5 to use, in place of the flat trees of van Emde Boas, we use Johnson's improved flat trees [31]. With such an implementation, a sequence of k operations, all of one type (insertions, deletions, or searches), can be performed in total time $O(k \log n/k)$. If the cost function is simple, this time can be further reduced to $O(k \log \log n/k)$.

Theorem 10. The RNA structure computation of recurrence 12, for sparse set of M fragments, and convex or concave cost functions with the closest zero property, can be performed in time $O(n + M \log M \log \log \min(M, n^2/M))$.

Proof: Consider the time for the top level of the recursion, which we denote $T(0)$. The algorithm from the previous section consists of, for each column, performing a sequence of searches, and then a sequence of insertions and deletions. Let the number of searches in column i be denoted s_i, and the number of insertions and deletions be denoted d_i. Then $\sum d_i$ is at most twice the total number of points in set A, and $\sum s_i$ is the total number of points in set C. The time taken is $\sum s_i \log \log n/s_i + \sum d_i \log \log n/d_i$. In the function $f(x) = x \log \log n/x$, the $\log \log n/x$ term decreases as x increases, and so the function as a whole is sublinear and therefore convex. Because of this convexity the total time taken at the

given recursive stage in the algorithm can be reduced to

$$\begin{aligned} T(0) &= \sum_{i=1}^{n} O(f(s_i) + f(d_i)) \\ &\leq O(nf(\sum_{i=1}^{n} s_i/n)) \\ &= O(nf(M/n)) \\ &= O(n(M/n)\log\log\frac{n}{M/n}) \\ &= O(M\log\log n^2/M). \end{aligned}$$

An identical analysis applies to each even level recursive subproblem, with n replaced in the bound by the size of the matrix for the subproblem. Similar bounds hold for the odd levels.

Now let us consider the sum of the times for all stages at a given level $2i$. As before, the analysis for odd levels is similar. Let M_j, for j from 1 to 2^{2i}, be the number of points in subproblem j. Further, at the given level, there will be 2^{2i} subproblems, each of having 2^i rows and columns. Then, by convexity, the total time for the level is

$$\begin{aligned} T(i) &= O(\sum_{j=1}^{2^{2i}} M_j \log\log\frac{(n/2^i)^2}{M_j}) \\ &\leq O(2^{2i}(M/2^{2i})\log\log\frac{(n/2^i)^2}{M/2^{2i}}) \\ &= O(M\log\log n^2/M). \end{aligned}$$

There are $O(\log M)$ levels in the recursion, and each takes time $O(M\log\log n^2/M)$. Therefore the total time is $O(M\log M\log\log\min(M, n^2/M))$. •

Theorem 11. The RNA structure computation of recurrence 12, for a sequence of length n, with M possible base pairs, and arbitrary convex or concave cost functions, can be performed in time $O(n + M\log M\log\log\min(M, n^2/M))$.

Proof: As we showed in chapter 5, k operations of the same type in the data structure may be performed in time $O(k \log n/k)$. Thus the analysis is identical to that of theorem 10, with $\log \log$ replaced by $\log$. •

10. CONCLUSIONS AND FURTHER RESEARCH

We have described algorithms for sequence alignment with gaps, and for RNA secondary structure computation. These algorithms assume convex, concave, or mixed convex and concave costs. Some perform their computations on all potential matching pairs of symbols in the input strings (or potential base pairs in the RNA structure problem). Others achieve even greater efficiency by considering only a sparse subset of the potential pairs. In particular, we described algorithms for the following problems:

(1) Sequence alignment with concave gap costs, in time $O(mn)$.

(2) RNA structure with convex or concave loop length costs, in time $O(n^2 \log^2 n)$. For simple cost functions, the time becomes $O(n^2 \log n \log\log n)$. These bounds were recently improved by Aggarwal and Park [3] to $O(n^2 \log n)$.

(3) Sequence alignment with mixed costs, in time $O(mns\,\alpha(n/s))$.

(4) RNA structure with mixed costs, in time $O(n^2 s\,\log n\,\alpha(n/s))$.

(5) Sparse sequence alignment, with concave costs in time $O(M \log M)$, and with convex costs in time $O(M \log M \alpha(M))$.

(6) Sparse RNA structure computation, with convex or concave costs, in time $O(M \log M \log\min(M, n^2/M))$. For simple cost functions, the time becomes $O(M \log M \log\log\min(M, n^2/M))$.

A number of related problems remain open for further research.

The most obvious is to ask whether any of the above bounds can be further improved. For instance, when $M = nm$ the sparse alignment algorithms are worse than their non-sparse counterparts by a logarithmic factor; perhaps this factor can be removed by, as in the RNA structure computation, replacing the $\log M$ term by $\log\min(M, nm/M)$. However this would seem to require a different approach than the present divide and conquer. A similar question is whether the factor of $\alpha(n)$

can be removed in the convex alignment times, reducing them to the same as the concave times. Such an improvement would also likely give a similar improvement for the mixed cost algorithm time bounds.

It may prove possible to reduce the non-sparse RNA structure computation time to $O(n^2)$, which would be optimal since that many values must be computed in the recurrence. In the convex case perhaps an $O(n^2\alpha(n))$ time algorithm can be found. Baruch Schieber (private communication) has claimed some results in this direction.

Let us turn now to widening the set of assumptions made in our algorithms. A particular special case that we omitted is sparse alignment or RNA structure computation with mixed convex and concave cost functions. The difficulty here is that, unlike the case for convex or concave costs, a submatrix of a mixed cost least weight subsequence is not itself mixed. Perhaps our techniques can be extended to a broader class of cost functions that is closed under submatrix reductions. A related question, raised in the chapter on mixed cost functions, is whether we can improve the obvious dynamic programming algorithms when the cost function $w(x, y) = g(y - x)$ is a function of one variable, but with no further assumptions.

In chapter 2, we mentioned the problem of computing a *circular alignment* for two strings; that is, the minimum cost alignment between any circular permutations of the two strings. For single insertions and deletions, a $O(nm \log n)$ time algorithm was known [33], and it seems likely that the linear cost sparse alignment algorithm of Eppstein et al. [12] reduces this to $O(M \log M \log\log \min(M, nm/M))$. However these algorithms depend on a non-crossing property of shortest paths in the dynamic programming matrix which may not hold in general, and in particular is not likely to hold for convex costs. For concave costs, the non-crossing property may hold, but only if we make some further assumptions about the cost of substitutions.

Finally, we must consider the space bounds for our algorithms. Recall that,

for linear cost sequence alignment, Hirschberg [25] gave a linear space algorithm remaining within the best known time bound of $O(nm)$. However no such improvement is known for our algorithms. In general, our space bounds are all currently $O(n^2)$ for the RNA structure problem, and $O(nm)$ for sequence alignment. For the sparse RNA computation, the space bound is $O(M)$, and for sparse sequence alignment it is $O(M \log M)$. The last bound, in particular, may be greater than the $O(nm)$ non-sparse bound. For all of these bounds it remains open whether the space can be improved, even at some small cost in time.

BIBLIOGRAPHY

[1] A. Aggarwal and M.M. Klawe, Applications of Generalized Matrix Searching to Geometric Algorithms, Discr. Appl. Math., to appear.

[2] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber, Geometric Applications of a Matrix-Searching Algorithm, Algorithmica 2, 1987, pp. 209–233.

[3] Alok Aggarwal and James Park, Searching in Multidimensional Monotone Matrices, 29th IEEE Symp. Found. Comput. Sci., 1988, pp. 497–512.

[4] Stephen F. Altschul and David J. Lipman, Trees, Stars, and Multiple Biological Sequence Alignment, SIAM J. Appl. Math., to appear.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[6] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, Data Structures and Algorithms, Addison-Wesley, 1983.

[7] A. Apostolico and C. Guerra, The Longest Common Subsequence Problem Revisited, Algorithmica 2, 1987, pp. 315–336.

[8] W. Bains, MULTAN: A Program to Align Multiple DNA Sequences, Nucl. Acids Res. 14, 1986, pp. 159–177.

[9] Geoffrey J. Barton and Michael J.E. Sternberg, A Strategy for the Rapid Multiple Alignment of Protein Sequences, J. Mol. Biol. 198, 1987, pp. 327–337.

[10] J.L. Bentley and J.B. Saxe, Decomposable Searching Problems I: Static-to-Dynamic Transformation. J. Algorithms 1(4), December 1980, pp. 301–358.

[11] H.S. Bilofsky, C. Burks, J.W. Fickett, W.B. Goad, F.I. Lewitter, W.P. Rindone, C.D. Swindel, and C.S. Tung, The GenBank Genetic Sequence Databank, Nucl. Acids Res. 14, 1986, pp. 1–4.

[12] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano, Sparse Dynamic Programming I: Linear Cost Functions, manuscript.

[13] Michael J. Fischer and R. Wagner, The String to String Correction Problem, J. ACM 21, 1974, pp. 168–178.

[14] Walter M. Fitch, Weighted Parsimony, Workshop on Algorithms for Molecular Genetics, Washington D.C., 1988.

[15] Walter M. Fitch and Temple F. Smith, Proc. Nat. Acad. Sci. USA 80, 1983, pp. 1382–1385.

[16] Zvi Galil and Raffaele Giancarlo, Data Structures and Algorithms for Approximate String Matching, J. Complexity 4, 1988, pp. 33–72.

[17] Zvi Galil and Raffaele Giancarlo, Speeding Up Dynamic Programming with Applications to Molecular Biology, Theor. Comput. Sci., to appear.

[18] Zvi Galil and Kunsoo Park, An Improved Algorithm for Approximate String Matching, 16th Int. Conf. Automata, Languages and Programming, 1989, to appear.

[19] Zvi Galil and Y. Rabani, On the Space Requirement for Computing Edit Distances with Convex or Concave Gap Costs, in preparation.

[20] Michael R. Garey and David S. Johnson, Computers and Intractibility: A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.

[21] O. Gotoh, An Improved Algorithm for Matching Biological Sequences, J. Mol. Biol. 162, 1982, pp. 705–708.

[22] O. Gotoh, Alignment of Three Biological Sequences with an Efficient Traceback Procedure. J. Theor. Biol. 121, 1986, pp. 327–337.

[23] G.H. Hamm and G.N. Cameron, The EMBL Data Library, Nucl. Acids Res. 14, 1986, pp. 5–9.

[24] J.P. Haton, Practical Application of a Real-Time Isolated-Word Recognition System using Syntactic Constraints, IEEE Trans. Acoustics, Speech and Signal Proc. ASSP-22(6), 1974, pp. 416–419.

[25] D.S. Hirschberg, A Linear Space Algorithm for Computing Maximal Common Subsequences, Comm. ACM 18, 1975, pp. 341–343.

[26] D.S. Hirschberg, Algorithms for the Longest Common Subsequence Problem, J. ACM 24, 1977, pp. 664-675.

[27] D.S. Hirschberg and L.L. Larmore, The Least Weight Subsequence Problem, 26th IEEE Symp. Found. Comput. Sci., 1985, 137–143, and SIAM J. Comput. 16, 1987, pp. 628–638.

[28] M.K. Hobish, The Role of the Computer in Estimates of DNA Nucleotide Sequence Divergence, in S.K. Dutta, ed., DNA Systematics, Volume I: Evolution, CRC Press, 1986.

[29] A.J. Hoffman, On Simple Linear Programming Problems, Convexity, Proc. Symp. Pure Math. 7, AMS, 1961, pp. 317–327.

[30] J.W. Hunt and T.G. Szymanski, A Fast Algorithm for Computing Longest Common Subsequences, C. ACM 20(5), 1977, pp. 350–353.

[31] Donald B. Johnson, A Priority Queue in Which Initialization and Queue Operations Take $O(\log \log D)$ Time, Math. Sys. Th. 15, 1982, pp. 295–309.

[32] M.I. Kanehisi and W.B. Goad, Pattern Recognition in Nucleic Acid Sequences II: An Efficient Method for Finding Locally Stable Secondary Structures, Nucl. Acids Res. 10(1), 1982, pp. 265–277.

[33] Zvi M. Kedem and Henry Fuchs, On Finding Several Shortest Paths in Certain Graphs, 18th Allerton Conf., 1980, pp. 677–686.

[34] Maria M. Klawe, Speeding Up Dynamic Programming, manuscript.

[35] Maria M. Klawe and D. Kleitman, An Almost Linear Algorithm for Generalized Matrix Searching, preprint, 1987.

[36] Donald E. Knuth, Optimum Binary Search Trees, Acta Informatica 1, 1973, pp. 14–25.

[37] Donald E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, 1973.

[38] Donald E. Knuth and Michael F. Plass, Breaking Paragraphs into Lines, Software Practice and Experience 11, 1981, pp. 1119–1184.

[39] Richard E. Ladner and Michael J. Fischer, Parallel Prefix Computation, J. ACM 27(4), 1980, pp. 831–838.

[40] Gad M. Landau, String Matching in Erroneous Input, Ph.D. dissertation, Computer Science Dept., Tel-Aviv Univ., 1986.

[41] Gad M. Landau and Uzi Vishkin, Efficient String Matching in the Presence of Errors, 26th IEEE Symp. Found. Comput. Sci., 1985, pp. 126–136.

[42] Gad M. Landau and Uzi Vishkin, Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm, 18th ACM Symp. Theory of Computing, 1986, pp. 220–230.

[43] Gad M. Landau and Uzi Vishkin, Fast Parallel and Serial Approximate String Matching, J. Algorithms, to appear.

[44] Gad M. Landau, Uzi Vishkin, and Ruth Nussinov, Efficient String Matching with k Differences for Nucleotide and Amino Acid Sequences, Nucl. Acids Res., 1986.

[45] V.I. Levenshtein, Binary Codes Capable of Correcting, Deletions, Insertions and Reversals, Sov. Phys. Dokl. 10, 1966, pp. 707–710.

[46] David J. Lipman and W.L. Pearson, Rapid and Sensitive Protein Similarity Searches, Science 2, 1985, pp. 1435–1441.

[47] D. Maier, The Complexity of Some Problems on Subsequences and Supersequences, J. ACM 25, 1978, pp. 322–336.

[48] T. Maniatis, Recombinant DNA, in D.M. Prescott, ed., Cell Biology, Academic Press, New York, 1980.

[49] Hugo Martinez, Extending RNA Secondary Structure Predictions to Include Pseudoknots, Workshop on Algorithms for Molecular Genetics, Washington D.C., 1988.

[50] W.J. Masek and M.S. Paterson, A Faster Algorithm Computing String Edit Distances, J. Comp. Sys. Sci. 20, 1980, pp. 18–31.

[51] A.M. Maxam and W. Gilbert, Sequencing End-Labeled DNA with Base Specific Chemical Cleavages, Meth. Enzymol. 65, 1980, p. 499.

[52] E.M. McCreight, A Space Economical Suffix Tree Construction Algorithm, J. ACM 23, 1976, pp. 262-272.

[53] W. Miller and E.W. Myers, Sequence Comparison with Concave Weighting Functions, Bull. Math. Biol. 50(2), 1988, pp. 97–120.

[54] G. Monge, Déblai et Remblai, Mémoires de l'Académie des Sciences, Paris, 1781.

[55] 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. Mol. Biol. 48, 1970, p. 443.

[56] Ruth Nussinov, George Pieczenik, Jerrold R. Griggs, and Daniel J. Kleitman, Algorithms for Loop Matchings, SIAM J. Appl. Math. 35(1), 1978, pp. 68–82.

[57] Ruth Nussinov and A. Jacobson, Fast Algorithm for Predicting the Secondary Structure of Single-Stranded RNA, Proc. Nat. Acad. Sci. USA 77, 1980, pp. 6309–6313.

[58] T.A. Reichert, D.N. Cohen, and A.K.C. Wong, An Application of Information Theory to Genetic Mutations and the Matching of Polypeptide Sequences, J. Theor. Biol. 42, 1973, pp. 245–261.

[59] H. Sakoe and S. Chiba, A Dynamic-Programming Approach to Continuous Speech Recognition, Proc. Int. Cong. Acoustics, Budapest, 1971, Paper 20 C 13.

[60] F. Sanger, S. Nicklen, and A.R. Coulson, Chain Sequencing with Chain-Terminating Inhibitors, Proc. Nat. Acad. Sci. USA 74, 1977, 5463.

[61] David Sankoff, Matching Sequences under Deletion-Insertion Constraints, Proc. Nat. Acad. Sci. USA 69, 1972, pp. 4–6.

[62] David Sankoff, Simultaneous Solution of the RNA Folding, Alignment and Protosequence Problems, SIAM J. Appl. Math. 45(5), 1985, pp. 810–825.

[63] David Sankoff, Robert J. Cedergren, and Guy Lapalme, Frequency of Insertion-Deletion, Transversion, and Transition in the Evolution of 5S Ribosomal RNA, J. Mol. Evol. 7, 1976, pp. 133–149.

[64] David Sankoff, Joseph B. Kruskal, Sylvie Mainville, and Robert J. Cedergren, Fast Algorithms to Determine RNA Secondary Structures Containing Multiple Loops, in D. Sankoff and J.B. Kruskal, editors, Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, 1983, pp. 93–120.

[65] David Sankoff and Joseph B. Kruskal, editors, Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, 1983.

[66] P.H. Sellers, On the Theory and Computation of Evolutionary Distance, SIAM J. Appl. Math. 26, 1974, pp. 787–793.

[67] Robert E. Tarjan, Data Structures and Network Algorithms, SIAM, 1985.

[68] Peter van Emde Boas, Preserving Order in a Forest in Less Than Logarithmic Time, 16th IEEE Symp. Found. Comput. Sci., 1975, and Info. Proc. Lett. 6, 1977, pp. 80–82.

[69] V.M. Velichko and N.G. Zagoruyko, Automatic Recognition of 200 Words, Int. J. Man-Machine Studies 2, 1970, pp. 223–234.

[70] T.K. Vintsyuk, Speech Discrimination by Dynamic Programming, Cybernetics 4(1), 1968, 52–57; Russian Kibernetika 4(1), 1968, pp. 81–88.

[71] R.A. Wagner, On the Complexity of the Extended String-to-String Correction Problem, 7th ACM Symp. Theory of Computing, 1975, pp. 218–223.

[72] Michael S. Waterman, General Methods of Sequence Comparison, Bull. Math. Biol. 46, 1984, pp. 473–501.

[73] Michael S. Waterman and Temple F. Smith, RNA Secondary Structure: A Complete Mathematical Analysis, Math. Biosciences 42, 1978, pp. 257–266.

[74] Michael S. Waterman and Temple F. Smith, New Stratigraphic Correlation Techniques, J. Geol. 88, 1980, pp. 451–457.

[75] Michael S. Waterman and Temple F. Smith, Rapid Dynamic Programming Algorithms for RNA Secondary Structure, Adv. Appl. Math. 7, 1986, pp. 455–464.

[76] Michael S. Waterman, Temple F. Smith, and W.A. Beyer, Some Biological Sequence Matrices, Adv. Math. 20, 1976, pp. 367–387.

[77] P. Wiener, Linear Pattern Matching Algorithms, 14th Symposium on Switching and Automata Theory, 1973, pp. 1-11.

[78] Robert Wilber, The Concave Least Weight Subsequence Problem Revisited, J. Algorithms 9(3), 1988, pp. 418–425.

[79] W.J. Wilbur and D.J. Lipman, Rapid Similarity Searches of Nucleic Acid and Protein Data Banks, Proc. Nat. Acad. Sci. USA 80, 1983, pp. 726–730.

[80] W.J. Wilbur and David J. Lipman, The Context Dependent Comparison of Biological Sequences, SIAM J. Appl. Math. 44(3), 1984, pp. 557–567.

[81] F.F. Yao, Efficient Dynamic Programming Using Quadrangle Inequalities, 12th ACM Symp. Theory of Computing, 1980, pp. 429–435.