# CSE 584A Class 18

Jeremy Buhler

March 30, 2016

## 1 Why Go Beyond Dynamic Programming?

**Limitations of Smith-Waterman**

- unsuitable for long/many sequences – time complexity is $\Theta(n^2)$ for two sequences of length $n$.
- only returns single (optimal) local alignment – not too helpful for large comparisons intended to find many distinct features!

**Idea**: do fast preprocessing to help us decide where to apply sensitive (but expensive) DP alignment.

**Generate and Filter** method:

1. Generate *candidate patterns* that may indicate the presence of an alignment nearby
2. Filter the candidates by searching for high-scoring alignments near each one
3. Repeat step 2 as needed with progressively more sensitive algorithms

**Two views: flow-chart, DP matrix**

**Key Engineering Issues**

- How do we generate candidates efficiently?

- How do we eliminate *redundant* candidates (that lead to same alignments)?
- How do we generate alignments only near candidates? (next time)
- How do we score these alignments?
- Which alignments, if any, do we report?
- (Last two are beyond scope of this course, but for more info, see, e.g., Durbin, Eddy, Krogh, and Mitchison's textbook *Biological Sequence Analysis*.)

## 2 Candidate Generation

We'll describe the classic approach that BLAST takes to candidate generation.

- **What might be evidence that two sequences are locally similar?**
- think DNA in particular
- One plausible observation: a high-scoring alignment on DNA contains many matching bases, so it is likely to have 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.
- *Strategy*: every word match of sufficient length ($\geq k$ for some $k$) is a candidate.
- Goes back at least as far as FASTA (Pearson and Lipman 1985), by far most popular approach today (BLAST)
- You already know how to find $k$-mer matches, or indeed exact matches of any length $\geq k$, quickly.
- BLAST actually fixes the match size to exactly $k$, hashes the query sequence, and then scans the database against the hash table. No database indexing!
- BLAT, for example, actually indexes the database first, but in a much simpler way that only accelerates lookups for a predetermined word size $k$.

## 3 Word Matching Properties

Word matching is fast, but is it any good?

- Word matching is parameterized by word length $k$
- How do we choose $k$? Sensitivity / specificity tradeoff
- **Specificity**: given two i.i.d. random, unrelated DNA sequences $s$, $t$ with equal base freqs, how long a match is likely to occur by chance alone?

$$\begin{aligned}\Pr(s[i] = t[i]) &= \frac{1}{4} \\ \Pr(s[1 \ldots k] = t[1 \ldots k]) &= \left(\frac{1}{4}\right)^k\end{aligned}$$

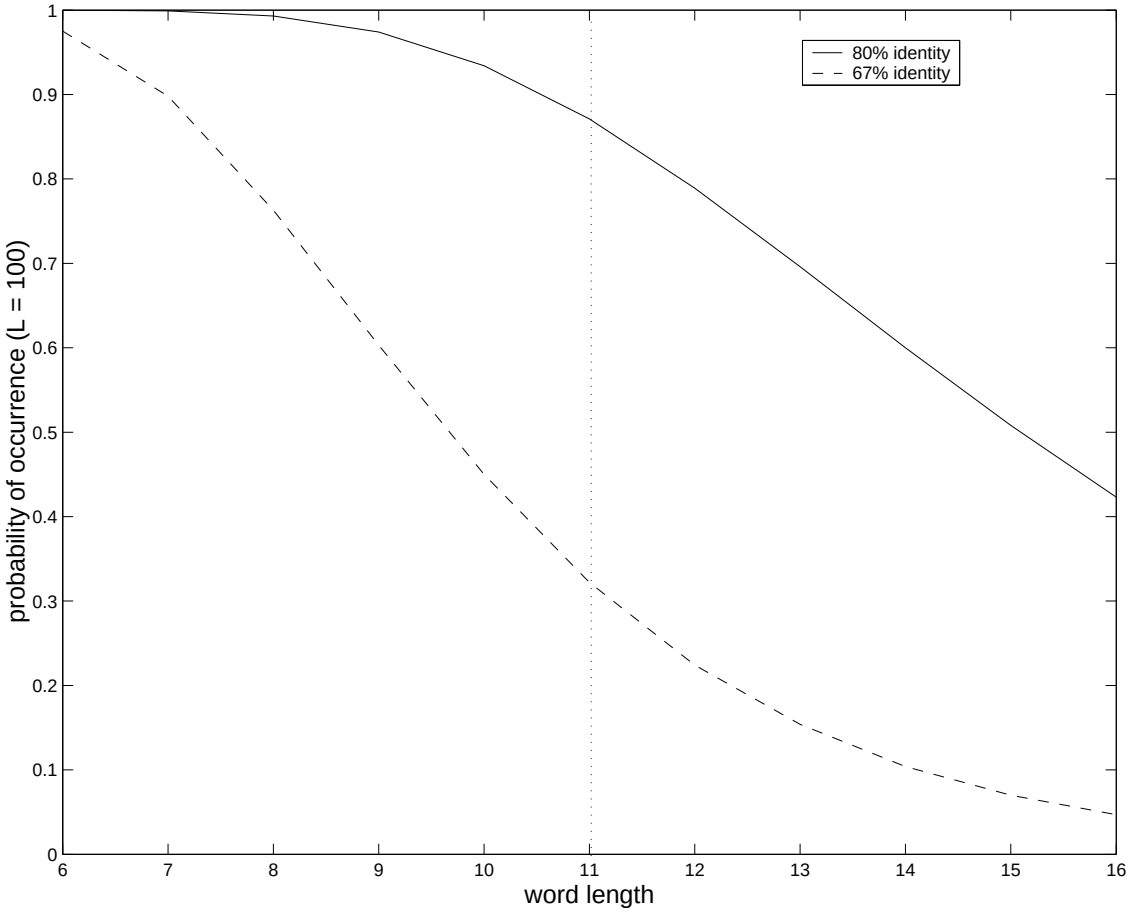
Let $M = |s| \cdot |t|$, and let $k = \log_4 M$. Then

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

- 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.
- Number of matches grows exponentially as $k$ decreases – about $|s||t|/4^k$ $k$-mer matches expected by chance alone.
- same example: about 1,000 15-mer matches, 60,000 12-mer matches, 1 million 10-mer matches
- In practice, nonuniform and non-i.i.d. sequence has a higher rate of chance matches
- Doesn't count biologically meaningful but uninteresting matches (e.g. tandem repeats, low-complexity sequence)

OK, what about the **sensitivity of word matching**?

- Harder to analyze – what is frequency of word matches in pairs of conserved features (orthologous or paralogous)?
- Rough idea: consider the set of all alignments of some length $N$ (length of alignment path) that contain exactly $fN$ matching base pairs. $100f$ is alignment's *percent identity*.
- Assume that the non-matching positions (mutations) are distributed *uniformly at random* throughout the alignment. (Harder than real aignments, but we'll assume this.)
- Suppose we can detect these alignments only if they contain at least one word match of length $\geq k$.
- How large a value of $k$ is likely to detect "most" alignments?




- Practical result: a match length of around 12 bases catches most alignments with 75-80% identity. However, to discover matches with 67% identity, a better length would be 8-9 bases.
- BLASTN: 11 bases; RepeatMasker: down to 8 bases for old repeats

What is practical implication?

- For homology search (BLAST), word match lengths expected in real similarities are well below lengths that occur frequently by chance alone.
- Hence, most word matches found will be spurious.
- Need fast filtering and other tricks to maintain efficiency!
- In contrast, when assembling reads (typical identity 95% or more), can safely raise threshold enough to prevent most spurious matches.

# 4 What About Protein?

For protein, word matches are typically much shorter and may not be exact.

- larger alphabet, so less chance of matches between unrelated sequences.
- comparable sensitivity/specificity tradeoff happens when $k = 3$ to 4.
- much more information in mismatches for amino acids! e.g., leucine-isoleucine vs leucine-tryptophan
- for this reason, BLAST looks not only for word matches to pattern but also for words that score highly when aligned to some word in the pattern (*neighborhood* strategy)
- implementation: enumerate all "good" matches to each pattern word, and put those in the hash table
- This works great, provided neighborhood is not too large.

## 5 Reducing Redundant Candidates

Word matching with a fixed length produces many redundant candidates. Example [**long exact match with many successive word matches**]:

(This is less of an issue if we use a word matching technique that detects variable-length, maximal exact matches.) How can we discover and remove redundant matches on the fly? Nice technique described in 1995 paper by Pevzner and Waterman.

- **Defn**: a *diagonal* of a DP matrix (as for Smith-Waterman) is the set of all cells ($i$,$j$) for which $j - i$ is constant. [**picture**]

- Every word match has a unique diagonal.
- Moreover, nearby word matches on same diagonal cause us to explore roughly the same region of the DP matrix.
- **Conclude**: eliminate nearby matches on the same diagonal

How do we track nearby matches on the same diagonal? Remember that we process the corpus in order from lower to higher indices.

- Allocate an array $A$ indexed by diagonal (from $-m$ to $n$) **[picture here]**
- Suppose we find a word match on diagonal $d$ at position $j$ in the corpus.
- Look at $A[d]$. If $j - A[d]$ is small (e.g. less than word length plus a small constant), word match at $j$ is redundant to some match found recently on diagonal $d$.
- Otherwise, set $A[d] \leftarrow j$ and process match at $j$
- **Note**: BLAST tracks not location of last word match but end of last ungapped extension for redundancy filtering.

This approach is practical if total number of diagonals in matrix is small (at most hundreds of thousands).
If pattern much shorter than corpus, can actually use array of size proportional to pattern by observing that some diagonals are "finished" early on in the algorithm and reusing their array slots.

**What else can we do with diagonal tracking technology?**

- Here's an idea: instead of processing each individual word match, only apply DP algorithm if *two* such matches appear close together on same diagonal
- can easily detect such "double hits" with above technique
- two nearby hits less likely by chance than one hit, so can obtain same specificity with shorter word matches (good!)
- BLAST uses this "two-hit" technique for protein sequences