

PLAINS Alignment Program

By Ofer H. Gill
March 9, 2004

PLAINS Alignment Program

- (1) Introduction
- (2) Literature PLAINS Derives From
- (3) The PLAINS Algorithm
- (4) The PLAINS Lab Results
- (5) Conclusions / Future Work

(1) Introduction

(a) Why Align?

2. The Scoring System

3. Gap Function Selection

4. What is PLAINS?

Why Align?

- We align different sequences of DNA because we'd like to learn what they have in common. Common regions in two sequences often represent a presence of the same gene.
- It's also important to learn how two sequences differ, as in what one sequence may have that the other one does not.
- An intuitive way to address these issues is to create a scoring system in aligning the two sequences. This system will reward matches and penalize mismatches, insertions, and deletions. This will allow us to see firsthand areas the sequences have in common, and areas where they differ.

The Scoring System

- In observing the DNA sequence evolution of certain species from a common ancestor. Many regions of DNA letters (or nucleotides) stay the same, but many also get replaced, inserted or deleted. (Indel is a shorthand term for “insertion or deletion.”)
- In these evolutions, replacements typically occur more frequently than indel. Therefore, replacements should be penalized less than indels.
- Also, a replacement region is typically 1 letter long, whereas an indel is typically 5-20 letters long. Therefore, it's common to penalize replacements per letter, but penalize indels per region of this gap.

The Scoring System

- Hence, typically a constant ms is the amount penalized per pair of letters that mismatched.
- And, a value $w(i)$, is the amount penalized per entire gap, where i is the length of this gap.
- As the discussion seems to lend, the choice of function for $w()$ is more difficult to find than the choice for constant value ms . Hence, choosing a good gap penalty function is a big issue in alignment.

Gap Function Selection

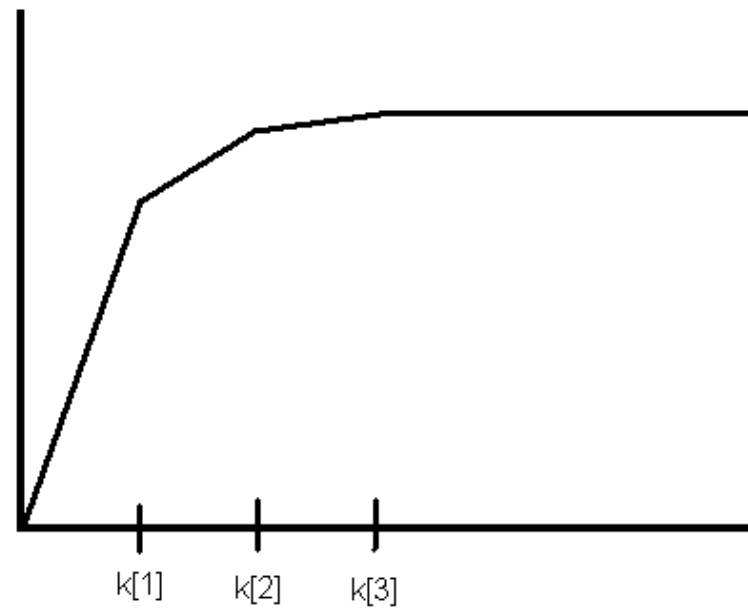
- Linear (aka affine) gap penalty function $a(i)$ lends itself to a simple alignment solution using Dynamic Programming (Smith-Watermann Algorithm) with:
 - $a(i) = w_0 + i * w_c$
- Here, w_0 is the cost of creating a gap, and w_c is the rate-increase for extending the gap. This dynamic program resembles LCS and Edit Distance very closely.
- Linear gap functions also have other intuitive algorithmic advantages, including space reduction (discussed later).
- Linear gap functions work fine in practice for aligning protein sequences, because they typically have gaps of length 1-3. DNA however, typically has gaps that range from 5-20 letters. The larger gap lengths and higher variation in gap lengths isn't encouraged by affine penalties.

Gap Function Selection

- Because for DNA, the probability of an i -length gap is classifiable by a binomial distribution (using multiplications), but our scoring system uses rewards and penalties (with additions), and the conversion from probabilities to scores is done by a log function, it shouldn't be a surprise why experts would guess that the best gap penalty function for DNA should be a log function of some sort.
- Log gap functions in alignments typically take much longer runtime and more memory, and in most cases, aren't practical, even for 4000 letter sequences.
- Piecewise-linear function $w(i)$ with slopes that change at certain points lets you approximate values for any function (including log functions), and they have most of the algorithmic advantages of linear functions. Therefore PLAINS uses piecewise-linear gap functions.

Gap Function Selection

Example of Piecewise-Linear Gap function



What is PLAINS?

- **P**iecewise **L**inear **A**lignment with Important Nerve Seeker (An “important nerve” is a nickname for a “fancy-looking alignment”)
- PLAINS performs pairwise alignment for DNA sequences.
- PLAINS works on DNA sequences of lengths varying from 100 to 4000.
- PLAINS can be used with a set of prespecified gap parameters, or it can look for a set of gap parameters that produces the best alignment.
- PLAINS can be run in text-mode, or as a graphical tool within Valis (a suite of Bioinformatics tools invented in NYU).

(2) Literature PLAINS Derives From

(a) Needleman-Wunsch General Gap Function
Alignment

2. Smith-Waterman Linear Gap Function
Alignment

3. Hirschberg/MM Linear Space Algorithm

4. Miller-Myers Forward Gap Searching Algorithm

General Alignment Outline

- Given two sequences X and Y of lengths m and n , a fixed mismatch penalty ms , and a general gap function $w()$, we'd like to find the alignment for X with Y .
- $V(i,j)$ is the score found for bits $X[1..i]$ and $Y[1..j]$.
- $E(i,j)$ and $F(i,j)$ are scores obtained where we end with vertical or horizontal gaps.
- $G(i,j)$ is the score where $X[i]$ and $Y[j]$ were aligned (match or mismatch).
- We seek $V(m,n)$, and can backtrack the V , E , F , and G tables to obtain our alignment.

Needleman-Wunsch Algorithm

- For general gap function $w()$, we have V , E , F , and G computed as follows:
 1. $V(0, 0) = 0$
 2. $V(i, 0) = E(i, 0) = -w(i)$
- $V(0, j) = F(0, j) = -w(j)$
- $V(i, j) = \max\{E(i, j), F(i, j), G(i, j)\}$
- $G(i, j) = V(i-1, j-1) + s(X[i], Y[j])$
- (Note: $s(X[i], Y[j]) = 1$ if $X[i] == Y[j]$, 0 otherwise)
- $E(i, j) = \max_{0 \leq k \leq j-1} [V(i, k) - w(j-k)]$
- $F(i, j) = \max_{0 \leq k \leq i-1} [V(k, j) - w(i-k)]$

Needleman-Wunsch Algorithm

- This algorithm runs in $O(m^2n)$ time and uses $O(mn)$ space. For a regular computer aligning 4000 nucleotide sequences, this runtime and memory usage isn't practical.

Smith-Waterman Algorithm

- Using linear gap function $a(i) = w_o + w_c * i$, with w_o and w_c are mentioned earlier, the computation for V , E , F , and G now become that of Gotoh's, which is:
 1. $V(0, 0) = 0$
 2. $V(i, 0) = E(i, 0) = -w_o - i * w_c$
 3. $V(0, j) = F(0, j) = -w_o - j * w_c$
 4. $V(i, j) = \max\{E(i, j), F(i, j), G(i, j)\}$
 5. $G(i, j) = V(i-1, j-1) + s(X[i], Y[j])$
 6. $E(i, j) = -w_c + \max\{E(i, j-1), V(i, j-1) - w_o\}$
 7. $F(i, j) = -w_c + \max\{F(i-1, j), V(i-1, j) - w_o\}$

Smith-Waterman Algorithm

- This algorithm uses $O(mn)$ time and $O(mn)$ space. In terms of runtime, it will run for 4000 nucleotide sequences on a regular computer, but the memory usage is still not practical...

Hirschberg/MM Linear-Space Algorithm

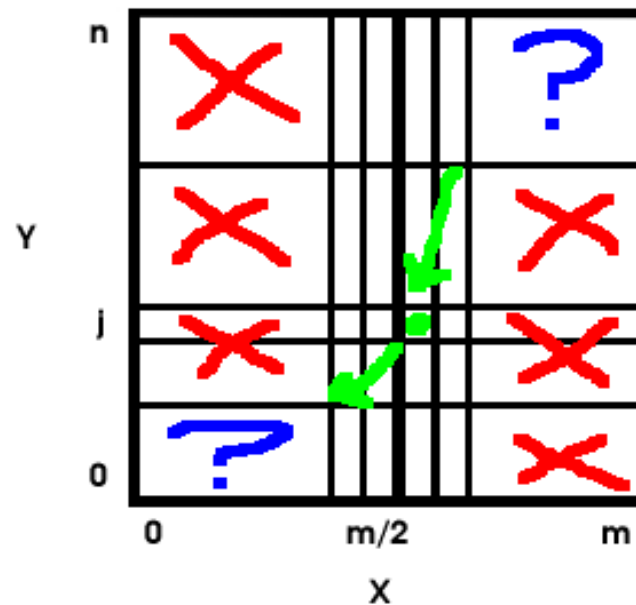
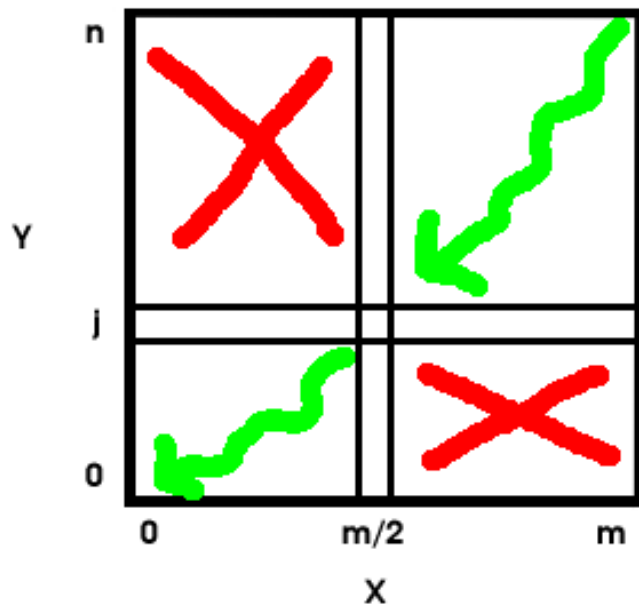
- Hirschberg uses a linear-space algorithm (later refined by Miller-Myers) that works with linear gap function $a()$ as follows:
 - In addition to our X, Y, V, E, F, G items, create strings X_r and Y_r (the reversal of X and Y), and create tables $V_r, E_r, F_r,$ and G_r for computations performed over X_r and Y_r . Hence we will work in reverse using a similar way to working forwards.
 - Instead of saving all m columns of our table, we're only going to save the t most recently computed columns (where t is some constant integer ≥ 2). We use the t most recently computed columns of our forward and reverse tables to get certain info on the first and last bits of X and Y .

Hirschberg/MM Linear-Space Algorithm

- If we compute the forward tables for $X[1..m/2]$ and $Y[1..n]$, and the reverse tables for $X[m/2..m]$ and $Y[1..n]$, saving the t most recently computed columns gives us many possible table-traces and alignments involving $X[(m/2)-t..(m/2)+t]$.
- Using this, if we can find the bits in Y that align with $X[(m/2)-t..(m/2)+1+t]$ in our alignment, we can proceed recursively on trying to align the bits in X and Y to the left/right of these known bits, and hence obtain the overall alignment using $O(t*n) = O(n)$ space.
- For Linear Gap function $a()$, the “magic” bits of Y to select are based on finding a k from among 0 to n such that:
 - $V(m/2, k) + V^r(m/2, n-k)$ is maximized
- This is the EVALUATION CRITERIA

Hirschberg/MM Linear-Space Algorithm

Hirschberg/MM algorithm. X marks areas not in the alignment.



Hirschberg/MM Linear-Space Algorithm

- Here's the Hirschberg/MM Linear-Space algorithm in full detail:

1. $\text{OPTA}(l, l', r, r', t)$ {

- if $(l > l')$ then align $Y[r\dots r']$ against gaps and return that; (Base case)
- else if $(r > r')$ then align $X[l\dots l']$ against gaps and return that; (Base case)
- else if $(l + l' - l \leq t)$ then
 - compute V for entries $X[l\dots l']$ and $Y[r\dots r']$ and trace the result to get an alignment A . We don't throw away any columns doing this, so we can get the full alignment for $X[l\dots l']$ and $Y[r\dots r']$. (This is another base case.) We return A ;
- else do as follows on the next slide;

Hirschberg/MM Linear-Space Algorithm

- $h = (l' - 1) / 2$;
- In $O(r' - r) = O(n)$ space, compute V on entries $X[l...h]$ and $Y[r...r']$ and compute V^r on entries $(X[h+1...l'])^r$ and $(Y[r...r'])^r$. Then, find an index k^* such that $X[l...h]$ aligned with $Y[r...k^*]$ and $X[h+1...l']$ aligned with $Y[k^*+1...r']$ gives the best score based on our EVALUATION CRITERIA. Trace the last t columns in V and the last t columns in V^r (or first depending on how you see it) in order to find the alignments for $X[h-(t-1)...h]$ with $Y[q_1...k^*]$ and $X[h+1...h+t]$ with $Y[k^*+1...q_2]$. (Note: q_1 and q_2 are determined from the positions in Y we are when we can trace no further.) Let's call these combined alignments L_2 .
- Call $\text{OPTA}(l, h-t, r, q_1, t)$. Let's call the alignment from this L_1
- Call $\text{OPTA}(h+t+1, l', q_2, r', t)$. Let's call the alignment from this L_3
- We glue L_1 followed by L_2 followed by L_3 to make an alignment L . We output L .

}

Hirschberg/MM Linear-Space Algorithm

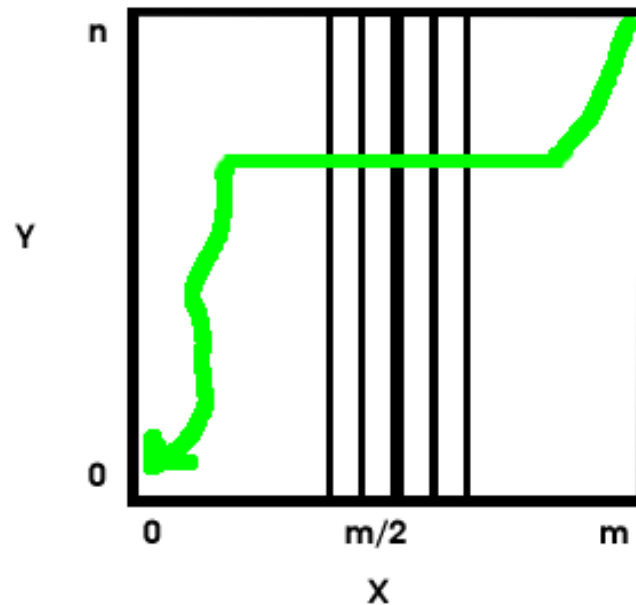
- We call $\text{OPTA}(1, m, 1, n, t)$ to get the alignment of X with Y.
- 2. If $T(m, n)$ is the runtime of $\text{OPTA}(1, m, 1, n, t)$, we can express $T(m, n)$ as
 - $T(m, n) \leq \max_{0 \leq k \leq n} [T(m/2, n-k) + T(m/2, k)] + O(mn)$
 - It turns out $T(m, n) = O(mn)$.
- Hence, we get an optimal alignment in the linear gap model in $O(mn)$ time and $O(n)$ space. (Quadratic time and linear space.)
- We get the correct alignment with linear gap functions because there, $V(m, n) = \max_{0 \leq k \leq n} [V(m/2, k) + V^r(m/2, n-k)]$. This relation doesn't hold true for general gap functions, or even for piecewise-linear functions, which typically need more than one column's results to compute info in a current column.

Hirschberg/MM Linear-Space Algorithm

- One particular issue that arises for general and piecewise-linear gap functions is that of correctly “bridging together” solutions from the forward and reverse tables in order to obtain the correct alignment.
- For alignments with mainly matches and mismatches, this isn't a problem.
- However, for alignments that are gap heavy, particularly having X against a gap with the gap starting in the first half of X, and ending in the second half of X, having forward and reverse computations for the first and second halves of X performed independently won't work here (since $w(i+j)$ might not necessarily be equal to $w(i)+w(j)$). But PLAINS works around this by having forward and reverse computations depend on each other. (More on this later...)

Hirschberg/MM Linear-Space Algorithm

When we've got a long gap from the first half of X to the second.



Miller-Myers Forward Gap Searching Algorithm

- Using general gap function $w()$, and the Needleman-Wunsch model, let's fix ourselves over a single row j at the moment (so $V(i) = V(i,j)$ and $F(i) = F(i,j)$)
- The Needleman-Wunsch assumption is that:
 - $F(i) = \max_{0 \leq k \leq i-1} [V(k) - w(i-k)]$
- Now, suppose we have $\text{Cand_F}(i)$ and $\text{ind_F}(i)$ for all i . These values correspond respectively to the best $F(i)$ score found yet, and the “temporary index winner” that the score comes from. (Essentially, the chosen k value in the max function above.)
- Next, let $\text{cand}(k,i) = V(i) - w(i-k)$.
- Then, we can restate $F(i)$ as:
 - $F(i) = \max_{(1 \leq k < j)} (\text{cand}(k,j))$

Miller-Myers Forward Gap Searching Algorithm

- We now have a new way to obtain the correct value for each $F(i)$.
 - For each i from 0 to m , we evaluate all $i' > i$ checking to see if $\text{cand}(i, i') > \text{cand}(\text{ind_F}(i'), i')$. If so, we change $\text{ind_F}(i')$ to i , and $\text{Cand_F}(i')$ to $\text{cand}(i, i')$.
- This uses the same time as our original Needleman-Wunsch algorithm, however...
 - If at some i th iteration, we know that for all i' from a to b , $\text{ind_F}(i, i')$ is some value k , then instead of maintaining $b-a+1$ ind_F entries, we need only one entry (let's have it be a struct block of some sort). And, this block can specify that k is the “temporary index winner” for all values from a to b by having its v -value = k , its l -value = a , and its r -value = b . With this in mind, we can cut down the # of $\text{ind_F}()$ entries we'd need to search over by making a simple linked list L of blocks. (We assume cand_F can be found in $O(1)$ once the necessary ind_F entry is known.)

Miller-Myers Forward Gap Searching Algorithm

- Next, note that in practice, general gap function $w()$ is convex (positive derivate, negative double derivative, and the piecewise-linear functions used in PLAINS are no exception). When this happens, suppose that for some i and i' , that: $\text{cand}(i,i') \leq \text{cand_F}(i')$, then, for all $i'' > i'$, $\text{cand}(i,i'') \leq \text{cand_F}(i'')$. In other words, if i isn't the $\text{ind_F}(i')$ value, it's not the ind_F value for any $i'' > i'$. This is based on the convex-ness of w , and helps save us computations. Therefore, the v -values in our L list should decrease as we go from left to right.
 - Example: | 5 5 5 5 5 | 4 4 4 | 3 3 3 3 3 3 | 2 2 | 1 1 1 1 1 |
- With this in mind, on an i th iteration, when determining which entries will have i as their ind_F value, we can search elements in L left to right and stop at the point when i “loses” to a temp index winner.

Miller-Myers Forward Gap Searching Algorithm

- You might ask: “For this i , if we found an element in L where for its l and r values, $\text{cand}(i,l) > \text{cand_F}(l)$ and $\text{cand}(i,r) \leq \text{cand_F}(r)$, what do we do then?” Well, we can use binary-search over the interval $[l,r]$ to find the largest i' value such that $\text{cand}(i,i') > \text{cand_F}(i')$ by doing binary search over the indices from l to r . We now have the foundation to Miller-Myers Forward Gap Searching Algorithm. The details to computing each $F(i)$ is now:
- **(Step 1) L begins empty, and for $F(0)$, we set it to whatever is the base value (as defined by our functions). We then put in L a block with $l=1$, $r=\text{max}$, and $v=0$. (For now, max can be m , but in general, it should be set to whatever j -value is the largest we'd like to get forward-gap computations up to, which could be larger than m without affecting computation time and space...)**
- **(Step 2) For each i from 1 to m , we do:**
- **(Step 2a) Look at the leftmost block b of L . We set $F(i)$ to $\text{cand}(b.v, i)$. We increment $b.l$ by 1. (So $b.l$ is now $i+1$)**
 - **Next, if $b.l > b.r$, we delete block b from L , and set b to the next leftmost block in L , and set $b.l$ to $i+1$.**

Miller-Myers Forward Gap Searching Algorithm

- (Step 2b) We check if $\text{cand}(i, i+1) > \text{cand}(b.v, i+1)$. If not, we go no further and jump to the next iteration of i . (Because i is not going to be the ind_F winner for any $i' > i$.) Otherwise we do step 2c.
- (Step 2c) while ($\text{cand}(i, b.r) > \text{cand}(b.v, b.r)$) and L isn't empty do:
 - delete b from L , set b to the new leftmost element of L (if it exists)
- (Step 2d) if L is empty, then insert into L one block b , and set $b.l$ to $i+1$, $b.r$ to max , and $b.v$ to i , and jump to the next iteration of i ; else do step 2e.
- (Step 2e) Insert a new block c as the leftmost element of L , and set $c.l$ to $i+1$, $c.r$ to $b.l - 1$, and $c.v$ to i (Note that block b is the block just next to c ...)
- (Step 2f) We do binary search over the interval $b.l$ thru $b.r$ to find the largest number i' such that $\text{cand}(i, i') > \text{cand}(b.v, i')$. If this number i' doesn't exist in block b , we jump to the next iteration of i , otherwise, we do step 2g.
- (Step 2g) Set $c.r$ to i' , set $b.l$ to $i'+1$. Go to the next iteration of i .

Miller-Myers Forward Gap Searching Algorithm

- Notice that each i th iteration adds at most 1 element to L , and deletes at most m elements, inspects at most one more element than the # it possibly inserts or deletes, and performs at most one binary search within an element (with its $r-l+1 \leq m$, so binary search-time is $O(\log m)$). And, we started the algorithm with only one element in L . Amortized analysis of this algorithm using the # of elements in L as potential value dictates that our overall runtime is $O(m \log m)$. (Better than $O(m^2)$ from Needleman-Wunsch)
- Going back to the two-dimensional tables, assume we use a different linked list $L(j)$ for each row j to compute F values, and we compute E values column-wise in a similar manner to F (using a different linked-list $R(i)$ for each column i). Then for general gap functions, our runtime now becomes $O(mn \log m)$.

Before we go on...

- It might appear like, in using Hirschberg/MM Linear-Space reduction in conjunction with Miller-Myers Forward Gap Search algorithm, we run the risk of violating the $O(n)$ space assumption, because even though we're saving only the t most recently computed columns of our tables, our n linked lists $L()$ have $O(m)$ elements, and our m linked lists $R()$ have $O(n)$ elements, hence we get $O(mn)$ space on Linked-Lists alone. True... However....
- PLAINS uses both Hirschberg/MM Linear-Space reduction in conjunction with Miller-Myers Forward Gap Search algorithm, but for p -part piecewise-linear gap function $ww()$, not general gap function $w()$. As we'll show later, a linked list in this assumption has $O(p)$ elements. Therefore, this and the assumption that we compute tables column-by-column (allowing us to use a single linked list R over all columns instead of m of them) implies $O(pn)$ space used for Linked Lists instead of $O(mn)$. (For fixed p , this is linear space...)

(3) The PLAINS Algorithm

- (a) Memory/Space Reductions in Using Piecewise Linear Functions
- 2. Bridging the Gaps for Gapped Alignments
- 3. Gap Parameter Optimization
- 4. ColorGrid Generation

Memory/Space Reductions in Using Piecewise Linear Functions

- PLAINS uses $O(np)$ memory, where p is the # of parts to piecewise-linear function $ww()$. $O(n)$ space gets used in the forward/reverse tables due to Hirschberg/MM Linear Space reduction.
- The reasoning for this is: In using Miller-Myers Forward Gap Searching on p -part $ww()$ function, it turns out that, at any moment of time, a linked list $L(j)$ must always have at most p elements in it because p -line functions imply at most p different curves get the “highest” answer when you plot out the curves. So each list uses $O(p)$ space, and there are n of them. Hence $O(np)$ space used overall.

Memory/Space Reductions in Using Piecewise Linear Functions

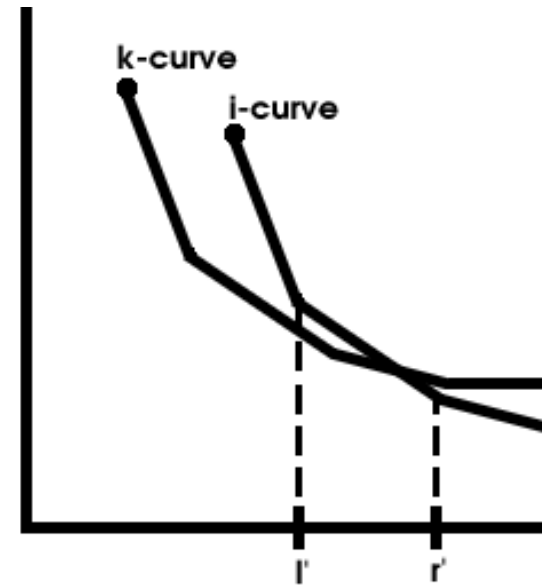
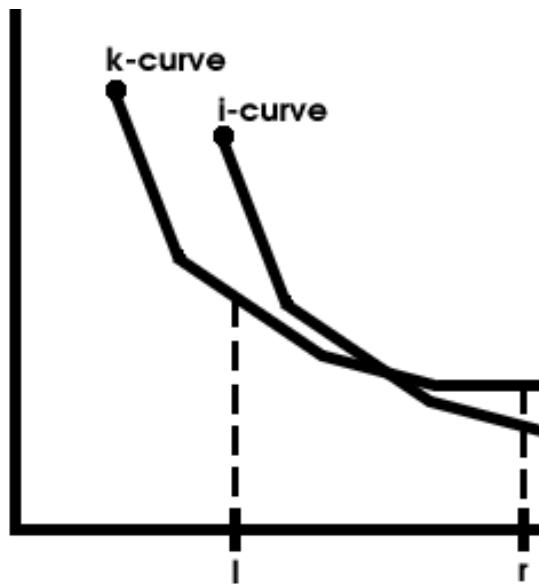
- Also, when updating a list $L(j)$, if we need to do binary search on one of its elements el to determine where curve-1 intersects curve-2, then... Instead of doing a binary search over the interval $el.l$ thru $el.r$, we can take advantage of the fact that our curves have p -lines.
 - We do binary search over the lines in curve-1 covered by $el.l$ thru $el.r$ to get the line on that curve of the intersection in question. Let's call the x -endpoints of that line l and r . (Note $el.l \leq l$ and $r \leq el.r$, so we're getting “closer to the answer.”)
 - We then do binary search on curve-2 over its lines covered by the l thru r to get the line on that curve of the intersection in question.
 - At this point know which lines on curve-1 and curve-2 are where the intersection happens. We can find the point of intersection of two lines in $O(1)$ time.

Memory/Space Reductions in Using Piecewise Linear Functions

- Binary search over the lines of curve-1 and curve-2 takes $O(\log p)$ time, since both curves have p lines.
- Hence, our overall computation time reduces from $O(mn \log m)$ time to $O(mn \log p)$ time.

Memory/Space Reductions in Using Piecewise Linear Functions

Binary search over the lines in the two curves:



Bridging the Gaps for Gapped Alignments

- As mentioned earlier, when using non-linear gap functions, the Hirschberg/MM linear space reduction runs the risk of not being accurate, particularly due to having the forward and backward tables being computed independently and the fact that $w(i + j)$ isn't always $w(i) + w(j)$ in bridging horizontal gaps. This case gets worse harmful when the optimal alignment involves X against a really really gap, with the gap starting in the first half of X , and ending in the second half. So, what to do?
- Suppose that we save the t most recently computed entries in computing V , V_r , F , and F_r plus corresponding linked lists L and L_r (the Hirschberg/MM assumption). And, suppose all forward tables get completed before we begin the reverse tables. Then...

Bridging the Gaps for Gapped Alignments

- For all j from 0 to n , at the time we finished computing V and F , because we made $m/2$ iterations, linked list $L(j)$ contains the best alignments involving gaps for all i from $m/2$ to m , where the gaps began at i' less than $m/2$. Therefore, there's no purpose in throwing away $L(j)$ when hopping over to the reverse table computations.
- So, as we go thru computing V_r , F_r , and L_r , we maintain an entry $cr(j)$ for the best solution involving X vs. gap for row j , and use L to correctly consider “combined-score solution” $V(i,j) + V_r(i',n-j) + ww(m-i'-i)$ for $cr(j)$. (Forward table and reverse table entry plus penalty for horizontal gap going between them...) $cr(j)$ also saves indices i and i' . (We still only save the t most recent columns in computation...) Hence, after we finish computing the reverse tables $cr(j).score$ is the best “combined-score solution”, and $cr(j).l$ and $cr(j).r$ denote the left/right endpoints for the bits of X corresponding to the best horizontal gap for this solution.

Bridging the Gaps for Gapped Alignments

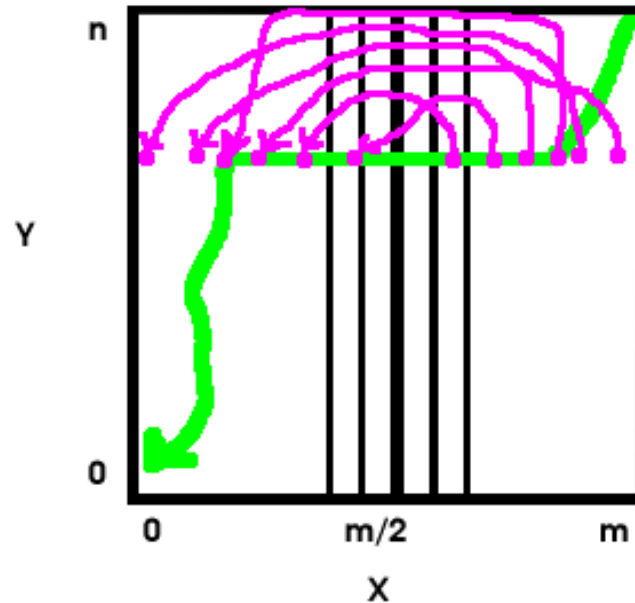
- Then in Hirschberg/MM, our EVALUATION CRITERIA is find k such that $\max\{V(m/2,k)+V(m/2,n-k), cr(k).score\}$ is maximized.
 - In the event the first of the two in the $\max\{, \}$ function is chosen, we get an alignment without a horizontal gap over the middle bits of X (which is similar to the typical linear gap penalty alignment assumption).
 - And when the latter of the two in $\max\{, \}$ is chosen, we have a horizontal gap using the middle bits of X , and can then use $cr(k).l$ and $cr(k).r$ to get the endpoints for the bits of X corresponding to this gap (which is then used to help construct the alignment).

Bridging the Gaps for Gapped Alignments

- By evaluation midpoint horizontal gaps independently of V and V_r , and using it in the EVALUATION CRITERIA of k , we obtain the alignment that would be seen if we used quadratic space (or an alternate alignment that ends up having the exact same score).

Bridging the Gaps for Gapped Alignments

- Using a Linked List to account for gaps independent for forward/reverse table computations.



Gap Parameter Optimization

- Using the assumption that log functions give the best alignments for DNA sequences, PLAINS models piecewise-linear functions to resemble log functions (within some epsilon of error). And shown before, piecewise-linear functions allow significant reductions in runtime and space over general gap functions, so this is to be a good idea.
- All log gap functions $l(i)$ can be generalized as $\alpha \cdot \ln(i+1) + \beta$, where α and β are positive real constants, and \ln is log using base e . (And we take $\ln(i+1)$ instead of $\ln(i)$ so that the y-intercept is β ...)
- In converting $l(i)$ to piecewise-linear $ww(i)$, we use x-value “evaluation points” such that, for each of these points i' , $l(i') = ww(i')$. In $ww()$, we set the slopes so that they connect the y-values of these “evaluation points.”

Gap Parameter Optimization

- In optimizing the $w_w()$ function, Plains assumes p is some fixed value (giving us quadratic time and linear space).
- Plains sets the evaluations points using some d value. Once a d value is chosen, the evaluations points are $x=d, 2d, \dots, p*d$. It turns out that varying d varies the alignments obtained.
- Furthermore, varying the mismatch penalty m_s also varies the alignments obtained.
- In summary, Plains uses fixed p and optimizes four variables: α , β , d , and m_s . α and β dictate the log function $l()$, d dictates the evaluation points the $w_w()$ function uses to approximate $l()$, and m_s is the mismatch penalty (which may or may not be smaller than the gap penalty, depending on the situation).

Gap Parameter Optimization

- Although varying four variables varies the alignments obtained, how do we decide on the quality of the alignment?
- Well, on a given set of values for alpha, beta, d, and ms, PLAINS:
 - First discard leftmost and rightmost bits of X and Y that would align against gaps BEFORE performing the alignment. This serves to assure the alignment better focuses on matches. (Let m' and n' denote the lengths of X and Y after the necessary leftmost and rightmost bits are removed.)
 - For some fixed constant N (typically 50), PLAINS creates m'/N blocks, the first corresponding to the first N bits of X, and second corresponding to the next N bits of X, etc. For the i th block, let $px(i)$ denote the percentage of bits of X in that block that the alignment found as a match.

Gap Parameter Optimization

- Similarly, PLAINS creates n'/N blocks, each corresponding to different continuous N bits of Y . And for the i th block, $py(i)$ is the percentage of bits of Y in that block that the alignment found as a match.
- Let ppx be the average of each $px()$ value, and let ppy be the average of each $py()$ value. The alignment is given a quality value = $(ppx + ppy)/2$
- The alignment of highest quality value is considered optimal. The quality value favors alignments with matches distributed throughout X and Y , instead of all being in one area.
- The optimization of the quality value over our four variables α , β , d and ms is done using the ideas from the Ameoba algorithm outlined in “Numerical Recipes.”

Gap Parameter Optimization

- Except for the few tossed-out leftmost and rightmost bits of X and Y, PLAINS assumes most of the bits of both sequences are intended to participate in the alignment, this is why the quality value was done in the way described here.
- In order to prevent PLAINS from throwing out too many leftmost/rightmost bits in X and Y, PLAINS will sometimes maximize quality value * $(m'/N + n'/N)$ instead of just the quality value alone.

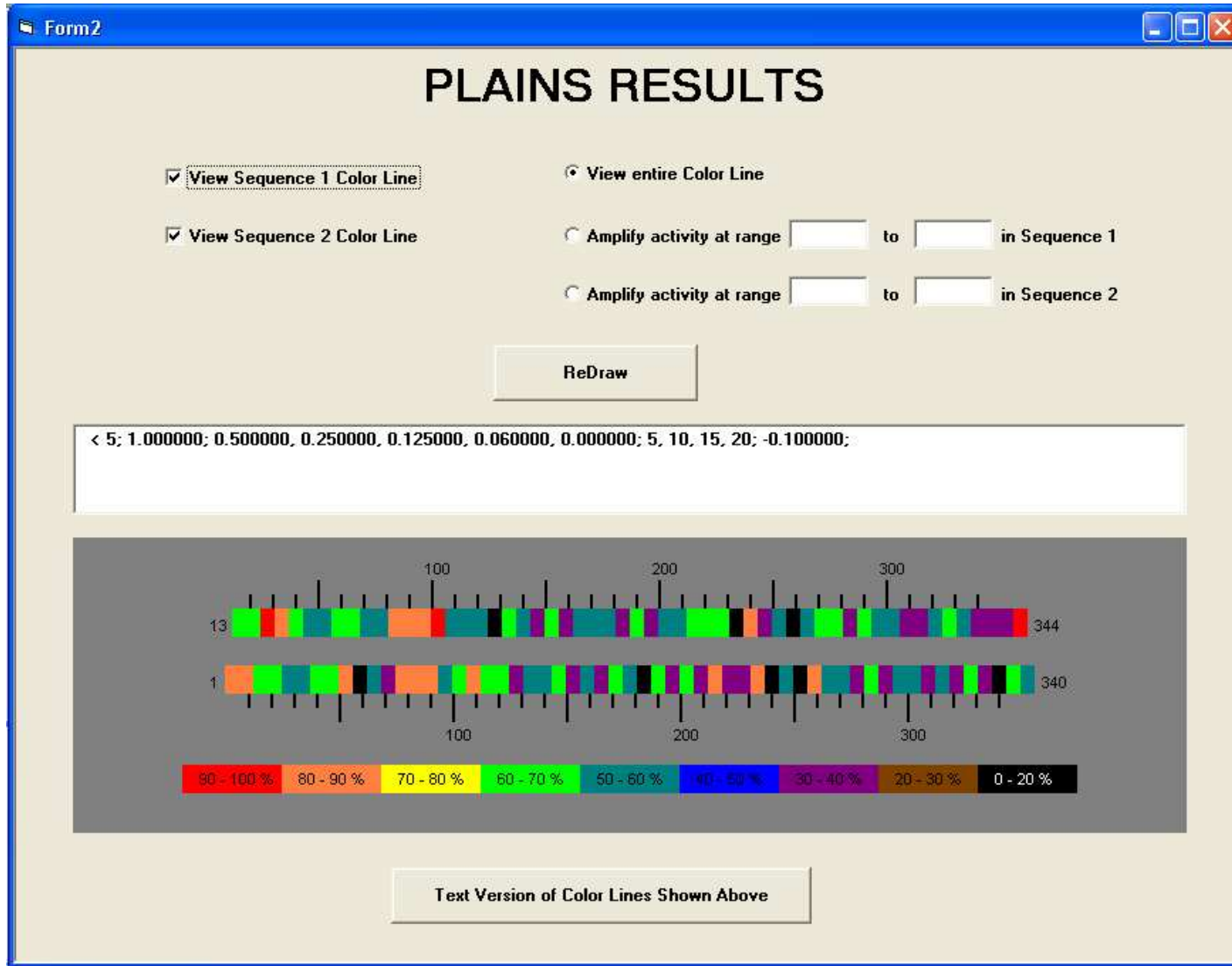
ColorGrid Generation

- For the PLAINS version used in Valis, users can view the match ratios of X and Y with respect to the alignment in a visually beautiful ColorGrid.
- For some fixed M (typically 50), the grid for X works by plotting m'/M different colors in M different adjacent spots. Going left to right, the i th color corresponds to the percentage of bits in $X[(i-1)*(m'/M)...i*(m'/M)]$ that got matched in performing the alignment. This is sort of like the $px()$ idea used in evaluating the quality scores, except that there, each spot drew from a fixed # of bits from X, and the # of spots was variable. For the ColorGrid, each spot draws from a variable # of bits from X, and the # of spots is fixed (due to screen resolution limitations).
- The grid for Y works similar in idea to the grid for X.

ColorGrid Generation

- Users may change the range of bits represented by the X and Y Colorgrids. This lets users zoom in on certain areas to better see how the matches came out without explicitly looking at the gigantic alignment itself.
- And, whenever the viewing range of bits for one sequence is changed, the appropriate corresponding viewing range on the other sequence is also chosen (and this can be found based on the alignment).

ColorGrid Generation



(4) Lab Results

- A number of tests were performed on PLAINS, and similar alignments tools for pairs of DNA of up to 4000 nucleotides: LALIGN, EMBOSS, and ALION.
- Each test involved a pair of sequences from human and mouse genes.
- The test sequences ranged from those of near perfect similarity (ABCB9) to those of less than 50% similarity (human Homolog vs. mouseHomolog)
- The test sequences ranged from those with 300 nucleotides (5_McrBC) to those with 4000 nucleotides (human K-Channel vs. mouse RTK)
- For each test, the alignment each tool produced was evaluated based on its quality value * $(m'/N + n'/N)$. Tables on the next few pages show “quality value” | “ $(m'/N + n'/N)$.” The winning tool for each test is colored in red text.

Lab Results

ABCB9 results

Test Name	Alignment Details	PLAINS score	LALIGN/PLALIGN score	EMBOSS Water score	ALION score
ABCB9_12	ENST00000280559.2 ENSG00000150967.3 vs. ENST00000280560.2 ENSG00000150967.3	0.971957 89	0.971957 89	0.971957 89	0.971957 89
ABCB9_13	ENST00000280559.2 ENSG00000150967.3 vs. ENST00000325185.1 ENSG00000150967.3	0.877103 60	0.886178 58	0.871913 62	1.000000 18
ABCB9_23	ENST00000280560.2 ENSG00000150967.3 vs. ENST00000325185.1 ENSG00000150967.3	0.876763 60	0.883990 55	0.866670 60	1.000000 18

Lab Results

FRS3, Kcnk4, TAP1 results

Test Name	Alignment Details	PLAINS score	LALIGN/PLALIGN score	EMBOSS Water score	ALION score
FRS3	ENST00000259748 vs. ENSMUST00000024034	0.829463 58	0.817739 58	0.818428 58	0.787394 58
Kcnk4	ENST00000308892 vs. ENSMUST00000025908	0.842636 47	0.823901 47	0.832076 47	0.819643 47
TAP1	ENST00000320763.2 ENSG00000168394.3 vs. ENSMUST00000041633.1 ENSMUSG00000037321.1	0.818187 86	0.791649 86	0.797230 86	0.788393 86

5_McrBC_boundary_377 results

Test Name	Alignment Details	PLAINS score	LALIGN/PLALIGN score	EMBOSS Water score	ALION score
5_McrBC(111 vs 205)	111 vs. 205	0.590240 14	0.612901 2	0.574996 14	0.389526 12
5_McrBC(1024 vs 2048)	1024 vs. 2048	0.617473 14	0.692941 4	0.543457 14	0.365021 14
5_McrBC(1492 vs 1776)	1492 vs. 1776	0.615291 10	0.613462 4	0.550830 12	0.366426 14

Lab Results

humanKchannel vs. mouseRTK results

Test Name	Alignment Details	PLAINS Score	LALIGN/PLALIGN score	EMBOSS Water score	ALION score
hmK1	ENST00000298972.1 ENSG00000166006.1 vs. ENSMUST00000053393.1 ENSMUSG00000028072.1	0.543813 76	0.639384 7	0.551375 75	0.358284 70
hmK2	ENST00000298972.1 ENSG00000166006.1 vs. ENSMUST00000049672.1 ENSMUSG00000028072.1	0.615124 79	0.639384 7	0.549123 79	0.357861 72
hmK3	ENST00000298972.1 ENSG00000166006.1 vs. ENSMUST00000029713.1 ENSMUSG00000028072.1	0.617398 78	0.639384 7	0.547342 78	0.360198 72
hmK4	ENST00000298972.1 ENSG00000166006.1 vs. ENSMUST00000029712.1 ENSMUSG00000028072.1	0.608504 82	0.563876 22	0.548796 84	0.361252 72
hmK5	ENST00000263372.1 ENSG00000099337.1 vs. ENSMUST00000053393.1 ENSMUSG00000028072.1	0.630907 59	0.600486 14	0.564175 56	0.399509 36
hmK6	ENST00000263372.1 ENSG00000099337.1 vs. ENSMUST00000049672.1 ENSMUSG00000028072.1	0.507006 60	0.592992 24	0.568675 49	0.378579 36
hmK7	ENST00000263372.1 ENSG00000099337.1 vs. ENSMUST00000029713.1 ENSMUSG00000028072.1	0.555818 37	0.571686 30	0.564002 46	0.384351 36
hmK8	ENST00000263372.1 ENSG00000099337.1 vs. ENSMUST00000029712.1 ENSMUSG00000028072.1	0.473252 65	0.618221 10	0.556955 64	0.391364 36

Test Name	Alignment Details	humanHomol vs. mouseHomol results			
		PLAINS Score	LALIGN/PLALIGN score	EMBOSS Water score	ALION score
hmHomol1	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000054934.1 ENSMUSG00000015711.1	0.602781 29	0.574405 16	0.563539 39	0.383670 28
hmHomol2	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000061644.1 ENSMUSG00000015711.1	0.650563 65	0.570130 45	0.551929 64	0.377675 46
hmHomol3	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000015855.1 ENSMUSG00000015711.1	0.555657 59	0.573338 28	0.562868 70	0.364252 54
hmHomol4	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000021748.1 ENSMUSG00000021311.1	0.635373 93	0.592903 14	0.543280 112	0.362714 88
hmHomol5	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000021747.1 ENSMUSG00000021311.1	0.642750 116	0.592903 14	0.560178 110	0.381152 88
hmHomol6	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000043502.1 ENSMUSG00000021311.1	0.557470 115	0.592903 14	0.560175 109	0.360800 89
hmHomol7	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000021746.1 ENSMUSG00000021311.1	0.631675 106	0.592903 14	0.553961 114	0.358819 88
hmHomol8	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000053346.1 ENSMUSG00000021311.1	0.582116 117	0.592903 14	0.543648 116	0.356754 88
hmHomol9	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000021744.1 ENSMUSG00000021311.1	0.630010 85	0.592903 14	0.547691 110	0.369185 88
hmHomol10	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000021745.1 ENSMUSG00000021311.1	0.520469 91	0.592903 14	0.547691 110	0.376902 88
hmHomol11	ENST00000327711.2 ENSG00000182473.2 vs. ENSMUST00000047611.1 ENSMUSG00000041429.1	0.544972 44	0.572299 27	0.554358 45	0.361517 34

Lab Results

- From these results, clearly the only adversary to PLAINS is EMBOSS. PLAINS gives a better result than EMBOSS at least 60% of the time. Whenever PLAINS doesn't give the best result, it comes second to EMBOSS.
- Furthermore, when PLAINS doesn't give the best result, it still comes awfully close to EMBOSS.
- In analyzing the test results results more closely, PLAINS and EMBOSS clearly outperform LALIGN and ALION at least 90% of the time.
- The success of PLAINS shows some promise in its use in industrial-style alignments.

(5) Conclusions / Future Work

- Although Plains runs slower than other tools, it runs a constant factor slower. This has to do with the fact that Hirschberg/MM Linear Space reduction and Miller-Myers Forward Gap Search add on constant factors to the overall runtime. The tools PLAINS was pitted against either use much simpler tactics, or have hidden databases and data structures for quick lookups.
- Also optimizing gap/mismatch parameters naturally take longer than using a prespecified set of gap/mismatch parameters due to the # of extra evaluations needed. PLAINS allows users to do alignments both of these ways (so the user can decide if he wants better results, or just plain quick results).
- For 4000 nucleotide sequences running on a regular Pentium II-600 Mhz machine, PLAINS runs for 5 minutes when you give it a prespecified set of gap/mismatch parameters, and 2 hours when you ask it to optimize gap/mismatch parameters and align with the best set it finds. Not too bad all things considered...

(5) Conclusions / Future Work

- With PLAINS working well on 4000 nucleotide sequences in comparison to similar tools, the next natural step is to scale PLAINS upwards to allow for larger sequences.
- As PLAINS stands, it wouldn't be practical to use it for million nucleotide sequences. It's apparent extra modifications such as searching sequences for high regions of matches, and combining smaller alignments into bigger one will certainly enter the picture here.
- Furthermore, it might be helpful to add scoring matrices to PLAINS if, for example, we'd like C-T matches to be scored differently than C-G matches. (At the moment, PLAINS gives all matches a score of 1, and all mismatches a score of ms.)

Still awake?
This is the end of the talk!