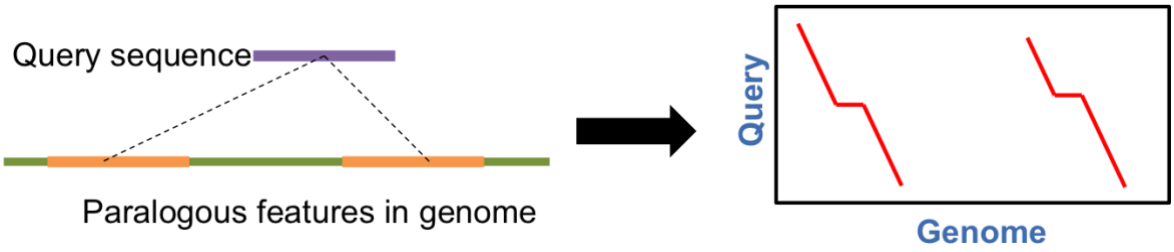
# From Smith-Waterman to BLAST

Jeremy Buhler

July 23, 2015

Smith-Waterman is *the* fundamental tool that we use to decide how similar two sequences are. Isn't that all that BLAST does?

- In principle, it is possible to build a biosequence database search tool that uses *only* Smith-Waterman (and associated Karlin-Altschul significance testing) to do its work.
- If you want to try such a program, Bill Pearson's ssearch is a good example.
- If you really, really want to use "pure" Smith-Waterman on a large database, you can run the algorithm on specialized hardware such as a Graphical Processing Unit (GPU) or a Field-Programmable Gate Array (FPGA)]
  - Example: Oliveira FF, Dias LA, Fernandes MAC. Proposal of Smith-Waterman algorithm on FPGA to accelerate the forward and backtracking steps. PLoS One. 2022 Jun 30;17(6):e0254736.
- However, most people don't want to buy specialized hardware and find that pure Smith-Waterman in software is *much too slow* for their search needs, and so they use BLAST or a related tool (FASTA, BLAT, etc.).
- There is one other important limitation of Smith-Waterman: given two sequences, it returns only *one* alignment between them.
- For genomic applications, one would of like to find *all* sufficiently good matches between the query and (say) a chromosome.
- For example, we'd like to see both of the alignments in the following comparison:




- We may want to see all "good" feature matches, even if they don't all have the optimal score.

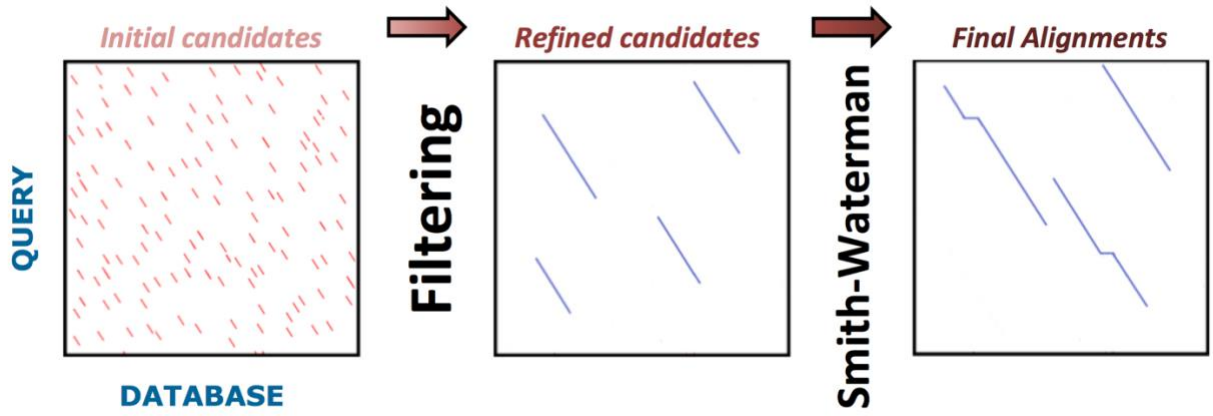
Today, we'll talk about how BLAST (and friends) address the two key limitations of Smith-Waterman: speed and returning multiple hits from one database sequence.

# 1. Beyond Dynamic Programming?

The key idea of BLAST is to do some fast preprocessing up front to strictly limit the fraction of the database that is subjected to more sensitive (but expensive) Smith-Waterman alignment. This strategy is called **generate and filter**.

1. Generate a list of *candidate patterns* that may indicate the presence of a high-scoring alignment nearby.
2. Filter the candidate list by searching for evidence of high-scoring alignments near each one. Discard candidates that do not yield such evidence.
3. Repeat step 2 as needed on the remaining candidates, using progressively more sensitive filters, until the candidate set is small enough that we can just run Smith-Waterman on each candidate.

It may help to visualize this strategy as running inside Smith-Waterman's dynamic programming matrix:

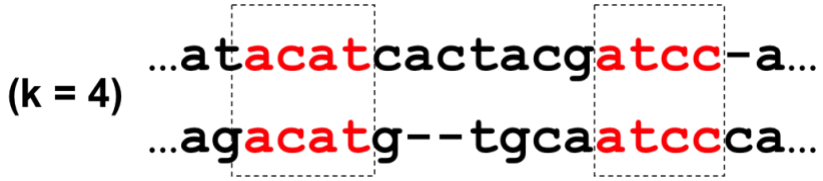



- To realize this strategy, we need to answer two questions:
    a. What kind of candidate patterns should we look for?
    b. How do we find the best alignment "near" a candidate?

# 2. Candidate Generation

We'll describe the approach that NCBI *blastn* takes to candidate generation.

- **What might be evidence that two sequences are locally similar?**
- Think about DNA in particular...
- Observe that a high-scoring alignment between DNA sequences contains many matching bases, so it is likely to exhibit several matches in a row.
- **Defn**: a $k$-mer match ("word match") between two sequences is a string of $k$ contiguous residues that occurs somewhere in both sequences.




- We propose to treat every word match of sufficient length ($\geq$ some fixed $k$) as a candidate.

The first question to ask is: can we locate $k$-mer matches between two sequences more efficiently than finding high-scoring alignments?

- Here's a quick-and-dirty strategy that works well.
- First, make a list of all the $k$-mers in the query sequence.
- Then, create a table $T$ with $4^k$ entries, one for each possible $k$-mer.
- If a $k$-mer $w$ is present in the query, then entry $T[w]$ lists its position(s) in the query.
- When searching a database $D$, for each position $d$ into the database, check whether the $k$-mer $w_d = D[d..d+k-1]$ appears in $T$.
- If so, each query position in $T[w_d]$ begins a $k$-mer match between query and the database at position $d$.
- NCBI *blastn* works this way, except that it uses a more space-efficient data structure called a *hash table* to store $T$ in space proportional to the query size, which may be much less than $4^k$.

In general, the "build a table" abstraction lets us spend time proportional to one sequence's length to construct the table, and then spend time proportional to the other sequence's length to search it. This yields an algorithm whose cost is proportional to the *sum*, not the product, of lengths.

- If we want to compare a bunch of queries to the same database, it might be faster to build a table from the $k$-mers in the database once and scan each query against this table, rather than scanning the whole database for each query.
- This is exactly what BLAT does to support large numbers of queries against the UCSC Genome Database. The table is stored in memory or on other fast storage such as an SSD.
- There are also much fancier, more space-efficient index structures than a simple table, such as the "virtual suffix tree" used by short-read alignment tools like Bowtie, but the principle is the same.

Generating $k$-mer word matches can be done very fast! But is it any good?

- We have one parameter to play with — the word length $k$.
- By changing this parameter, we can trade off between *sensitivity* (ability to find high-scoring alignments) and *specificity* (ability to discard sequences without such alignments).
- First, let's quantify specificity.
- Given two unrelated DNA sequences $S, T$, how long a match is likely to occur between them by chance alone?
- To make this question concrete, we assume that $S$ and $T$ are i.i.d. random with equal base frequencies.

$$\begin{aligned} \Pr(S[i] = T[i]) &= \frac{1}{4} \\ \Pr(S[i \ldots i+k-1] = T[j \ldots j+k-1]) &= \left(\frac{1}{4}\right)^k. \end{aligned}$$

- Let $M = |S| \cdot |T|$, and let $k = \log_4 M$. Then

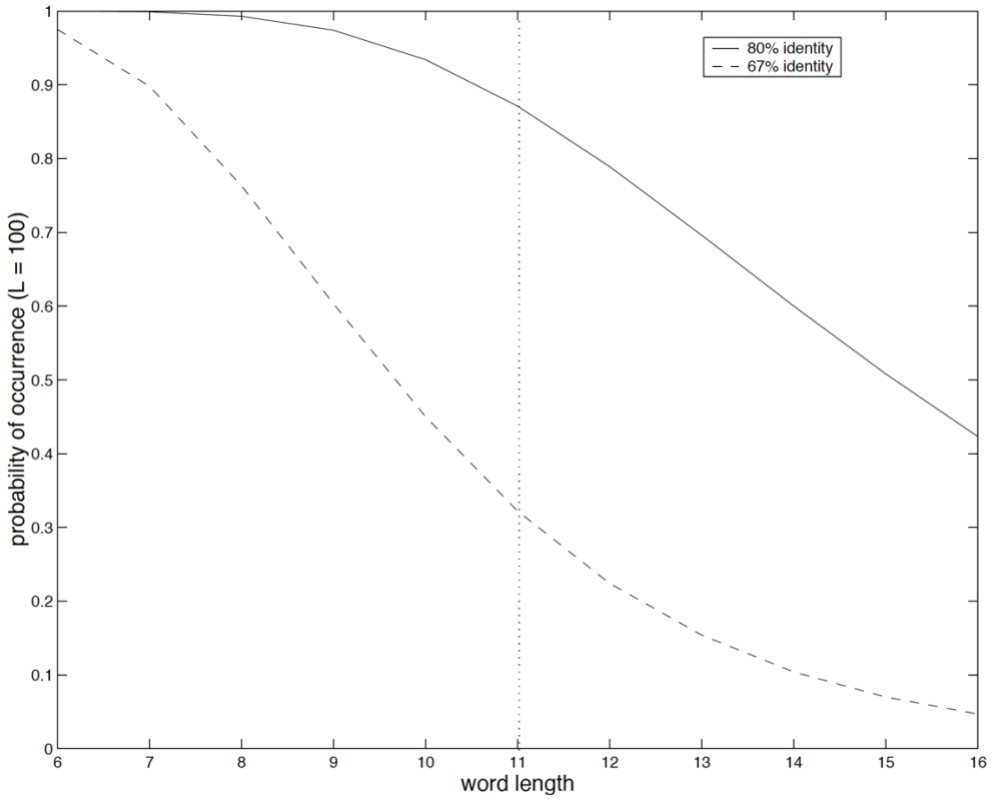
$$\begin{aligned} E[\#\text{ matches of length } = k] &= \sum_{i,j} \Pr\left(S[i \ldots i+k-1] = T[j \ldots j+k-1]\right) \\ &\approx M\left(\frac{1}{4}\right)^k \\ &= M \cdot 4^{-\log_4 M} \\ &= \frac{M}{M} \\ &= 1. \end{aligned}$$

- For example, if we search a 1000-base pattern against a 1 gigabase database, we shouldn't be surprised to see a word match of length $\log_4 10^{12} \approx 20$ bases.
- The number of expected matches grows exponentially as $k$ decreases. Roughly $|S||T|/4^k$ $k$-mer matches are expected by chance alone.
- For the above sequence sizes, that's about 1,000 15-mer matches, 60,000 12-mer matches, or 1 million 10-mer matches.
- In practice, nonuniform and non-i.i.d. sequences exhibit a higher rate of chance matches.
- All these matches are "junk" — they do *not* arise from an interesting alignment.

Now, let's think about how to quantify the *sensitivity* of word matching.

- This is harder to analyze — what is the frequency of $k$-mer matches in pairs of conserved biological sequence features?
- To get a rough idea, we can study the set of all alignments of some length $L$ that contain exactly $f \cdot L$ matching residue pairs. Such alignments have $100f$ percent identity.
- Assume that the non-matching residue pairs are distributed *uniformly at random* throughout the alignment. (This is actually a more challenging model than many real biological sequence alignments.)
- Our strategy can detect these alignments only if they contain at least one $k$-mer match.
- How large a value of $k$ is likely to detect "most" alignments? We can address this question through fancy math or via simulation.

- The following plot, produced by simulation, shows the probability that a random alignment of length $L = 100$ from the above model contains a word match, as a function of the word length. The solid line is for alignments with $f = 0.8$; the dashed line is for $f = 0.67$.



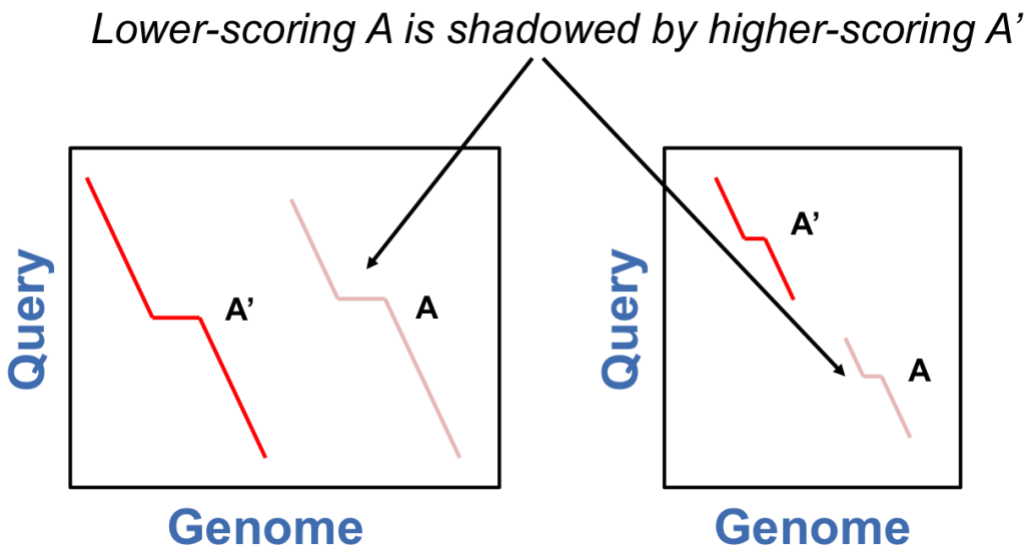

- In practice, a match length of around $k = 12$ bases catches most alignments with 75-80% identity. However, to discover most matches with 67% identity, a better length would be 8-9 bases.
- NCBI *blastn* uses a match length of 11 bases. Hence, we expect it to find most medium- to high-similarity alignments in DNA, but *almost all* candidates it generates will be false positives.
- RepeatMasker, which tries hard to find very diverged repeat alignments, uses match lengths as small as 8 bases.
- BLAT, which runs super-fast but does not try very hard to find distant homologies, uses matches of 15 bases or more.

Lest you think that we've said all there is to say about word-matching algorithms, the strategy used for protein is slightly different because we can't rely on seeing many exact residue matches. Also, there are tricks we can (must?) do to avoid generating many candidates that will lead to the same alignment. But for now, let's move on.

# 3. Using the Smith-Waterman Algorithm as a Filter

Each candidate match indicates that we should search the dynamic programming matrix "nearby" for high-scoring local alignments. This idea raises some technical issues.

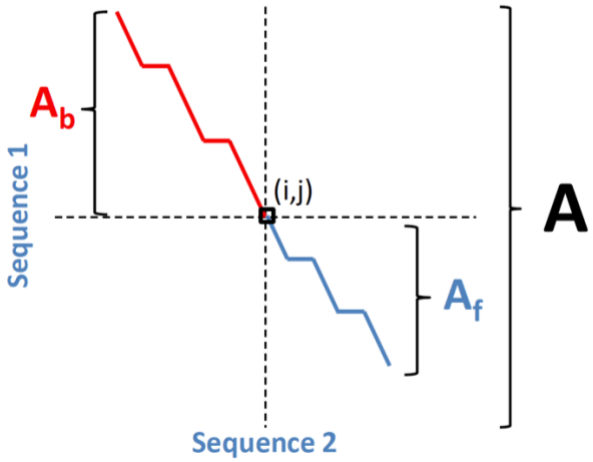
- First, what shape of region do we use when searching for alignments around a candidate?
- Second, how large should this region be?
- The third issue is a bit more subtle.
- Recall that Smith-Waterman only returns a single alignment (in this case, per candidate, since we run it once for each candidate).
- Suppose some alignment $A$ generates a candidate, but the region we search contains both $A$ *and* some other alignment $A'$ with a higher score than $A$.
- If we return only a single alignment per region, we will see $A'$ but not $A$.
- This is called the "shadowing problem" — $A'$ is said to shadow $A$.




- Examples of shadowed features might include part of a tandem duplication, as at left, or all but one exon of a multi-exon gene, as at right.

There are many ways to address these issues. I'll sketch one consistent approach, which isn't quite what NCBI BLAST uses but can be found in other BLAST-like tools.

- Suppose our candidate match is centered on bases $S[i], T[j]$ in sequences $S$ and $T$, respectively.
- We will compute an optimal local alignment between $S$ and $T$ that *passes through* entry $(i, j)$ of the DP matrix.
- This is not hard to do — just compute two "half-anchored" alignments.
- The first alignment $A_f$ is a best alignment starting from $(i, j)$ and ending anywhere below and to the right of it.
- The second alignment $A_b$ is a best alignment ending at $(i, j)$ and starting anywhere above and to the left of it.
- The desired alignment $A$ is just the concatenation of $A_f$ with $A_b$.




- (*Note*: to compute $A_b$, we can run the dynamic programming algorithm backwards, starting from the lower-right corner of the matrix.)

The above strategy is said to *pin* the alignment to position $(i, j)$. Why might it be a good idea?

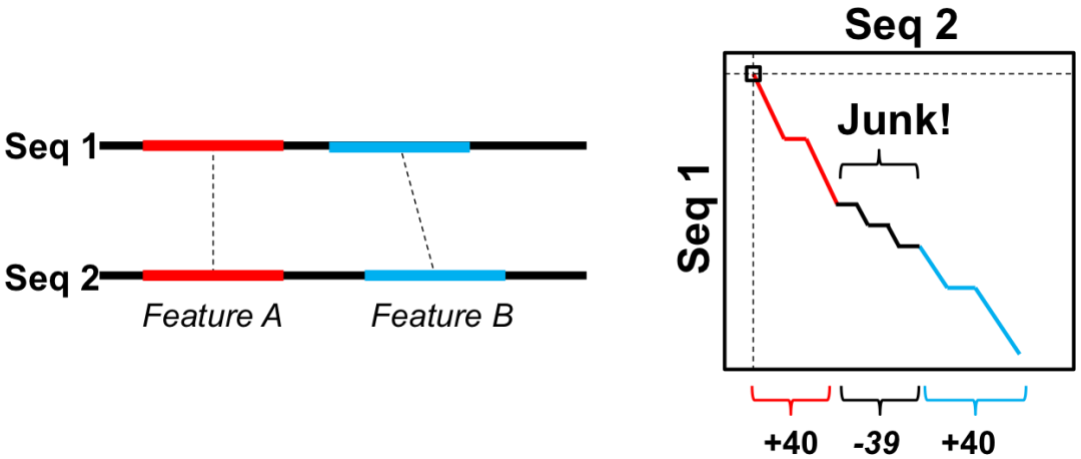
- Pinning anchors the alignment window near the candidate.
- More precisely, it nails down at least one endpoint of each of two dynamic programming problems for that candidate.
- Pinning addresses shadowing: nearby alignments that do not pass through the candidate will not be found.

The one thing pinning doesn't tell us is: how large a region must be searched by each of the two dynamic programming problems?

- In theory, we could simply bound the two search regions by the ends of the two sequences.
- But as the following figure shows, this would cost about as much as not filtering at all!


*DP fill region ≥ half of matrix*

- Moreover, if we do not limit the search region, we can have the "opposite" problem to shadowing.
- Consider aligning two sequences that contain two nearby matching features (for example, genomic fragments with two adjacent homologous exons).



- Suppose we generate a candidate match in the first feature and then search for high-scoring alignments passing through this candidate.
- An optimal alignment might subsume both features, even if it includes a completely spurious and biologically meaningless alignment of the sequence between them!
- In the example above, the penalty $(-39)$ for the junk alignment between the two features is less than the bonus $(+40)$ for the second feature match, so the second feature's alignment is linked to that of the first feature.
- This is called the *chaining* problem.

To avoid chaining and limit the size of the region to be explored by dynamic programming, we need a little more "secret sauce."

# 4. Banded Alignment with X-Drop

The key idea for bounding the search region will be to ignore alignments that are "not promising." That is, don't pursue alignments that are unlikely to be the best one passing through the candidate.

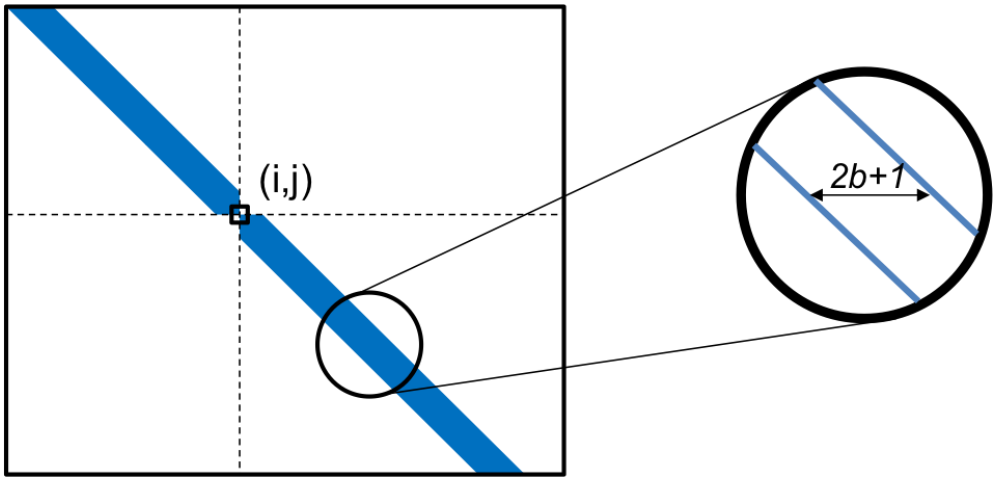
- One heuristic that is intuitively plausible is "don't consider alignments that contain very large gaps."
- Firstly, gaps are expensive under most common scoring systems, so an alignment with a large gap will likely have a poor score.
- Secondly, if we have to create a large gap to form an alignment, we should be suspicious that there is chaining going on. It would be better to stop before the gap and discover each of the two parts separately.

How can we limit the size of gaps in an alignment?

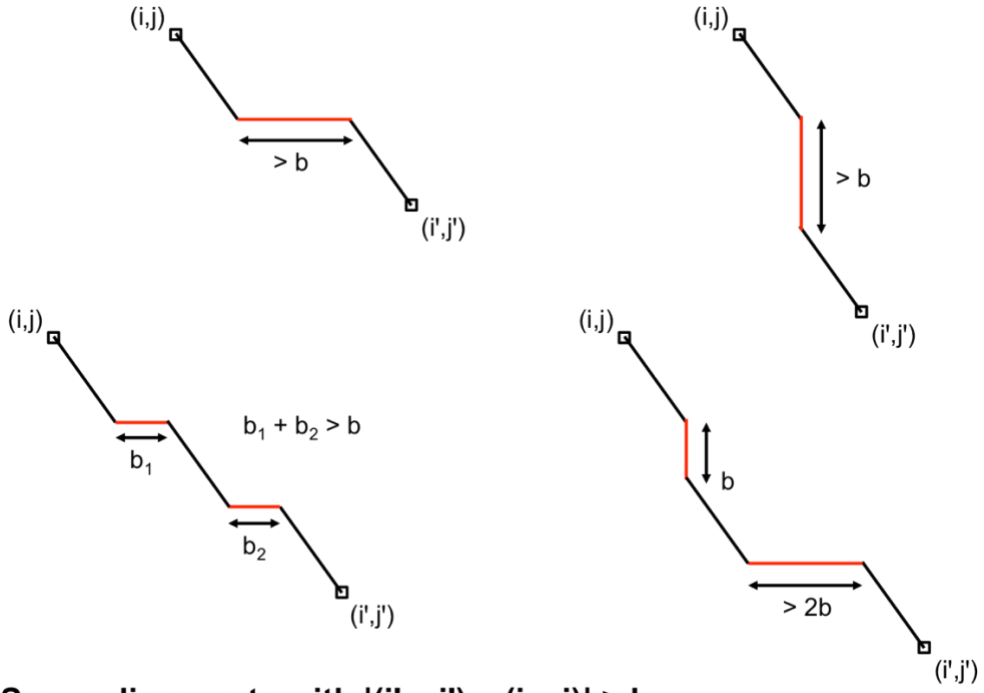
- For any entry $(i, j)$ in the dynamic programming matrix, call the difference $j - i$ its *diagonal*.
- Diagonals correspond (surprise!) to diagonal lines of entries in the matrix.
- Our heuristic is: if an alignment is pinned to entry $(i, j)$, do not let it pass through any entry $(i', j')$ for which

$$|(j' - i') - (j - i)| > b.$$

- The region of all DP entries that satisfy this constraint for a given $(i, j)$ form a *diagonal band* in the matrix:




- The parameter $b$ is called the *bandwidth* of the search, and it is typically at most a few tens of bases. The band itself is actually $2b + 1$ entries wide.
- If we restrict alignments to lie completely inside the band, it is impossible to have a contiguous gap, or a series of adjacent gaps, of length more than $2b$ residues in one sequence.
- The following figure shows a few such "impossible" alignments for bandwidth $b$:

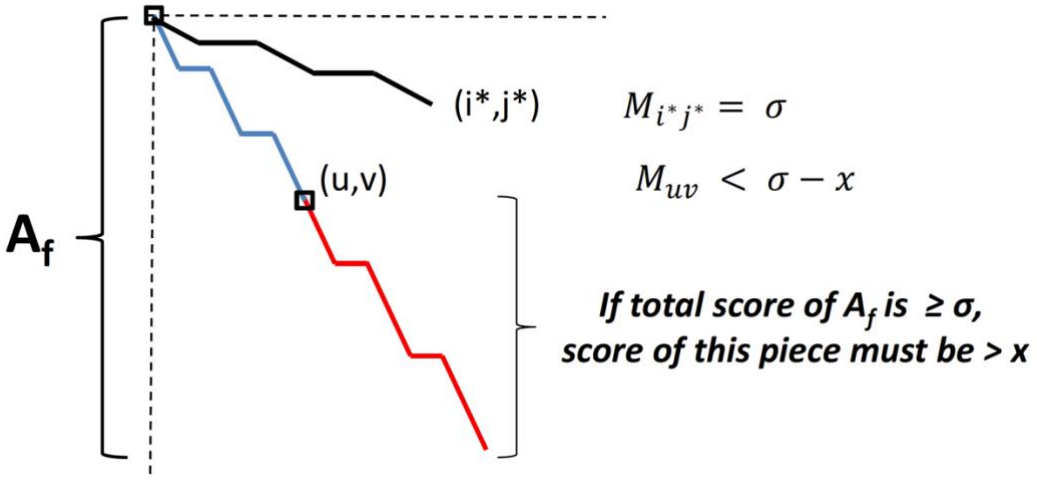



**Some alignments with |(j' – i') – (j – i)| > b**

- The variant of the Smith-Waterman algorithm that restricts alignments to a diagonal band is known as *banded alignment*.
- Banded alignment guarantees that the number of DP entries computed is at most proportional to the length of the shorter sequence (times $b$), rather than the product of the sequence lengths.

Banded alignment already provides a huge speedup over the naive approach, but we can do even better.

- Suppose that (say) the DP search for the forward alignment $A_f$ starting from entry $(i, j)$ has at some point discovered an entry $(i^*, j^*)$ with score $\sigma$.
- Now suppose that we compute another entry $(u, v)$ and find that its score is $< \sigma - x$.
- If an optimal alignment starting at $(i, j)$ passes through $(u, v)$, then the portion induced by the path *starting* at $(u, v)$ must have score *greater than* $x$ — otherwise, it couldn't be optimal!




- If $x$ is a big number (typically a few tens for DNA alignment), we think it unlikely that an alignment scoring less than $\sigma - x$ will "improve" enough to become optimal after further extension.
- Moreover, if it were to improve, then we would worry about chaining — the part after $(u, v)$ might be a distinct feature.
- Hence, we would be justified in suppressing any alignments starting from $(u, v)$.

This observation leads to a heuristic called the "X-drop".

- Track the best score $\sigma^*$ for any entry seen so far in the current dynamic programming search. (Initially, $\sigma^* = 0$.)
- If an entry $(u, v)$ receives a score $< \sigma^* - x$, for some value $x$ chosen *a priori*, then replace this score with $-\infty$.
- Every alignment path passing through this entry will now receive score $-\infty$, so Smith-Waterman will never return such a path as optimal (since we can always return an empty alignment with score 0 instead!).
- If, when filling in the matrix, we find that every entry on a given row of the band is $-\infty$, then all future rows will also be $-\infty$, and we we can stop the alignment algorithm altogether.

The combination of banded alignment and X-drop typically limits the search area for filtering to just a small region around each candidate. BLAST uses a slightly different X-drop strategy that does not rely on banding, but the basic idea is the same.