Chapter 1

The Fragment Alignment Algorithm

In this supplement, we describe the sparse dynamic programming algorithm used to find the best possible alignment of fragments under a collection of linear cost functions. We use the definitions and ideas presented in the paper by Eppstein et al (1992). We just present the basic concepts here, the details of which can be found in the above paper.

1.1 The Problem Statement

In this section, we present a formal statement of the problem.

Let $X = x_1 x_2 \cdots x_n$ and $Y = y_1 y_2 \cdots y_n$ be the two input sequences. We define a fragment $f = (i, j, k, l, score, direction)$ where $x_i \cdots x_j$ aligns with $y_k \cdots y_l$. The score of the alignment is given by *score*. The variable *direction* can take two values. A value of $direction = positive$ implies that its a forward alignment (ie. x_i aligns with y_k and x_j aligns with y_l) and a value of $direction = negative$ implies that its a reverse alignemnt (ie. x_i aligns with y_l and x_j aligns with y_k). Henceforth, we shall refer to a fragment as $f = (i, j, k, l)$, whenever we do not care about its *score* and *direction*.

A fragment $f_1 = (i_1, j_1, k_1, l_1, score_1, direction_1)$ is said to be *above* a fragment $f_2 = (i_2, j_2, k_2, l_2, score_2, direction_2)$ whenever $j_1 < i_2$, and, either $l_1 < k_2$ or $l_2 < k_1$, and then f_2 is said to be *below* f_1. An *alignment* A is defined as a sequence of fragments $A = f_1, f_2, \cdots, f_k$ such that f_i is above f_{i+1} for $i = 1, \cdots, k-1$. The score of A is defined as $score(A) = (\Sigma_{i=1}^{M} score_i) - (\Sigma_{i=1}^{M-1} GAP(f_i, f_{i+1}))$, where $GAP(f_i, f_j)$ is a linear gap cost function that gives us the gap cost associated with linking fragment f_j to fragment f_i.

Note that this definition of the terms *above* and *below* is different from the one presented

in Eppstein's paper. If we assume that the sequence X is lined up vertically from top to bottom and the sequence Y is lined up horizontally from left to right, then the fragment f_1 could either be above and to the left of fragment f_2 or be above and to the right of fragment f_2. Intuitively, this allows us to find alignments with fragments whose coordinates on the sequence X are continuously increasing, but whose coordinates on the sequence Y may follow any pattern along the alignment.

The fragment alignment problem can then be stated as : Given a set of fragments $F = \{f_1, \cdots, f_M\}$, find the alignment with maximum score.

1.2 Definitions and Terminology

In this section, we present some definitions and facts which will be used in the description of the algorithm.

The algorithm uses the ideas of sparse dynamic programming developed in Eppstein's paper. Since we now allow a fragment f_{i+1} in the alignment to link to a fragment f_i that is above and either to the left or to the right of fragment f_{i+1}, we have eight different linear gap cost functions as described below. For each of these eight cases, we have different data structures over which we execute Eppstein's algorithm as described in a subsequent section.

The eight different linear gap cost functions can be described by three bits b_1, b_2, b_3 as follows. Assume that we need to find the gap cost of linking fragment f_i to fragment f_j. The first bit $b_1 = 1$ if f_i is above and to the left of f_j, and $b_1 = 0$ if f_i is above and to the right of f_j. The second bit $b_2 = 1$ if $direction_i = positive$ and $b_2 = 0$ if $direction_i = negative$. The third bit $b_3 = 1$ if $direction_j = positive$ and $b_3 = 0$ if $direction_j = negative$.

We define a point as a pair (x, y) where x is an index in the sequence X and y is an index in the sequence Y. A point $P = (x, y)$ is said to be in the left influence region of a fragment $f = (i, j, k, l)$ if P is below and to the left of the point (j, l) and $(y - x) <= (l - j)$. Similarly, P is said to be in the right influence region of f if P is below and to the left of the point (j, l) and $(y - x) > (l - j)$.

Let $score(f, x, y)$ be the score at a point (x, y) due to the fragment f, that is, score (f,x,y) = {the total score of the alignment upto fragment f } - { the gap cost from the point (j, l) to the point (x, y) }.

The crucial fact that allows us to use sparse DP is the following (from Eppstein et al 1992):

FACT : Let $P = (i, j)$ be a point in the left influence (right influence respectively) region of both fragment f_1 and f_2 and assume that $score(f_1, i, j) >= score(f_2, i, j)$ given a linear gap cost function. Then $score(f_1, x, y) >= score(f_2, x, y)$ for any point (x, y) in the common left (right respectively) influence region of fragments f_1 and f_2.

Let $totalscore(f)$ denote the total score of the alignment upto and including the fragment f. The recurrences for finding the best alignment can then be written as :

$$LI(f) = \max_{f' : f \text{ is under the left influence of } f'} (totalscore(f') - GAP(f', f))$$

$$RI(f) = \max_{f' : f \text{ is under the right influence of } f'} (totalscore(f') - GAP(f', f))$$

$$totalscore(f) = \max(LI(f), RI(f)) + score(f)$$

A fragment f' is said to *left dominate* a region R if, for any fragment f starting in that region, $LI(f) = totalscore(f') - GAP(f', f)$. We say that f owns the region R. Similarly, we define *right domination*. Using the above fact, it is clear that if we keep track of which fragment dominates every region, we can compute the recurrences $LI(f)$ and $RI(f)$ above by just looking up the fragments that left and right dominate the region into which the upper left corner of the new fragment f falls. We can then use these values of $LI(f)$ and $RI(f)$ in the third recurrence.

1.3 Data Structures

We now describe the data structures used for keeping track of the domination of left influence regions by fragments. The data structures for the right influence regions are similar and are not described here. The details can again be found in Eppstein's paper.

For the left influence regions, we use two binary trees to keep track of the column and diagonal boundaries of the regions respectively. A binary tree called $CBOUND$ stores in a sorted order the columns where vertical region boundaries intersect the current row. Another binary tree called $DBOUND$ stores in a sorted order the diagonals where the region boundaries intersect the current row. By current row, we mean the row that is being processed by the algorithm. As we shall see later, the algorithm processes points row wise.

A fragment is said to be *active* if it is the owner of some region. All currently active fragments are present in a doubly linked list called $OWNER$. The order in which the fragments appear in this list is the order in which the regions appear on the current row.

We have 8 different copies of the three data structures described above, one for every case.

In addition to the above data structures, we use another binary tree for storing points $P = (x, y)$. We call this the *point tree*. This point tree is common for all the eight cases. The point tree is sorted in the increasing order of x and points with the same value of x are sorted in the increasing order of y.

In addition to storing the endpoints of fragments, the point tree also stores intersection points. Intersection points are defined as points where two or more region boundaries intersect. These points are inserted into the point tree when found, and each intersection point is marked with the three bit case number $b_1b_2b_3$ that generated that point.

1.4 The Algorithm

We now present a description of the algorithm.

The algorithm starts off by reading the information about the fragments from the input file (which is generated by CHAOS). For every fragment $f = (i, j, k, l, score, direction)$, we insert four points into the point tree as described below :

1. (i, k) which is the upper left corner of the fragment.
2. (i, l) which is the upper right corner of the fragment.
3. (j, k) which is the lower left corner of the fragment.
4. (j, l) which is the lower right corner of the fragment.

Points are then processed one by one from the point tree row wise and the following updates to the above data structures are made.

If the point is the upper left corner of a fragment, we find the fragment which owns the region in which this point falls by searching the $CBOUND$ and $DBOUND$ trees. We do this for both the left and right influence regions, and we repeat this for all cases $b_1b_2b_3$ where $b_1 = 1$ and $b_3 = 1$ if the current fragment is positive and 0 otherwise. We thus get 4 fragments, out of which we choose the one which gives the maximum score, and link the current fragment to that fragment.

If the point is the upper right corner of a fragment, we repeat the above steps with $b_1 = 0$ instead of $b_1 = 1$.

If the point is the lower right corner of a fragment, we modify the above data structures for all cases $b_1 b_2 b_3$ where $b_1 = 1$ and $b_2 = 1$ if the current fragment is positive, and 0 otherwise. Thus we modify the data structures for 2 cases in all (given by the two values of b_3). The modifications are done as follows.

We first find out whether the current fragment is the owner of some new region. If not, we do nothing. Else we insert the current fragment in the OWNER list in its appropriate place. We also insert new column and diagonal boundaries in $CBOUND$ and $DBOUND$. Insertion of new column and diagonal boundaries may create new intersection points. It may also cause the deletion of other intersection points. There are many different cases involved in the processing of the lower right corner of a fragment. We do not provide the details of those cases here, they can be found in the Eppstein paper.

If the point is the lower left corner of a fragment, we repeat the above steps for $b_1 = 0$ instead of $b_1 = 1$.

If the point is an intersection point, then three active regions meet there. Out of these three regions, one region ceases to exist after the current row. We have to determine which region of the other two continues to exist after the current row. This involves checking which fragment in owner is better for a point on the next row, and then using FACT 1, we can conclude that that fragment will be better for all points in the common region of the two fragments. Thus we remove the other fragment from the OWNER list, and modify the other data structures accordingly.

Thus we can process the points in the point tree one by one depending on the type of point. Finally, when we have processed all points from the point tree, we will have the alignments in the form of a fragment pointing backwards to the previous fragment in the alignment that gave the best score for that fragment and so on. By following these links of back pointers, we can reconstruct the alignments.

Note that we have used a little trick for the processing of cases with $b_1 = 0$. Consider a fragment $f = (i, j, k, l)$. Instead of inserting the point (i, l) as the upper right corner in the point tree, we insert the point $(i, -l)$ in the point tree. Similarly, for the lower left corner, we insert the point $(j, -k)$ in the point tree. This is because for cases where $b_1 = 0$, the successive fragment appears below and to the left of the previous fragment. Hence we ought to process that row in the reverse order, that is, from right to left. But instead, if we use the mirror image of the coordinates by negating the column numbers, we can then process the points in the usual left to right order. We also make use of the fact that the

processing for one case is entirely independent of the processing for the other cases since we have different data structures for every case. Hence negating the column numbers maintains the correctness of the algorithm.

1.5 Timing Analysis

In this section, we examine the time taken by our algorithm.

Consider that we have M fragments in all. We make use of the following two lemmas from Eppstein et al 1992.

Lemma 1: The total number of active regions is at most $2M$.

Lemma 2: The total number of active intersection points is bounded above by $4M$.

Using these two lemmas, we can claim the following.

Lemma 3: For every case, the size of the trees $CBOUND$ and $DBOUND$ is $O(M)$.
Proof: From Lemma 1, we have that the total number of regions is $O(M)$ for every case. Every region can have at most one column boundary, and at most one diagonal boundary. Hence proved.

Lemma 4: The size of the point tree is at most $O(M)$.
Proof: For every fragment, we have a constant number (four) of points in the point tree. The number of intersection points is $O(M)$ by Lemma 2. Hence the overall size of the point tree is $O(M)$.

Now consider the time taken to process a point in the point tree. When we process a point, there are many different cases involved depending on the type of point and the location of the point. But all these cases essentially involve a constant number of insertions and/or deletions into the three binary trees (CBOUND, DBOUND, and point tree), which can be accomplished in time $O(logM)$, and into the $OWNER$ list, which can be accomplished in constant time (because we know the exact place in the list where the insertion or deletion is to be made). Thus processing each point takes time $O(logM)$.

Since the total number of points is $O(M)$, it follows that the overall time taken by the algorithm is $O(MlogM)$.