

Alignment Theory and Applications

By Ofer H. Gill
November 5, 2003

Alignment Theory

- (1) Why Align?
- (2) Sequence Alignment: An Intuition
- (3) Why PL (Piecewise-Linear)?
- (4) Reducing to Linear Space
- (5) Forward-Gap Searching Algorithm
- (6) PLALIGN's Algorithm

Section 1: Why Align?

DNA Alignment

- When two organisms have a common continuous region of base pairs (and sometimes even subcontinuous), it often implies that those organisms have a shared trait.
- Similarly, when a continuous/subcontinuous region of base pairs is present in one organism, but not the other, it often implies a trait present in one organism, but not the other.
- Common regions and inserted/deleted regions can be revealed by alignment.

DNA Alignment

- When an organism evolves, base pairs of DNA get deleted, inserted, or substituted.
- Therefore, in comparing DNA sequence of different species, any base pairs of DNA that are inserted, deleted, or substituted reveals evolutionary information for those species. (For example, human vs. chimp.)
- For this reason, alignment can help us trace the evolution of various species.

DNA Alignment

- In determining the DNA of a whole genome, you only get to sequence small fragments (~500bp) with the state-of-the-art technology.
- Aligning overlapping fragments is the “glue” in learning the entire sequence from these fragments. (Though you have to be careful in regards to how you glue due to repeats...)

Protein Alignment

- When two proteins have a common continuous or subcontinuous region of amino acids, it often implies that those proteins have the same secondary structure (and possibly even tertiary). Aligning protein sequences is useful in revealing this information.
- In some applications (like deciding if a certain medicine for mice works on humans), protein alignment may be preferred to DNA alignment, since protein sequences are typically shorter, and correspond more closely to known important organism functions. However, protein alignment sometimes misses critical information that DNA alignment reveals (framing problem).

Additional Notes

- This presentation deals mainly with local (Smith-Waterman) alignment of substrings, not global (Needleman-Wunsch) alignment of an entire sequence. We generally would like to align parts of two sequences that are highly similar (otherwise we're just aligning random junk, and the result isn't anything biologically meaningful).
- We can construct global alignments a number of ways:
 - Search for highly related regions in our sequences, local align those, then combine local alignments with an algorithm for nonsimilar regions to make a global alignment.
 - DNA sequences may have duplications, tandem repeats, and reversals. Finding such areas are biologically meaningful. We can use string matching algorithms to catch such things, and then proceed with local alignments over highly similar areas as stated above.

Section 2:
Sequence Alignment: An Intuition

Why Dynamic Programming?

- Suppose X and Y are our inputs, and we seek a number called $FOO(X, Y)$.
- And, suppose that it's possible to conclude the $FOO(X[1..i], Y[1..j])$ based on things such as:
 - $FOO(X[1..i-1], Y[1..j-1])$
 - $FOO(X[1..i], Y[1..j-1])$
 - $FOO(X[1..i-1], Y[1..j])$
 - $FOO(X[1..i/5], Y[1..j/2])$
 - $FOO(X[1..i/2], Y[1..j/5])$
 - Etc....

Why Dynamic Programming?

- If for all $i' < I$, and $j' < j$, $\text{FOO}(X[1..I'], Y[1..j'])$, $\text{FOO}(X[1..I], Y[1..j'])$, and $\text{FOO}(X[1..I'], Y[1..j])$ are known and saved in memory, we can get $\text{FOO}(X[1..i], Y[1..j])$ in time proportional to how long it takes to look up the necessary and previously found FOO entries.
- Furthermore, once $\text{FOO}(X[1..i], Y[1..j])$ is computed, we can save that in memory to help us compute later FOO values and prevent recomputing $\text{FOO}(X[1..i], Y[1..j])$ again at a later point.

Why Dynamic Programming?

- Dynamic Programming is based on the idea of saving extra information in memory to prevent recomputations, hence speeding up computation time.
- Dynamic Programming is one way of solving FOO, but depending on what FOO is, there may be other ways that are equally effective or better... (Binary trees, Heaps, Linked Lists, Google search, Bud Mishra, etc.)

Longest Common Subsequence

- Given two strings X and Y for lengths m and n , we wish to find the longest subsequence they have in common. And, let $V(i, j)$ denote our result, over $X[1\dots i]$ and $Y[1\dots j]$.
- Formula is:
 - $V(i, 0) = 0$, for all $i \geq 0$
 - $V(0, j) = 0$, for all $j \geq 0$
 - And for all $i > 0$ and $j > 0$,
 - $V(i, j) = 1 + V(i-1, j-1)$, if $X[i] == Y[j]$
 - $V(i, j) = \max\{V(i-1, j), V(i, j-1)\}$, if $X[i] != Y[j]$

Longest Common Subsequence

t	10	0	0	1	1	2	3	4	4	4	↙5
y	9	0	0	1	1	2	3	↙4	←4	←4	5
q	8	0	0	1	1	2	↙3	3	3	4	5
b	7	0	0	1	1	↓2	2	2	3	4	5
t	6	0	0	1	1	↓2	2	2	3	4	5
t	5	0	0	1	1	↓2	2	2	3	4	5
a	4	0	0	1	1	↓2	2	2	3	4	4
z	3	0	0	1	1	↓2	2	2	3	3	3
c	2	0	0	1	←1	↙2	2	2	2	2	2
b	1	0	0	↙1	1	1	1	1	1	1	1
b	1	0	←0	0	0	0	0	0	0	0	0
	0	0	1	2	3	4	5	6	7	8	9
			g	b	e	c	q	y	z	a	t

Longest Common Subsequence

- We can, in addition to computing max function, save how we got the max function. This gives us the arrows in the dynamic table.
- Computing the table numbers and arrows takes $O(mn)$ time and $O(mn)$ space. (Quadratic time and space.)
- Tracing the arrows to obtain the $LCS(X, Y)$ afterwards takes $O(m + n)$ time. (This is no big deal.)

Edit Distance

- Given X and Y , and assuming it takes one operation to perform an insertion, deletion, or substitution, we wish to find the minimum number of operations to transform X into Y , also known as the edit distance from X to Y .
- Edit distance from X to Y is also the edit distance from Y to X , since a deletion on one string corresponds to an insertion on the other, and vice versa.

Edit Distance

- Let $V(i, j)$ denote our answer for $X[1\dots i]$ and $Y[1\dots j]$, then the Formula is:
 - $V(i, 0) = i$, for all $i \geq 0$
 - $V(0, j) = j$, for all $j \geq 0$
 - And for all $i > 0$ and $j > 0$,
 - if $X[i] == Y[j]$, then $V(i, j) = V(i-1, j-1)$
 - if $X[i] != Y[j]$, then
 - $V(i, j) = 1 + \min\{V(i-1, j), V(i, j-1), V(i-1, j-1)\}$

Edit Distance

t	10	10	10	9	9	9	9	8	8	9	↙9
y	9	9	9	8	8	8	8	7	8	↓9	8
q	8	8	8	7	7	8	7	8	8	↓8	7
b	7	7	7	6	7	7	7	7	7	↓7	6
t	6	6	6	6	6	6	6	6	6	↓6	5
t	5	5	5	5	5	5	5	5	6	↓5	4
a	4	4	4	4	4	4	4	5	5	↙4	5
z	3	3	3	3	3	3	3	4	↙4	5	6
c	2	2	2	2	2	↙2	←3	←4	5	6	7
b	1	1	1	↙1	←2	3	4	5	6	7	8
	0	0	←1	2	3	4	5	6	7	8	9
	0	0	1	2	3	4	5	6	7	8	9
			g	b	e	c	q	y	z	a	t

Edit Distance

- Just like LCS, we can save arrows with the table entries, to compute the table in $O(mn)$ time, and trace a solution for the table in $O(m + n)$ time. (So we use quadratic time and space overall.)

Edit Distance

- Edit distance of X and Y is like an alignment between X and Y , where:
 - **Insertion** corresponds to a character of X aligned with a gap (dashed character).
 - **Deletion** corresponds to a character of Y aligned with a gap.
 - **Substitution** corresponds to two different characters of X and Y aligned with each other.
 - **Matches** corresponds to two matching characters of X and Y aligned with each other.

Edit Distance

- For the example X and Y given, (X = gbecqyzat, Y = bczattbqyt). The edit distance table gives us the following alignment for X and Y show below.

```
g b e c q y z a - - - - - t
| | | | | | | |
- b - c - - z a t t b q y t
```

Sequence Alignment Observation

- An alternate way of thinking of edit distance is that we wish to inspect matches, mismatches and gaps for X and Y.
- Instead of computing an edit distance, we can create a scoring model to get the alignment, where one point is given for each match found in X and Y, and no points are given to mismatches and gaps. In this case, maximizing the score corresponds to minimizing the ED.
- We can tweak the scoring model for matches, mismatches, gaps in order to get different alignments of X and Y (allowing us to focus on various areas in X and Y).
- If finding the most matches in X and Y is our only issue, and we don't care about mismatches and gaps, then LCS is our answer. (But in practice, we DO care about block matches and gaps!)

Sequence Alignment Observation

- When aligning, the matches we seek, whether in DNA or proteins, typically occur in blocks (substrings), not at various discrete points. Therefore, we want a scoring scheme that aligns matches in blocks, not scattered around. We get this by deducting points from our score for any gaps, and the longer the gap, the larger the penalty (hence encouraging blocks of matches). Therefore, it's typical to model a function for gaps, where this function's penalty value increases with a gap's length.
- Furthermore, longer gaps often reveal important differences between proteins and DNA. To allow for this in our alignment, we decrease the extra penalty for each length increase in the gap. (So, for example, we could have the extra penalty from going from a 200,000 to a 200,001 length gap be smaller than the extra penalty in going from a 2 to a 3 length gap.)
- Combining the two things mentioned above, we get that a gap function whose penalty increases with length, but the rate of increase decreases. (In other words, positive derivative, and negative double-derivative.)

The General Gap Alignment Model

- Behaves much like LCS and ED, except we keep track of more at each table entry.
- $E(i, j)$ denotes the score if we align $Y[j]$ against a gap.
- $F(i, j)$ denotes the score if we align $X[i]$ against a gap.
- $G(i, j)$ denotes the score if we align $X[i]$ with $Y[j]$ (whether or not they are equal to each other).
- $V(i, j)$, our score for $X[1..i]$ and $Y[1..j]$ is set to whichever of $E(i, j)$, $F(i, j)$ or $G(i, j)$ is highest.
- For simplicity, I'll assume we get one point if $X[i]$ and $Y[j]$ match, zero points if they mismatch. (It's common to assume this, but we can later change the points for these if you like...)

The General Gap Alignment Model

- A gap is penalized based on how long it is. Let $w(i)$ denote the nonnegative penalty given for a gap of length i . (w is some math function.)
- For reasons discussed earlier, $w(i)$ will typically increase as i increases, but the rate of increase lowers as i increase (in some cases, the curve for $w(i)$ even eventually flattens out as i increases).

The General Gap Alignment Model

- Our score function V is hence derived as follows:
 - $V(0, 0) = 0$
 - $V(i, 0) = E(i, 0) = -w(i)$
 - $V(0, j) = F(0, j) = -w(j)$
 - $V(i, j) = \max\{E(i, j), F(i, j), G(i, j)\}$
 - $G(i, j) = V(i-1, j-1) + s(X[i], Y[j])$
 - (Note: $s(X[i], Y[j]) = 1$ if $X[i] == Y[j]$, 0 otherwise)
 - $E(i, j) = \max_{0 \leq k \leq j-1} [V(i, k) - w(j-k)]$
 - $F(i, j) = \max_{0 \leq k \leq i-1} [V(k, j) - w(i-k)]$

The General Gap Alignment Model

- The algorithm described here uses $O(mn)$ space. To compute $V(m, n)$, we look at $m + n + 1$ previously computed values. (Hence, our lookbehind size for each entry in the V table is $O(m + n)$.) Therefore, our overall runtime is $O(mn * (m + n)) = O(m^2n + mn^2)$. (Cubic runtime and quadratic space.)
- The lookbehind size for each entry in V for LCS and ED is $O(1)$.

The General Gap Alignment Model

- However, quadratic space and cubic runtime for general gap formula w is pretty large. Can we do better?
- If we restrict w , this answer is yes.

Linear Gap Formula Alignment

- When $w(i)$ is a linear formula (of form $w_c i + w_o$), we have a way to reduce runtime by reducing the number of lookbehinds.
- In this case, the gap penalty starts at some value w_o and increases at a constant rate of w_c for each new increase in the gap length.
- Here, because the gap penalty always increases by w_c once the gap is longer than one, we don't need to worry where a gap begins; only if it already began, or a new gap is started.

Linear Gap Formula Alignment

- Our formula, now becomes the Gotoh formula of:
 - $V(0, 0) = 0$
 - $V(i, 0) = E(i, 0) = -w_o - i * w_c$
 - $V(0, j) = F(0, j) = -w_o - j * w_c$
 - $V(i, j) = \max\{E(i, j), F(i, j), G(i, j)\}$
 - $G(i, j) = V(i-1, j-1) + s(X[i], Y[j])$
 - $E(i, j) = -w_c + \max\{E(i, j-1), V(i, j-1) - w_o\}$
 - $F(i, j) = -w_c + \max\{F(i-1, j), V(i-1, j) - w_o\}$

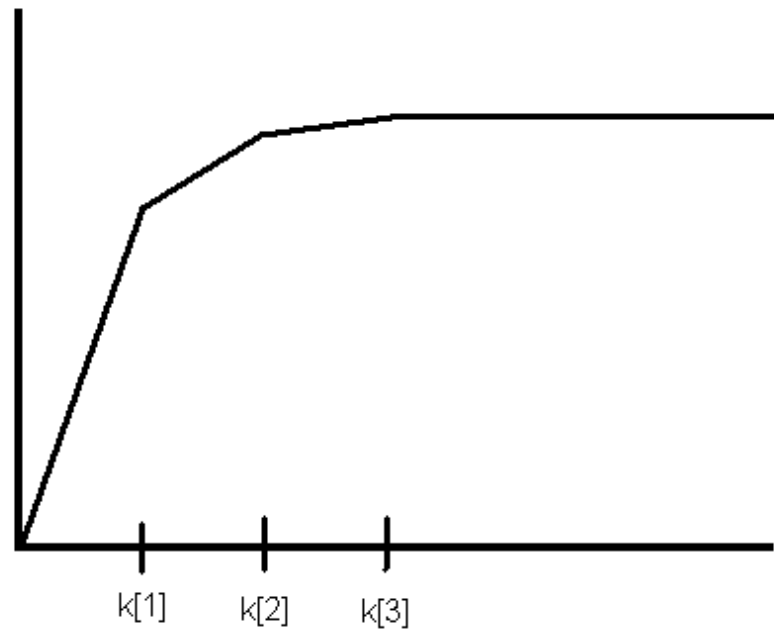
Linear Gap Formula Alignment

- Here, we see that each entry in V is computed using $5 = O(1)$ lookbehinds. Therefore, the overall runtime is $O(mn)$. The space used is still $O(mn)$. Hence, we still use quadratic space, but have reduced our runtime to quadratic time.
- Problem: A linear gap function w , in spite being easy to compute, sacrifices a key feature, the ability to decrease the rate of gap penalty increase as our gap gets larger. Is there a way around this?

Section 3: Why PL (Piecewise-Linear)?

Piecewise-Linear Gap Formula Alignment

- Piecewise-Linear Gap functions look something like the following:



Why PL (Piecewise-Linear)?

- One might note that the Linear Gap Formula Alignment was easier to compute than the General Gap Formula Alignment.
- Piecewise-Linear functions allow us to emulate a general function using a sequence of lines. And, the Piecewise-Linear Gap formula has the same ease of computation as the Linear Gap formula (or at least within a factor of p , where p is the number of lines in the Piecewise-Linear Gap formula).
- Therefore, assuming we're willing to sacrifice some accuracy in gap functions (which CAN be fine-tuned by varying p), piecewise-linear gap functions achieve the same effect as general gap functions, but more efficiently.

Piecewise-Linear Gap Formula Definition

- Let $w(x)$ denote a Piecewise-Linear Gap function of length x .
- Suppose there are p different slopes to this formula, and w_0 is the cost of opening a gap (the y-intercept), $wc[i]$ is slope of the i th line in the w formula, and $k[i]$ is the x-value where the i th line ends (and assume that $k[0] = 0$ and $k[p] = \text{infinity}$, as this should make sense...).
- Next, assume that $t[i]$ denotes y-values corresponding to each $k[i]$. Then:
 - $t[0] = w_0$
 - $t[1] = w_0 + (k[1] - k[0]) * wc[1] = t[0] + (k[1] - k[0]) * wc[1]$
 - $t[i] = w_0 + \sum_{u=1 \text{ to } i} ((k[u] - k[u-1]) * wc[u]) = t[i-1] + (k[i] - k[i-1]) * wc[i]$
- The entries of t can hence be found in a forward manner using w_0 , and the entries of k and wc in a total of $O(p)$ time and space.

Piecewise-Linear Gap Formula Definition

- From this, we can compute $ww(x)$ as follows:
 - If for some i , $k[i] \leq x < k[i+1]$, then:
 - $ww(x) = w_0 + [\sum_{u=1 \text{ to } i} ((k[u] - k[u-1]) * wc[u])] + (x - k[i]) * wc[i+1] =$
 - $= t[i] + (x - k[i]) * wc[i+1]$
 - For a given x , to get the i such that $k[i] \leq x < k[i+1]$, we'll use a hashtable h where, when $k[i] \leq x < k[i+1]$, then $h[x] = i+1$.
 - And hence, $i = h[x] - 1$.
 - Also, on the side, note that for all i , $ww(k[i]) = t[i]$.

Piecewise-Linear Gap Formula

- Let $H+1$ be the hashtable size. We can compute entries $h[0]$ up to $h[H]$ in $O(H)$ time and space at the beginning of our algorithm (before computing V), hence saving time later on for when we need to obtain $ww(x)$ for an arbitrary x .
- Hence, for any piecewise-linear function with p lines, characterized by y -intercept w_0 , an array of slopes wc , and an array k denoting x -values where each line ends, we can compute the needed t and h entries in $O(p + H)$ time and space.
- Then from w_0 , wc , k , t , and h , we compute $ww(x)$ in constant time by doing:
 - Let $i = h[x] - 1$, then
 - $ww(x) = t[i] + (x - k[i])*wc[i+1] = t[h[x] - 1] + (x - k[h[x] - 1])*wc[h[x]]$
- For a given X and Y of lengths m and n (with $m \geq n$), assume that any gap larger than n lies on the p th line. Therefore H can be set to n . ($h[x] = p$ whenever $x > n$, so we don't need to explicitly save more than n entries of h .) Hence, letting us compute t and h entries in $O(p + n)$ time and space.

Piecewise-Linear Gap Formula

- Also, it's common practice to use $p < n$. With this in mind, if we use the General Gap Formula, where we substitute $w(x)$ with $ww(x)$, we can get the same result in the same time and space (cubic time and quadratic space). For the moment this is clearly no improvement over using a general gap formula $w(x)$. However, $ww(x)$ is a key stepping stone, and the next few sections will illustrate how to cut down the time and space needed based from $ww(x)$...

Section 4: Reducing to Linear Space

Reducing to Linear Space

- If, for the LCS, ED, or Linear Gap Alignment problems, we sought only $V(m, n)$, not an actual alignment, we could easily reduce to $O(n)$ space and $O(mn)$ time by only keeping the two most recently computed columns of V (and E , F , and G).
- (For the moment, I'll put my attention into the linear gap model) Hirschberg has a way to obtain an alignment from the Linear Gap Alignment problem in $O(n)$ space and $O(mn)$ time. To do this, we note the following:
 - For X and Y , let X^r and Y^r denote the reversals of X and Y , and let $V^r(i, j)$ denote the score for $X^r[1..i]$ and $Y^r[1..j]$. We can compute V^r in the same way as V , and $V^r(i, j)$ gives us the alignment of the last i characters of X with the last j characters of Y .

Reducing to Linear Space

- Then, we can compute $V(m/2, n)$ and $V^r(m/2, n)$. And, we note that
 - $V(m, n) = \max_{0 \leq k \leq n} [V(m/2, k) + V^r(m/2, n-k)]$
- This means that computing $V(m/2, n)$ and $V^r(m/2, n)$ allows us to find $V(m, n)$.
- Essentially, we split X into half, and look for a way to split Y such that the left part of Y aligns with the first half of X , and the right part of Y aligns with the second half. By finding the split of Y so that our calls to $V(m/2, n)$ and $V^r(m/2, n)$ are maximized, we essentially found the character in Y such that our optimal result in the table for $V(m, n)$ that goes thru the halfpoint (give or take 1) for X .

Reducing to Linear Space

- We record whatever alignment data we found from the calls to $V(m/2, n)$ and $V^r(m/2, n)$. If we use linear space in these calls (recording only the most recent columns), we only have the ending portion of the alignment from the $V(m/2, n)$ call, and the starting portion of the alignment from the $V^r(m/2, n)$ call. Once we found the correct point in Y to split, we can repeat ourselves recursively on the left parts of X and Y , and on the right parts of X and Y . This gives us the alignments for the rest of the bits of X and Y . We can then glue these results to get the optimal alignment for X and Y .

Reducing to Linear Space

- Hirshberg's OPTA algorithm (assuming we use at most t columns) works as follows:
 - $\text{OPTA}(l, l', r, r', t)$ {
 - 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 $(l + l' - 1 \leq t)$ 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;

Reducing to Linear Space

- $h = (l' - 1) / 2$;
 - 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. 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 .
- }

Reducing to Linear Space

- For the Linear Gap Model, we call $\text{OPTA}(1, m, 1, n, t)$ to get the alignment of X with Y .
- 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 $O(tn)$ space. (Quadratic time and linear space.)
- t (chosen to be between 2 and m) doesn't affect the runtime. Furthermore, if t is constant, we get linear space.

Fundamental Laws of Computers

- Let me step aside from our discussion for a moment to instill/install some Ofer words of wisdom. Here are the two Fundamental Laws of Computers:
- (First Fundamental Law of Computers) Backup your work often!
 - Have at least two copies of your work on different machines or media types (hard disks, floppy disks, key drives, remote servers, Internet accounts, etc.) at ALL times. Three copies is preferable. You never know when an unexpected problem creeps into your computer or disk or the Internet, erasing your data. The extra copies prevent you from starting from scratch, which is often costly... (Note: I've made a backup copy of this presentation 2 seconds after typing up the previous sentence, so as not to contradict myself.)
 - If this is tough to do, just remember every day to: Brush your teeth, floss, shower, backup your data! Eat, sleep, exercise, backup your data! Smile, laugh, love, relax, backup your data! Check your mail/email, say hello to your friends, shag your girlfriend, wife, or mistress, backup your data!

Fundamental Laws of Computers

- (Second Fundamental Law of Computers) One Product Never Fits All!
 - Ever got that latest antivirus program guaranteed to work on all machines, only to find out it works on all machines but yours? And to make matter worse, explaining the problem to the vendors of the software and your machine only ends up puzzling them because you did everything right, therefore it SHOULD work fine, but doesn't and they don't know why. (Or worse yet, the vendors of your software blame the problem on the vendors of your machine, and the vendors of your machine blame the problem on the vendors of your software.) Clearly the term “Device-Independent Software” is a big pile of baloney (with swiss cheese, in a kaiser roll topped with mustard, lettuce, and tomato).
 - Similarly the sorting algorithm, or binary-search algorithm, which should work in any case, doesn't quite give you the solution needed for YOUR problem, forcing you to tweak the algorithm. Hence one product never fits all!
 - Although the Linear Space reduction mentioned earlier works fine with Linear Gap Model, there are some problems that arise when applying it to the Piecewise-Linear Gap Model, forcing us to tweak it in order to fit...

What goes wrong when using PL in Linear Space?

- In linear space, two big problems affect piecewise linear functions, but not linear functions:
 - (1) The way we have the formulation for V in the piecewise-linear gap model set up, whenever we compute an entry in V , we need to look at some entry from ALL of the previous columns of V (in the linear gap model, we only need entries from the most recent column of V). But, if we want t to be constant (hence using linear space), it implies we've thrown away columns that we still need!!! But there's a way around this... Forward-Gap Searching (described in the next section) allows us to correctly use our piecewise-linear gap model while only requiring entries from the most recent column (plus as an added bonus, it reduces computation time).

What goes wrong when using PL in Linear Space?

(2) Even with the first problem solved, there's still the issue of gluing solutions between the V and V^r tables. Because most General Gap Functions (and Piecewise-Linear Gap Functions for that matter) have a negative double-derivative, this means that the extra gap cost of extending a gap decreases for larger gaps. Therefore, in OPTA, if we happen to have a gap that starts in the V portion, and ends in the V^r portion, (and this gap is length a in the V portion, and length b in the V^r portion), the max function for finding k^* in OPTA should "know" that this is one gap of size $a+b$, not two gaps of sizes a and b . ($w(a+b) \leq w(a) + w(b)$, so may have $w(a+b) < w(a) + w(b)$...) And, this could affect the correct selection of k^* , hence affecting the overall alignment. (Note: In linear functions, the extra gap cost for extending a gap is always the same value, regardless of how big the gap. That's why this problem doesn't apply to it.) There's a fix to this problem also. This one also uses Forward-Gap Searching, but in a way to "bridge" solutions in the V and V^r tables as they're being computed. (More on this later...)

Section 5:
Forward-Gap Searching Algorithm

Forward-Gap Searching Algorithm

- Suppose that we wish to align two sequences X and Y (of lengths m and n , with $m \geq n$), and we wish to do it using an arbitrary gap function w . (We'll see about piecewise-linear gap function w later...) For now, we won't worry about using $O(mn)$ space. The standard solution involves each table entry in V looking at all previous columns in the same row, and all previous rows in the same column, resulting in a $O(m^2 \cdot n)$ runtime.
- However, assuming w is a convex function, we can do faster than this. The speedup is based on the following assumptions: (For simplicity, assume i is fixed, so we compute $E(j)$ instead of $E(i,j)$)

Forward-Gap Assumption # 1

- First off, we compute candidates for each E entry in a forward manner instead of a backwards one (hence the term Forward-Gap Searching Algorithm).
- We do this by maintaining $\text{Cand_E}(j)$ and $\text{ind_E}(j)$ for all j . These values correspond respectively to the best $E(j)$ score found yet, and the value before j where this such most recent score corresponds to.
- Next, let $\text{cand}(k,j) = V(j) - w(j-k)$.
- Then, we can restate the prior general gap model's definition of:
 - $E(j) = \max_{(1 \leq k < j)} (V(k) - w(j-k))$ to instead be $E(j) = \max_{(1 \leq k < j)} (\text{cand}(k,j))$.
- Now, suppose we don't know which k gives the max $\text{cand}(k, j)$, but we have a k' that gives the best $\text{cand}(k', j)$ we found so far, then we set $\text{Cand_E}(j)$ to $\text{cand}(k', j)$ and $\text{ind_E}(j)$ to k' .
- So when we compute $E(j)$, then for all $jp > j$, we check if $\text{cand}(j,jp)$ is larger than $\text{Cand_E}(jp)$. If yes, we change $\text{Cand_E}(jp)$ to $\text{cand}(j,jp)$ and set $\text{ind_E}(jp)$ to j , otherwise we do nothing. This forward method doesn't change runtime yet, but it's a setup for what we're about to do next...

Forward-Gap Assumption # 2

- Next, suppose for some j and j_p , that: $\text{cand}(j, j_p) \leq E(j_p)$, then, for all $j_{pp} > j_p$, $\text{cand}(j, j_{pp}) \leq E(j_{pp})$.
- In other words, if j isn't the $\text{ind}_E(j_p)$ value, it's not the ind_E value for any $j_{pp} > j_p$. This is based on the convex-ness of w , and helps save us computations.

Forward-Gap Assumption # 3

- Next, note that Assumption #2 implies that if, at some point in the algorithm before computing E values beyond j, we were to look at the ind_E values for all $j_p > j$, we'd be seeing a list of gradually decreasing numbers. (Some books call it nonincreasing, but I find that term too confusing...) This list of gradually decreasing numbers can be split up into blocks, where the values of all the blocks are equal. Example:
 - The list: 5 5 5 5 5 4 4 4 3 3 3 3 3 3 2 2 1 1 1 1 1 is split into the blocks:
 - 5 5 5 5 5 | 4 4 4 | 3 3 3 3 3 | 2 2 | 1 1 1 1 1
- Next, to same time and space, instead of listing out all these blocks, we can simply make a doubly-linked list L, where each element in L represented a block and has three values (l, r, and v):
 - l is the index where the block begins
 - r is the index where the block ends
 - v is the value of ind_E for all entries in this block.

Forward-Gap Assumption # 3 (Continued...)

- Now, instead of maintaining `ind_E` arrays, `L` can be used instead. The advantage besides the space saved, is that for a given `jp` value (assuming we have instant access to its block in `L`), we have $O(1)$ access to the endpoints of its block (so we know where to look when we need the leftmost index of `E` that selects a different `ind_E` value for `Cand_E`).
- Similarly, we don't need to explicitly maintain `cand_E` values, since (provided we have correct entry in the `L` list) `ind_E` can be found from an `L` element's `v` values, and from `ind_E`, we can compute `cand_E` in $O(1)$ time (since we have $O(1)$ access to the `V` table and can compute a `w`-value in $O(1)$ time).

Forward-Gap Assumption # 4

- So now, note that if we found a $jp > j$ such that $\text{cand}(j, jp) > \text{cand_E}(jp)$, then the corresponding block full of jp values as the ind_E has to have its l value set to $jp+1$ (or this block has to be deleted entirely), and ALL blocks to the left of this block can be deleted from L , and replaced with one block with $l=j+1$, $r=jp$, and $v=j$.
- This will help us cut time during future iterations over j .
- Also, if for some $jp > j$, $\text{cand}(j, jp) \leq \text{cand_E}(jp)$, then we need not inspect any blocks to the right of the block that jp is represented by.

Forward-Gap Assumption # 5

- Lastly, if we found a block where for the l and r values of this block, $\text{cand}(j,l) > \text{cand_E}(l)$ and $\text{cand}(j,r) \leq \text{cand_E}(r)$, we can search for the largest jp value (with $l \leq jp < r$) such that $\text{cand}(j,jp) > \text{cand_E}(jp)$ by doing binary search over the indices from l to r . Again, this is a time-cutting method. (Piecewise linear gap model cuts this time further. More on this very soon... All in good space... I mean time.)

Forward-Gap Search Psuedocode

- So, to compute any E entry, we proceed as follows:
- (Step 1) L begins empty, and for E(0), we set it to whatever is the base value (as defined by our functions). We then put in L a block with $l=1$, $r=\max$, and $v=0$. (For now, \max can be n , 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 n without affecting computation time and space...)
- (Step 2) For each j from 1 to n , we do:
- (Step 2a) Look at the leftmost block b of L. We set $E(j)$ to $\text{cand}(b.v, j)$. We increment $b.l$ by 1. (So $b.l$ is now $j+1$)
 - 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 $j+1$.

Forward-Gap Search Psuedocode

- (Step 2b) We check if $\text{cand}(j, j+1) > \text{cand}(b.v, j+1)$. If not, we go no further and jump to the next iteration of j . (Because j is not going to be the ind_E winner for any $j_p > j$.) Otherwise we do step 2c.
- (Step 2c) while $(\text{cand}(j, b.r) > \text{cand}(b.v, b.r))$ 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 $j+1$, $b.r$ to max , and $b.v$ to j , and jump to the next iteration of j ; else do step 2e.

Forward-Gap Search Psuedocode

- (Step 2e) Insert a new block c as the leftmost element of L , and set $c.l$ to $j+1$, $c.r$ to $b.l - 1$, and $c.v$ to j (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 jp such that $cand(j, jp) > cand(b.v, jp)$. If this number jp doesn't exist in block b , we jump to the next iteration of j , otherwise, we do step 2g.
- (Step 2g) Set $c.r$ to jp , set $b.l$ to $jp+1$. Go to the next iteration of j .

Forward-Gap Search Pseudocode

- Now to analyze runtime for this algorithm. Note that after step 1, L has only one element in it. Also note that, on any iteration of j in step 2, when comparing # of blocks of L before and after the iteration:
 - If only one block of L is looked at during the iteration, the # of elements of L increases by at most 1.
 - If two blocks of L are looked at during the iteration, the # of elements of L stays the same.
 - For any $r > 2$, if r blocks of L are looked at during the iteration, the # of elements of L decreases by $r-2$.

Forward-Gap Search Analysis

- With this, and the fact that the maximum # of elements in L is n , and we begin with one element of L , this means:
- Total time to do all iterations of j for steps 2a thru 2e is $O(n)$. (Essentially $O(1)$ amortized time per iteration.)
- As for step 2f, the interval of $b.l$ thru $b.r$ is size $O(n)$. So doing a binary search takes $O(\log n)$ time. Step 2g takes $O(1)$ time. Therefore, time for all iterations of j to do steps 2f and 2g is $O(n \log n)$.
- Hence, overall runtime is $O(n \log n)$. (Which is better than the $O(n^2)$ time of the conventional method.) Space used is $O(n)$. (Since the L list's size is at most n .)

Forward-Gap Search Analysis

- The time-saving computation method for F function is similar to that for E (and since F iterates over a row, not a column like E does, the computation for F takes $O(m \log m)$ time and $O(m)$ space).
- Now, note that we only dealt with one column for the E function. To account for all columns of E, we maintain m doubly-linked lists L_E , each list is for each column (we essentially treat forward computations for each column of the E function separately). Similarly, we account for all rows of F by maintaining n doubly-linked lists L_F , each list for each row. This lets us interweave computations of E and F. G and V are computed same way as in the conventional method.
- However, upon more careful inspection, if we compute table entries column by column, we need maintain only one doubly-linked list L_E and can reuse it for each next column. However, we'd still need n doubly-linked lists L_F , one per row. On the bright side, forward-gap computations reduce the number of table lookbehinds we've made to a constant number. (In truth, only the G entry needs to make a lookbehind in this model...)
- We need $O(n^2)$ extra space to maintain the one L_E list, and all n L_F lists. (But the table for V typically needs $O(mn)$ space for general function w anyhow...)

Forward-Gap Search Analysis

- And, total time to compute E entries in two dimensions is $O(m * n \log n)$ time, and total time to compute F entries in two dimensions can be done in $O(n * m \log m)$ time, and total time to compute G entries in two dimensions is $O(mn)$ time (just like before). Hence, total time to compute the V entries is $O((mn \log m) + (mn \log n)) = O(mn \log m)$ time.
- Thus, we now have a method for general convex gap function w that completes computation in time $O(mn \log m)$, and $O(mn)$ space.

Piecewise-Linear Functions

Observations

- We'll use ww instead of w for the Forward-Gap Search Algorithm, and note that:
 - For the general gap function w , the concatenation of the best-solution curves corresponding to the blocks of a list L is itself a curve that resembles w , except that some intervals are folded, and the remaining parts of the curve might be moved up or down...
 - This is same idea applies to piecewise-linear gap function ww . However, note that for the best-solution piecewise-linear formula, each line in this formula lies in at most one block. (If a line was to originate from two blocks, the earlier block of the two would prevent the later block from even being created, since new blocks are only made when their scores beat earlier solutions, not meet it) Since there are at most p lines in ww , there are hence at most p blocks in L . So our L_E and L_F lists each take $O(p)$ space.
 - With one L_E list, and n L_F lists, each using $O(p)$ space, we get $O(np)$ space overall with all the lists. Couple this with constant lookbehinds needed in forward-gap search, and it's looking very hopeful to achieve $O(np)$ space.

Piecewise-Linear Functions

Observations

- In analyzing the L_E list,
 - In steps 2a thru 2e of the earlier algorithm, our list has at most p blocks, and hence makes $O(p)$ work per iteration (not $O(n)$), but we still make n iterations per j value, and hence, total time to do all iterations of j for steps 2a thru 2e is still $O(n)$.
 - Step 2f, where we do a binary search over the interval $b.l$ thru $b.r$ to find the largest number jp such that $\text{cand}(j, jp) > \text{cand}(b.v, jp)$, simply involves taking the lines corresponding to this block, and determining which ww line from the $E_{\text{candidate}}$ solution intersects with line from that generated at spot j . There are p lines to consider in the former and latter, and once the pair of lines for the intersection in question is known, it takes $O(1)$ time to find jp . Binary search over the p lines in the former and p lines of the latter, making comparisons in $\text{cand}()$ entries as a way of deciding if to go left/right (each comparison taking $O(1)$ time), is sufficient to find the value between $b.l$ and $b.r$ that is jp . Such a search takes $O(\log p)$ per iteration. Since step 2f is ran $O(n)$ times, and hence time here for all iterations is $O(n \log p)$.
 - Runtime for step 2g is unchanged.
 - Hence overall work on the L_E list takes $O(n \log p)$ time. Thus, overall time to compute all V entries is $O(mn + m*n \log p + n*m \log p) = O(mn \log p)$

Section 6:
PLALIGN's Algorithm

PLALIGN's Algorithm

- In addition to using the techniques mentioned in the previous sections for piecewise-linear gap model (Hirschberg and Forward-Gap Seeking) and using $O(np)$ space in the process, all of which PLALIGN does, there's still the issue about gluing alignments where a horizontal gap from a V table extends into the V^r table.
- However, the L_F lists from the previous section are a step in the right direction. When using them, instead of setting the max value to the # of columns of the V table (which is at most m), we set the max value to the # of columns of the V table plus the # of columns of the V^r table. This allows us to, while computing the V table, anticipate gaps that stretch over into the V^r table (and this doesn't affect asymptotic runtime or space).
- Then, while computing the V^r table, for each row, we keep track of the crossover between V and V^r with the best combined score (so that the go-between horizontal gap is evaluated and penalized independently of both the V and V^r tables, and scores from the V and V^r tables are added in separate from the gap). This can be done without affecting asymptotic runtime and space.

PLALIGN's Algorithm

- Next, in $O(n)$ time, we compare the best crossovers for each row against each other to find the best of the best. The winner yields specific entries in V and V^r where to backtrack from, and this, combined with the go-between crossover, gives the overall alignment. (Note a zero-gap crossover is perfectly valid. In this case, there's no go-between gap penalty, since there is no horizontal gap in going from the V to the V^r table.)
- This is can be done (and is done) in PLALIGN without affecting asymptotic runtime and space or the earlier mentioned methods.
- Overall time used here is $O(mn \log p)$, and space used is $O(np)$.
- p is typically ≤ 10 in most applications. With this in mind, we get quadratic time and linear space (worst case scenario!!!). (A definite improvement over the earlier cubic time and quadratic space.)

Still awake?
This is the end of the talk!