Piecewise-Linear Gap Formula Alignment in Linear Space

By Ofer H. Gill
April 29, 2003

Piecewise-Linear Gap Formula Alignment in Linear Space

- The Sequence Alignment intuition
- Why perform Protein Alignment?
- The Protein Alignment General Formula
- Linear Gap Formula Alignment
- Piecewise-Linear Gap Formula Alignment
- Reducing to Linear Space
- Conclusions & Open Questions

The Sequence Alignment Intuition

- Longest Common Subsequence (LCS)
- Edit Distance
- Sequence Alignment in Comparision.

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...i] and Y[1...j].
- Formula is:

  V(i, 0) = 0, for all i >= 0

  V(0, j) = 0, for all j >= 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 $X[1...i]$ and $Y[1...j]$, then the Formula is:

  $V(i, 0) = i$, for all $i >= 0$

  $V(0, j) = j$, for all $j >= 0$

  And for all $i > 0$ and $j > 0$,

  if $X[i] == Y[i]$, 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 in Comparision

- 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 we might care more about block matches and gaps!)

Why perform protein alignment?

- We wish to find common points in living organisms.
- We wish to find differing points in living organisms.
- We wish to trace evolution of certain specifics.
- The Biology Influence

Why perform protein alignment?

- We wish to find common points in living organisms.
  - This allows us to learn about one organism from what we know of another. (It's easier for us to experiment with medical treatments on mice than humans, so knowing common traits in mice and humans can hint us in on treatments that work for humans.)

Why perform protein alignment?

- We wish to find differing points in living organisms.
  - If something differs between two organisms, we'd like to know what it is. (If a medication works on a mouse, but not a human, we'd like to know what differing parts in the mice and humans that cause this.)

Why perform protein alignment?

- We wish to trace evolution of certain specifics.
  - We'd like to know what, in the evolution of species accounted for certain traits. (In observing the human genes versus that of our ape ancestors or the chimpanzee, we'd like to know what's the same and what's different. Specifically, what gives us the ability to reason over apes and chimps.)

Why perform protein alignment?

- The Biology Influence
  - Common points and traits in proteins typically occur in blocks (substrings), not at various discrete points. Therefore, we want a scoring scheme that aligns proteins so that matches are 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).
  - For very aligning long differing proteins, finding area of matches is best done by a local (Smith-Waterman) alignment of substrings, not a global (Needleman-Wunsch) alignment of the entire proteins.

Why perform protein alignment?

- The Biology Influence
  - Although we penalize gaps, longer gaps often reveal important differences between proteins. To allow for this in our alignment, we decrease the extra penalty for each length increase in the gap. (So, for exmaple, 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.)

The Protein Alignment General Formula

- 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 Protein Alignment General Formula

- 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 Protein Alignment General Formula

- 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)
  - $E(i, j) = \max_{0<=k<=j-1}[V(i, k) - w(j-k)]$
  - $F(i, j) = \max_{0<=k<=i-1}[V(k, j) - w(i-k)]$

The Protein Alignment General Formula

- 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 Protein Alignment General Formula

- 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

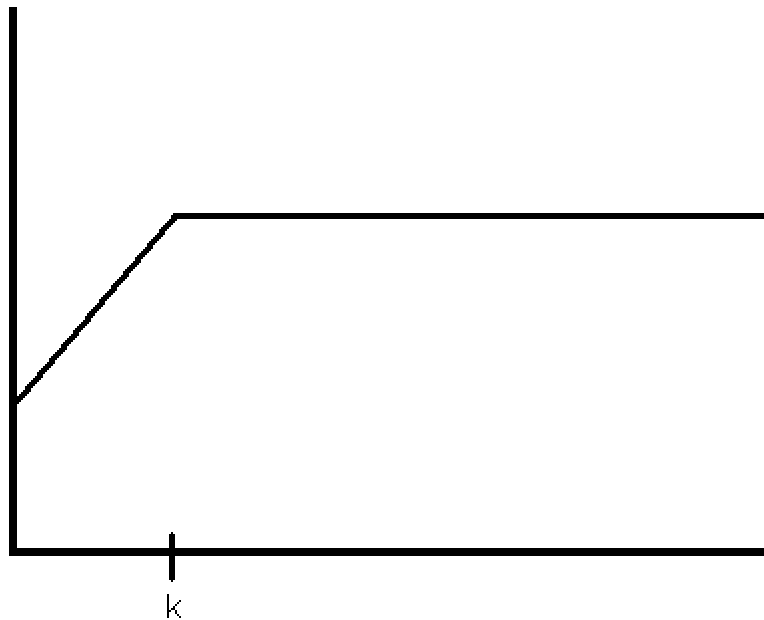
- Our formula now becomes:
  - $V(0, 0) = 0$
  - $V(i, 0) = E(i, 0) = -w_o - i*w_c$
  - $V(0, j) = F(0, j) = -w_o - j*w_c$
  - $V(i, j) = \max\{E(i, j), F(i, j), G(i, j)\}$
  - $G(i, j) = V(i-1, j-1) + s(X[i], Y[j])$
  - $E(i, j) = -w_c + \max\{E(i, j-1), V(i, j-1) - w_o\}$
  - $F(i, j) = -w_c + \max\{F(i-1,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

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 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?

Piecewise-Linear Gap Formula Alignment

- For a general gap penalty function g(i), one workaround is to create a piecewise-linear gap function w where, for each i, w(i) approximates (within some reasonable range) what g(i) would be.
- First, let's consider a two-piece linear gap-function like the one shown here:



Piecewise-Linear Gap Formula Alignment

- To keep things simple for now, we assume $w(i) = w_{c[0]}i+w_o$ if $i < k$, and $w(i) = w_{c[1]}*(i-k) + w_{c[0]}k + w_o$ if $i >= k$. w_o is the cost for starting a gap, $w_{c[0]}$ is the rate of penalty increase for any gap below size k, and $w_{c[1]}$ is the rate of penalty increase for any gap of size k or larger.
- In the figure shown on the previous slide, $w_{c[1]}$ is zero. (We can do that if we want...)
- In this case, we'll use five tables to derive the V table (not three as before). The five are G, E_0, E_1, F_0, and F_1.

Piecewise-Linear Gap Formula Alignment

- $G(i, j)$ behaves same as before.
- $E_0(i, j)$ denotes the score if we align Y[j] against a gap of length less than k.
- $E_1(i, j)$ denotes the score if we align Y[j] against a gap of length k or larger.
- $F_0(i, j)$ and $F_1(i, j)$ behave similarly, but for aligning X[i] against a gap.

Piecewise-Linear Gap Formula Alignment

- In this model, for E_0 and F_0, the gap penalty always increases by $w_{c[0]}$ once the gap is longer than one but smaller than k, we don't need to worry where a gap begins (E_1 and F_1 will account for if the gap exceeds size k). So for E_0 and F_0, we only need to worry if a gap has already began, or a new gap is started.
- Also, for E_1 and F_1, once the gap is larger than k, we don't need to worry where it began, and the gap lenalty always increases by $w_{c[1]}$. So, we only need to worry about if a gap larger than k was already begun, or if a new gap of size at least k is started, based on E_0 and F_0, which are gaps of size at least one. (My reasoning for basing E_1 and F_1 from E_0 and F_0, and not V, which gives the same answer, will become clearer later.)

Piecewise-Linear Gap Formula Alignment

- Our base values are:
  - $V(0, 0) = 0$
  - $V(i, 0) = E_0(i, 0) = -w_o - i*w_{c[0]}$ if $i < k$
  - $V(i, 0) = E_1(i, 0) = -w_o - k*w_{c[0]} - (i-k)*w_{c[1]}$ if $i >= k$
  - $V(0, j) = F_0(0, j) = -w_o - j*w_{c[0]}$ if $j < k$
  - $V(0, j) = F_1(0, j) = -w_o - k*w_{c[0]} - (j-k)*w_{c[1]}$ if $j >= k$

Piecewise-Linear Gap Formula Alignment

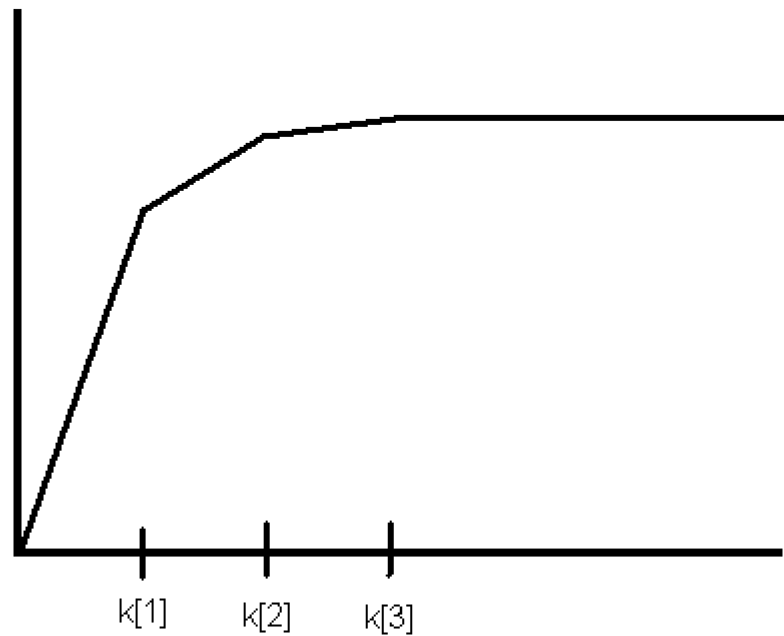
- The remaining values are:
  - $V(i, j) = \max\{F_1(i, j), E_1(i, j), F_0(i, j), E_0(i, j), G(i, j)\}$
  - $G(i, j) = V(i-1, j-1) + s(X[i], Y[j])$
  - $E_0(i, j) = -w_{c[0]} + \max\{E_0(i, j-1), V(i, j-1) - w_o\}$
  - $E_1(i, j) = \max\{E_1(i, j-1) - w_{c[1]}, E_0(i, j-(k-1)) - (k-1)*w_{c[0]}\}$ if $j >= k$, -infinity otherwise
  - $F_0(i, j) = -w_{c[0]} + \max\{F_0(i-1, j), V(i-1, j) - w_o\}$
  - $F_1(i, j) = \max\{F_1(i-1, j) - w_{c[1]}, F_0(i-(k-1), j) - (k-1)*w_{c[0]}\}$ if $i >= k$, -infinity otherwise

Piecewise-Linear Gap Formula Alignment

- For the two-piece linear gap function, we perform O(1) lookbehinds to get each V(i, j) entry. Therefore, our runtime is O(mn). Our space here is also O(mn). (We use the same time and space as the linear gap function.)

Piecewise-Linear Gap Formula Alignment

- What about if there's more than two pieces in the piecewise linear formula?



Piecewise-Linear Gap Formula Alignment

- To generalize, we'll assume we're given a p+1 part piecewise-linear function w containing a penalty for starting a gap called w_o, and p+1 different slopes named $w_{c[0]}$, $w_{c[1]}$, ... $w_{c[p]}$, as well as p values k[1], k[2], ... k[p] where the slopes change. (See drawing in previous slide.)

- Assuming k[0] = 0 and k[p+1] = infinity, we can write w(i) as follows:

  – If there exists a u such that k[u] <= i < k[u+1], then

    - $w(i) = w_o + (\text{sum}_{v=1 \text{ to } u}[(k[v] - k[v-1]) * w_{c[v-1]}]) + (i - k[u]) * w_{c[u]}$

- In this case, we use the ith and jth entries of tables E_0, E_1, ... E_p and F_0, F_1, ... F_p to compute the scores for aligning X[i] or Y[j] against gaps, and E_u and F_u corresponds to the u-th slope in our gap function. The train of thought for constructing these tables is similar to the two-part piecewise linear case.

Piecewise-Linear Gap Formula Alignment

- So, our basis values for V are now:
  - $V(0, 0) = 0$
  - If there exists a u such that $k[u] <= i < k[u+1]$, then
    - $V(i, 0) = E_u(i, 0) = -w(i)$
  - If there exists a u such that $k[u] <= j < k[u+1]$, then
    - $V(0, j) = F_u(0, j) = -w(j)$

Piecewise-Linear Gap Formula Alignment

- The rest of V (assuming k[0] = 1 in the E and F tables) is:
  - $V(i, j) = \max\{F_p(i, j), E_p(i, j), \ldots F_1(i, j), E_1(i, j), F_0(i, j), E_0(i, j), G(i, j)\}$
  - $G(i, j) = V(i-1, j-1) + s(X[i], Y[j])$
  - $E_0(i, j) = -w_{c[0]} + \max\{E_0(i, j-1), V(i, j-1) - w_o\}$
  - $F_0(i, j) = -w_{c[0]} + \max\{F_0(i-1, j), V(i-1, j) - w_o\}$
  - For $u > 0$, $E_u(i, j) = \max\{E_u(i, j-1) - w_{c[u]}, E_{u-1}(i, j-(k[u]-k[u-1])) - (k[u]-k[u-1])*w_{c[u-1]}\}$ if $j \geq k[u]$, -infinity otherwise
  - For $u > 0$, $F_u(i, j) = \max\{F_u(i-1, j) - w_{c[u]}, F_{u-1}(i-(k[u]-k[u-1]), j) - (k[u]-k[u-1])*w_{c[u-1]}\}$ if $i \geq k[u]$, -infinity otherwise

Piecewise-Linear Gap Formula Alignment

- Assuming that p is a variable, just like m and n, then each V(i, j) entry uses 2p + 1 = O(p) lookbehinds to compute its value, and we use O(mnp) memory overall. Our runtime is O(mnp). Hence we use cubic time and memory under this assumption.
- However, assuming p is constant, we use O(mn) time and O(mn) space. (Quadratic time and space, just like the linear gap formula model.) And in most cases, the chosen gap function uses p <= 10, so we get quadratic time and space for a piecewise linear gap function that can be adjusted to come close to resembling any arbitrary function whose slope increase rate decreases as the gapsize increases.
- So we have piecewise-linear gap formula alignment in quadratic space. Can we do better? (Answer is yes. Look at title of this presentation for a hint...)

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

  $$V(m, n) = \max_{0<=k<=n}[V(m/2, k) + V^r(m/2, n-k)]$$

- 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:
  - OPTA(l, l', r, r', t) {
    - if (l > l') then align Y[r...r'] against gaps and return that; (Base case)
    - else if (r > r') then align X[l...l'] against gaps and return that; (Base case)
    - else if (1 + l' – l <= t) then
      - compute V for entries X[l...l'] and Y[r...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...l'] and Y[r...r']. (This is another base case.) We return A;
    - else do as follows on the next slide;

Reducing to Linear Space

h = (l' – l) / 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 OPTA(l, h-t, r, q_1, t). Let's call the alignment from this L_1
- Call 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 OPTA(1, m, 1, n, 2) to get the alignment of X with Y.
- If T(m, n) is the runtime of OPTA(1, m, 1, n, 2), we can express T(m, n) as
  - $T(m, n) <= \max_{0<=k<=n}[T(m/2, n-k) + T(m/2, k)] + O(mn)$
  - It turns out T(m,n) = O(mn).
- Hence, we found an optimal alignment in the linear gap model in O(mn) time and O(n) space. (Quadratic time and linear space.)
- t doesn't affect the runtime, but in truth, we use O(tn) space. But if t is constant, this is ok.

Reducing to Linear Space

- For the two-piece piecewise linear gap function, we will need to look k-1 entries to the left, so we must be sure this isn't deleted from memory. Calling OPTA(1, m, 1, n, k) using the two-piece piecewise linear gap function will make sure this doesn't happen.
- (Assume k[0] = 0) For arbitrary piecewise linear gap functions, let $d = \max_{1<=u<=p}[k[u] - k[u-1]]$. We will need to look at most d entries to the left, so we must be sure this isn't deleted from memory. Calling OPTA(1, m, 1, n, d+1) using the arbitrary piecewise linear gap function will make sure this doesn't happen. (And assuming d is bounded by a constant, and so is p, we get O(mn) runtime in O(n) space. Hence, we get efficient Piecewise Linear Gap Alignment in Linear Space.)

Reducing to Linear Space

- However, there are two things we must be careful about with piecewise linear functions that don't affect linear functions.
  - In piecewise linear functions, if two answers give the same result for V, E_u, or F_u , we want the one such that X is aligned with the longest possible gap (which is why in my code, the max function is made so that in the case of ties, the longest possible gap is the result chosen as winner).
  - Also, with linear functions, the extra gap cost for extending a gap is always the same value, regardless of how big the gap. In piecewise linear functions, 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.

Reducing to Linear Space

- Remember that $w(a+b) <= w(a) + w(b)$ when the rate of increase for w goes down as the gaplength grows, so we might have $w(a+b) < w(a) + w(b)$, and we need to compensate for this!
- Solution: For each j and j', after we compute $V(h, j)$ and $V^r(h, j')$ in OPTA, we can create compensation functions c and c'.
- For each u, we record the gaplength a for the choice of $F_u(h, j)$ (and we can add gaplength tracing information into all the entries of V without affecting the asymptotic runtime), then we set $c(u, j) = a$.
- Similarly, we can compute c' from V'.

Reducing to Linear Space

- Hence, after computing $V(m/2, n)$ and $V^r(m/2, n)$. We can correct for a gap that stretches from V to V^r by instead finding k^* by doing:
  - $\max_{0<=k<=n}[V(m/2, k) + V^r(m/2, n-k), \max_{0<=u<=p,\ 0<=v<=p} [F_u(m/2, k) + F_v(m/2, n-k) + w(c(u, k)) + w(c'(v, n-k)) – w(c(u, k) + c'(v, n-k))]]$
- Therefore, we use the max function shown above in order to select k^*.
- This helps to account for "bridges" between the left and right alignments.
- If p is constant, this won't affect the asymptotic runtime.

Conclusions & Open Questions

- For protein pairs X and Y with large blocks of matches and gaps, in comparing the piecewise-linear gap formula with linear space to the cubic time, quadratic space general gap formula (where we set the genenral gap formula to behave like the piecewise linear one) the scores obtained are the same, and the alignments reveal the important blocks of common matches, and gaps (even if one or two bits differ in the alignments).

- Where to now?

  - Try to model non-linear gap functions in linear space and quadratic time (for example, the logarithm gap function).
  - Try aligning more than two proteins.
  - Parallelize the computations. (Surely there's room for parallelism in OPTA...)